File No. S360-29
Form C28-8201-1

OS

IBM Systems Reference Library

IBM System/360

PL/I Reference Manual

This publication provides the rules for writing PL/I programs that are to be compiled using the PL/I (F) Compiler under the IBM System/360 Operating System.

Second Edition (March 1968)

This edition, C28-8201-1, obsoletes the previous edition, C28-8201-0. Chapters 14 and 15 are completely new and should be reviewed in their entirety; other changes are indicated by a vertical line to the left of the changed text, while changes to illustrations are indicated by the symbol • to the left of the caption.

Specifications contained herein are subject to change from time to time. Any such change will be reported in subsequent revisions or Technical Newsletters.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special print chain.

A form is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to IBM United Kingdom Laboratories Ltd., Programming Publications, Hursley Park, Winchester, Hampshire, England.

PREFACE

This publication is planned for use as a reference book by the PL/I programmer. It is not intended to be a tutorial publication, but is designed for the reader who already has a knowledge of the language and who requires a source of reference material.

It is divided into two parts. Part I contains discussions of concepts of the language. Part II contains detailed rules and syntactic descriptions.

Although implementation information is included, the book is not a complete description of any implementation environment. In general, it contains information needed in writing a program; it does not contain all of the information required to execute a program.

The following features, discussed in this publication, are implemented in the fourth version of the F Compiler but are not implemented in the third version:

- Based storage facilities:

  The BASED, POINTER, AREA, and OFFSET attributes;

  The ADDR, NULL, NULLO, and EMPTY built-in functions;

  The AREA condition;

  Area-to-area and locator-to-locator assignment;

  The LOCATE statement;

  The IN option on the ALLOCATE and FREE statements;

  The SET option on the READ, LOCATE, and ALLOCATE statements;

  The REFER option in a based structure declaration;

  The pointer qualification symbol.

- Multitasking facilities:

  The TASK, EVENT, and PRIORITY options on the CALL statement;

  The EVENT option on the DISPLAY statement with the REPLY option;

  The TASK option in the OPTIONS list;

  The TASK attribute;

  Explicit declaration of event variables;

  The use of event arrays in the WAIT statement, and the use of an expression specifying the number of events to be waited for;

  The STATUS and PRIORITY built-in functions and pseudo-variables;

  The EXCLUSIVE file attribute;

  The UNLOCK statement;

  The NOLOCK option on the READ statement.

- Data interchange: the COBOL option in the ENVIRONMENT attribute.

- Carriage control: the CTLASA and CTL360 options in the ENVIRONMENT attribute.

- The STRINGRANGE condition.

- Omission of the data list from a data-directed PUT statement.

- Use of the LINESIZE option for any stream output file; use of the SKIP option and the COLUMN and SKIP format items, in GET statements for stream input files.

- Use of VARYING strings in the INTO and FROM options of record-oriented input/output statements.

- Omission of the KEY option from the DELETE statement (to allow deletion from SEQUENTIAL UPDATE files for INDEXED data sets).

REQUISITE PUBLICATION

For information necessary to compile, linkage edit, and execute a program, the reader should be familiar with the following publication:

IBM System/360 Operating System, PL/I (F) Programmer's Guide, Form C28-6594

RECOMMENDED PUBLICATIONS

The following publications contain other information that might be valuable to the PL/I programmer or to a programmer who is learning PL/:

A PL/I Primer, Form C28-6808

A Guide to PL/I for Commercial Programmers, Form C20-1651

A Guide to PL/I for FORTRAN Users, Form C20-1637

CONTENTS

TABLES

PL/I is a programming language designed to cover as wide a range of programming applications as possible. A basic belief underlying the design of PL/I is that programmers have common problems, regardless of the different applications with which they may be concerned.

The language also is designed to reduce the cost of programming, including the cost of training programmers, the cost of debugging, and, in particular, the cost of program maintenance.

Training programmers to use a particular language can often be expensive, particularly if each programmer must be taught the entire language, even if he need use only a part of it. One of the prime features in the design of PL/I is modularity; in general, a programmer need know only as much of the language as he requires to solve his problems.

Another factor that contributes to programming cost is that a program frequently must be rewritten, sometimes because the system under which it is used has changed, sometimes because the program is to be run on a new machine. It is not uncommon to find that rewriting a program costs as much as writing it in the first place.

Two basic characteristics of PL/I are intended to reduce the need to rewrite complete programs if either the machine environment or the application environment changes. These characteristics are the block structure used in the language and its machine independence.

A PL/I program is composed of blocks of statements called procedure blocks (or procedures) and begin blocks, each of which defines a region of the program. A single program may consist of one procedure or of several procedures and begin blocks. Either a procedure block or a begin block can contain other blocks; a begin block must be contained in a procedure block. Each external procedure, that is, a procedure that is not contained in another procedure, is compiled separately. The same external procedure might be used in a number of different programs. Consequently, a necessary change made in that one block effectively makes the change in all programs that use it.

PL/I is much less machine dependent than most commonly used programming languages. In the interest of efficiency, however, certain features are provided that allow machine dependence for those cases in which complete independence would be too costly.

The variety of features provided by PL/I, as well as the simplicity of the concepts underlying them, demonstrate the versatility of the language, its universality, and the ease with which different subsets can be defined to meet the needs of different users.

USE OF THIS PUBLICATION

This publication is designed as a reference book for the PL/I programmer. Its two-part format allows a presentation of the material in such a way that references can be found quickly, in as much or as little detail as the user needs.

Part I, "Concepts of PL/I," is composed of discussions and examples that explain the different features of the language and their interrelationships. To reduce the need for cross references and to allow each chapter to stand alone as a complete reference to its subject, some information is repeated from one chapter to another. Part I can, nevertheless, be read sequentially in its entirety.

Part II, "Rules and Syntactic Descriptions," provides a quick reference to specific information. It includes less information about interrelationships, but it is organized so that a particular question can be answered quickly. Part II is organized purely from a reference point of view; it is not intended for sequential reading.

For example, a programmer would read Chapter 5 in Part I, "Statement Classification," for information about the interactions of different statements in a program; but he would look in Section J of Part II, "Statements," to find all the rules for the use of a specific statement, its effect, options allowed, and the format in which it is written.

In the same manner, he would read Chapter 4 in Part I, "Expressions," for a discussion of the concepts of data conversion, but he would use Section F of Part II, "Problem Data Conversion," to determine the exact results of a particular type of conversion.

An explanation of the syntax language used in this publication to describe elements of PL/I is contained in Part II, Section A, "Syntax Notation."

IMPLEMENTATION CONSIDERATIONS

This publication reflects features of the fourth version of the F Compiler. No attempt is made to provide complete implementation information; this publication is designed for use in conjunction with IBM System/360 Operating System: PL/I (F) Programmer's Guide, Form C28-6594. Discussion of implementation is limited to those features that are required for a full explanation of the language. For example, references to certain parameters of the Data Definition (DD) job control language statement are essential to an explanation of record-oriented input and output file organization.

Implementation features identified by the phrase "for System/360 implementations..." apply to all implementations for IBM System/360 computers. Features identified by the phrase "for the F Compiler..." apply specifically to the IBM F Compiler under the IBM System/360 Operating System.

A separate publication, IBM System/360: PL/I Subset Reference Manual, Form C28-8202, provides the same type of implementation information as it applies to the D Compiler used under the IBM System/360 Disk and Tape Operating Systems.

PART I

The modularity of PL/I, the ease with which subsets can be defined to meet different needs, becomes apparent when one examines the different features of the language. Such modularity is one of the most important characteristics of PL/I.

This chapter contains brief discussions of most of the basic features to provide an overall description of the language. Each is treated in more detail in subsequent chapters. An annotated example in Chapter 14, "A PL/I Program," illustrates the use of many of these features.

MACHINE INDEPENDENCE

No language can be completely machine independent, but PL/I is much less machine dependent than most commonly used programming languages. The methods used to achieve this show in the form of restrictions in the language. The most obvious example is that data with different characteristics cannot in general share the same storage; to equate a floating-point number with a certain number of alphabetic characters would be to make assumptions about the representation of these data items which would not be true for all machines.

It is recognized that the price entailed by machine independence may sometimes be too high. In the interest of efficiency, certain features such as the UNSPEC built-in function and record-oriented data transmission, do permit a degree of machine dependence.

PROGRAM STRUCTURE

A PL/I program consists of one or more blocks of statements called procedures. A procedure may be thought of as a subroutine. Procedures may invoke other procedures, and these procedures or subroutines may either be compiled separately, or may be nested within the calling procedure and compiled with it. Each procedure may contain declarations that define names and control allocation of storage.

The rules defining the use of procedures, communication between procedures, the meaning of names, and allocation of storage are fundamental to the proper understanding of PL/I at any level but the most elementary. These rules give the programmer considerable control over the degree of interaction between subroutines. They permit flexible communication and storage allocation, at the same time allowing the definition of names and allocation of storage for private use within a procedure.

By giving the programmer freedom to determine the degree to which a subroutine is self-contained, PL/I makes it possible to write procedures which can freely be used in other environments, while still allowing interaction in procedures where interaction is desirable.

MULTIPROGRAMMING

By means of the PL/I multitasking facilities, the programmer can specify that an invoked procedure is to be executed concurrently with the invoking procedure, thus making use of the multiprogramming capabilities of the system. In this way, the central processing unit can be occupied with one part of the program while the input/output channels are occupied with other parts of the program; this can reduce the overall amount of waiting time during execution.

Concurrent execution of different parts of a program does not imply that the program cannot be coordinated. The programmer can specify that execution of a procedure will be suspended at a specified point until some point in another procedure has been reached, or until an input/output operation has been completed.

DATA TYPES AND DATA DESCRIPTION

The characteristic of PL/I that most contributes to the range of applications for which it can be used is the variety of data types that can be represented and manipulated. PL/I deals with arithmetic data, string data (bit and character), and program control data, such as labels. Arithmetic data may be represented in a variety of ways; it can be binary or

decimal, fixed-point or floating-point, real or complex, and its precision may be specified.

PL/I provides features to perform arithmetic operations, operations for comparisons, logical manipulation of bit strings, and operations and functions for assembling, scanning, and subdividing character strings.

The compiler must be able to determine, for every name used in a program, the complete set of attributes associated with that name. The programmer may specify these attributes explicitly by means of a DECLARE statement, the compiler may determine all or some of the attributes by context, or the attributes may be assumed by default.

DEFAULT ASSUMPTIONS

An important feature of PL/I is its default philosophy. If all the attributes associated with a name, or all the options permitted in a statement, are not specified by the programmer, attributes or options may be assigned by the compiler. This default action has two main consequences. First, it reduces the amount of declaration and other program writing required; second, it makes it possible to teach and use subsets of the language for which the programmer need not know all possible alternatives, or even that alternatives exist.

Since defaults are based on assumptions about the intent of the programmer, errors or omissions may be overlooked, and incorrect attributes may be assigned by default. To reduce the chance of this, the F Compiler optionally provides an attribute listing, which can be used to check the names in the program and the attributes associated with them.

STORAGE ALLOCATION

PL/I goes beyond most other languages in the flexibility of storage allocation that it provides. Dynamic storage allocation is comparatively difficult for an assembly language programmer to handle for himself; yet it is automatically provided in PL/I. There are four different storage classes: AUTOMATIC, STATIC, CONTROLLED, and BASED. In general, the default storage class in PL/I is AUTOMATIC. This class of storage is allocated whenever the block in which the variables are declared is activated. At that time the bounds of arrays and the lengths of strings are calculated. AUTOMATIC storage is freed and is available for re-use whenever control leaves the block in which the storage is allocated.

Storage also may be declared STATIC, in which case it is allocated when the program is loaded; it may be declared CONTROLLED, in which case it is explicitly controlled by the programmer with ALLOCATE and FREE statements, independent of the invocation of blocks; or it may be declared BASED, which gives the programmer an even higher degree of control.

The existence of several storage classes enables the programmer to determine for himself the speed, storage space, or programming economy that he needs for each application. The cost of a particular facility will depend upon the implementation, but it will usually be true that the more dynamic the storage allocation, the greater the overhead in execution time.

EXPRESSIONS

Calculations in PL/I are specified by expressions. An expression has a meaning in PL/I that is similar to that of elementary algebra. For example:

```
A + B * C
```

This specifies multiplication of the value of B by the value of C and adding the value of A to the result. PL/I places few restrictions on the kinds of data that can be used in an expression. For example, it is conceivable, though unlikely, that A could be a floating-point number, B a fixed-point number, and C a character string.

When such mixed expressions are specified, the operands will be converted so that the operation can be evaluated meaningfully. Note, however, that the rules for conversion must be considered carefully; converted data may not have the same value as the original. And, of course, any conversion requires additional compiler-generated coding, which increases execution time.

The results of the evaluation of expressions are assigned to variables by means of the assignment statement. An example of an assignment statement is:

```
X = A + B * C;
```

This means: evaluate the expression on the right and store the result in X. If the attributes of X differ from the attributes of the result of the expression, conversion will again be performed.

DATA COLLECTIONS

PL/I permits the programmer many ways of describing and operating on collections of data, or data aggregates. Arrays are collections of data elements, all of the same type, collected into lists or tables of one or more dimensions. Structures are hierarchical collections of data, not necessarily all of the same type. Each level of the hierarchy may contain other structures of deeper levels. The deepest levels of the hierarchy represent elementary data items or arrays.

An element of an array may be a structure; similarly, any level of a structure may be an array. Operations can be specified for arrays, structures, or parts of arrays or structures. For example:

```
A = B + C;
```

In this assignment statement, A, B, and C could be arrays or structures.

INPUT AND OUTPUT

Facilities for input and output allow the user to choose between factors such as simplicity, machine independence, and efficiency. There are two broad classes of input/output in PL/I: stream-oriented and record-oriented.

Stream-oriented input/output is almost completely machine independent. On input, data items are selected one by one from what is assumed to be a continuous stream of characters that are converted to internal form and assigned to variables specified in a list. Similarly, on output, data items are converted one by one to external character form and are added to a conceptually continuous stream of characters. Within the class of stream input/output, the programmer can choose different levels of control over the way data items are edited and selected from or added to the stream.

For printing, the output stream may be considered to be divided into lines and pages. An output stream file may be declared to be a print file with a specified line size and page size. The programmer has facilities to detect the end of a page and to specify the beginning of a line or a page. These facilities may be used in subroutines that can be developed into a report generating system suitable for a particular installation or application.

Record-oriented input/output is machine dependent. It deals with collections of data, called records, and transmits these a record at a time without any data conversion; the external representation is an exact copy of the internal representation. Because the aggregate is treated as a whole, and because no conversion is performed, this form of input/output is potentially more efficient than stream-oriented input/output, although the actual efficiency of each class will, of course, depend on the implementation.

Stream-oriented input and output usually sacrifices efficiency for ease of handling. Each data item is transmitted separately and is examined to determine if data conversion is required. Record-oriented input and output, on the other hand, provides faster transmission by transmitting data as entire records, without conversion.

COMPILE-TIME OPERATIONS

Most programming is concerned only with operations upon data. PL/I permits a compile-time level of operation, in which preprocessor statements specify operations upon the text of the source program itself. The simplest, and perhaps the commonest preprocessor statement is %INCLUDE (in general, preprocessor statements are preceded by a percent sign). This statement causes text to be inserted into the program, replacing the %INCLUDE statement itself. A typical use could be to copy declarations from an installation's standard set of definitions into the program.

Another function provided by compile-time facilities is the selective compilation of program text. For example, it might specify the inclusion or deletion of debugging statements.

Since a simple but powerful part of the PL/I language is available for compile-time activity, the generation, or replacement and deletion, of text can become more elaborate, and more subtle transformations can be performed. Such transformations might then be considered to be installation-defined extensions to the language.

INTERRUPT ACTIVITIES

Modern computing systems provide facilities for interrupting the execution of a program whenever an exceptional condition arises. Further, they allow the program to deal with the exceptional condition and to return to the point at which the interrupt occurred.

PL/I provides facilities for detecting a variety of exceptional conditions. It allows the programmer to specify, by means of a condition prefix, whether certain interrupts will or will not occur if the condition should arise. And, by use of an ON statement, he can specify the action to be taken when an interrupt does occur.

There are few restrictions in the format of PL/I statements. Consequently, programs can be written without consideration of special coding forms or checking to see that each statement begins in a specific column. As long as each statement is terminated by a semicolon, the format is completely free. Each statement may begin in the next column or position after the previous statement, or any number of blanks may intervene.

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. For a given external procedure, 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 alphabetic characters.

There are 29 alphabetic characters beginning with the currency symbol ($), the number sign (#), and the commercial "at" sign (@), which precede the 26 letters of the English alphabet in the IBM System/360 collating sequence in Extended Binary-Coded-Decimal Interchange Code (EBCDIC). For use with languages other than English, the first three alphabetic characters can be used to cause printing of letters that are not included in the standard English alphabet.

There are ten digits. The decimal digits are the digits 0 through 9. A binary digit is either a 0 or a 1.

There are 21 special characters. They are as follows:

Name	Character
Blank	
Equal sign or assignment symbol	=
Plus sign	+
Minus sign	-
Asterisk or multiply symbol	*
Slash or divide symbol	/
Left parenthesis	(
Right parenthesis	)
Comma	,
Point or period	.
Single quotation mark or apostrophe	'
Percent symbol	%
Semicolon	;
Colon	:
"Not" symbol	¬
"And" symbol	&
"Or" symbol	|
"Greater than" symbol	>
"Less than" symbol	<
Break character[1]	_
Question mark	?

Special characters are combined to create other symbols. For example, <= means "less than or equal to," ¬= means "not equal to." The combination ** denotes exponentiation (X**2 means X^2). Blanks are not permitted in such composite symbols.

An alphameric character is either an alphabetic character or a digit, but not a special character.

Note: The question mark, at present, has no specific use in the language, even though it is included in the 60-character set.

48-CHARACTER SET

The 48-character set is composed of 48 characters of the 60-character set. In all but four cases, the characters of the reduced set can be combined to represent the missing characters from the larger set. For example, the percent symbol (%) is not included in the 48-character set, but a double slash (//) can be used to represent it. The four characters that are not duplicated are the commercial "at" sign, the number sign, the break character, and the question mark.

The restrictions and changes for this character set are described in Part II,

---

[1]The break character is the same as the typewriter underline character. It can be used with a name, such as GROSS_PAY, to improve readability.

Section B, "Character Sets with EBCDIC and Card-Punch Codes."

USING THE CHARACTER SET

All the elements that make up a PL/I program are constructed from the PL/I character sets. There are two exceptions: character-string constants and comments may contain any character permitted by a particular machine configuration.

Certain characters perform specific functions in a PL/I program. For example, many characters function as operators.

There are four types of operators: arithmetic, comparison, bit-string, and string.

The arithmetic operators are:

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

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"

The bit-string operators are:

- `¬` denoting "not"
- `&` denoting "and"
- `|` denoting "or"

The string operator is:

- `||` denoting concatenation

Table 2-1 shows some of the functions of other special characters.

Table 2-1. Some Functions of Special Characters

Name	Character	Use
comma	,	Separates elements of a list
period	.	Indicates decimal point or binary point; connects elements of a qualified name
semicolon	;	Terminates statements
assignment symbol	=	Indicates assignment of values[1]
colon	:	Connects prefixes to statements; can be used in specification for bounds of an array
blank		Separates elements of a statement
single quotation mark	'	Encloses string constants and picture specification
parentheses	( )	Enclose lists; specify information associated with various keywords; in conjunction with operators and operands, delimit portions of a computational expression
arrow	->	Denotes pointer qualification
percent symbol	%	Indicates statements to be executed by the compiler preprocessor

[1]Note that the character = can be used as an equal sign and as an assignment symbol.

Identifiers

In a PL/I program, names or labels are given to data, files, statements, and entry points of different program areas. In creating a name or label, a programmer must observe the syntactic rules for creating an identifier.

An identifier is a single alphabetic character or a string of alphameric and break characters, not contained in a comment or constant, and preceded and followed by a blank or some other delimiter; the initial character of the string must be alphabetic. For System/360 implementation, the length must not exceed 31 characters.

Language keywords also are identifiers. A keyword is an identifier that, when used in proper context, has a specific meaning to the compiler. A keyword can specify such things as the action to be taken, the nature of data, the purpose of a name. For example, READ, DECIMAL, and ENDFILE are keywords. Some keywords can be abbreviated. A complete list of keywords and their abbreviations is contained in Part II, Section C, "Keywords and Keyword Abbreviations."

Note: PL/I keywords are not reserved words. They are recognized as keywords by the compiler only when they appear in their proper context. In other contexts they may be used as programmer-defined identifiers.

No identifier can exceed 31 characters in length; for the F Compiler, some identifiers, as discussed in later chapters, cannot exceed seven characters in length. This limitation is placed upon certain names, called external names, that may be referred to by the operating system or by more than one separately compiled procedure. If an external name contains more than seven characters, it is truncated by the compiler, which concatenates the first four characters with the last three characters.

Examples of identifiers that could be used for names or labels:

```
A

FILE2

LOOP_3

RATE_OF_PAY

#32
```

The Use of Blanks

Blanks may be used freely throughout a PL/I program. They may or may not surround operators and most other delimiters. In general, any number of blanks may appear wherever one blank is allowed, such as between words in a statement.

One or more blanks must be used to separate identifiers and constants that are not separated by some other delimiter or by a comment. However, identifiers, constants (except character-string constants) and composite operators (for example, ¬=) cannot contain blanks.

Other cases that require or permit blanks are noted in the text where the feature of the language is discussed. Some examples of the use of blanks are:

AB+BC is equivalent to AB + BC

TABLE(10) is equivalent to TABLE (10)

FIRST,SECOND is equivalent to FIRST, SECOND

ATOB is not equivalent to A TO B

Comments

Comments are permitted wherever blanks are allowed in a program, except within data items, such as a character string. A comment is treated as a blank and can therefore be used in place of a required separating blank. Comments do not otherwise affect execution of a program; they are used only for documentation purposes. Comments may be punched into the same cards as statements, either inserted between statements or in the middle of them.

The general format of a comment is:

```
/*   character-string   */
```

The character pair /* indicates the beginning of a comment. The same character pair reversed, */, indicates its end. No blanks or other characters can separate the two characters of either composite pair; the slash and the asterisk must be immediately adjacent. The comment itself may contain any characters except the */ combination, which would be interpreted as terminating the comment.

Example:

```
/*  THIS WHOLE SENTENCE COULD BE
    INSERTED AS A COMMENT  */
```

Any characters permitted for a particular machine configuration may be used in comments.

BASIC PROGRAM STRUCTURE

A PL/I program is constructed from basic program elements called statements. There are two types of statements: simple and compound. These statements make up larger program elements called groups and blocks.

SIMPLE AND COMPOUND STATEMENTS

There are three types of simple statements: keyword, assignment, and null, each of which contains a statement body that is terminated by a semicolon.

A keyword statement has a keyword to indicate the function of the statement; the statement body is the remainder of the statement.

The assignment statement contains the assignment symbol (=) and does not have a keyword.

The null statement consists only of a semicolon and indicates no operation; the semicolon is the statement body.

Examples of simple statements are:

```
GO TO LOOP_3;    (GO TO is a keyword; the
                  blank between GO and TO
                  is optional. The state-
                  ment body is LOOP_3;)

A = B + C;       (assignment statement)
```

A compound statement is a statement that contains one or more other statements as a part of its statement body. There are two compound statements: the IF statement and the ON statement. The final statement of a compound statement is a simple statement that is terminated by a semicolon. Hence, the compound statement is terminated by this semicolon. The IF statement can contain two simple statements as shown in the following example:

```
IF A>B THEN A = B+C; ELSE GO TO
   LOOP_3;
```

This example can also be written as follows:

```
IF A>B
    THEN A=B+C;
    ELSE GO TO LOOP_3;
```

Following are examples of the ON statement:

```
ON OVERFLOW GO TO OVFIX;

ON UNDERFLOW;
```

The contained statement in the second example is the null statement represented by a semicolon only; it indicates that no action is to be taken when an UNDERFLOW interrupt occurs.

Statement Prefixes

Both simple and compound statements may have one or more prefixes. There are two types of prefixes; the label prefix and the condition prefix.

A label prefix identifies a statement so that it can be referred to at some other point in the program. A label prefix is an identifier that precedes the statement and is connected to the statement by a colon. Any statement may have one or more labels. If more than one are specified, they may be used interchangeably to refer to that statement.

A condition prefix specifies whether or not interrupts are to result from the occurrence of the named conditions. Condition names are language keywords, each of which represents an exceptional condition that might arise during execution of a program. Examples are OVERFLOW and SIZE. The OVERFLOW condition arises when the exponent of a floating-point number exceeds the maximum allowed (representing a maximum value of about 10^{75}). The SIZE condition arises when a value is assigned to a variable with loss of high-order digits or bits.

A condition name in a condition prefix may be preceded by the word NO to indicate that, effectively, no interrupt is to occur if the condition arises. If NO is used, there can be no intervening blank between the NO and the condition name.

A condition prefix consists of a list of one or more condition names, separated by commas and enclosed in parentheses. One or more condition prefixes may be attached to a statement, and each parenthesized list must be followed by a colon. Condition prefixes precede the entire statement, including any possible label prefixes for the statement. For example:

```
(SIZE,NOOVERFLOW):COMPUTE:A = B * C ** D;
```

The single condition prefix indicates that an interrupt is to occur if the SIZE condition arises during execution of the assignment statement, but that no interrupt is to occur if the OVERFLOW condition arises. Note that the condition prefix precedes the label prefix COMPUTE.

Since intervening blanks between a prefix and its associated statement are ignored, it is often convenient to punch the condition prefix into a separate card that precedes the card into which the statement is punched. Thus, after debugging, the prefix can be easily removed. For example:

```
(NOCONVERSION):

(SIZE,NOOVERFLOW):

COMPUTE:  A = B * C ** D;
```

Note that there are two condition prefixes. The first specifies that no interrupt is to occur if an invalid character is encountered during an attempted data conversion.

Condition prefixes are discussed in Chapter 11, "Exceptional Condition Handling and Program Checkout."

GROUPS AND BLOCKS

A group is a sequence of statements headed by a DO statement and terminated by a corresponding END statement. It is used for control purposes. A group also may be called a DO-group.

A block is a sequence of statements that defines an area of a program. It is used to delimit the scope of a name and for control purposes. A program may consist of one or more blocks. Every statement must appear within a block. There are two kinds of blocks: begin blocks and procedure blocks. A begin block is delimited by a BEGIN statement and an END statement. A procedure block is delimited by a PROCEDURE statement and an END statement. Every begin block must be contained within some procedure block.

Execution passes sequentially into and out of a begin block. However, a procedure block must be invoked by execution of a statement in another block. The first procedure in a program to be executed is invoked automatically by the operating system. For System/360 implementations, this first procedure must be identified by specifying OPTIONS (MAIN) in the PROCEDURE statement.

A procedure block may be invoked as a task, in which case it is executed concurrently with the invoking procedure. Tasks are discussed in Chapter 15, "Multitasking."

CHAPTER 3: DATA ELEMENTS

Data is generally defined as a representation of information or of value.

In PL/I, reference to a data item, arithmetic or string, is made by using either a variable or a constant (the terms are not exactly the same as in general mathematical usage).

A variable is a symbolic name having a value that may change during execution of a program.

A constant (which is not a symbolic name) has a value that cannot change.

The following statement has both variables and constants:

```
AREA = RADIUS**2*3.1416;
```

AREA and RADIUS are variables; the numbers 2 and 3.1416 are constants. The value of RADIUS is a data item, and the result of the computation will be a data item that will be assigned as the value of AREA. The number 3.1416 in the statement is itself the data item; the characters 3.1416 also are written to refer to the data item.

If the number 3.1416 is to be used in more than one place in the program, it may be convenient to represent it as a variable to which the value 3.1416 has been assigned. Thus, the above statement could be written as:

```
PI = 3.1416;
AREA = RADIUS**2*PI;
```

In this statement, only the digit 2 is a constant.

In preparing a PL/I program, the programmer must be familiar with the types of data that are permitted, the ways in which data can be organized, and the methods by which data can be referred to. The following paragraphs discuss these features.

DATA TYPES

The types of data that may be used in a PL/I program fall into two categories: problem data and program control data. Problem data is used to represent values to be processed by a program. It consists of two data types, arithmetic and string. Program control data is used by the programmer to control the execution of his program. Program control data consists of the following types: label, event, task, locator, and area.

A constant does more than state a value; it demonstrates various characteristics of the data item. For example, 3.1416 shows that the data type is arithmetic and that the data item is a decimal number of five digits and that four of these digits are to the right of the decimal point.

The characteristics of a variable are not immediately apparent in the name. Since these characteristics, called attributes, must be known, certain keywords and expressions may be used to specify the attributes of a variable in a DECLARE statement. The attributes used to describe each data type are discussed briefly in this chapter. A complete discussion of each attribute appears in Part II, Section I, "Attributes."

PROBLEM DATA

The types of problem data are arithmetic and string.

ARITHMETIC DATA

An item of arithmetic data is one with a numeric value. Arithmetic data items have the characteristics of base, scale, precision, and mode. The characteristics of data items represented by an arithmetic variable are specified by attributes declared for the name, or assumed by default.

The base of an arithmetic data item is either decimal or binary.

The scale of an arithmetic data item is either fixed-point or floating-point. A fixed-point data item is a number in which the position of the decimal or binary point is specified, either by its appearance in a constant or by a scale factor declared for a variable. A floating-point data item is a number followed by an optionally signed exponent. The exponent specifies the assumed position of the decimal or binary

point, relative to the position in which it appears.

The precision of an arithmetic data item is the number of digits the data item may contain, in the case of fixed-point, or the minimum number of significant digits (excluding the exponent) to be maintained, in the case of floating-point. For fixed-point data items, precision can also specify the assumed position of the decimal or binary point, relative to the rightmost digit of the number.

Whenever a data item is assigned to a fixed-point variable, the declared precision is maintained. The assigned item is aligned on the decimal or binary point. Leading zeros are inserted if the assigned item contains fewer integer digits than declared; trailing zeros are inserted if it contains fewer fractional digits. A SIZE error may occur if the assigned item contains too many integer digits; truncation on the right may occur if it contains too many fractional digits.

The mode of an arithmetic data item is either real or complex. A real data item is a number that expresses a real value. A complex data item is a pair of numbers: the first is real and the second is imaginary. For a variable representing complex data items, the base, scale, and precision of the two parts must be identical.

Base, scale, and mode of arithmetic variables are specified by keywords; precision is specified by parenthesized decimal integer constants.

In the following sections, the real arithmetic data types discussed are decimal fixed-point, sterling fixed-point, binary fixed-point, decimal floating-point, and binary floating-point. Any of these, except sterling fixed-point, can be used as the real part of a complex data item. The imaginary part of a complex number is discussed in the section "Complex Arithmetic Data," in this chapter.

Complex arithmetic variables must be explicitly declared with the COMPLEX attribute. Real arithmetic variables may be explicitly declared to have the REAL attribute, but it is not necessary to do so, since any arithmetic variable is assumed to be real unless it is explicitly declared complex.

Decimal Fixed-Point Data

A decimal fixed-point constant consists of one or more decimal digits with an optional decimal point. If no decimal point appears, the point is assumed to be immediately to the right of the rightmost digit. In most uses, a sign may optionally precede a decimal fixed-point constant.

Examples of decimal fixed-point constants as written in a program are:

```
3.1416

455.3

732

003

5280

.0012
```

The keyword attributes for declaring decimal fixed-point variables are DECIMAL and FIXED. Precision is stated by two decimal integers, separated by a comma and enclosed in parentheses. The first, which must be unsigned, specifies the total number of digits; the second, the scale factor, may be signed and specifies the number of digits to the right of the decimal point. If the variable is to represent integers, the scale factor and its preceding comma can be omitted. The attributes may appear in any order, but the precision specification must follow either DECIMAL or FIXED (or REAL or COMPLEX).

Following are examples of declarations of decimal fixed-point variables:

```
DECLARE A FIXED DECIMAL (5,4);

DECLARE B FIXED (6,0) DECIMAL;

DECLARE C FIXED (7,-2) DECIMAL;
```

The first DECLARE statement specifies that the identifier A is to represent decimal fixed-point items of not more than five digits, four of which are to be treated as fractional, that is, to the right of the assumed decimal point. Any item assigned to A will be converted to decimal fixed-point and aligned on the decimal point. The second DECLARE statement specifies that B is to represent integers of no more than 6 digits. Note that the comma and the zero are unnecessary; it could have been specified B FIXED DECIMAL (6). The third DECLARE statement specifies a negative scale factor of -2; this means that the assumed decimal point is two places to the right of the rightmost digit of the item.

The maximum number of decimal digits allowed for System/360 implementations is 15. Default precision, assumed when no specification is made, is (5,0). The

internal coded arithmetic form of decimal fixed-point data is packed decimal. Packed decimal is stored two digits to the byte, with a sign indication in the rightmost four bits of the rightmost byte. Consequently, a decimal fixed-point data item is always stored as an odd number of digits, even though the declaration of the variable may specify the number of digits (p) as an even number. When the declaration specifies an even number of digits, the extra digit place is in the high-order position, and it participates in any operations performed upon the data item, such as in a comparison operation. Any arithmetic overflow or assignment into an extra high-order digit place can be detected only if the SIZE condition is enabled.

Sterling Fixed-Point Data

PL/I has a facility for handling constants stated in terms of sterling currency value. The data may be written in a program with pounds, shillings, and pence fields, each separated by a period. Such data is converted and maintained internally as a decimal fixed-point number representing the equivalent in pence. A sterling data constant ends with the letter L, representing the pounds symbol. All three fields (pounds, shillings, and pence) must be present in a sterling constant. Note that the pence field is one or more decimal digits with an optional decimal point (the integer part must be less than 12 and cannot be omitted).

Examples of sterling fixed-point constants as written in a program are:

```
101.13.8L

1.10.0L

0.0.2.5L

2.4.6L
```

The third example represents twopence-halfpenny. The last example represents two pounds, four shillings, and six pence. It is converted and stored internally as 534 (pence).

There are no keyword attributes for declaring sterling variables, but a variable can be declared with a sterling picture, or sterling values may be expressed in pence as decimal fixed-point data. The precision of a sterling constant is the precision of its value expressed in pence.

Binary Fixed-Point Data

A binary fixed-point constant consists of one or more binary digits with an optional binary point, followed immediately by the letter B, with no intervening blank. In most uses, a sign may optionally precede the constant.

Examples of binary fixed-point constants as written in a program are:

```
10110B

11111B

101B

111.01B

1011.111B
```

The keyword attributes for declaring binary fixed-point variables are BINARY and FIXED. Precision is specified by two decimal integer constants, enclosed in parentheses, to represent the maximum number of binary digits and the number of digits to the right of the binary point, respectively. If the variable is to represent integers, the second digit and the comma can be omitted. The attributes can appear in any order, but the precision specification must follow either BINARY or FIXED (or REAL or COMPLEX).

Following is an example of declaration of a binary fixed-point variable:

```
DECLARE FACTOR BINARY FIXED (20,2);
```

FACTOR is declared to be a variable that can represent arithmetic data items as large as 20 binary digits, two of which are fractional. The decimal equivalent of that value range is from -262,144.00 through +262,143.75.

The maximum number of binary digits allowed for System/360 implementations is 31. Default precision is (15,0). The internal coded arithmetic form of binary fixed-point data is a fixed-point binary full word. A full word is 31 bits plus a sign bit. Any binary fixed-point data item is always stored as 31 digits, even though the declaration of the variable may specify fewer digits. The declared number of digits are considered to be in the low-order positions, but the extra high-order digits participate in any operations performed upon the data item. Any arithmetic overflow into such extra high-order digit positions can be detected only if the SIZE condition is enabled.

An identifier for which no declaration is made is assumed to be a binary fixed-point variable, with default precision, if its first letter is any of the letters I through N.

Decimal Floating-Point Data

A decimal floating-point constant is written as a field of decimal digits followed by the letter E, followed by an optionally signed decimal integer exponent. The first field of digits may contain a decimal point. The entire constant may be preceded by a plus or minus sign. Examples of decimal floating-point constants as written in a program are:

```
15E-23

15E23

4E-3

48333E65

438E0

3141593E-6

.003141593E3
```

The last two examples represent the same value.

The keyword attributes for declaring decimal floating-point variables are DECIMAL and FLOAT. Precision is stated by a decimal integer constant enclosed in parentheses. It specifies the minimum number of significant digits to be maintained. If an item assigned to a variable has a field width larger than the declared precision of the variable, truncation may occur on the right. The least significant digit is the first that is lost. Attributes may appear in any order, but the precision specification must follow either DECIMAL or FLOAT (or REAL or COMPLEX).

Following is an example of declaration of a decimal floating-point variable:

```
DECLARE LIGHT_YEARS DECIMAL FLOAT(5);
```

This statement specifies that LIGHT_YEARS is to represent decimal floating-point data items with an accuracy of at least five significant digits.

The maximum precision allowed for decimal floating-point data items for System/360 implementations is (16); the exponent cannot exceed two digits. A value range of approximately 10^{-78} to 10^{75} can be expressed by a decimal floating-point data item. Default precision is (6). The internal coded arithmetic form of decimal floating-point data is normalized hexadecimal floating-point, with the point assumed to the left of the first hexadecimal digit. If the declared precision is less than or equal to (6), short floating-point form is used; if the declared precision is greater than (6), long floating-point form is used.

An identifier for which no declaration is made is assumed to be a decimal floating-point variable if its first letter is any of the letters A through H, O through Z, or one of the alphabetic extenders, $, #, @.

Binary Floating-Point Data

A binary floating-point constant consists of a field of binary digits followed by the letter E, followed by an optionally signed decimal integer exponent followed by the letter B. The exponent is a string of decimal digits and specifies an integral power of two. The field of binary digits may contain a binary point. A binary floating-point constant may be preceded by a plus or minus sign. Examples of binary floating-point constants as written in a program are:

```
101101E5B

101.101E2B

11101E-28B
```

The keyword attributes for declaring binary floating-point variables are BINARY and FLOAT. Precision is expressed as a decimal integer constant, enclosed in parentheses, to specify the minimum number of significant digits to be maintained. The attributes can appear in any order, but the precision specification must follow either BINARY or FLOAT (or REAL or COMPLEX). Following is an example of declaration of a binary floating-point variable:

```
DECLARE S BINARY FLOAT (16);
```

This specifies that the identifier S is to represent binary floating-point data items with 16 digits in the binary field.

The maximum precision allowed for binary floating-point data items for System/360 implementations is (53); default precision is (21). The exponent cannot exceed three decimal digits. A value range of approximately 2^{-260} to 2^{252} can be expressed by a

binary floating-point data item. The internal coded arithmetic form of binary floating-point data is normalized hexadecimal floating-point. If the declared precision is less than or equal to (21), short floating-point form is used; if the declared precision is greater than (21), long floating-point form is used.

Complex Arithmetic Data

In the complex mode, an arithmetic data item is considered to consist of two parts, the first a real part and the second a signed imaginary part. There are no complex constants in PL/I. The effect is obtained by writing a real constant and an imaginary constant.

An imaginary constant is written as a real constant of any type (except sterling fixed-point) immediately followed by the letter I.

Examples of imaginary constants as written in a program are:

```
27I

3.968E10I

11011.01BI
```

Each of these is considered to have a real part of zero. Although complex constants cannot be written with a nonzero real part, PL/I provides the facility to express such values in the following form:

```
real-constant{+|-}imaginary-constant
```

Thus a complex value could be written as 38+27I.

The keyword attribute for declaring a complex variable is COMPLEX. A complex variable can have any of the attributes valid for the different types of real arithmetic data. Each of the base, scale, and precision attributes applies to both fields.

Unless a variable is explicitly declared to have the COMPLEX attribute, it is assumed to represent real data items.

Numeric Character Data

A numeric character data item (also known as a numeric field data item) is the value of a variable that has been declared with the PICTURE attribute and a numeric picture specification. The data item is the character representation of a decimal fixed-point or floating-point value.

A numeric picture specification describes a character string to which only data that has, or can be converted to, an arithmetic value is to be assigned. A numeric picture specification cannot contain either of the picture characters A or X, which are used for non-numeric picture-character strings. The basic form of a numeric picture specification is one or more occurrences of the digit-specifying picture character 9 and an optional occurrence of the picture character V, to indicate the assumed location of a decimal point. The picture specification must be enclosed in single quotation marks. For example:

```
'999V99'
```

This numeric picture specification describes a data item consisting of up to five decimal digits in character form, with a decimal point assumed to precede the rightmost two digits.

Repetition factors may be used in numeric picture specifications. A repetition factor is a decimal integer constant, enclosed in parentheses, that indicates the number or repetitions of the immediately following picture character. For example, the following picture specification would result in the same description as the example shown above:

```
'(3)9V(2)9'
```

The format for declaring a numeric character variable is:

```
DECLARE identifier PICTURE
   'numeric-picture-specification';
```

For example:

```
DECLARE PRICE PICTURE '999V99';
```

This specifies that any value assigned to PRICE is to be maintained as a character string of five decimal digits, with an assumed decimal point preceding the rightmost two digits. Data assigned to PRICE will be aligned on the assumed point in the same way that point alignment is maintained for fixed-point decimal data.

The numeric picture specification can specify all of the arithmetic attributes of data in much the same way that they are specified by the appearance of a constant. Only decimal numeric data can be represented by picture characters. Complex data can

be declared by specifying the COMPLEX attribute along with a single picture specification that describes either a fixed-point or a floating-point data item.

It is important to note that, although numeric character data has arithmetic attributes, it is not stored in coded arithmetic form. In System/360 implementations, numeric character data is stored in zoned decimal format; before it can be used in arithmetic computations, it must be converted either to packed decimal or to hexadecimal floating-point format. Such conversions are done automatically, but they require extra execution time.

Although numeric character data is in character form, like character strings, and although it is aligned on the decimal point like coded arithmetic data, it is processed differently from the way either coded arithmetic items or character strings are processed. Editing characters can be specified for insertion into a numeric character data item, and such characters are actually stored within the data item. Consequently, when the item is printed or treated as a character string, the editing characters are included in the assignment. If, however, a numeric character item is assigned to another numeric character or arithmetic variable, the editing characters will not be included in the assignment; only the actual digits and the location of the assumed decimal point are assigned.

Consider the following example:

```
DECLARE PRICE PICTURE '$99V.99',
        COST CHARACTER (6),
        VALUE FIXED DECIMAL (6,2);

PRICE = 12.28;

COST = '$12.28';
```

In the picture specification for PRICE, the currency symbol ($) and the decimal point (.) are editing characters. They are stored as characters in the data item. They are not, however, a part of its arithmetic value. After execution of the second assignment statement, the actual internal character representation of PRICE and COST can be considered identical. If they were printed, they would print exactly the same. They do not, however, always function the same. For example:

```
VALUE = PRICE;

COST = PRICE;

VALUE = COST;

PRICE = COST;
```

After the first two assignment statements are executed, the value of VALUE would be 0012.28 and the value of COST would be '$12.28'. In the assignment of PRICE to VALUE, the currency symbol and the decimal point are considered to be editing characters, and they are not part of the assignment; the arithmetic value of PRICE is converted to internal coded arithmetic form. In the assignment of PRICE to COST, however, the assignment is to a character string, and the editing characters of a numeric picture specification always participate in such an assignment. No conversion is necessary because PRICE is stored in character form.

The third and fourth assignment statements would cause errors. The value of COST cannot be assigned to VALUE because the currency symbol in the string makes it invalid as an arithmetic constant. The value of COST cannot be assigned to PRICE for exactly the same reason. Only values that are of arithmetic type, or that can be converted to arithmetic type, can be assigned to a variable declared with a numeric picture specification.

Note: Although the decimal point can be an editing character or an actual character in a character string, it will not cause an error in converting to arithmetic form, since its appearance is valid in an arithmetic constant. The same would be true of a valid plus or minus sign, since arithmetic constants can be preceded by signs.

Other editing characters, including zero suppression characters, drifting characters, and insertion characters, can be used in numeric picture specifications. For complete discussions of picture characters, see Part II, Section D, "Picture Specification Characters" and the discussion of the PICTURE attribute in Part II, Section I, "Attributes."

STRING DATA

A string is a contiguous sequence of characters (or binary digits) that is treated as a single data item. The length of the string is the number of characters (or binary digits) it contains.

There are two types of strings: character strings and bit strings.

Character-String Data

A character string can include any digit, letter, or special character recognized as a character by the particular machine configuration. Any blank included in a character string is an integral character and is included in the count of length. A comment that is inserted within a character string will not be recognized as a comment. The comment, as well as the comment delimiters (/* and */), will be considered to be part of the character-string data.

Character-string constants, when written in a program, must be enclosed in single quotation marks. If a single quotation mark is a character in a string, it must be written as two single quotation marks with no intervening blank. The length of a character string is the number of characters between the enclosing quotation marks. If two single quotation marks are used within the string to represent a single quotation mark, they are counted as a single character.

Examples of character-string constants are:

```
'LOGARITHM TABLE'

'PAGE 5'

'SHAKESPEARE''S ''''HAMLET'''''

'AC438-19'

(2)'WALLA '
```

The third example actually indicates SHAKESPEARE'S ''HAMLET'' WITH A LENGTH OF 24. In the last example, the parenthesized number is a repetition factor, which indicates repetition of the characters that follow. This example specifies the constant 'WALLA WALLA ' (the blank is included as one of the characters to be repeated). The repetition factor must be an unsigned decimal integer constant, enclosed in parentheses.

The keyword attribute for declaring a character-string variable is CHARACTER. Length is declared by an expression or a decimal integer constant, enclosed in parentheses, which specifies the number of characters in the string. The length specification must follow the keyword CHARACTER. For example:

```
DECLARE NAME CHARACTER (15);
```

This DECLARE statement specifies that the identifier NAME is to represent character-string data items, 15 characters in length. If a character string shorter than 15 characters were to be assigned to NAME, it would be left adjusted and padded on the right with blanks to a length of 15. If a longer string were assigned, it would be truncated on the right. (Note: If such truncation occurs, no interrupt will result as it might for truncation of arithmetic data, and there is no ON condition in PL/I to deal with string truncation.)

Character-string variables may also be declared to have the VARYING attribute, as follows:

```
DECLARE NAME CHARACTER (15) VARYING;
```

This DECLARE statement specifies that the identifier NAME is to be used to represent varying-length character-string data items with a maximum length of 15. The actual length attribute for NAME at any particular time is the length of the data item assigned to it at that time. The programmer need not keep track of the length of a varying-length character string; this is done automatically. The length at any given time can be determined by the programmer, however, by use of the LENGTH built-in function, as discussed in Chapter 9, "Editing and String Handling." Note for the F Compiler that until a varying-length string variable is given an initial value, its length is set to zero.

Character-string data in System/360 implementations is maintained internally in character format, that is, each character occupies one byte of storage. The maximum length allowed for variables declared with the CHARACTER attribute is 32,767. The maximum length allowed for a character-string constant after application of repetition factors varies according to the amount of storage available to the compiler, but it never will be less than 1,007 (see IBM System/360 Operating System: PL/I (F), Programmer's Guide, Form C28-6594). The minimum length for a character string is zero.

Character-string variables also can be declared using the PICTURE attribute of the form:

```
PICTURE 'character-picture-specification'
```

The character picture specification is a string composed of the picture specification characters A, X, and 9. The string of picture characters must be enclosed in single quotation marks, and it must contain at least one A or X and no other picture characters except 9. The character A specifies that the corresponding position in the described field will contain an alphabetic character or blank. The character X

specifies that any character may appear in the corresponding position in the field. The picture character 9 specifies that the

corresponding position will contain a numeric character or blank. For example:

```
DECLARE PART_NO PICTURE 'AA9999X999';
```

This DECLARE statement specifies that the identifier PART_NO will represent character-string data items consisting of two alphabetic characters, four numeric characters, one character that may be any character, and three numeric characters.

Repetition factors are used in picture specifications differently from the way they are used in string constants. Repetition factors must be placed inside the quotation marks. The repetition factor specifies repetition of the immediately following picture character. For example, the above picture specification could be written:

```
'(2)A(4)9X(3)9'
```

The maximum length allowed for a picture specification is the same as that allowed for character-string constants, as discussed above.

Note that, for character picture specifications, the picture character 9 specifies a digit or a blank, while, for numeric picture specifications, the same character specifies only a digit.

Bit-String Data

A bit-string constant is written in a program as a series of binary digits enclosed in single quotation marks and followed immediately by the letter B.

Examples of bit-string constants as written in a program are:

```
'1'B

'11111010110001'B

(64)'0'B
```

The parenthesized number in the last example is a repetition factor which specifies that the following series of digits is to be repeated the specified number of times. The example shown would result in a string of 64 binary zeros.

A bit-string variable is declared with the BIT keyword attribute. Length is specified by an expression or a decimal integer constant, enclosed in parentheses, to specify the number of binary digits in the string. The letter B is not included in the length specification since it is not part of the string. The length specification must follow the keyword BIT. Following is an example of declaration of a bit-string variable:

```
DECLARE SYMPTOMS BIT (64);
```

Like character strings, bit strings are assigned to variables from left to right. If a string is longer than the length declared for the variable, the rightmost digits are truncated; if shorter, padding, on the right, is with zeros.

A bit-string variable may be given the VARYING attribute to indicate it is to be used to represent varying-length bit strings. Its application is the same as that described for character-string variables in the preceding section.

With System/360 implementations, bit strings are stored eight bits to a byte. The maximum length allowed for a bit-string variable with the F Compiler is 32,767. The maximum length allowed for a bit-string constant after application of repetition factors depends upon the amount of storage available to the compiler, but it will never be less than 8,056 (1,007 bytes). The minimum length for a bit string is zero.

PROGRAM CONTROL DATA

The types of program control data are label, event, task, locator, and area.

LABEL DATA

A label data item is a label constant or the value of a label variable.

A label constant is an identifier written as a prefix to a statement so that, during execution, program control can be transferred to that statement through a reference to its label. A colon connects the label to the statement.

```
ABCDE: DISTANCE = RATE*TIME;
```

In this example, ABCDE is the statement label. The statement can be executed either by normal sequential execution of instructions or by transferring control to this statement from some other point in the program by means of a GO TO statement.

As used above, ABCDE can be classified further as a statement-label constant. A

statement-label variable is an identifier that refers to statement-label constants. Consider the following example:

```
LBL_A:   statement;
             .
             .
             .
LBL_B:   statement;
             .
             .
             .
         LBL_X = LBL_A;
             .
             .
             .
         GO TO LBL_X;
             .
             .
             .
```

LBL_A and LBL_B are statement-label constants because they are prefixed to statements. LBL_X is a statement-label variable. By assigning LBL_A to LBL_X, the statement GO TO LBL_X causes a transfer to the LBL_A statement. Elsewhere, the program may contain a statement assigning LBL_B to LBL_X. Then, any reference to LBL_X would be the same as a reference to LBL_B. This value of LBL_X is retained until another value is assigned to it.

A statement-label variable must be declared with the LABEL attribute, as follows:

```
DECLARE LBL_X LABEL;
```

EVENT DATA

Event variables are used to coordinate the concurrent execution of a number of procedures, or to allow a degree of overlap between a record-oriented input/output operation and the execution of other statements in the procedure that initiated the operation.

A variable is given the EVENT attribute by its appearance in an EVENT option or a WAIT statement, or by explicit declaration, as in the following example:

```
DECLARE ENDEVT EVENT;
```

For detailed information, see Chapter 15, "Multitasking," and "The EVENT Option" in Chapter 8, "Input and Output."

TASK DATA

Task variables are used to control the relative priorities of different tasks (i.e., concurrent separate executions of a procedure or procedures).

A variable is given the TASK attribute by its appearance in a TASK option, or by explicit declaration, as in the following example:

```
DECLARE ADTASK TASK;
```

For detailed information, see Chapter 15, "Multitasking."

LOCATOR DATA

There are two types of locator data: pointer and offset.

The value of a pointer variable is effectively an address of a location in storage, and so it can be used to qualify a reference to a variable that may have been allocated storage in several different locations, all of which are immediately accessible. Since based storage is so allocated, reference to a based variable must be qualified in some way; with the F Compiler, this qualification must be provided by a pointer variable.

The value of an offset variable specifies a location relative to the start of a reserved area of storage and remains valid when the address of the area itself changes.

Locator variables can be declared as in the following example:

```
DECLARE HEADPTR POINTER,
        FIRST OFFSET (AREA1);
```

In this example, AREA1 is the name of the reserved area of storage that will contain the location specified by FIRST.

A variable can also be given the POINTER attribute by its appearance in the BASED attribute, by its appearance on the left-hand side of a pointer qualification symbol, or by its appearance in a SET option.

For detailed information, see Chapter 14, "Based Storage and List Processing."

AREA DATA

Area variables are used to describe areas of storage that are to be reserved for the allocation of based variables. An area can be assigned or transmitted complete with its contained allocations; thus, a set of based allocations can be treated as one unit for assignment and input/output while each allocation retains its individual identity.

A variable is given the AREA attribute either by its appearance in the OFFSET attribute or an IN option, or by explicit declaration, as in the following example:

```
DECLARE AREA1 AREA(2000),
        AREA2 AREA;
```

The number of bytes of storage to be reserved can be stated explicitly, as it has been for AREA1 in the example; otherwise a default size is assumed. For the F Compiler, this default size is 1000 bytes.

For detailed information, see Chapter 14, "Based Storage and List Processing."

DATA ORGANIZATION

In PL/I, data items may be single data elements, or they may be grouped together to form data collections called arrays and structures. A variable that represents a single element is an element variable (also called a scalar variable). A variable that represents a collection of data elements is either an array variable or a structure variable.

Any type of problem data or program control data can be collected into arrays or structures.

ARRAYS

Data elements having the same characteristics, that is, of the same data type and of the same precision or length, may be grouped together to form an array. An array is an n-dimensional collection of elements, all of which have identical attributes. Only the array itself is given a name. An individual item of an array is referred to by giving its relative position within the array.

Consider the following two declarations:

```
DECLARE LIST (8) FIXED DECIMAL (3);

DECLARE TABLE (4,2) FIXED DECIMAL (3);
```

In the first example, LIST is declared to be a one-dimensional array of eight elements, each of which is a fixed-point decimal item of three digits. In the second example, TABLE is declared to be a two-dimensional array, also of eight fixed-point decimal elements.

The parenthesized number or numbers following the array name in a DECLARE statement is the dimension attribute specification. It must follow the array name, with or without an intervening blank. It specifies the number of dimensions of the array and the bounds, or extent, of each dimension. Since only one bounds specification appears for LIST, it is a one-dimensional array. Two bounds specifications, separated by a comma, are listed for TABLE; consequently, it is declared to be a two-dimensional array.

The bounds of a dimension are the beginning and the end of that dimension. The extent is the number of integers between, and including, the lower and upper bounds. If only one integer appears in the bounds specification for a dimension, the lower bound is assumed to be 1. The one dimension of LIST has bounds of 1 and 8; its extent is 8. The two dimensions of TABLE have bounds of 1 and 4 and 1 and 2; the extents are 4 and 2.

If the lower bound of a dimension is not 1, both the upper bound and the lower bound must be stated explicitly, with the two numbers connected with a colon. For example:

```
DECLARE LIST_A (4:11);

DECLARE LIST_B (-4:3);
```

In the first example, the bounds are 4 and 11; in the second they are -4 and 3. Note that the extents are the same; in each case, there are 8 integers from the lower bound through the upper bound. It is important to note the difference between the bounds and the extent of an array. In the manipulation of array data (discussed in Chapter 4, "Expressions") involving more than one array, the bounds -- not merely the extents -- must be identical. Although LIST, LIST_A, and LIST_B all have the same extent, the bounds are not identical.

The bounds of an array determine the way elements of the array can be referred to. For example, assume that the following data

items are assigned to the array LIST, as declared above:

```
20 5 10 30 630 150 310 70
```

The different elements would be referred to as follows:

Reference	Element
LIST (1)	20
LIST (2)	5
LIST (3)	10
LIST (4)	30
LIST (5)	630
LIST (6)	150
LIST (7)	310
LIST (8)	70

Each of the numbers following the name LIST is a subscript. A parenthesized subscript following an array name, with or without an intervening blank, specifies the relative position of a data item within the array. A subscripted name, such as LIST(4), refers to a single element and is an element variable. The entire array can be referred to by the unsubscripted name of the array, for example, LIST. In this case, LIST is an array variable. Note the difference between a subscript and the dimension attribute specification. The latter, which appears in a declaration, specifies the dimensionality and the number of elements in an array. Subscripts are used in other references to identify specific elements within the array.

The same data assigned to LIST_A and LIST_B, as declared above, would be referred to as follows:

Reference	Element	Reference
LIST_A (4)	20	LIST_B (-4)
LIST_A (5)	5	LIST_B (-3)
LIST_A (6)	10	LIST_B (-2)
LIST_A (7)	30	LIST_B (-1)
LIST_A (8)	630	LIST_B (0)
LIST_A (9)	150	LIST_B (1)
LIST_A (10)	310	LIST_B (2)
LIST_A (11)	70	LIST_B (3)

Assume that the same data were assigned to TABLE, which is declared as a two-dimensional array. TABLE can be illustrated as a matrix of four rows and two columns, as follows:

TABLE(m,n)	(m,1)	(m,2)
(1,n)	20	5
(2,n)	10	30
(3,n)	630	150
(4,n)	310	70

An element of TABLE is referred to by a subscripted name with two parenthesized subscripts, separated by a comma. For example, TABLE (2,1) would specify the first item in the second row, in this case, the data item 10.

Note: The use of a matrix to illustrate TABLE is purely conceptual. It has no relationship to the way in which the items are actually organized in storage. Data items are assigned to an array in row major order, that is, with the right-most subscript varying most rapidly. For example, assignment to TABLE would be to TABLE(1,1), TABLE(1,2), TABLE(2,1), TABLE(2,2) and so forth.

Arrays are not limited to two dimensions. The PL/I F Compiler allows as many as 32 dimensions to be declared for an array. In a reference to an element of any array, a subscripted name must contain as many subscripts as there are dimensions in the array.

Examples of arrays in this chapter have shown arrays of arithmetic data. Other data types may be collected into arrays. String arrays, either character or bit, are valid, as are arrays of statement labels.

Expressions as Subscripts

The subscripts of a subscripted name need not be constants. Any expression that yields a valid arithmetic value can be used. If the evaluation of such an expression does not yield an integer value, the fractional portion is ignored. For System/360 implementations, the integer value is converted, if necessary, to a fixed-point binary number of precision (15,0), since subscripts are maintained internally as binary integers.

Subscripts are frequently expressed as variables or other expressions. Thus, TABLE(I,J*K) could be used to refer to the different elements of TABLE by varying the values of I, J, and K.

Cross Sections of Arrays

Cross sections of arrays can be referred to by substituting an asterisk for a subscript in a subscripted name. The asterisk then specifies that the entire extent is to be used. For example, TABLE(*,1) refers to all of the elements in the first column of TABLE. It specifies the cross section consisting of TABLE(1,1), TABLE(2,1), TABLE(3,1), and TABLE(4,1). The subscripted name TABLE(2,*) refers to all of the data items in the second row of TABLE. TABLE(*,*) refers to the entire array.

Note that a subscripted name containing asterisk subscripts represents, not a single data element, but an array with as many dimensions as there are asterisks. Consequently, such a name is not an element expression, but an array expression.

STRUCTURES

Data items that need not have identical characteristics, but that possess a logical relationship to one another, can be grouped into aggregates called structures.

Like an array, the entire structure is given a name that can be used to refer to the entire collection of data. Unlike an array, however, each element of a structure also has a name.

A structure is a hierarchical collection of names. At the bottom of the hierarchy is a collection of elements, each of which represents a single data item or an array. At the top of the hierarchy is the structure name, which represents the entire collection of element variables. For example, the following is a collection of element variables that might be used to compute a weekly payroll:

```
LAST_NAME
FIRST_NAME
REGULAR_HOURS
OVERTIME_HOURS
REGULAR_RATE
OVERTIME_RATE
```

These variables could be collected into a structure and given a single structure name, PAYROLL, which would refer to the entire collection.

```
                    PAYROLL

LAST_NAME     REGULAR_HOURS     REGULAR_RATE

FIRST_NAME    OVERTIME_HOURS    OVERTIME_RATE
```

Any reference to PAYROLL would be a reference to all of the element variables. For example:

```
GET DATA (PAYROLL);
```

This input statement could cause data to be assigned to each of the element variables of the structure PAYROLL.

It often is convenient to subdivide the entire collection into smaller logical collections. In the above examples, LAST_NAME and FIRST_NAME might make a logical subcollection, as might REGULAR_HOURS and OVERTIME_HOURS, as well as REGULAR_RATE and OVERTIME_RATE. In a structure, such subcollections also are given names.

```
                    PAYROLL

    NAME          HOURS           RATE

    FIRST         REGULAR         REGULAR
    LAST          OVERTIME        OVERTIME
```

Note that the hierarchy of names can be considered to have different levels. At the first level is the structure name (called a major structure name); at a deeper level are the names of substructures (called minor structure names); and at the deepest are the element names (called elementary names). An elementary name in a structure can represent an array, in which case it is not an element variable, but an array variable.

The organization of a structure is specified in a DECLARE statement through the use of level numbers. A major structure name must be declared with the level number 1. Minor structures and elementary names must be declared with level numbers arithmetically greater than 1; they must be decimal integer constants. A blank must separate the level number and its associated name.

For example, the items of a weekly payroll could be declared as follows:

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST,
            3 FIRST,
          2 HOURS,
            3 REGULAR,
            3 OVERTIME,
          2 RATE,
            3 REGULAR,
            3 OVERTIME;
```

Note: In an actual declaration of the structure PAYROLL, attributes would be specified for each of the elementary names. The pattern of indention in this example is used only for readability. The statement could be written in a continuous string as DECLARE 1 PAYROLL, 2 NAME, 3 LAST, etc.

PAYROLL is declared as a major structure containing the minor structures NAME, HOURS, and RATE. Each minor structure contains two elementary names. A programmer can refer to the entire structure by the name PAYROLL, or he can refer to portions of the structure by referring to the minor structure names. He can refer to an element by referring to an elementary name.

Note that in the declaration, each level number precedes its associated name and is separated from the name by a blank. The numbers chosen for successively deeper levels need not be the immediately succeeding integers. They are used merely to specify the relative level of a name. A minor structure at level n contains all the names with level numbers greater than n that lie between that minor structure name and the next name with a level number less than or equal to n. PAYROLL might have been declared as follows:

```
DECLARE 1 PAYROLL,
          4 NAME,
            5 LAST,
            5 FIRST,
          2 HOURS,
            6 REGULAR,
            5 OVERTIME,
          2 RATE,
            3 REGULAR,
            3 OVERTIME;
```

This declaration would result in exactly the same structuring as the previous declaration.

The description of a major structure name is terminated by the declaration of another item with a level number 1, by the declaration of another item with no level number, or by a semicolon terminating the DECLARE statement.

Level numbers are specified with structure names only in DECLARE statements. In references to the structure or its elements, no level numbers are used.

Qualified Names

A minor structure or a structure element can be referred to by the minor structure name or the elementary name alone if there is no ambiguity. Note, however, that each of the names REGULAR and OVERTIME appears twice in the structure declaration for PAYROLL. A reference to either name would be ambiguous without some qualification to make the name unique.

PL/I allows the use of qualified names to avoid this ambiguity. A qualified name is an elementary name or a minor structure name that is made unique by qualifying it with one or more names at a higher level. In the PAYROLL example, REGULAR and OVERTIME could be made unique through use of the qualified names HOURS.REGULAR, HOURS.OVERTIME, RATE.REGULAR, and RATE.OVERTIME.

The different names of a qualified name are connected by periods. Blanks may or may not appear surrounding the period. Qualification is in the order of levels; that is, the name at the highest level must appear first, with the name at the deepest level appearing last.

Any of the names in a structure, except the major structure name itself, need not be unique within the procedure in which it is declared. For example, the qualified name PAYROLL.HOURS.REGULAR might be required to make the reference unique (another structure, say WORK, might also have the name REGULAR in a minor structure HOURS; it could be made unique with the name WORK.HOURS.REGULAR). All of the qualifying names need not be used, although they may be, if desired. Qualification need go only so far as necessary to make the name unique. Intermediate qualifying names can be omitted. The name PAYROLL.LAST is a valid reference to the name PAYROLL.NAME.LAST.

ARRAYS OF STRUCTURES

A structure name, either major or minor, can be given a dimension attribute in a DECLARE statement to declare an array of structures. An array of structures is an array whose elements are structures having identical names, levels, and elements. For example, if a structure, WEATHER, were used to process meteorological information for each month of a year, it might be declared as follows:

```
DECLARE 1 WEATHER(12),
        2 TEMPERATURE,
          3 HIGH DECIMAL FIXED(4,1),
          3 LOW DECIMAL FIXED(3,1),
        2 WIND_VELOCITY,
          3 HIGH DECIMAL FIXED(3),
          3 LOW DECIMAL FIXED(2),
        2 PRECIPITATION,
          3 TOTAL DECIMAL FIXED(3,1),
          3 AVERAGE DECIMAL FIXED(3,1);
```

Thus, a programmer could refer to the weather data for the month of July by specifying WEATHER(7). Portions of the July weather could be referred to by TEMPERATURE(7), WIND_VELOCITY(7), and PRECIPITATION(7), but TOTAL(7) would refer to the total precipitation during the month of July.

TEMPERATURE.HIGH(3), which would refer to the high temperature in March, is a subscripted qualified name.

The need for subscripted qualified names becomes more apparent when an array of structures contains minor structures that are arrays. For example, consider the following array of structures:

```
DECLARE 1 A (6,6),
          2 B (5),
            3 C,
            3 D,
          2 E;
```

Both A and B are arrays of structures. To identify a data item, it may be necessary to use as many as three names and three subscripts. For example, A(1,1).B(2).C identifies a particular C that is an element of B in a structure in A.

So long as the order of subscripts remains unchanged, subscripts in such references may be moved to the right or left and attached to names at a lower or higher level. For example, A.B.C(1,1,2) and A(1,1,2).B.C have the same meaning as A(1,1).B(2).C for the above array of structures. Unless all of the subscripts are moved to the lowest or highest level, the qualified name is said to have interleaved subscripts; thus, A.B(1,1,2).C has interleaved subscripts.

An array declared within an array of structures inherits dimensions declared in the containing structure. For example, in the above declaration for the array of structures A, the array B is a three-dimensional structure, because it inherits the two dimensions declared for A. If B is unique and requires no qualification, any reference to a particular B would require three subscripts, two to identify the specific A and one to identify the specific B within that A.

OTHER ATTRIBUTES

Keyword attributes for data variables such as BINARY and DECIMAL are discussed briefly in the preceding sections of this chapter. Other attributes that are not peculiar to one data type may also be applicable. A complete discussion of these attributes is contained in Part II, Section I, "Attributes." Some that are especially applicable to a discussion of data type and data organization are DEFINED, LIKE, ALIGNED, UNALIGNED, and INITIAL.

The DEFINED Attribute

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

```
DECLARE LIST (100,100),
   LIST_ITEM (100,100) DEFINED LIST;
```

LIST is a 100 by 100 two-dimensional array. LIST_ITEM is an identical array defined on LIST. A reference to an element in LIST_ITEM is the same as a reference to the corresponding element in LIST.

The DEFINED attribute, along with the POSITION attribute, can be used to subdivide or overlay a data item. For example:

```
DECLARE LIST CHARACTER (50),
   LISTA CHARACTER(10) DEFINED LIST,
   LISTB CHARACTER(10) DEFINED LIST
      POSITION(11),
   LISTC CHARACTER(30) DEFINED LIST
      POSITION(21);
```

LISTA refers to the first ten characters of LIST. LISTB refers to the second ten characters of LIST. LISTC refers to the last thirty characters of LIST.

The DEFINED attribute may also be used to specify parts of an array through use of iSUB variables, in order to constitute a new array. The iSUB variables are dummy variables where i can be specified as any decimal integer constant from 1 through n (where n represents the number of dimensions for the defined item). The value of the dummy variable (iSUB) ranges from the lower bound to the upper bound of the dimension specified by n. For example:

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

B is a subset of A consisting of every even element in the diagonal of the array, A. In other words, B(1) corresponds to A(2,2), B(2) corresponds to A(4,4).

The LIKE Attribute

The LIKE attribute is used to indicate that the name being declared is to be given the same structuring as the major structure or minor structure name following the attribute LIKE. For example:

```
DECLARE 1 BUDGET,
          2 RENT,
          2 FOOD,
            3 MEAT,
            3 EGGS,
            3 BUTTER,
          2 TRANSPORTATION,
            3 WORK,
            3 OTHER,
          2 ENTERTAINMENT,
        1 COST_OF_LIVING LIKE BUDGET;
```

This declaration for COST_OF_LIVING is the same as if it had been declared:

```
DECLARE 1 COST_OF_LIVING,
          2 RENT,
          2 FOOD,
            3 MEAT,
            3 EGGS,
            3 BUTTER,
          2 TRANSPORTATION,
            3 WORK,
            3 OTHER,
          2 ENTERTAINMENT;
```

Note: The LIKE attribute copies structuring, names, and attributes of the structure below the level of the specified name only. No dimensionality of the specified name is copied. For example, if BUDGET were declared as 1 BUDGET(12), the declaration of COST_OF_LIVING LIKE BUDGET would not give the dimension attribute to COST_OF_LIVING. To achieve dimensionality of COST_OF_LIVING, the declaration would have to be DECLARE 1 COST_OF_LIVING(12) LIKE BUDGET.

A minor structure name can be declared LIKE a major structure or LIKE another minor structure. A major structure name can be declared LIKE a minor structure or LIKE another major structure.

The ALIGNED and UNALIGNED Attributes

The ALIGNED and UNALIGNED attributes are used to specify the positioning in storage of data elements, to influence speed of access or storage economy respectively.

Note: Use of the UNALIGNED attribute allows data interchange with FORTRAN files. See 'Managing Programs' in the PL/I (F) Programmer's Guide, Form C28-6594.

ALIGNED in System/360 implementations specifies that the data element is to be aligned on the storage boundary corresponding to its data type requirement.

UNALIGNED in System/360 implementations specifies that each data element is to be stored contiguously with the data element preceding it: a character-string item is to be mapped on the next byte boundary, a bit-string item is to be mapped on the next bit, and a word and doubleword item is to be mapped on the next byte boundary.

Defaults are applied at element level. The default for bit-string data, character-string data, and numeric character data is UNALIGNED; for all other types of data, the default is ALIGNED.

ALIGNED or UNALIGNED can be specified for element, array, or structure variables. The application of either attribute to a structure is equivalent to applying the attribute to all contained elements that are not explicitly declared ALIGNED or UNALIGNED.

The following example illustrates the effect of ALIGNED and UNALIGNED declarations for a structure and its elements:

```
DECLARE 1 STRUCTURE,
  2 X BIT(2),        /* UNALIGNED BY
                        DEFAULT */
  2 A ALIGNED,       /* ALIGNED EXPLICITLY */
    3 B,             /* ALIGNED FROM A */
    3 C UNALIGNED,   /* UNALIGNED
                        EXPLICITLY */
      4 D,           /* UNALIGNED FROM C */
      4 E ALIGNED,   /* ALIGNED EXPLICITLY */
      4 F,           /* UNALIGNED FROM C */
    3 G,             /* ALIGNED FROM A */
  2 H;               /* ALIGNED BY DEFAULT */
```

Although UNALIGNED causes economic use of data storage, for System/360 implementations it will increase the amount of code generated to access data items that are not aligned on the required byte boundaries.

The INITIAL Attribute

The INITIAL attribute specifies an initial value to be assigned to a variable at the time storage is allocated for it. For example:

```
DECLARE   NAME  CHARACTER(10)  INITIAL
   ('JOHN DOE');

DECLARE PI FIXED DECIMAL (5,4) INITIAL
   (3.1416);
```

```
DECLARE TABLE (100,100)  INITIAL  CALL
   SUBR(ALPHA);
```

When storage is allocated for NAME, the character string 'JOHN DOE' (padded to 10

characters) will be assigned to it. When PI is allocated, it will be initialized to the value 3.1416. Either value may be retained throughout the program, or it may be changed during execution. The third example illustrates the CALL option. It indicates that the procedure SUBR is to be invoked to perform the initialization.

For a variable that is allocated when the program is loaded, that is, a static variable, which remains in allocation throughout execution of the program, any value specified in an INITIAL attribute is assigned only once. For automatic variables, which are allocated at each activation of the declaring block, any specified initialization is assigned with each allocation. For controlled variables, which are allocated at the execution of ALLOCATE statements, any specified initialization is assigned with each allocation. Note, however, that this initialization can be overridden in the ALLOCATE statement. The F Compiler does not allow the INITIAL attribute to be specified for based variables.

The INITIAL attribute cannot be given for entry names, file names, DEFINED data, entire structures, parameters, task data, or event data.

Note: The CALL option cannot be used with the INITIAL attribute for static data.

The INITIAL attribute cannot be used without the CALL option for pointer, offset, or area data. An area variable is automatically initialized with the value of the EMPTY built-in function, on allocation, after which any specified INITIAL CALL is applied.

The INITIAL attribute can be specified for arrays, as well as for element variables. In a structure declaration, only elementary names can be given the INITIAL attribute.

An array or an array of structures can be partly initialized or fully initialized. For example:

```
DECLARE A(15) CHARACTER(13) INITIAL
          ('JOHN DOE', 'RICHARD ROW',
           'MARY SMITH'),
        B (10,10) DECIMAL FIXED(5)
          INITIAL((25)0,(25)1,(50)0),
        1 C(8),
          2 D INITIAL (0),
          2 E INITIAL((8)0);
```

In this example, only the first three elements of A are initialized; the rest of the array is uninitialized. The array B is fully initialized, with the first 25 elements initialized to 0, the next 25 to 1, and the last 50 to 0. The parenthesized numbers (25, 25, and 50) are iteration factors, that specify the number of elements to be initialized. In the structure C, where the dimension (8) has been inherited by D, only the first element of D is initialized; where the dimension (8) has been inherited by E, all the elements of E are initialized.

When an array of structures is declared with the LIKE attribute to obtain the same structuring as a structure whose elements have been initialized, it should be noted that only the first structure in this array of structures will be initialized. For example:

```
DECLARE 1 G,
          2 H INITIAL(0),
          2 I INITIAL(0),
        1 J(8) LIKE G;
```

In this example, only J(1).H and J(1).I are initialized in the array of structures.

For STATIC arrays, iteration factors must be decimal integer constants; for arrays of other storage classes, iteration factors may be constants, variables, or expressions.

The iteration factor should not be confused with the string repetition factor discussed earlier in this chapter. Consider the following example:

```
DECLARE TABLE (50) CHARACTER (10)
   INITIAL ((10)'A',(25)(10)'B',
   (24)(1)'C');
```

This INITIAL attribute specification contains both iteration factors and repetition factors. It specifies that the first element of TABLE is to be initialized with a string consisting of 10 A's, each of the next 25 elements is to be initialized with a string consisting of 10 B's, and each of the last 24 elements is to be initialized with the single character C. In the INITIAL attribute specification for a string array, a single parenthesized factor preceding a string constant is assumed to be a string repetition factor (as in (10)'A'). If more than one appears, the first is assumed to be an iteration factor, and the second a string repetition factor. For this reason (as in (24)(1)'C'), a string repetition factor of 1 must be inserted if a single string constant is to be used to initialize more than one element.

The CALL option can be used to initialize arrays, except for arrays of static storage class.

An expression is a representation of a value. A single constant or a variable is an expression. Combinations of constants and/or variables, along with operators and/or parentheses, are expressions. An expression that contains operators is an operational expression. The constants and variables of an operational expression are called operands.

Examples of expressions are:

```
27

LOSS

A+B

(SQTY-QTY)*SPRICE
```

Any expression can be classified as an element expression (also called a scalar expression), an array expression, or a structure expression. An element expression is one that represents an element value. An array expression is one that represents an array value. A structure expression is one that represents a structure value.

For the F Compiler, array variables and structure variables cannot appear in the same expression. Element variables and constants, however, can appear in either array expressions or structure expressions. An elementary name within a structure or a subscripted name that specifies a single element of an array is an element expression.

Note: If an elementary name of a structure is given the dimension attribute, that elementary name is an array variable and can appear only in array expressions.

In the examples that follow, assume that the variables have attributes declared as follows:

```
DECLARE A(10,10) BINARY FIXED (31),
   B(10,10) BINARY FIXED (31),
   1 RATE, 2 PRIMARY DECIMAL FIXED (4,2),
           2 SECONDARY DECIMAL FIXED (4,2),
   1 COST, 2 PRIMARY DECIMAL FIXED (4,2),
           2 SECONDARY DECIMAL FIXED (4,2),
   C BINARY FIXED (15),
   D BINARY FIXED (15);
```

Examples of element expressions are:

```
C * D

A(3,2) + B(4,8)

RATE.PRIMARY - COST.PRIMARY

A(4,4) * C

RATE.SECONDARY / 4

A(4,6) * COST.SECONDARY
```

All of these expressions are element expressions because each operand is an element variable or constant (even though some may be elements of arrays or elementary names of structures); hence, each expression represents an element value.

Examples of array expressions are:

```
A + B

A * C - D

B / 10B
```

All of these expressions are array expressions because at least one operand of each is an array variable; hence, each expression represents an array value. Note that the third example contains the binary fixed-point constant 10B.

Examples of structure expressions are:

```
RATE * COST

RATE / 2
```

Both of these expressions are structure expressions because at least one operand of each is a structure variable; hence, each expression represents a structure value.

USE OF EXPRESSIONS

Expressions that are single constants or single variables may appear freely throughout a program. However, the syntax of many PL/I statements allows the appearance of operational expressions, so long as evaluation of the expression yields a valid value.

In syntactic descriptions used in this publication, the unqualified term

"expression" refers to an element expression, an array expression, or a structure expression. For cases in which the kind of expression is restricted, the type of restriction is noted; for example, the term "element-expression" in a syntactic description indicates that neither an array expression nor a structure expression is valid.

Note: Although operational expressions can appear in a number of different PL/I statements, their most common occurrences are in assignment statements of the form:

```
A = B + C;
```

The assignment statement has no PL/I keyword. The assignment symbol (=) indicates that the value of the expression on the right (B + C) is to be assigned to the variable on the left (A). For purposes of illustration in this chapter, some examples of expressions are shown in assignment statements.

DATA CONVERSION IN OPERATIONAL EXPRESSIONS

An operational expression consists of one or more single operations. A single operation is either a prefix operation (an operator preceding a single operand) or an infix operation (an operator between two operands). The two operands of any infix operation, when the operation is performed, usually must be of the same data type, as specified by the attributes of a variable or the notation used in writing a constant.

The operands of an operation in a PL/I expression are automatically converted, if necessary, to a common representation before the operation is performed. General rules for conversion of different data types are discussed in the following paragraphs and in a later section of this chapter, "Concepts of Data Conversion." Detailed rules for specific cases, including rules for computing the precision or length of a converted item, can be found in Part II, Section F, "Problem Data Conversion."

Data conversion is mainly confined to problem data. The only conversion possible with program control data is conversion between offset and pointer types.

PROBLEM DATA CONVERSION

Data conversion can be applied to all types of problem data, as listed below.

Bit-String to Character-String

The bit 1 becomes the character 1; the bit 0 becomes the character 0.

Character-String to Bit-String

The character string should contain the characters 1 and 0 only, in which case the character 1 becomes the bit 1, and the character 0 becomes the bit 0. The CONVERSION condition is raised by an attempt to convert any character other than 1 or 0 to a bit.

Character-String to Arithmetic

The character string must be in the form of a signed or unsigned arithmetic constant (or an expression representation of a COMPLEX data item). The constant may be surrounded by blanks, but blanks must not be imbedded in a number. Any character other than those allowed in arithmetic data will raise the CONVERSION condition if conversion is attempted.

Note: In the conversion, for an infix operation, of a character string that represents a fixed-point constant -- either decimal or binary -- any fractional portion will be lost if it is converted to fixed-point. For the F Compiler, integer digits will be truncated if the character string contains more than 5 decimal integer digits or 15 binary digits. If the conversion is to floating-point, it will retain its fractional value. Rules for the precision of such conversion are listed in Part II, Section F, "Problem Data Conversion."

Arithmetic to Character-String

The value of an internal coded arithmetic operand is converted to its character representation. The converted field is a

character string in the form of a valid arithmetic constant. The length of the character string is dependent upon the precision of the arithmetic data item.

Bit-String to Arithmetic

A bit string is interpreted as an unsigned binary integer and is converted to fixed-point binary of positive value. The base and scale are further converted, if necessary.

Arithmetic to Bit-String

The absolute value is converted, if necessary, to a real fixed-point binary integer. Ignoring the plus sign, the integer is then interpreted as a bit string. The length of the bit string is dependent upon the precision of the original unconverted arithmetic data item.

Arithmetic Mode Conversion

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

A real data item is converted to a complex data item by adding an imaginary part of zero.

Arithmetic Base and Scale Conversion

The precision of the result of an arithmetic base or scale conversion is dependent upon the precision of the original arithmetic data item. The rules are listed in Part II, Section F, "Problem Data Conversion."

LOCATOR DATA CONVERSION

Only offset to pointer conversion occurs as a result of an operational expression (locator variables are restricted to = and ¬= comparison operations), but either of the following types of conversion can result from assignment. (See also Chapter 14, "Based Storage and List Processing.")

Offset to Pointer

An offset value is converted to pointer by combining the offset value with the pointer value relating to the start of the area named in the OFFSET attribute.

Pointer to Offset

A pointer value is converted to offset by effectively deducting the pointer value for the start of the area from the pointer value to be converted. This conversion is limited to pointer values that relate to addresses within the area named in the OFFSET attribute.

CONVERSION Y ASSIGNMENT

In addition to conversion performed as the result of an operation in the evaluation of an expression, conversion will also occur when a data item -- or the result of an expression evaluation -- is assigned to a variable whose attributes differ from the attributes of the item assigned. The rules for such conversion are generally the same as those discussed above and in Part II, Section F, "Problem Data Conversion."

EXPRESSION OPERATIONS

An operational expression can specify one or more single operations. The class of operation is dependent upon the class of operator specified for the operation. There are four classes of operations -- arithmetic, bit-string, comparison, and concatenation.

ARITHMETIC OPERATIONS

An arithmetic operation is one that is specified by combining operands with one of the following operators:

+ - * / **

The plus sign and the minus sign can appear either as prefix operators (associated with and preceding a single operand, such as +A or -A) or as infix operators (associated with and between two operands, such as A+B or A-B). All other arithmetic operators can appear only as infix operators.

An expression of greater complexity can be composed of a set of such arithmetic operations. Note that prefix operators can precede and be associated with any of the operands of an infix operation. For example, in the expression A*-B, the minus sign preceding the variable B indicates that the value of A is to be multiplied by the negative value of B.

More than one prefix operator can precede and be associated with a single variable. More than one positive prefix operator will have no cumulative effect, but two consecutive negative prefix operators will have the same effect as a single positive prefix operator. For example:

-A The single minus sign has the effect of reversing the sign of the value that A represents.

--A One minus sign reverses the sign of the value that A represents. The second minus sign again reverses the sign of the value, restoring it to the original arithmetic value represented by A.

---A Three minus signs reverse the sign of the value three times, giving the same result as a single minus sign.

Data Conversion in Arithmetic Operations

The two operands of an arithmetic operation may differ in type, base, mode, precision, and scale. When they differ, conversion takes place according to rules listed below. Certain other rules -- as stated below -- may apply in cases of exponentiation.

TYPE: Character-string operands, numeric character field operands (digits recorded in character form), and bit-string operands are converted to internal coded arithmetic type. The result of an arithmetic operation is always in coded arithmetic form. Note that type conversion is the only conversion that can take place in an arithmetic prefix operation.

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

MODE: If the modes of the two operands 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). The exception to this rule is in the case of exponentiation when the second operand (the exponent of the operation) is fixed-point real with a scale factor of zero. In such a case, no conversion is necessary.

PRECISION: If only precisions differ, no type conversion is necessary.

SCALE: If the scales of the two operands differ, the fixed-point operand is converted to floating-point scale. The exception to this rule is in the case of exponentiation when the first operand is of floating-point scale and the second operand (the exponent of the operation) is fixed-point with a scale factor of zero, that is, a fixed-point integer constant or a variable that has been declared with precision (p,0). In such a case, no conversion is necessary, but the result will be floating-point.

If both operands of an exponentiation operation are fixed-point, conversions may occur, as follows:

1. Both operands are converted to floating-point if the exponent has a precision other than (p,0).

2. The first operand is converted to floating-point unless the exponent is an unsigned fixed-point integer constant.

3. The first operand is converted to floating-point if precisions indicate that the result of the fixed-point exponentiation would exceed the maximum number of digits allowed for the implementation (for System/360, 15 decimal digits or 31 binary digits). Further details and examples of conversion in exponentiation are included in the section "Concepts of Data Conversion" in this chapter.

Results of Arithmetic Operations

The "result" of an arithmetic operation, as used in the following text, may refer to an intermediate result if the operation is only one of several operations specified in a single operational expression. Any result may require further conversion if it is an intermediate result that is used as an operand of a subsequent operation or if it is assigned to a variable.

After required conversions have taken place, the arithmetic operation is performed. If maximum precision is exceeded and truncation is necessary, the truncation is performed on low-order fractional digits, regardless of base or scale of the operands. In some cases involving fixed-point data, however, high-order digits may sometimes be lost when scale factors are such that point alignment does not allow for the declared number of integer digits.

The base, scale, mode, and precision of the result depend upon the operands and the operator involved.

For prefix operations, the result has the same base, scale, mode, and precision as the converted operand. Note that the result of -A, where A is a string, is an arithmetic result, since A must first be converted to coded arithmetic form before the operation can be performed.

For infix operations, the result depends upon the scale of the operands in the following ways:

FLOATING-POINT: If the converted operands of an infix operation are of floating-point scale, the result is of floating-point scale, 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.

FIXED-POINT: If the converted operands of an infix operation are of fixed-point scale, the result is of fixed-point scale, and the base and mode of the result are the common base and mode of the operands. The precision of a fixed-point result depends upon operands, according to the rules listed below.

In the formulas for computing precision, the symbols used are as follows:

- p represents the total number of digits of the result
- q represents the scale factor of the result
- p_1 represents the total number of digits of the first operand
- q_1 represents the scale factor of the first operand
- p_2 represents the total number of digits of the second operand
- q_2 represents the scale factor of the second operand

ADDITION AND SUBTRACTION: The total number of digits in the result is equal to 1 plus the number of integer digits of the operand having the greater number of integer digits plus the number of fractional digits of the operand having the greater number of fractional digits. The total number of positions cannot exceed the maximum number of digits allowed (15 decimal digits, 31 binary digits). The scale factor of the result is equal to the larger scale factor of the two operands.

Formulas:

$$p = 1 + \text{maximum } (p_1-q_1, p_2-q_2) + \text{maximum } (q_1, q_2)$$

$$q = \text{maximum } (q_1, q_2)$$

Example:

```
12354.2385 + 222.11111
  A    B      C    D
```

The total number of digits in the result would be equal to 1 plus the number of digits in A plus the number of digits in D. The scale factor of the result would be equal to the number of digits in D. Precision of the result would be (11,5).

MULTIPLICATION: The total number of digits in the result is equal to one plus the number of digits in operand one plus the number of digits in operand two. The total number of digits cannot exceed the maximum number of digits allowed for the implementation (15 decimal, 31 binary). The scale factor of the result is the sum of the scale factors of the two operands.

Formulas:

$$p = p_1 + p_2 + 1$$

$$q = q_1 + q_2$$

Example:

```
345.432 * 22.45
 A   B     C  D
```

The total number of digits in the result would be equal to 1 plus the sum of the number of digits in A, B, C, and D. The scale factor of the result would be the sum of the number of digits in B and D. Precision of the result would be (11,5).

DIVISION: The total number of digits in the quotient is equal to the maximum allowed by the implementation (15 decimal, 31 binary). The scale factor of the quotient is dependent upon the number of integer digits of the dividend (A in the example below), and the number of fractional digits of the divisor (D in the example

below). The scale factor is equal to the total number of digits of the result minus the sum of A and D.

Formulas:

p = 15 decimal, 31 binary

q = 15 (or 31)-((p_1-q_1) + q_2)

Example:

```
  432.432 / 2
   A   B   C  D
```

The total number of digits in the quotient would be 15 (the maximum number allowed). The scale factor would be 15 minus the sum of 3 (A, the number of integer digits in the dividend) and zero (D, the number of fractional digits in the divisor). Precision of the quotient would be (15,12).

Note that any change in the number of integer digits in the dividend or any change in the number of fractional digits in the divisor will change the precision of the quotient, even if all additional digits are zeros.

Examples:

00432.432 / 2

432.432 / 2.0000

Precision of the quotient of the first example would be (15,10); scale factor is equal to 15-(5+0). Precision of the quotient of the second example would be (15,8); scale factor is equal to 15-(3+4).

Caution: In the use of fixed-point division operations, care should be taken that declared precision of variables and apparent precision of constants will not give a result with a scale factor that can force the result of subsequent operations to exceed the maximum number of digits allowed by the implementation.

EXPONENTIATION: If the second operand (the exponent) is an unsigned nonzero real fixed-point constant of precision (p,0), the total number of positions in the result is equal to one less than the product of a number that is one greater than the number of digits in the first operand multiplied by the value of the second operand (the exponent). The scale factor of the result is equal to the product of the scale factor of the first operand multiplied by the value of the second operand (the exponent).

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

1. Real mode, x**y
   a. If x=0 and y>0, the result is 0.
   b. If x=0 and y≤0, the ERROR condition is raised.
   c. If x≠0 and y=0, the result is 1.
   d. If x<0 and y is not fixed-point with precision (p,0), the ERROR condition is raised.
2. Complex mode, x**y
   a. If x=0 and y has its real part >0 and its imaginary part =0, the result is 0.
   b. If x=0 and the real part of y ≤0 or the imaginary part of y =0, the ERROR condition is raised.

(As pointed out under "Data Conversion in Arithmetic Operations," if the exponent is not an unsigned real fixed-point integer constant, or if the total number of digits of the result would exceed 15 decimal digits or 31 binary digits, the first operand is converted to floating-point scale, and the rules for floating-point exponentiation apply.)

Formulas:

p = ((p_1+1)*(value-of-exponent))-1

q = q_1 * (value-of-exponent)

Example:

```
  32 ** 5
```

The total number of digits in the result would be 14. This is arrived at by multiplying a number equal to one plus the number of digits in the first operand (1+2) by the value of the exponent and subtracting one. The scale factor of the result would be zero (0 * 5, scale factor of the first operand multiplied by the value of the exponent).

BIT-STRING OPERATIONS

A bit-string operation is one that is specified by combining operands with one of the following operators:

¬ & |

The first operator, the "not" symbol, can be used as a prefix operator only. The second and third operators, the "and" symbol and the "or" symbol, can be used as

infix operators only. (The operators have the same function as in Boolean algebra.)

Operands of a bit-string operation are, if necessary, converted to bit strings before the operation is performed. If the operands of an infix operation are of unequal length, the shorter is extended on the right with zeros.

The result of a bit-string operation is a bit string equal in length to the length of the operands (the two operands, after conversion, always are the same length). If either is a varying-length bit string, the result is of varying length.

Bit-string operations are performed on a bit-by-bit basis. The effect of the "not" operation is bit reversal; that is, the result of ¬1 is 0; the result of ¬0 is 1. The result of an "and" operation is 1 only if both corresponding bits are 1; in all other cases, the result is 0. The result of an "or" operation is 1 if either or both of the corresponding bits are 1; in all other cases, the result is 0. The following table illustrates the result for each bit position for each of the operators:

A	B	¬A	¬B	A&B	A|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

More than one bit-string operation can be combined in a single expression that yields a bit-string value.

In the following examples, if the value of operand A is '010111'B, the value of operand B is '111111'B, and the value of operand C is '110'B, then:

¬ A yields '101000'B

¬ C yields '001'B

C & B yields '110000'B

A | B yields '111111'B

C | B yields '111111'B

A | (¬C) yields '011111'B

¬((¬C)|(¬B)) yields '110111'B

COMPARISON OPERATIONS

A comparison operation is one that is specified by combining operands with one of the following operators:

< ¬< <= = ¬= >= > ¬>

These operators specify "less than," "not less than," "less than or equal to," "equal to," "not equal to," "greater than or equal to," "greater than," and "not greater than."

There are three types of comparisons:

1. Algebraic, which involves the comparison of signed arithmetic values in internal coded arithmetic form. If operands differ in base, scale, precision, or mode, they are converted according to the rules for arithmetic operations. Numeric character data is converted to coded arithmetic before comparison.

2. Character, which involves left-to-right, character-by-character comparisons of characters according to the collating sequence.

3. Bit, which involves left-to-right, bit-by-bit comparison of binary digits.

If the operands of a comparison are not immediately compatible (that is, if their data types are appropriate to different types of comparison), the operand of the lower comparison type is converted to conform to the comparison type of the operand of the higher type. The priority of comparison types is (1) algebraic (highest), (2) character string, (3) bit string. Thus, for example, if a bit string were to be compared with a fixed decimal value, the bit string would be converted to arithmetic (i.e., fixed binary) for algebraic comparison with the decimal value (which would also be converted to fixed binary for the comparison).

If operands of a character-string comparison, after conversion, are of different lengths, the shorter operand is extended on the right with blanks. If operands of a bit-string comparison are of different lengths, the shorter is extended on the right with zeros.

The result of a comparison operation always is a bit string of length one; the value is '1'B if the relationship is true, or '0'B if the relationship is not true.

The most common occurrences of comparison operations are in the IF statement, of the following format:

```
IF A = B

    THEN action-if-true

    ELSE action-if-false
```

The evaluation of the expression A = B yields either '1'B or '0'B. Depending upon the value, either the THEN portion or the ELSE portion of the IF statement is executed.

Comparison operations need not be limited to IF statements, however. The following assignment statement could be valid:

```
X = A < B;
```

In this example, the value '1'B would be assigned to X if A is less than B; otherwise, the value '0'B would be assigned. In the same way, the following assignment statement could be valid:

```
X = A = B;
```

The first symbol (=) is the assignment symbol; the second (=) is the comparison operator. If A is equal to B, the value of X will be '1'B; if A is not equal to B, the value of X will be '0'B.

Only the comparison operations of "equal" and "not equal" are valid for comparisons of complex operands, or comparisons of locator operands. Comparison operations with program control data other than locator data are not allowed.

CONCATENATION OPERATIONS

A concatenation operation is one that is specified by combining operands with the concatenation symbol:

||

It signifies that the operands are to be joined in such a way that the last character or bit of the operand to the left will immediately precede the first character or bit of the operand to the right, with no intervening bits or characters.

The concatenation operator can cause conversion to string type since concatenation can be performed only upon strings, either character strings or bit strings. If both operands are character strings or if both operands are bit strings, no conversion takes place. Otherwise both operands are converted to character strings.

The results of concatenation operations are as follows:

Bit String: A bit string whose length is equal to the sum of the lengths of the two bit-string operands.

Character String: A character string whose length is equal to the sum of the lengths of the two character-string operands. If an operand requires conversion for the concatenation operation, the result is dependent upon the length of the character string to which the operand is converted.

For example, if A has the attributes and value of the constant '010111'B, B of the constant '101'B, C of the constant 'XY,Z', and D of the constant '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'

    B||D yields '101AA/BB'
```

Note that, in the last example, the bit string '101'B is converted to the character string '101' before the concatenation is performed. The result is a character string consisting of eight characters.

Note: If either of the operands of a concatenation operation has the VARYING attribute, the result will be a VARYING string. When VARYING strings are concatenated, the intermediate string created has a length equal to the sum of the maximum lengths. If the maximum lengths are known at compile time and their sum exceeds 32767, then a truncated intermediate string of length 32767 will be created and an error message produced. If the maximum length of either operand is not known at compile time and their sum exceeds 32767, a truncated intermediate string of length 32767 will be created but there will be no diagnostic message.

COMBINATIONS OF OPERATIONS

Different types of operations can be combined within the same operational expression. Any combination can be used. For example, the expression shown in the following assignment statement is valid:

```
RESULT = A + B < C & D;
```

Each operation within the expression is

evaluated according to the rules for that kind of operation, with necessary data conversions taking place before the operation is performed.

Assume that the variables given above are declared as follows:

```
DECLARE RESULT BIT(3),
        A FIXED DECIMAL(1),
        B FIXED BINARY (3),
        C CHARACTER(2), D BIT(4);
```

- The decimal value of A would be converted to binary base.
- The binary addition would be performed, adding A and B.
- The binary result would be compared with the converted binary value of C.
- The bit-string result of the comparison would be extended to the length of the bit string D, and the "and" operation would be performed.
- The result of the "and" operation, a bit string of length 4, would be assigned to RESULT without conversion, but with truncation on the right.

The expression in this example is described as being evaluated operation-by-operation, from left to right. Such would be the case for this particular expression. The order of evaluation, however, depends upon the priority of the operators appearing in the expression.

Priority of Operators

In the evaluation of expressions, priority of the operators is as follows:

```
**   prefix+   prefix-   ¬   (highest)
*  /                          |
infix+   infix-               |
||                            |
<  ¬<  <=  =  ¬=  >=  >  ¬>   |
 &                            V
 |                        (lowest)
```

If two or more operators of the highest priority appear in the same expression, the order of priority of those operators is from right to left; that is, the rightmost exponentiation or prefix operator has the highest priority. Each succeeding exponentiation or prefix operator to the left has the next highest priority.

For all other operators, if two or more operators of the same priority appear in the same expression, the order of priority of those operators is from left to right.

Note that the order of evaluation of the expression in the assignment statement:

```
RESULT = A + B < C & D;
```

is the result of the priority of the operators. It is as if various elements of the expression were enclosed in parentheses as follows:

```
(A) + (B)
(A + B) < (C)
(A + B < C) & (D)
```

The order of evaluation (and, consequently, the result) of an expression can be changed through the use of parentheses. The above expression, for example, might be changed as follows:

```
(A + B) < (C & D)
```

The order of evaluation of this expression would yield a bit string of length one, the result of the comparison operation. In such an expression, those expressions enclosed in parentheses are evaluated first, to be reduced to a single value, before they are considered in relation to surrounding operators. Within the language, however, no rules specify which of two parenthesized expressions, such as those in the above example, would be evaluated first.

The value of A would be converted to fixed-point binary, and the addition would be performed, yielding a fixed-point binary result (RESULT_1). The value of C would be converted to a bit string (if valid for such conversion) and the "and" operation would be performed.

At this point, the expression would have been reduced to:

```
RESULT_1 < RESULT_2
```

RESULT_2 would be converted to binary, and the algebraic comparison would be performed, yielding the bit-string result of the entire expression.

The priority of operators is defined only within operands (or sub-operands). It does not necessarily hold true for an entire expression. Consider the following example:

```
A + (B < C) & (D || E ** F)
```

The priority of the operators specifies, in this case, only that the exponentiation will occur before the concatenation. It does not specify the order of the operation

in relation to the evaluation of the other operand (A + (B < C)).

Any operational expression (except a prefix expression) must eventually be reduced to a single infix operation. The operands and operator of that operation determine the attributes of the result of the entire expression. For instance, in the first example of combining operations (which contains no parentheses), the "and"

operator is the operator of the final infix operation; in this case, the result of evaluation of the expression is a bit string of length 4. In the second example (because of the use of parentheses), the operator of the final infix operation is the comparison operator, and the evaluation yields a bit string of length 1.

In general, unless parentheses are used within the expression, the operator of lowest priority determines the operands of the final operation. For example:

```
A + B ** 3 || C * D - E
```

In this case, the concatenation operator indicates that the final operation will be:

```
(A + B ** 3) || (C * D - E)
```

The evaluation will yield a character-string result.

Subexpressions can be analyzed in the same way. The two operands of the expression can be defined as follows:

```
A + (B ** 3)

(C * D) - E
```

ARRAY EXPRESSIONS

An array expression is a single array variable or an expression that includes at least one array operand. Array expressions may also include operators -- both prefix and infix -- element variables and constants.

Evaluation of an array expression yields an array result. All operations performed on arrays are performed on an element-by-element basis, in row-major order. Therefore, all arrays referred to in an array expression must be of identical bounds.

Although comparison operators are valid for use with array operands, an array operand cannot appear in the IF clause of an IF statement. Only an element expression is valid in the IF clause, since the IF statement tests a single true or false result.

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

PREFIX OPERATORS AND ARRAYS

The result of the operation of a prefix operator on an array is an array of identical bounds, each element of which is the result of the operation having been performed upon each element of the original array. For 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

Infix operations that include an array variable as one operand may have an element or another array as the other operand.

Array and Element Operations

The result of an operation in which an element and an array are connected by an infix operator is an array with bounds identical to the original array, each element of which is the result of the operation performed upon the corresponding element of the original array and the single element. For example:

If A is the array	5	10	8
	12	11	3
then A*3 is the array	15	30	24
	36	33	9

The element of an array-element operation can be an element of the same array. For example, the expression A*A(2,3) would give the same result in the case of the array A above, since the value of A(2,3) is 3.

Consider the following assignment statement:

```
A = A * A(1,2);
```

Again, using the above values for A, the newly assigned value of A would be:

```
50   100  800

1200 1100 300
```

Note that the original value for A(1,2), which is 10, is used in the evaluation for only the first two elements of A. Since the result of the expression is assigned to A, changing the value of A, the new value of A(1,2) is used for all subsequent operations. The first two elements are multiplied by 10, the original value of A(1,2); all other elements are multiplied by 100, the new value of A(1,2).

Array and Array Operations

If two arrays are connected by an infix operator, the two arrays must be of identical bounds. The result is an array with bounds identical to those of the original arrays; the operation is performed upon the corresponding elements of the two original arrays.

Note that the arrays must have identical bounds. They must have the same number of dimensions, and corresponding dimensions must have identical lower bounds and identical upper bounds. For example, the bounds of an array declared X(10,6) are not identical to the bounds of an array declared Y(2:11,3:8) although the extents are the same for corresponding dimensions, and the number of elements is the same.

Examples of array infix expressions are:

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

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

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

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

Array and Structure Operations

For the F Compiler, no reference can be made to both an array and a structure in the same expression or in the same assignment statement.

Data Conversion in Array Expressions

The examples in this discussion of array expressions have shown only single arithmetic operations. The rules for combining operations and for data conversion of operands are the same as those for element operations.

STRUCTURE EXPRESSIONS

A structure expression is a single structure variable or an expression that includes at least one structure operand and does not contain an array operand. Element variables and constants can be operands of a structure expression. Evaluation of a structure expression yields a structure result. A structure operand can be a major structure name or a minor structure name.

Although comparison operators are valid for use with structure operands, a structure operand cannot appear in the IF clause of an IF statement. Only an element expression is valid in the IF clause, since the IF statement tests a single true or false result.

All operations performed on structures are performed on an element-by-element basis. Except in a BY NAME assignment (see below), all structure variables appearing in a structure expression must have identical structuring.

Identical structuring means that the structures must have the same minor structuring and the same number of contained elements and arrays and that the positioning of the elements and arrays within the structure (and within the minor structures if any) must be the same. Arrays in corresponding positions must have identical bounds. Names do not have to be the same. Data types of corresponding elements do not

have to be the same, so long as valid conversion can be performed.

PREFIX OPERATORS AND STRUCTURES

The result of the operation of a prefix operator on a structure is a structure of identical structuring, each element of which is the result of the operation having been performed upon each element of the original structure.

Note: Since structures may contain elements of many different data types, a prefix operation in a structure expression would be meaningless unless the operation can be validly performed upon every element represented by the structure variable, which is either a major structure name or a minor structure name.

INFIX OPERATORS AND STRUCTURES

Infix operations that include a structure variable as one operand may have an element or another structure as the other operand.

Structure operands in a structure expression need not be major structure names. A minor structure name, at any level, is a structure variable. Thus, if M.N is a minor structure in the major structure M, the following is a structure expression:

```
M.N & '1010'B
```

Structure and Element Operations

When an operation has one structure and one element operand, it is the same as a series of operations, one for each element in the structure. Each sub-operation involves a structure element and the single element.

Consider the following structure:

```
1 A
  2 B
    3 C
    3 D
    3 E
  2 F
    3 G
    3 H
    3 I
```

If X is an element variable, then A * X is equivalent to:

```
A.C * X
A.D * X
A.E * X
A.G * X
A.H * X
A.I * X
```

Structure and Structure Operations

When an operation has two structure operands, it is the same as a series of element operations, one for each corresponding pair of elements.

For example, if A is the structure shown in the previous example and if M is the following structure:

```
1 M
  2 N
    3 O
    3 P
    3 Q
  2 R
    3 S
    3 T
    3 U
```

then A || M is equivalent to:

```
A.C || M.O
A.D || M.P
A.E || M.Q
A.G || M.S
A.H || M.T
A.I || M.U
```

Structure ssignment BY NAME

One exception to the rule that operands of a structure expression must have the same structuring is the case in which the structure expression appears in an assignment statement with the BY NAME option.

The BY NAME appears at the end of a structure assignment statement and is preceded by a comma. Examples are shown below.

Consider the following structures and assignment statements:

```
1 ONE          1 TWO          1 THREE
  2 PART1        2 PART1        2 PART1
    3 RED          3 BLUE         3 RED
    3 ORANGE       3 GREEN        3 BLUE
  2 PART2          3 RED          3 BROWN
    3 YELLOW     2 PART2        2 PART2
    3 BLUE         3 BROWN        3 YELLOW
    3 GREEN        3 YELLOW       3 GREEN
```

```
ONE = TWO, BY NAME;
ONE.PART1 = THREE.PART1, BY NAME;
ONE = TWO + THREE, BY NAME;
```

The first assignment statement would be the same as the following:

```
ONE.PART1.RED = TWO.PART1.RED;
```

```
ONE.PART2.YELLOW = TWO.PART2.YELLOW;
```

The second assignment statement would be the same as the following:

```
ONE.PART1.RED = THREE.PART1.RED;
```

The third assignment statement would be the same as the following:

```
ONE.PART1.RED = TWO.PART1.RED
                    + THREE.PART1.RED;
```

```
ONE.PART2.YELLOW = TWO.PART2.YELLOW
                    + THREE.PART2.YELLOW;
```

The BY NAME option can appear in an assignment statement only. It indicates that assignment of elements of a structure is to be made only for those elements whose names are common to both structures. Except for the highest-level qualifier specified in the assignment statement, all qualifying names must be identical.

If an operational expression appears in an assignment statement with the BY NAME option, operation and assignment are performed only upon those elements whose names have been declared in each of the structures. In the third assignment statement above, no operation is performed upon ONE.PART2.GREEN and THREE.PART2.GREEN, because GREEN does not appear as an elementary name in PART2 of TWO.

OPERANDS OF EXPRESSIONS

An operand of an expression can be a constant, an element variable, an array variable, or a structure variable. An operand can also be an expression that represents a value that is the result of a computation, as shown in the following assignment statement:

```
A = B * SQRT(C);
```

In this example, the expression SQRT(C) represents a value that is equal to the square root of the value of C. Such an expression is called a function reference.

FUNCTION REFERENCE OPERANDS

A function reference consists of a name and, usually, a parenthesized list of one or more variables, constants, or other expressions. The name is the name of a block of coding written to perform specific computations upon the data represented by the list and to substitute the computed value in place of the function reference.

Assume, in the above example, that C has the value 16. The function reference SQRT(C) causes execution of the coding that would compute the square root of 16 and replace the function reference with the value 4. In effect, the assignment statement would become:

```
A = B * 4;
```

The coding represented by the name in the function reference is called a function. The function SQRT is one of the PL/I built-in functions. Built-in functions, which provide a number of different operations, are a part of the PL/I language. A complete discussion of each appears in Part II, Section G, "Built-In Functions and Pseudo-Variables." In addition, a programmer may write functions for other purposes (as described in Chapter 10, "Subroutines and Functions"), and the names of those functions can be used in function references.

The use of a function reference is not limited to operands of operational expressions. A function reference is, in itself, an expression and can be used wherever an expression is allowed. It cannot be used in those cases where a variable represents a receiving field, such as to the left of an assignment symbol.

There are, however, ten built-in functions that can be used as pseudo-variables. A pseudo-variable is a built-in function name that is used in a receiving field. Consider the following example:

```
DECLARE A CHARACTER(10),
        B CHARACTER(30);

SUBSTR(A,6,5) = SUBSTR(B,20,5);
```

In this assignment statement, the SUBSTR built-in function name is used both in a normal function reference and as a pseudo-variable.

The SUBSTR built-in function extracts a substring of specified length from the named string. As a pseudo-variable, it indicates the location, within a named string, that is the receiving field.

In the above example, a substring five characters in length, beginning with character 20 of the string B, is to be assigned to the last five characters of the string A. That is, the last five characters of A are to be replaced by characters 20 through 24 of B. The first five characters of A remain unchanged, as do all of the characters of B.

All ten of the built-in functions that can be used as pseudo-variables are discussed in Part II, Section G, "Built-In Functions and Pseudo-Variables." No programmer-written function can be used as a pseudo-variable.

CONCEPTS OF DATA CONVERSION

Data conversion is the transformation of the representation of a value from one form to another. PL/I makes very few restrictions upon the use of the available forms of data representation or upon the mixing of different representations within an expression.

Programmers who wish to make use of this freedom must understand that mixed expressions imply conversions. If conversions take place at execution time, they will slow down the execution, sometimes significantly. Unless care is taken, conversions can result in loss of precision and can cause unexpected results. A lack of understanding of conversions can lead to logical errors and inaccuracies that are sometimes hard to trace.

This section is concerned primarily with the concepts of conversion operations. Specific rules for each kind of conversion are listed in Part II, Section F, "Problem Data Conversion." Earlier sections of this chapter discuss circumstances under which conversion can occur during evaluation of expressions. This section deals with the processes of the conversion.

The subject of conversion can be considered in two parts, first, determining the target attributes, and, second, the conversion operation with known source and target attributes. This section deals with determining target attributes. Rules for conversion operations are given in Part II, Section F, "Problem Data Conversion." Within each section, here and in Part II, arithmetic conversion and type conversion are considered separately.

The target of a conversion is the receiving field to which the converted value is assigned. In the case of a direct assignment, such as A = B, in which conversion must take place, the variable to the left of the assignment symbol (in this case, A) is the target. Consider the following example, however:

```
DECLARE  A CHARACTER(8),
         B FIXED DECIMAL(3,2),
         C FIXED BINARY(10);

A = B + C;
```

During the evaluation of the expression B+C and during the assignment of that result, there are four different targets, as follows:

1. The compiler-created temporary to which the converted binary equivalent of B is assigned

2. The compiler-created temporary to which the binary result of the addition is assigned

3. The temporary to which the converted decimal fixed-point equivalent of the binary result is assigned

4. A, the final destination of the result, to which the converted character-string equivalent of the decimal fixed-point representation of the value is assigned

The attributes of the first target are determined from the attributes of the source (B), from the operator, and from the attributes of the other operand (if one operand of an arithmetic infix operator is binary, the other is converted to binary before evaluation). The attributes of the second target are determined from the attributes of the source (C and the converted representation of B). The attributes of the third target are determined in part from the source (the second target) and in part from the attributes of the eventual target (A). (The only attribute determined from the eventual target is DECIMAL, since a binary arithmetic representation must be converted to decimal representation before it can be converted to a character string.) The attributes of the fourth target (A) are known from the DECLARE statement.

When an expression is evaluated, the target attributes usually are partly derived from the source, partly from the operation being performed, and partly from the attributes of a second operand. Some assumptions may be made, and some implementation restrictions (for example, maximum precision) and conventions exist. After an expression is evaluated, the result may be further converted. In this case, the target attributes usually are independent of the source. Since the process of determining target attributes is different for expression operands and for the results of expression evaluation, the two cases are dealt with separately.

A conversion always involves a source data item and a target data item, that is, the original representation of the value and the converted representation of the value. All of the attributes of both the source data item and the target data item are known, or assumed, at compile time.

It is possible for a conversion to involve intermediate results whose attributes may depend upon the source value. For example, conversion from character string to arithmetic may require an intermediate conversion and, thus, an intermediate result, before final conversion is completed. The final target attributes in such cases, however, are always determined from the source data item and are independent of the values of the variables.

It should be realized that constants also have attributes; the constant 1.0 is different from the constants 1, '1'B, '1', 1B, or 1E0. Constants may be converted at compile time or at execution time, but in either case, the rules are the same.

TARGET ATTRIBUTES FOR TYPE CONVERSION

When an expression operand requires type conversion, some target attributes must be assumed or deduced from the source. Some of these assumptions can be made based on the operator, as shown in Table 4-1.

Table 4-1. Target Types for Expression Operands

Operator	Target Type
+ - * / **	coded arithmetic
& | ¬	bit string
||	character string (unless both operands are bit strings)
> < >= <= = ¬= ¬> ¬<	arithmetic, unless both operands are strings; then character string, unless both operands are bit strings; then bit string

BIT TO CHARACTER AND CHARACTER TO BIT

In the conversion of bit to character, and character to bit, the length of the target (in bits or characters) is the same as the length of the source (in bits or characters).

ARITHMETIC TO STRING

In the conversion of arithmetic to bit-string or character-string data, the length of the target is deduced from the precision of the source. Algorithms for determining the length of the target are given below under the headings "Lengths of Bit-String Targets" and "Lengths of Character-String Targets." In the case of expression operands, there is no truncation of the resulting character-string value, since the length of the target is the length of the intermediate string.

STRING TO ARITHMETIC

In the conversion of bit-string or character-string data to arithmetic, the string must consist of digits that represent a valid arithmetic constant. The compiler has no way of determining the attributes of the constant represented by the string; therefore, attributes must be assumed for the target.

In the case of character-string to arithmetic conversion, the attributes assumed for the target are those attributes that would have been assumed if a fixed-point decimal integer of precision (15,0) had appeared in place of the string. Similarly, for a bit-string source that is to be converted to arithmetic type, the attributes of the target are the attributes that would have been given to the target if a fixed-point binary integer of precision (31,0) had appeared in place of the bit string.

Target Attributes for Arithmetic Expression Operands

Except for exponentiation, the target attributes for arithmetic conversion are assumed as follows:

BINARY	unless both operands are DECIMAL, in which case no base conversion is performed
FLOAT	unless both operands are FIXED, in which case no scale conversion is performed
COMPLEX	unless both operands are REAL, in which case no mode conversion is performed
precision of source	unless base or scale conversion is performed (see Table 4-2, "Precision for Arithmetic Conversion")

In the case of exponentiation, the base and precision are determined as for other operations. The target scale of the first operand is always FLOAT unless the first operand source is FIXED and the second operand (the exponent) is an unsigned fixed-point integer constant with a value small enough that the result of the exponentiation will not exceed the maximum number of digits allowed (for System/360 implementations, 31, if binary, or 15, if decimal). The target scale of the second operand is FLOAT unless it is an integer constant or a variable of precision (p,0). If either of the operands is COMPLEX, the target mode is COMPLEX for both operands unless the second operand is a REAL integer constant or variable of precision (p,0). In either case, the target mode for the second operand is REAL (that is, its mode is not converted).

In the examples of exponentiation shown below, the variables are those named in the following DECLARE statement:

```
DECLARE A FIXED DECIMAL(2),
        B FIXED DECIMAL(3,2),
        C FLOAT DECIMAL(4),
        D FLOAT DECIMAL(7),
        E FIXED DECIMAL(8),
        F FIXED DECIMAL(15),
        G COMPLEX FLOAT DECIMAL(6);
```

Note: If only one digit appears in the precision attribute specification for a fixed-point variable, the scale factor is, by default, zero; the precision is (p,0).

D ** C	No conversion necessary. Both operands are floating-point.
A ** 4	No conversion necessary. Second operand is unsigned fixed-point integer constant, and the result will not exceed 15 digits.
D ** 5	No conversion necessary. First operand is floating-point; second is fixed-point with precision (p,0).
D ** A	No conversion necessary. First operand is floating-point; second is fixed-point with precision (p,0).
E ** A	First operand is converted to floating-point because second operand is not unsigned fixed-point integer constant. Second operand is not converted because it has precision (p,0).
D ** B	Second operand is converted to floating-point because it does not have precision (p,0). Even if B had an integer value with a fractional part of zero, it still would be converted, since its declared precision is (3,2).
G ** B	First operand is complex. Second operand is converted to floating-point complex because its precision is not (p,0).

Note: All of these examples would be the same if they had been declared binary rather than decimal, except that the maximum number of binary digits allowed is 31.

Precision and Length of Expression Operand Targets

The following rules apply to all calculations of precision and length:

1. Precision and length specifications are always integers. If any of the calculations given below produces a nonintegral value, the next largest integer is taken as the resulting precision. In the case of scale factors, which can be negative, it is the absolute (positive) value that is used to take the next largest integer; the result, of course, will be negative if the source scale factor is negative.

   The following illustrates how precision would be computed in a conversion from DECIMAL FIXED (13,-4) to BINARY FIXED:

   ```
   1 + 13 * 3.32 = 44.16   resulting number
                           of digits (p) is
                           45
   -4 * 3.32 = -13.28      resulting scale
                           factor (q) is
                           -14
   ```

   Thus, the resulting precision is (45,-14); however, due to rule 2 below, it becomes (31,-14).

2. There is an implementation-defined maximum for the precision of each arithmetic representation. If any calculation yields a value greater than the implementation-defined limit, then the implementation limit is used instead. In System/360 implementations, these limits are:

   FIXED DECIMAL -- 15 digits

   FIXED BINARY -- 31 digits

   FLOAT DECIMAL -- 16 digits

   FLOAT BINARY -- 53 digits

   Because of the particular values for these implementations, these limits will usually come into effect only for conversions from fixed-point decimal to fixed-point binary.

   The scale factor for both binary and decimal base has the range +127 to -128 in System/360 implementations. This limit will rarely concern the programmer.

Precision for Arithmetic Conversions

Table 4-2 gives the target precision for an operand if base or scale conversion occurs.

The target precision of one operand of an expression is not affected by the precision of the other operand. This can have a significant effect on accuracy, particularly if one of the operands is a constant.

Table 4-2. Precision for Arithmetic Conversion

Source Attributes	Target Attributes	Target Precision
DECIMAL FIXED(p,q)	DECIMAL FLOAT	p
DECIMAL FIXED(p,q)	BINARY FIXED	1+p*3.32, q*3.32
DECIMAL FIXED(p,q)	BINARY FLOAT	p*3.32
DECIMAL FLOAT(p)	BINARY FLOAT	p*3.32
BINARY FIXED(p,q)	BINARY FLOAT	p
BINARY FIXED(p,q)	DECIMAL FIXED	1+p/3.32, q/3.32
BINARY FIXED(p,q)	DECIMAL FLOAT	p/3.32
BINARY FLOAT(p)	DECIMAL FLOAT	p/3.32

Note: Conversion from floating-point to fixed-point scale will occur only when a target precision is known, as in assignment to a fixed-point variable. If the target precision is incapable of holding the floating-point value, truncation on both left and right will occur, and the SIZE condition will be raised (if enabled) if significant digits are lost.

Table 4-3. Lengths of Character-String Targets

Source Attributes	Conditions	Target Length
DECIMAL FIXED(p,q)	If p>=q>=0	p+3
	If q>p or q negative	p+3+k (where k = number of decimal digits to express scale factor)
DECIMAL FLOAT(p)		p+6
Numeric character field		Same as source

Lengths of Character-String Targets

The length of a character-string target is related to the precision of the decimal source, as shown in Table 4-3.

Note: If a binary data item is converted to character, it is first converted to decimal. The precision of this intermediate conversion result controls the length of the final character-string target. Algorithms for computing the intermediate precision of a decimal item converted from binary are shown in Table 4-2.

For complex coded arithmetic sources, the target length is one greater than twice the length of the target for the corresponding real source. For complex numeric character data, the target length is twice the length of the real part of the source.

Lengths of Bit-String Targets

When converting arithmetic operands to bit string, the arithmetic source is converted to a positive binary integer. The precision of the binary integer target is the same as the length of the bit-string target as given in Table 4-4.

Note that p-q represents the number of binary or decimal digits to the left of the point. This could be zero or negative, in which case no conversion is performed and, for the F Compiler, the final result is a null string.

Table 4-4. Lengths of Bit-String Targets

Source Attributes	Target Length
DECIMAL FIXED(p,q)	(p-q)*3.32
DECIMAL FLOAT(p)	p*3.32
BINARY FIXED(p,q)	p-q
BINARY FLOAT(p)	p

Conversion of the Value of an Expression

The result of a completely evaluated expression may require further conversion. The circumstances in which this can occur, and the target attributes for each situation, are given in Table 4-5. In addition, certain built-in functions cause conversion. Any subscript reference is converted to binary integer.

CONVERSION OPERATIONS

As in the case of determining target attributes, conversion operations may also be considered in two stages: type conversion and arithmetic conversion. For example, when a character-string source is converted to a coded arithmetic target, the string is first converted to an arithmetic form whose attributes are determined by the constant expressed by the string. This intermediate result is then converted (if necessary) to the attributes of the target. These two stages may not be separated in an actual implementation, but for the purpose of description it is convenient to consider them separately.

Table 4-5. Circumstances that Can Cause Conversion

The following may cause conversion to any target attributes:

Cause	Target Attributes
Assignment	Attributes of variable to the left of the assignment symbol
Argument to procedure with ENTRY declared	Attributes of corresponding parameter declared in ENTRY declaration
RETURN(expression)	Attributes specified in PROCEDURE or ENTRY statement

The following may cause conversion to character-string:

Statement	Option	String Length
OPEN	TITLE	Source, 8-character maximum
DISPLAY		Source, 100-character maximum
RECORD I/O	KEYFROM	Key length specified in DD statement
	KEY	Key length specified in DD statement (or eight characters in the case of the regional number)

The following may cause conversion to a binary integer whose precision, as defined for the F Compiler, is given below:

Statement	Option/Attribute	Precision
DECLARE/ALLOCATE	length	15
	bounds	15
	repetition factor	15
DELAY	milliseconds	31
FORMAT	iteration factor	15
(and format items	w	15
in GET and PUT)	d	7
	s	7
OPEN	LINESIZE	15
	PAGESIZE	15
I/O	SKIP	15
	LINE	15
	IGNORE	15

There are six cases of type conversion:

Arithmetic to character-string

Character-string to arithmetic

Arithmetic to bit-string

Bit-string to arithmetic

Character-string to bit-string

Bit-string to character-string

For specific rules for each of the cases of type conversion and for arithmetic conversion, see Part II, Section F, "Problem Data Conversion."

THE CONVERSION, SIZE, FIXEDOVERFLOW, AND OVERFLOW CONDITIONS

When data is converted from one representation to another, the CONVERSION or SIZE conditions may be raised. The OVERFLOW and FIXEDOVERFLOW conditions are raised only when the result of an arithmetic operation exceeds the implementation-defined limit. When an operand is convert-

ed from one representation to another, if the value of the result will not fit in the declared precision for the new representation, the SIZE condition is raised.

The SIZE condition is raised when significant digits are lost from the left-hand side of an arithmetic value. This can occur during conversion within an expression, or upon assigning the result of an expression. It is not raised in conversion to character string or bit string even if the value is truncated. It is raised on conversion to E or F format in edit-directed transmission if the field width specified will not hold the value of the list item. The SIZE condition is normally disabled, so an interrupt will occur only if the condition is raised within the scope of a SIZE prefix.

The CONVERSION condition is raised when the source field contains a character that is invalid for the conversion being performed. For example, CONVERSION would be raised if a character string being converted to arithmetic contains any character other than those allowed in arithmetic constants, or if a character string that is being converted to bit contains any character other than 0 and 1. Each invalid character raises the CONVERSION condition once, so a single conversion operation causes several interrupts if more than one invalid character is encountered. The CONVERSION condition is normally enabled, so when the condition is raised, an interrupt will occur. It can be disabled by a NOCONVERSION prefix, in which case an interrupt will not occur when the condition is raised.

Note that the OVERFLOW and FIXEDOVERFLOW conditions are raised when an implementation maximum is exceeded, while the SIZE condition is raised when a declared precision is exceeded.

CHAPTER 5: STATEMENT CLASSIFICATION

This chapter classifies statements according to their functions. Statements in each functional class are listed, the purpose of each statement is described, and examples of their use are shown.

A detailed description of each statement is not included in this chapter but may be found in Part II, Section J, "Statements."

CLASSES OF STATEMENTS

Statements can be grouped into the following six classes:

Descriptive

Input/Output

Data Movement and Computational

Control

Exception Control

Program Structure

The names of the classes have been chosen for descriptive purposes only; they have no fundamental significance in the language. Some statements are included in more than one class, since they can have more than one function.

DESCRIPTIVE STATEMENTS

When a PL/I program is executed, it may manipulate many different kinds of data. Each data item, except a constant, is referred to in the program by a name. The PL/I language requires that the properties (or attributes) of data items referred to must be known at the time the program is compiled. There are a few exceptions to this rule; the bounds of the dimensions of arrays, the length of strings, and some file attributes may be determined during execution of the program.

The DECLARE Statement

The DECLARE statement is the principal means of specifying the attributes of a name. A name used in a program need not always appear in a DECLARE statement; its attributes often can be determined by context. If the attributes are not specifically declared and if they cannot be determined by context, then default rules are applied. The combination of default rules and context determination can make it unnecessary, in some cases, to use a DECLARE statement.

DECLARE statements are always needed for fixed-point decimal and floating-point binary variables, character- and bit-string variables, label variables, arrays and structures, static, controlled, and based variables, offset variables, and all data with the PICTURE attribute. An ENTRY declaration must be made in a DECLARE statement for the name of any function that returns a value with attributes different from the default attributes that would be assumed for the name -- FIXED BINARY(15) if the first letter of the name is I through N; otherwise, DECIMAL FLOAT(6). (The default precisions are those defined for System/360 implementations.) An ENTRY declaration also must be made if arguments and parameters do not match exactly, as may be the case when constants are passed as arguments.

DECLARE statements may also be an important part of the documentation of a program; consequently, programmers may make liberal use of declarations, even when default attributes apply or when a contextual declaration is possible. Because there are no restrictions on the number of DECLARE statements, different DECLARE statements can be used for different groups of names. This can make modification easier and the interpretation of diagnostics clearer.

Other Descriptive Statements

The OPEN statement allows certain attributes to be specified for a file name and may, therefore, also be classified as a descriptive statement. The FORMAT statement may be thought of as describing the layout of data on an external medium, such as on a page or an input card.

INPUT/OUTPUT STATEMENTS

The principal statements of the input/output class are those that actually cause a transfer of data between internal storage and an external medium. Other input/output statements, which affect such transfers, may be considered input/output control statements.

In the following list, the statements that cause a transfer of data are grouped into two subclasses, RECORD I/O and STREAM I/O:

RECORD I/O Transfer Statements

- READ
- WRITE
- REWRITE
- LOCATE
- DELETE

STREAM I/O Transfer Statements

- GET
- PUT

I/O Control Statements

- OPEN
- CLOSE
- UNLOCK

An allied statement, discussed with these statements, is the DISPLAY statement.

There are two important differences between STREAM transmission and RECORD transmission. In STREAM transmission, each data item is treated individually, whereas RECORD transmission is concerned with collections of data items (records) as a whole. In STREAM transmission, each item may be edited and converted as it is transmitted; in RECORD transmission, the record on the external medium is an exact copy of the record as it exists in internal storage, with no editing or conversion performed.

As a result of these differences, record transmission is particularly applicable for processing large files that are written in an internal representation, such as in binary or packed decimal. Stream transmission may be used for processing keypunched data and for producing readable output, where editing is required. Since files for which stream transmission is used tend to be smaller, the larger processing overhead can be ignored.

RECORD I/O Transfer Statements

The READ statement transmits records directly into working storage or makes records available for processing. The WRITE statement creates new records, transferring collections of data to the output device. The LOCATE statement allocates storage for a variable within an output buffer, setting a pointer to indicate the location in the buffer, having previously caused any record already located in a buffer for this file to be written out.

The REWRITE statement alters existing records in an UPDATE file. The DELETE statement removes records from an UPDATE file.

STREAM I/O Transfer Statements

Only sequential files can be processed with the GET and PUT statements. Record boundaries generally are ignored; data is considered to be a stream of individual data items, either coming from (GET) or going to (PUT) the external medium.

The GET and PUT statements may transmit a list of items in one of three modes, data-directed, list-directed, or edit-directed. In data-directed transmission, the names of the data items, as well as their values, are recorded on the external medium. In list-directed transmission, the data is recorded externally as a list of constants, separated by blanks or commas. In edit-directed transmission, the data is recorded externally as a string of characters to be treated character by character according to a format list.

Data-directed transmission is most useful for reading a relatively small number of values and for producing self-annotated debugging output. List-directed input is suitable for reading in larger volumes of data punched in free form. Edit-directed transmission is used wherever format must be strictly controlled, for example, in producing reports and for reading cards punched in a fixed format.

Note: The GET and PUT statements can also be used for internal data movement, by specifying a string name in the STRING option instead of specifying the FILE option. Although the facility may be used

with READ and WRITE statements for moving data to and from a buffer, it is not actually a part of the input/output operation. GET and PUT statements with the STRING option are discussed in the section "Data Movement and Computational Statements," in this chapter.

Input/Output Control Statements

The OPEN statement associates a file name with a data set and prepares the data set for processing. It may also specify additional attributes for the file.

An OPEN statement need not always be written. Execution of any input or output transmission statement that specifies the name of an unopened file will result in an automatic opening of the file before the data transmission takes place.

The OPEN statement may be used to declare attributes for a file; for a PRINT file, the length of each printed line and the number of lines per page can be specified only in an OPEN statement. The OPEN statement can also be used to specify a name (in the TITLE option) other than the file name, as a link between the data set and the file.

The CLOSE statement dissociates a data set from a file. All files are closed at termination of a program, so a CLOSE statement is not always required.

The UNLOCK statement releases a record that has been temporarily locked by the task executing the UNLOCK statement, so that other concurrent tasks may resume access to the record. The UNLOCK statement is not always required; the unlocking operation is automatic when the task that locked the record deletes or rewrites it, or closes the file, or when the task is terminated.

The DISPLAY Statement

The DISPLAY statement is used to write messages on the console, usually to the operator. It may also be used, with the REPLY option, to allow the operator to communicate with the program by typing in a code or a message. The REPLY option may be used merely as a means of suspending program execution until the operator acknowledges the message.

DATA MOVEMENT AND COMPUTATIONAL STATEMENTS

Internal data movement involves the assignment of the value of an expression to a specified variable. The expression may be a constant or a variable, or it may be an expression that specifies computations to be made.

The most commonly used statement for internal data movement, as well as for specifying computations, is the assignment statement. The GET and PUT statements with the STRING option also can be used for internal data movement. The PUT statement can, in addition, specify computations to be made.

The Assignment Statement

The assignment statement, which has no keyword, is identified by the assignment symbol (=). It generally takes one of two forms:

```
A = B;

A = B + C;
```

The first form can be used purely for internal data movement. The value of the variable (or constant) to the right of the assignment symbol is to be assigned to the variable to the left. The second form includes an operational expression whose value is to be assigned to the variable to the left of the assignment symbol. The second form specifies computations to be made, as well as data movement.

Since the attributes of the variable on the left may differ from the attributes of the result of the expression (or of the variable or constant), the assignment statement can also be used for conversion and editing.

The variable on the left may be the name of an array or a structure; the expression on the right may yield an array or structure value. Thus the assignment statement can be used to move aggregates of data, as well as single items.

Multiple Assignment

The value of the expression in an assignment statement can be assigned to more than one variable in a statement of the following form:

```
A,X = B + C;
```

Such a statement is executed in exactly the same way as a single assignment, except that the value of B + C is assigned to both A and X. In general, it has the same effect as if the following two statements had been written:

```
A = B + C;

X = B + C;
```

Note: If multiple assignment is used for a structure assignment BY NAME, the elementary names affected will be only those that are common to all of the structures listed to the left of the assignment symbol.

The STRING Option

If the STRING option appears in a GET or PUT statement in place of a FILE option, execution of the statement will result only in internal data movement; neither input nor output is involved.

Assume that NAME is a string of 30 characters and that FIRST, MIDDLE, and LAST are string variables. Consider the following example:

```
GET STRING (NAME) EDIT
    (FIRST,MIDDLE, LAST)
    (A(12),A(1),A(17));
```

This statement specifies that the first 12 characters of NAME are to be assigned to FIRST, the next character to MIDDLE, and the remaining 17 characters to LAST.

The PUT statement with the string option specifies the reverse operation, that is, that the values of the specified variables are to be concatenated into a string and assigned as the value of the string named in the STRING option. For example:

```
PUT STRING (NAME) EDIT
    (FIRST,MIDDLE, LAST)
    (A(12),A(1),A(17));
```

This statement specifies that the values of FIRST, MIDDLE, and LAST are to be concatenated, in that order, and assigned to the string variable NAME.

Computations to be performed can be specified in a PUT statement by including operational expressions in the data list. Assume, for the following example, that the variables A, B, and C represent arithmetic data and BUFFER represents a character string:

```
PUT STRING (BUFFER) LIST (A*3,B+C);
```

This statement specifies that the character string assigned to BUFFER is to consist of the character representations of the value of A multiplied by 3 and the value of the sum of B and C.

Operational expressions in the data list of a PUT statement are not limited to PUT statements with the STRING option. Operational expressions can appear in PUT statements that specify output to a file.

CONTROL STATEMENTS

Statements in a PL/I program, in general, are executed sequentially unless the flow of control is modified by the occurrence of an interrupt or the execution of one of the following control statements:

```
GO TO

IF

DO

CALL

RETURN

END

STOP

EXIT
```

The GO TO Statement

The GO TO statement is most frequently used as an unconditional branch. If the destination of the GO TO is specified by a label variable, it may then be used as a switch by assigning label constants, as values, to the label variable.

If the label variable is subscripted, the switch may be controlled by varying the subscript. Since multidimensional label arrays are allowed, and since logical values may be used as subscripts, quite subtle switching can be effected. It is usually true, however, that simple control statements are the most efficient.

The keyword of the GO TO statement may be written either as two words separated by a blank or as a single word, GOTO.

The IF Statement

The IF statement provides the most common conditional branch and is usually used with a simple comparison expression following the word IF. For example:

```
IF A = B

   THEN action-if-true

   ELSE action-if-false
```

If the comparison is true, the THEN clause (the "action to be taken") is executed. After execution of the THEN clause, control branches around the ELSE clause (the "alternate action"), and execution continues with the next statement. Note that the THEN clause can contain a GO TO statement or some other control statement that would result in a different transfer of control.

If the comparison is not true, control branches around the THEN clause, and the ELSE clause is executed. Control then continues normally.

The IF statement might be as follows:

```
IF A = B

   THEN C = D;

   ELSE C = E;
```

If A is equal to B, the value of D is assigned to C, and control branches around the ELSE clause. If A is not equal to B, control branches around the THEN clause, and the value of E is assigned to C.

Either the THEN clause or the ELSE clause can contain some other control statement that causes a branch, either conditional or unconditional. If the THEN clause contains a GO TO statement, for example, there is no need to specify an ELSE clause. Consider the following example:

```
IF A = B

   THEN GO TO LABEL_1;

   next-statement
```

If A is equal to B, the GO TO statement of the THEN clause causes an unconditional branch to LABEL_1. If A is not equal to B, control branches around the THEN clause to the next statement, whether or not it is an ELSE clause associated with the IF statement.

Note: If the THEN clause does not cause a transfer of control and if it is not followed by an ELSE clause, the next statement will be executed whether or not the THEN clause is executed.

The expression following the IF keyword can be only an element expression; it cannot be an array or structure expression. It can, however, be a logical expression with more than one operator. For example:

```
IF A = B & C = D
   THEN GO TO R;
```

The same kind of test could be made with nested IF statements. The following three examples are equivalent:

```
IF A = B & C = D
   THEN GO TO R;
B = B + 1;

IF A = B
   THEN IF C = D
           THEN GO TO R;
B = B + 1;

IF A ¬= B THEN GO TO S;
IF C ¬= D THEN GO TO S;
GO TO R;
S: B = B + 1;
```

The DO Statement

The most common use of the DO statement is to specify that a group of statements is to be executed a stated number of times while a control variable is incremented each time through the loop. Such a group might take the form:

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

The statements to be executed iteratively must be delimited by the DO statement and an associated END statement. In this case, the group of statements will be executed ten times, while the value of the control variable I ranges from 1 through 10. The effect of the DO and END statements would be the same as the following:

```
   I = 0;
A: I = I + 1;
   IF I > 10 THEN GO TO B;
     .
     .
     .
   GO TO A;
B: next statement
```

Note that the increment is made before the control variable is tested and that, in general, control goes to the statement following the group only when the value of the control variable exceeds the limit set in the DO statement. If a reference is made to a control variable after the last iteration is completed, the value of the variable will be one increment beyond the specified limit.

The DO statement can also be used with the WHILE option and no control variable, as follows:

```
DO WHILE (A = B);
```

This statement, heading a group, causes the group to be executed repeatedly so long as the value of A remains equal to the value of B.

The WHILE option can be combined with a control variable of the form:

```
DO I = 1 TO 10 WHILE (A = B);
```

This statement specifies two tests. Each time that I is incremented, a test is made to see that I has not exceeded 10. An additional test then is made to see that A is equal to B. Only if both conditions are satisfied will the statements of the group be executed.

More than one successive iteration specification can be included in a single DO statement. Consider each of the following DO statements:

```
DO I = 1 TO 10, 13 TO 15;

DO I = 1 TO 10, 11 WHILE (A = B);
```

The first statement specifies that the DO group is to be executed a total of thirteen times, ten times with the value of I equal to 1 through 10, and three times with the value of I equal to 13 through 15. The second DO statement specifies that the group is to be executed at least ten times, and then (provided that A is equal to B) once more; if "BY 0" were inserted after "11", execution would continue with I set to 11 as long as A remained equal to B. Note that in both statements a comma is used to separate the two specifications. This indicates that a succeeding specification is to be considered only after the preceding specification has been satisfied.

The control variable of a DO statement can be used as a subscript in statements within the DO-group, so that each iteration deals with successive elements of a table or array. For example:

```
DO I = 1 TO 10;
   A(I) = I;
   END;
```

In this example, the first ten elements of A are set to 1,2,...,10, respectively.

The increment in the iteration specification is assumed to be one unless some other value is stated, as follows:

```
DO I = 2 TO 10 BY 2;
```

This specifies that the loop is to be executed five times, with the value of I equal to 2, 4, 6, 8, and 10.

Noniterative DO Statements

The DO statement need not specify repeated execution of the statements of a DO-group. A simple DO statement, in conjunction with a DO-group, can be used as follows:

```
DO;
 .
 .
 .
END;
```

The use of the simple DO statement in this manner merely indicates that the DO-group is to be treated logically as a single statement. It can be used to specify a number of statements to be executed in the THEN clause or the ELSE clause of an IF statement, thus maintaining sequential control without the use of a begin block. (Only a single statement, a DO-group, or a begin block can be specified in the THEN clause or in the ELSE clause.)

The CALL, RETURN, and END Statements

A subroutine may be invoked by a CALL statement that names an entry point of the subroutine. When the multitasking facilities are not in use, control is returned to the activating, or invoking, procedure when a RETURN statement is executed in the subroutine or when execution of the END statement terminates the subroutine. If the CALL statement contains one of the multitasking options, TASK, EVENT, or PRIORITY, the subroutine is executed by a subtask with its own separate flow of control; in this case, the RETURN or END statement merely terminates the separate flow of control established for the subtask. (See Chapter 15, "Multitasking.")

The RETURN statement with a parenthesized expression is used in a function procedure to return a value to a function reference. This form is used to return a value from a procedure that has been invoked by a function reference.

Normal termination of a program occurs as the result of execution of the final END statement of the main procedure or of a RETURN statement in the main procedure, either of which returns control to the operating system.

The STOP and EXIT Statements

The STOP and EXIT statements are both used to cause abnormal termination. The STOP statement terminates execution of the entire program, including all concurrent tasks. The EXIT statement terminates only the task that executes it, together with any attached tasks. (See Chapter 15, "Multitasking.")

EXCEPTION CONTROL STATEMENTS

The control statements, discussed in the preceding section, alter the flow of control whenever they are executed. Another way in which the sequence of execution can be altered is by the occurrence of a program interrupt caused by an exceptional condition that arises.

In general, an exceptional condition is the occurrence of an unexpected action, such as an overflow error, or of an expected action, such as an end of file, that occurs at an unpredictable time. A detailed discussion of the handling of these conditions appears in Chapter 11, "Exceptional Condition Handling and Program Checkout."

The three exception control statements are the ON statement, the REVERT statement, and the SIGNAL statement.

The ON Statement

The ON statement is used to specify action to be taken when any subsequent occurrence of a specified condition causes a program interrupt. ON statements may specify particular action for any of a number of different conditions. For all of these conditions, a standard system action exists as a part of PL/I, and if no ON statement is in force at the time an interrupt occurs, the standard system action will take place. For most conditions, the standard system action is to print a message and terminate execution.

The ON statement takes the form:

```
ON condition-name{SYSTEM;|on-unit}
```

The "condition name" is one of the keywords listed in Part II, Section H, "ON-Conditions." The "on-unit" is a single statement or a begin block that specifies action to be taken when that condition arises and an interrupt occurs. For example:

```
ON ENDFILE(DETAIL) GO TO NEXT_MASTER;
```

This statement specifies that when an interrupt occurs as the result of trying to read beyond the end of the file named DETAIL, control is to be transferred to the statement labeled NEXT_MASTER.

When execution of an on-unit is successfully completed, control will normally return to the point of the interrupt or to a point immediately following it, depending upon the condition that caused the interrupt.

An important use of the ON statement is for debugging. The CHECK condition causes debugging information to be printed whenever the value of one of a list of specified variables is changed or whenever a specified statement is executed.

The effect of an ON statement, the establishment of the on-unit, can be changed within a block (1) by execution of another ON statement naming the same condition with either another on-unit or the word SYSTEM, which re-establishes standard system action, or (2) by the execution of a REVERT statement naming that condition. On-units in effect at the time another block is activated remain in effect in the activated block, and in other blocks activated by it, unless another ON statement for the same condition is executed. When control returns to an activating block, on-units are re-established as they existed.

The REVERT Statement

The REVERT statement is used to cancel the effect of all ON statements for the same condition that have been executed in the block in which the REVERT statement appears.

The REVERT statement, which must specify the condition name, re-establishes the on-unit that was in effect in the activating block at the time the current block was invoked.

The SIGNAL Statement

The SIGNAL statement simulates the occurrence of an interrupt for a named condition. It can be used to test the coding of the on-unit established by execution of an ON statement. For example:

```
SIGNAL OVERFLOW;
```

This statement would simulate the occurrence of an overflow interrupt and would cause execution of the on-unit established for the OVERFLOW condition. If an on-unit has not been established, standard system action is taken.

PROGRAM STRUCTURE STATEMENTS

The program structure statements are those statements used to delimit sections of a program into blocks and groups, and to control the allocation of storage within a program. These statements are the PROCEDURE statement, the END statement, the ENTRY statement, the BEGIN statement, the DO statement, the ALLOCATE statement, and the FREE statement. The concept of blocks and groups is fundamental to a proper understanding of PL/I and is dealt with in detail in Chapters 6, 7, and 10.

Proper division of a program into blocks simplifies the writing and testing of the program, particularly when a number of programmers are co-operating in writing a single program. It may also result in more efficient use of storage, since dynamic storage of the automatic class is allocated on entry to the block in which it is declared.

The PROCEDURE Statement

The principal function of a procedure block, which is delimited by a PROCEDURE statement and an associated END statement, is to define a sequence of operations to be performed upon specified data. This sequence of operations is given a name (the label of the PROCEDURE statement) and can be invoked from any point at which the name is known.

Every program must have at least one PROCEDURE statement and one END statement. A program may consist of a number of separately written procedures linked together. A procedure may also contain other procedures nested within it. These internal procedures may contain declarations that are treated (unless otherwise specified) as local definitions of names. Such definitions are not known outside their own block, and the names cannot be referred to in the containing procedure. Storage associated with these names is generally allocated upon entry to the block in which such a name is defined, and it is freed upon exit from the block.

The sequence of statements defined by a procedure can be executed at any point at which the procedure name is known. This execution can be either synchronous (that is, the execution of the invoking procedure is suspended until control is returned to it) or asynchronous (that is, execution of the invoking procedure proceeds concurrently with that of the invoked procedure); for details of asynchronous operation, see Chapter 15, "Multitasking." A procedure is invoked either by a CALL statement or by the appearance of its name in an expression, in which case the procedure is called a function procedure. A function reference causes a value to be calculated and returned to the function reference for use in the evaluation of the expression. A function procedure cannot be executed asynchronously with the invoking procedure.

Communication between two procedures is by means of arguments passed from an invoking procedure to the invoked procedure, by a value returned from an invoked procedure, and by names known within both procedures. A procedure may therefore operate upon different data when it is invoked from different points. A value is returned from a function procedure to a function reference by means of the RETURN statement.

The ENTRY Statement

The ENTRY statement is used to provide an alternate entry point to a procedure and, possibly, an alternate parameter list to which arguments can be passed, corresponding to that entry point.

Note: It is important to distinguish between the ENTRY statement, which specifies an entry to the procedure in which it occurs, and the ENTRY attribute specification, which describes the attributes of parameters of procedures that are invoked from the procedure in which the ENTRY attribute specification appears.

The BEGIN Statement

Local definitions of names can also be made within begin blocks, which are delimited by a BEGIN statement and an associated END statement. Begin blocks, however, are executed in the normal flow of a program, either sequentially or as a result of a GO TO or an IF statement transfer. One of the most common uses of a begin block is as the on-unit of an ON statement, in which case it is not executed through normal flow of control, but only upon occurrence of the specified condition. It is also useful for delimiting a section of a program in which some automatic storage is to be allocated.

Each begin block must be nested within a procedure or another begin block.

The DO Statement

Another kind of program structure is provided by the DO-group, which is delimited by a DO statement and an associated END statement. A DO-group does not have any effect upon the allocation of storage or the meaning of names. A DO-group specifies that the statements contained within it are to be considered as an entity for the purpose of flow of control.

A DO statement may specify repeated execution of a sequence of statements until a criterion is satisfied, or it may indicate within an IF statement that a group of statements is to be taken together as the whole of the THEN clause or of the ELSE clause.

The ALLOCATE and FREE Statements

As with many other conventions in PL/I, the convention concerning storage allocation and the scope of definitions of names can be overridden by the programmer. The storage class attribute AUTOMATIC is assumed for most variables. However a variable can be declared STATIC, in which case it is allocated throughout the entire program; or it can be declared CONTROLLED, or BASED, in which case its allocation can be explicitly specified by the programmer.

The ALLOCATE statement is used to assign storage to controlled and based data, independent of block boundaries. The bounds of controlled arrays and the length of controlled strings, as well as their initial values, may also be specified at the time the ALLOCATE statement is executed. The FREE statement is used to free controlled and based storage after it has been allocated.

This section discusses how statements can be organized into blocks to form a PL/I program, how control flows within a program from one block of statements to another, and how storage may be allocated for data within a block of statements. The discussion in this chapter does not completely cover multitasking, which is discussed in detail in Chapter 15. However, the discussion generally applies to all blocks, whether or not they are executed concurrently.

BLOCKS

A block is a delimited sequence of statements that constitutes a section of a program. It localizes names declared within the block and limits the allocation of variables. There are two kinds of blocks: procedure blocks and begin blocks.

PROCEDURE BLOCKS

A procedure block, simply called a procedure, is a sequence of statements headed by a PROCEDURE statement and ended by an END statement, as follows:

```
label: [label:]... PROCEDURE;
                      .
                      .
                      .
                   END[label];
```

All procedures must be named because the procedure name is the primary point of entry through which control can be transferred to a procedure. Hence, a PROCEDURE statement must have at least one label. A label need not appear after the keyword END in the END statement, but if one does appear, it must match the label (or one of the labels) of the PROCEDURE statement to which the END statement corresponds. (There are exceptions; see "Use of the END Statement with Nested Blocks and DO-Groups" in this chapter.) An example of a procedure follows:

```
A:  READIN:  PROCEDURE
             statement-1
             statement-2
                .
                .
                .
             statement-n
             END READIN;
```

In general, control is transferred to a procedure through a reference to the name (or one of the names) of the procedure. Thus, the procedure in the above example would be given control by a reference to either of its names, A or READIN.

A PL/I program consists of one or more such procedures, each of which may contain other procedures and/or begin blocks.

BEGIN BLOCKS

A begin block is a set of statements headed by a BEGIN statement and ended by an END statement, as follows:

```
[label:]... BEGIN;
               .
               .
               .
            END [label];
```

Unlike a procedure block, a label is optional for a begin block. If one or more labels are prefixed to a BEGIN statement, they serve only to identify the starting point of the block. (Control may pass to a begin block without reference to the name of that block through normal sequential execution, although control can be transferred to a labeled BEGIN statement by execution of a GO TO statement.) The label following END is optional. However, a label can appear after END, matching a label of the corresponding BEGIN statement. (There are exceptions; see "Use of the END Statement with Nested Blocks and DO-Groups" in this chapter.) An example of a begin block follows:

```
B:  CONTROL:  BEGIN;
              statement-1
              statement-2
                 .
                 .
                 .
              statement-n
              END B;
```

Unlike procedures, begin blocks generally are not given control through special references to them. The normal sequence of control governing ordinary statement execution also governs the execution of begin blocks. Control passes into a begin block sequentially, following execution of the preceding statement.

Begin blocks are not essential to the construction of a PL/I program. However, there are times when it is advantageous to use begin blocks to delimit certain areas of a program. These advantages are discussed in this chapter and in Chapter 7, "Recognition of Names."

INTERNAL AND EXTERNAL BLOCKS

Any block can contain one or more blocks. That is, a procedure, as well as a begin block, can contain other procedures and begin blocks. However, there can be no overlapping of blocks; a block that contains another block must totally encompass that block.

A procedure block that is contained within another block is called an internal procedure. A procedure block that is not contained within another block is called an external procedure. There must always be at least one external procedure in a PL/I program. (Note: With System/360 implementations, each external procedure is compiled separately. Entry names of external procedures cannot exceed seven characters.)

Begin blocks are always internal; they must always be contained within another block.

Internal procedure and begin blocks can also be referred to as nested blocks. Nested blocks, in turn, may have blocks nested within them, and so on. The outermost block always must be a procedure. Consider the following example:

```
A: PROCEDURE;
   statement-a1
   statement-a2
   statement-a3
   B: BEGIN;
      statement-b1
      statement-b2
      statement-b3
      END B;
   statement-a4
   statement-a5
   C: PROCEDURE;
      statement-c1
      statement-c2
      D: BEGIN;
         statement-d1
         statement-d2
         statement-d3
         E: PROCEDURE;
            statement-e1
            statement-e2
            END E;
         statement-d4
         END D;
      END C;
   statement-a6
   statement-a7
   END A;
```

In the above example, procedure block A is an external procedure because it is not contained in any other block. Block B is a begin block that is contained in A; it contains no other blocks. Block C is an internal procedure; it contains begin block D, which, in turn, contains internal procedure E. This example contains three levels of nesting relative to A; B and C are at the first level, D is at the second level (but the first level relative to C) and E is at the third level (the second level relative to C, and the first level relative to D).

Use of the END Statement with Nested Blocks and DO-Groups (Multiple Closure)

The use of the END statement with a procedure, begin block, or DO-group is governed by the following rules:

1. If a label is not used after END, the END statement closes (i.e., ends) that unclosed block headed by the BEGIN or PROCEDURE statement, or that unclosed DO-group headed by the DO statement, that physically precedes, and appears closest to, the END statement.

2. If the optional label is used after END, the END statement closes that unclosed block or DO-group headed by the BEGIN, PROCEDURE, or DO statement that has a matching label, and that physically precedes, and appears closest to, the END statement. Any unclosed blocks or DO-groups nested within such a block or DO-group are automatically closed by this END statement; this is known as multiple closure.

From the second rule, it is evident that nested blocks sometimes make it possible for a single END statement to close more than one block. For example, assume that the following external procedure has been defined:

```
FRST: PROCEDURE;
      statement-f1
      statement-f2
      ABLK: BEGIN;
            statement-a1
            statement-a2
            SCND: PROCEDURE;
                  statement-s1
                  BBLK: BEGIN;
                        statement-b1
                        END;
                  END;
            statement-a3
            END ABLK;
      END FRST;
```

In this example, begin block BBLK and internal procedure SCND effectively end in the same place; that is, there are no statements between the END statements for each. This is also true for begin block ABLK and external procedure FRST. In such cases, it is not necessary to use an END statement for each block, as shown; rather, one END statement can be used to end BBLK and SCND, and another END can be used to end ABLK and FRST. In the first case, the statement would be END SCND, because one END statement with no following label would close only the begin block BBLK (see the first rule above). In the second case, only the statement END FRST is required; the statement END ABLK is superfluous. Thus, the example could be specified as follows:

```
FRST: PROCEDURE;
      statement-f1
      statement-f2
      ABLK: BEGIN;
            statement-a1
            statement-a2
            SCND: PROCEDURE;
                  statement-s1
                  statement-s2
                  BBLK: BEGIN;
                        statement-b1
                        statement-b2
                  END SCND;
            statement-a3
      END FRST;
```

ACTIVATION AND TERMINATION OF BLOCKS

ACTIVATION

Although the begin block and the procedure have a physical resemblance and play the same role in the allocation and freeing of storage, as well as in delimiting the scope of names, they differ in the way they are activated and executed. A begin block, like a single statement, is activated and executed in the course of normal sequential program flow (except when specified as an on-unit) and, in general, can appear wherever a single statement can appear. For a procedure, however, normal sequential program flow passes around the procedure, from the statement before the PROCEDURE statement to the statement after the END statement of that procedure. The only way in which a procedure can be activated is by a procedure reference.

A procedure reference is the appearance of an entry name (defined below) in one of the following contexts:

1. After the keyword CALL in a CALL statement
2. After the keyword CALL in the CALL option of the INITIAL attribute (see the discussion of the INITIAL attribute in Part II, Section I, "Attributes," for details)
3. As a function reference (see Chapter 10, "Subroutines and Functions," for details)

This chapter uses examples of the first of these; that is, with the procedure reference of the form:

```
CALL entry-name;
```

The material, however, is relevant to the other two forms as well.

An entry name is defined as either of the following:

1. The label, or one of the labels, of a PROCEDURE statement
2. The label, or one of the labels, of an ENTRY statement appearing within a procedure

The first of these is called the primary entry point to a procedure; the second is known as a secondary entry point to a procedure. The following is an example of a procedure containing secondary entry points:

```
          A: PROCEDURE;
             statement-1
             statement-2
       ERRT: ENTRY;
             statement-3
             statement-4
             statement-5
  NEXT: RETR: ENTRY;
             statement-6
             statement-7
             statement-8
             END A;
```

In this example, A is the primary entry point to the procedure, and ERRT, NEXT, and RETR specify secondary entry points. Actually, since they are both labels of the same ENTRY statement, NEXT and RETR specify the same secondary entry point.

When a procedure reference is executed, the procedure containing the specified entry point is activated and is said to be invoked; control is transferred to the specified entry point.[1] The point at which the procedure reference appears is called the point of invocation and the block in which the reference is made is called the invoking block. An invoking block remains active even though control is transferred from it to the block it invokes.

Whenever a procedure is invoked at its primary entry point, execution begins with the first executable statement in the invoked procedure. However, when a procedure is invoked at a secondary entry point, execution begins with the first executable statement following the ENTRY statement that defines that secondary entry point. Therefore, if all of the numbered statements in the last example are executable, the statement CALL A would invoke procedure A at its primary entry point, and execution would begin with statement-1; the statement CALL ERRT would invoke procedure A at the secondary entry point ERRT, and execution would begin with statement-3; either of the statements CALL NEXT or CALL RETR would invoke procedure A at its other secondary entry point, and execution would begin with statement-6. Note that any ENTRY statements encountered during sequential flow are never executed; control flows around the ENTRY statement as though the statement were a comment.

Any procedure, whether external or internal, can always invoke an external procedure, but it cannot always invoke an internal procedure that is contained in some other procedure. Those internal procedures that are at the first level of nesting relative to a containing procedure can always be invoked by that containing procedure, or by each other. For example:

```
PRMAIN: PROCEDURE;
        statement-1
        statement-2
        statement-3
        A: PROCEDURE;
           statement-a1
           statement-a2
           B: PROCEDURE;
              statement-b1
              statement-b2
           END A;
        statement-4
        statement-5
        C: PROCEDURE;
           statement-c1
           statement-c2
           END C;
        statement-6
        statement-7
        END PRMAIN;
```

In this example, PRMAIN can invoke procedures A and C, but not B; procedure A can invoke procedures B and C; procedure B can invoke procedure C; and procedure C can invoke procedure A but not B.

The foregoing discussion about the activation of blocks presupposes that a program has already been activated. A PL/I program becomes active when a calling program invokes the initial procedure. This calling program usually is the operating system, although it could be another program. For System/360 implementations, the initial procedure, called the main procedure, must be an external procedure whose PROCEDURE statement has the OPTIONS(MAIN) specification, as shown in the following example:

```
CONTRL: PROCEDURE OPTIONS(MAIN);
        CALL A;
        CALL B;
        CALL C;
        END CONTRL;
```

In this example, CONTRL is the initial procedure and it invokes other procedures in the program.

The following is a summary of what has been stated, or at least implied, about the activation of blocks:

- A program becomes active when the initial procedure is activated by the operating system.
- Except for the initial procedure, external and internal procedures contained in a program are activated only when they are invoked by a procedure reference.
- Begin blocks are activated through normal sequential flow or as on-units.

---

[1] This statement does not apply when the CALL statement specifies one of the multitasking options. See Chapter 15, "Multitasking."

- The initial procedure remains active for the duration of the program.
- All activated blocks remain active until they are terminated (see below).

TERMINATION

In general, a procedure block is terminated when, by some means other than a procedure reference, control passes back to the invoking block or to some other active block. Similarly, a begin block is terminated when, by some means other than a procedure reference, control passes to another active block. There are a number of ways by which such transfers of control can be accomplished, and their interpretations differ according to the type of block being terminated.

Note that when a block is terminated, any task attached by that block is terminated (see Chapter 15, "Multitasking").

Begin Block Termination

A begin block is terminated when any of the following occurs:

1. Control reaches the END statement for the block. When this occurs, control moves to the statement physically following the END, except when the block is an on-unit.
2. The execution of a GO TO statement within the begin block (or any block activated from within that begin block) transfers control to a point not contained within the block.
3. A STOP or EXIT statement is executed (thereby terminating execution).
4. Control reaches a RETURN statement that transfers control out of the begin block and out of its containing procedure as well.

A GO TO statement of the type described in item 2 can also cause the termination of other blocks as follows:

If the transfer point is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are terminated.

For example, if begin block B is contained in begin block A, then a GO TO statement in B that transfers control to a point contained in neither A nor B effectively terminates both A and B. This case is illustrated below:

```
FRST: PROCEDURE OPTIONS(MAIN);
      statement-1
      statement-2
      statement-3
      A: BEGIN;
         statement-a1
         statement-a2
         B: BEGIN;
            statement-b1
            statement-b2
            GO TO LAB;
            statement-b3
            END B;
         statement-a3
         END A;
      statement-4
      statement-5
LAB:  statement-6
      statement-7
      END FRST;
```

After FRST is invoked, the first three statements are executed and then begin block A is activated. The first two statements in A are executed and then begin block B is activated (A remaining active). When the GO TO statement in B is executed, control passes to statement-6 in FRST. Since statement-6 is contained in neither A nor B, both A and B are terminated. Thus, the transfer of control out of begin block B results in the termination of intervening block A as well as termination of block B.

Procedure Termination

A procedure is terminated when one of the following occurs:

1. Control reaches a RETURN statement within the procedure. The execution of a RETURN statement causes control to be returned to the point of invocation in the invoking procedure. If the point of invocation is a CALL statement, execution in the invoking procedure resumes with the statement following the CALL. If the point of invocation is one of the other forms of procedure references (that is, a CALL option or a function reference), execution of the statement containing the reference will be resumed.
2. Control reaches the END statement of the procedure. Effectively, this is equivalent to the execution of a RETURN statement.

3. The execution of a GO TO statement within the procedure (or any block activated from within that procedure) transfers control to a point not contained within the procedure.

4. A STOP or EXIT statement is executed (thereby terminating execution).

Items 1, 2, and 3 are normal procedure terminations; item 4 is abnormal procedure termination.

As with a begin block, the type of termination described in item 3 can sometimes result in the termination of several procedures and/or begin blocks. Specifically, if the transfer point specified by the GO TO statement is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are terminated. Consider the following example:

```
A: PROCEDURE OPTIONS(MAIN);
   statement-1
   statement-2
   B: BEGIN;
      statement-b1
      statement-b2
      CALL C;
      statement-b3
      END B;
   statement-3
   statement-4
   C: PROCEDURE;
      statement-c1
      statement-c2
      statement-c3
      D: BEGIN;
         statement-d1
         statement-d2
         GO TO LAB;
         statement-d3
         END D;
      statement-c4
      END C;
   statement-5
LAB: statement-6
   statement-7
   END A;
```

In the above example, A activates B, which activates C, which activates D. In D, the statement GO TO LAB transfers control to statement-6 in A. Since this statement is not contained in D, C, or B, all three blocks are terminated; A remains active. Thus, the transfer of control out of D results in the termination of intervening blocks B and C as well as the termination of block D.

Program Termination

A program is terminated when any one of the following occurs:

1. Control for the program reaches an EXIT statement. This is abnormal termination.

2. Control for the program reaches a STOP statement.[1] This also is abnormal termination.

3. Control reaches a RETURN statement or the final END statement in the main procedure. This is normal termination.

4. An on-unit for the ERROR condition is executed with normal return (that is, a GO TO statement does not transfer control out of the on-unit) or the FINISH condition is raised as a result of the standard system action for the ERROR condition.

Note: The termination of a program, whether normal or abnormal , raises the FINISH condition. The standard system action for this condition is to return control to the operating system control program. For normal termination, the control program will then pass control to the calling program, if any. For abnormal termination, it will terminate the job. (See Part II, Section H, "ON-Conditions.")

STORAGE ALLOCATION

Storage allocation is the process of associating an area of storage with a variable so that the data item(s) to be represented by the variable may be recorded internally. When storage has been associated with a variable, the variable is said to be allocated. Allocation for a given variable may take place statically, that is, before the execution of the program, or dynamically, during execution. A variable that is allocated statically remains allocated for the duration of the program. A variable that is allocated dynamically will relinquish its storage either upon the termination of the block containing that variable or at the request of the programmer, depending upon its storage class.

---

[1] When multitasking is in operation, the program (i.e., the major task) is terminated when any task reaches a STOP statement. See Chapter 15, "Multitasking."

The manner in which storage is allocated for a variable is determined by the storage class of that variable. There are four storage classes: static, automatic, controlled, and based. Each storage class is specified by its corresponding storage class attribute: STATIC, AUTOMATIC, CONTROLLED, and BASED, respectively, The last three define dynamic storage allocation.

Storage class attributes may be declared explicitly for element, array, and major structure variables. If a variable is an array or a major structure variable, the storage class declared for that variable applies to all of the elements in the array or structure.

All variables that have not been explicitly declared with a storage class attribute are assumed to have the AUTOMATIC attribute, with one exception: any variable that has the EXTERNAL attribute is assumed to have the STATIC attribute.

Static Storage

All variables that have the STATIC attribute are allocated storage before the execution of the program begins and they remain allocated for the duration of the program. For example:

```
OUTP: PROCEDURE;
      DECLARE X FIXED STATIC INITIAL (1);
        .
        .
        .
      PUT DATA (X);
        .
        .
        .
      X = X+1;
      END OUTP;
```

Before the execution of a program begins, all static variables are allocated and any initial values specified for them are assigned. Therefore, in the above example, the first time that procedure OUTP is invoked, X has the value 1 and execution of the PUT statement causes the item X=1 to be written. Before OUTP is terminated, the assignment statement X=X+1 increases the value of X by 1. If OUTP is invoked a second time, and if the value of X is not changed elsewhere in the program, X has the value 2 (X is not re-initialized to 1 because static variables are initialized only once before execution). When the PUT statement is executed for the second time, the item X=2 is written into the stream, etc. Thus, the static variable X might be used to record the number of times that OUTP is invoked.

Automatic Storage

A variable that has the AUTOMATIC attribute is allocated storage upon activation of the block in which that variable is declared. The variable remains allocated as long as the block remains active; it is freed when the block is terminated. Once a variable is freed, its value is lost.

Controlled Storage

A variable that has the CONTROLLED attribute is allocated storage only upon the execution of an ALLOCATE statement specifying that variable. Storage remains allocated for that variable until the execution of a FREE statement in which the variable is specified. This allocation remains even after termination of the block in which it is allocated. Thus, the allocation and freeing of storage for variables declared with the CONTROLLED attribute is directly under the control of the programmer.

A controlled variable may be stacked; that is, storage may be allocated for a controlled variable even when a previous allocation for that variable exists. In terms of ALLOCATE and FREE statements, stacking occurs when an allocated controlled variable is specified in an ALLOCATE statement without first having been specified in a FREE statement. When this occurs, the previous allocation is not released; its value remains the same but, for the time being, this value is not available to the programmer. Conceptually, the new allocation is stacked on top of the previous allocation, with the result that the previous allocation is "pushed-down" in the stack. Subsequent allocations are always added to the top of the stack.

Any reference to a stacked controlled variable always refers to the most recent allocation for that variable; i.e., to the allocation at the top of the stack. Thus, a FREE statement specifying a stacked controlled variable will cause the allocation at the top of the stack to be freed. When this occurs, the other allocations in the stack are "popped-up", the most recent previous allocation coming to the top and being available once again. When an allocation is popped up to the top of a stack, its value is the same as it was when it was pushed down.

Based Storage

Based storage is similar to controlled storage in that it can be allocated by the ALLOCATE statement and freed by the FREE statement; and more than one allocation can exist for one variable. However, the programmer has a much greater degree of control with based storage. For example, all current based allocations are available at any time: unique reference to a particular allocation is provided by a pointer value qualifying the based variable reference.

Based storage is the most powerful of the PL/I storage classes, but it must be used carefully; many of the safeguards against error that are provided for other storage classes cannot be provided for based.

For full details of based storage, see Chapter 14, "Based Storage and List Processing."

REACTIVATION OF AN ACTIVE PROCEDURE (RECURSION)

An active procedure that can be reactivated from within itself or from within another active procedure is said to be a recursive procedure; such reactivation is called recursion.

A procedure can be invoked recursively only if the RECURSIVE option has been specified in its PROCEDURE statement. This option also applies to the names of any secondary entry points that the procedure might have.

The environment (that is, values of automatic variables, etc.) of every invocation of a recursive procedure is preserved in a manner analogous to the stacking of allocations of a controlled variable. An environment can thus be thought of as being "pushed down" at a recursive invocation, and "popped up" at the termination of that invocation. Note that a label constant always contains information identifying the current invocation of the block that contains the label. Hence, if a label constant is assigned to a label variable in a particular invocation, a GO TO statement naming that variable in another invocation could restore the environment that existed when the assignment was performed.

Consider the following example:

```
RECURS: PROCEDURE RECURSIVE;
  DECLARE X STATIC EXTERNAL INITIAL (0);
    .
    .
    .
  X=X+1;
  PUT DATA (X);
  IF X =5 THEN GO TO LAB;
  CALL AGN;
  X =X-1;
  PUT DATA (X);
    .
    .
    .
  LAB: END RECURS;

AGN: PROCEDURE RECURSIVE;
  DECLARE X STATIC EXTERNAL INITIAL (0);
    .
    .
    .
  X=X+1;
  PUT DATA(X);
    .
    .
    .
  CALL RECURS;
  X=X-1;
  PUT DATA (X);
  END AGN;
```

In the above example, RECURS and AGN are both recursive procedures. Since X is static and has the INITIAL attribute, it is allocated and initialized before execution of the program begins.

The first time that RECURS is invoked, X is incremented by 1 and X=1 is transmitted by the PUT statement. Since X is less than 5, AGN is invoked. In AGN, X is incremented by 1 and X=2 is transmitted (also by a PUT statement). AGN then reinvokes RECURS.

This second invocation of RECURS is a recursive invocation, because RECURS is still active. X is incremented as before, and then X=3 is transmitted. X is still less than 5, so AGN is invoked again. Since AGN is active when invoked, this invocation of AGN is also recursive. X is incremented once again, X=4 is transmitted, and RECURS is invoked for the third time.

The third invocation of RECURS results in the transmission of X=5. But, since X is no longer less than 5, GO TO LAB is executed, and then RECURS is terminated. However, only the third invocation of RECURS is terminated, with the result that control returns to the procedure that invoked RECURS for the third time; that is, control returns to the statement following CALL RECURS in the second invocation of AGN. At this point X is decremented by 1 and X=4 is transmitted. Then the second

invocation of AGN is terminated, and control returns to the procedure that invoked AGN for the second time; that is, control returns to the statement following CALL AGN in the second invocation of RECURS. Here X is decremented again and X=3 is transmitted, after which the second invocation of RECURS is terminated and control returns to the first invocation of AGN. X is decremented again, X=2 is transmitted, the first invocation of AGN is terminated, and control returns to the first invocation of RECURS. X is decremented, X=1 is transmitted, X is reset to 0, and the first invocation of RECURS is terminated. Control then returns to the procedure that invoked RECURS in the first place.

Note the difference between recursive and reentrant procedures. A procedure is recursive only if the RECURSIVE option is specified in the PROCEDURE statement. Every procedure compiled by the F Compiler is reentrant; that is, it is a procedure that does not modify itself during its execution, so that subsequent execution of the procedure with the same data will always give the same result.

Effect of Recursion on Storage Classes

Allocation of static variables (as illustrated above) is not affected by recursion, because they are allocated storage outside the environment of a recursive procedure. Allocation of controlled variables is likewise unaffected because their allocation and release is completely under the control of the programmer. However, allocation of automatic variables is affected, because they are a part of the environment of a particular invocation and also because their allocation and release is not directly controlled by the programmer.

Each time a procedure is invoked recursively, storage for each automatic variable is reallocated, and the previous allocation is pushed down in a stack. Each time an activation of a recursive procedure is terminated, automatic storage is popped up to yield the next most recent generation of automatic storage. Hence, each generation of automatic storage is preserved as part of the environment of the corresponding recursive activation.

PROLOGUES AND EPILOGUES

Each time a block is activated, certain activities must be performed before control can reach the first executable statement in the block. This set of activities is called a prologue. Similarly, when a block is terminated, certain activities must be performed before control can be transferred out of the block; this set of activities is called an epilogue.

Prologues and epilogues are the responsibility of the compiler and not of the programmer. They are discussed here because knowledge of them may assist the programmer in improving the performance of his program.

Prologues

A prologue is a compiler-written routine logically appended to the beginning of a block and executed as the first step in the activation of a block. In general, activities performed by a prologue are as follows:

- Computing dimension bounds and string lengths for automatic and DEFINED variables and ENTRY declarations.
- Allocating storage for automatic variables and initialization, if specified.
- Determining which currently active blocks are known to the procedure, so that the correct generations of automatic storage are accessible, and the correct on-units may be entered.
- Allocating storage for dummy arguments that may be passed from this block.

The prologue may need to evaluate expressions defining lengths, bounds, iteration factors, and initial values. Note that if an item is referred to in an expression and the allocation or initialization of a second item depends on that expression, then the first item must be in no way dependent on the second item for its own allocation and initialization. Further, the first item must be in no way dependent on any other item that so depends on the second item. For example, the following declaration is invalid:

```
DCL A(B(1)) INITIAL(2),
    B(A(1)) INITIAL(3);
```

However, the following declaration is valid:

```
DCL N INITIAL(3),
    A(N),
    B CHAR(N);
```

Epiloques

An epilogue is a compiler-written routine logically appended to the end of a block and executed as the final step in the termination of a block. In general, the activities performed bv an epilogue are as follows:

- Re-establishing the on-unit environment existing before the block was activated.
- Releasing storage for all automatic variables allocated in the block.

A PL/I program consists of a collection of identifiers, constants, and special characters used as operators or delimiters. Identifiers themselves may be either keywords or names with a meaning specified by the programmer. The PL/I language is constructed so that the compiler can determine from context whether or not an identifier is a keyword, so there is no list of reserved words that must not be used for programmer-defined names. Any identifier may be used as a name; the only restriction is that at any point in a program a name can have one and only one meaning. For example, the same name cannot be used for both a file and a floating-point variable.

Note: The above is true so long as the 60-character set is used. Certain identifiers of the 48-character set cannot be used as programmer-defined identifiers in a program written using the 48-character set; these identifiers are: GT, GE, NE, LT, NG, LE, NL, CAT, OR, AND, NOT, and PT.

It is not necessary, however, for a name to have the same meaning throughout a program. A name declared within a block has a meaning only within that block. Outside the block it is unknown unless the same name has also been declared in the outer block. In this case, the name in the outer block refers to a different object. This enables programmers to specify local definitions and, hence, to write procedures or begin blocks without knowing all the names being used by other programmers writing other parts of the program.

Since it is possible for a name to have more than one meaning, it is important to define which part of the program a particular meaning applies to. In PL/I a name is given attributes and a meaning by a declaration (not necessarily explicit). The part of the program for which the meaning applies is called the scope of the declaration of that name. In most cases, the scope of a name is determined entirely by the position at which the name is declared within the program (or assumed to be declared if the declaration is not explicit). There are cases in which more than one generation of data may exist with the same name (such as in recursion); such cases are considered separately.

In order to understand the rules for the scope of a name, it is necessary to understand the terms "contained in" and "internal to."

Contained In:

> All of the text of a block, from the PROCEDURE or BEGIN statement through the corresponding END statement, is said to be contained in that block. Note, however, that the labels of the BEGIN or PROCEDURE statement heading the block, as well as the labels of any ENTRY statements that apply to the block, are not contained in that block. Nested blocks are contained in the block in which they appear.

Internal To:

> Text that is contained in a block, but not contained in any other block nested within it, is said to be internal to that block. Note that entry names of a procedure (and labels of a BEGIN statement) are not contained in that block. Consequently, they are internal to the containing block. Entry names of an external procedure are treated as if they were external to the external procedure.

In addition to these terms, the different types of declaration are important. The three different types -- explicit declaration, contextual declaration, and implicit declaration -- are discussed in the following sections.

EXPLICIT DECLARATION

A name is explicitly declared if it appears:

1. In a DECLARE statement
2. In a parameter list
3. As a statement label
4. As a label of a PROCEDURE or ENTRY statement

The appearance of a name in a parameter list is the same as if a DECLARE statement for that name appeared immediately following the PROCEDURE or ENTRY statement in which the parameter list occurs (though the same name may also appear in a DECLARE statement internal to the same block).

The appearance of a name as the label of either a PROCEDURE or ENTRY statement is

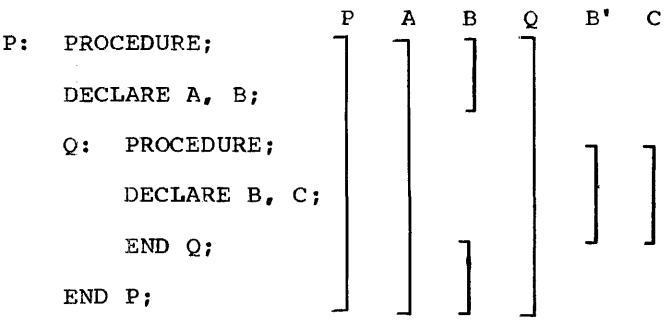
the same as if it were declared in a DECLARE statement immediately preceding the PROCEDURE statement for the procedure to which it refers.

The appearance of a statement label prefix constitutes explicit declaration equivalent to the declaration of a variable in a DECLARE statement internal to the same block as the statement to which it applies.

SCOPE OF AN EXPLICIT DECLARATION

The scope of an explicit declaration of a name is that block to which the declaration is internal, but excluding all contained blocks to which another explicit declaration of the same identifier is internal.

For example:



The lines to the right indicate the scope of the names. B and B' indicate the two distinct uses of the name B.

CONTEXTUAL DECLARATION

When a name appears in certain contexts, some of its attributes can be determined without explicit declaration. In such a case, if the appearance of a name does not lie within the scope of an explicit declaration for the same name, the name is said to be contextually declared.

A name that has not been declared explicitly will be recognized and declared contextually in the following cases:

1. A name that appears in a CALL statement, in a CALL option, or followed by a parenthesized list in a function reference (in a context where an expression is expected) is given the ENTRY and EXTERNAL attributes.

2. A name that appears in a FILE option, or a name that appears in an ON, SIGNAL, or REVERT statement for a condition that requires a file name, is given the FILE and EXTERNAL attributes.

3. A name that appears in an ON CONDITION, SIGNAL CONDITION, or REVERT CONDITION statement is recognized as a programmer-defined condition name.

4. A name that appears in an EVENT option or in a WAIT statement is given the EVENT attribute.

5. A name that appears in a TASK option is given the TASK attribute.

6. A name that appears in the BASED attribute, in a SET option, or on the left-hand side of a pointer qualification symbol is given the POINTER attribute.

7. A name that appears in an IN option, or in the OFFSET attribute is given the AREA attribute. Note, however, that all contextually declared area variables are given the AUTOMATIC attribute. The F Compiler implementation requires that the variable named in the OFFSET attribute must be based; if a nonbased area variable is named, the offset variable will be changed to a pointer variable. Hence, unless the variable named in the OFFSET attribute is explicitly declared, OFFSET effectively becomes POINTER, and a severe error occurs.

8. If an undeclared identifier appears:

   a. before the equal sign in an assignment statement, or

   b. before the assignment symbol in a DO statement (or in a repetitive specification), or

   c. in the data list of a GET statement

   and if that identifier is neither enclosed within an argument list nor immediately followed by an argument list, that identifier is contextually declared to be a variable and not a reference to a built-in function or pseudo-variable. This rule does not apply to the identifiers ONCHAR, ONSOURCE, and PRIORITY.

Examples of contextual declaration are:

```
READ FILE (PREQ) INTO (Q);

ON CONDITION (NEG) CALL CREDIT;
```

In these statements, PREQ is given the FILE attribute, NEG is recognized as a programmer-defined condition name, and CREDIT is given the ENTRY attribute. The EXTERNAL attribute is given to all three by default.

SCOPE OF A CONTEXTUAL DECLARATION

The scope of a contextual declaration is determined as if the declaration were made in a DECLARE statement immediately following the PROCEDURE statement of the external procedure in which the name appears.

Note that contextual declaration has the same effect as if the name were declared in the external procedure, even when the statement that causes the contextual declarations is internal to a block (called B, for example) that is contained in the external procedure. Consequently, the name is known throughout the entire external procedure, except for any blocks in which the name is explicitly declared. It is as if block B has inherited the declaration from the containing external procedure.

Since a contextual declaration cannot exist within the scope of an explicit declaration, it is impossible for the context of a name to add to the attributes established for that name in an explicit declaration.

For example, the following procedure is invalid:

```
P:  PROC (F);
      .
      .
      .
    READ FILE(F) INTO(X);
      .
      .
      .
    END P;
```

The identifier F is in a parameter list and is, therefore, explicitly declared. It is given the attributes REAL DECIMAL FLOAT by default. Since F is explicitly declared, its appearance in the FILE option does not constitute a contextual declaration. Such use of the identifier is in error.

IMPLICIT DECLARATION

If a name appears in a program and is not explicitly or contextually declared, it is said to be implicitly declared. The scope of an implicit declaration is determined as if the name were declared in a DECLARE statement immediately following the first PROCEDURE statement of the external procedure in which the name is used.

An implicit declaration causes default attributes to be applied, depending upon the first letter of the name. If the name begins with any of the letters I through N it is given the attributes REAL FIXED BINARY (15,0). If the name begins with any other letter including one of the alphabetic extenders $, #, or @, it is given the attributes REAL FLOAT DECIMAL (6). (The default precisions are those defined for System/360 implementations.)

EXAMPLES OF DECLARATIONS

Scopes of data declarations are illustrated in Figure 7-1. The brackets to the left indicate the block structure; the brackets to the right show the scope of each declaration of a name. In the diagram, the scopes of the two declarations of Q and R are shown as Q and Q' and R and R'.

P is declared in the block A and known throughout A since it is not redeclared.

Q is declared in A, and redeclared in B. The scope of the first declaration is all of A except B; the scope of the second declaration is block B only.

R is declared in block C, but a reference to R is also made in block B. The reference to R in block B results in an implicit declaration of R in A,the external procedure. Two separate names with different scopes exist, therefore. The scope of the explicitly declared R is C; the scope of the implicitly declared R is all of A except block C.

I is referred to in block C. This results in an implicit declaration in the external procedure A. As a result, this declaration applies to all of A, including the contained procedures B, C and D.

S is explicitly declared in procedure D and is known only within D.

Scopes of entry name and statement label declarations are illustrated in Figure 7-2. The example shows two external procedures. The names of these procedures, A and E, are assumed to be explicitly declared with the EXTERNAL attribute within the procedures to which they apply. In addition, E is contextually declared in A as an EXTERNAL entry name by its appearance in the CALL statement in block C. The contextual dec-

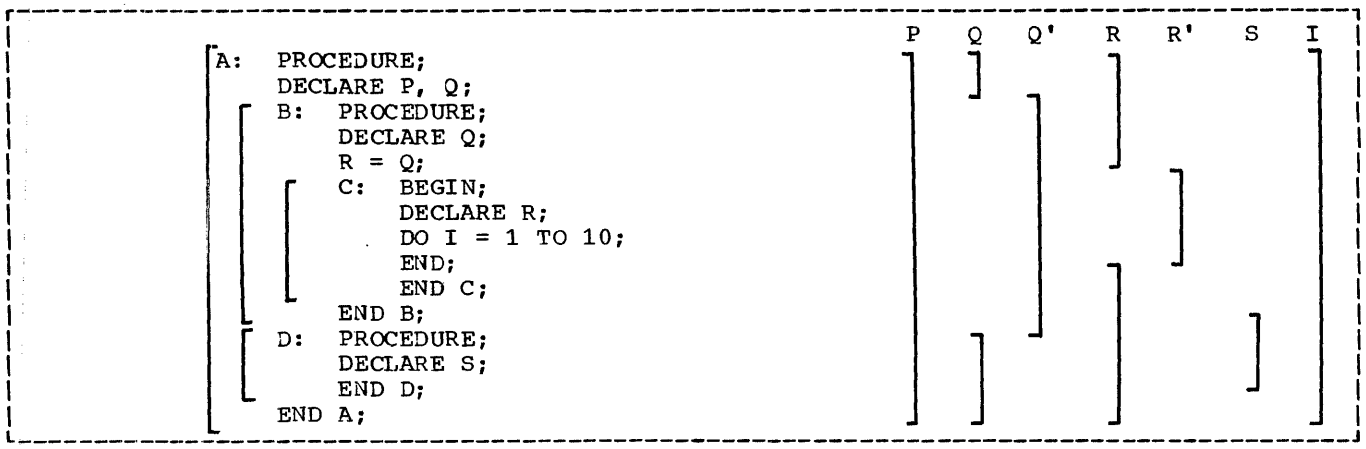


Figure 7-1. Scopes of Data Declarations

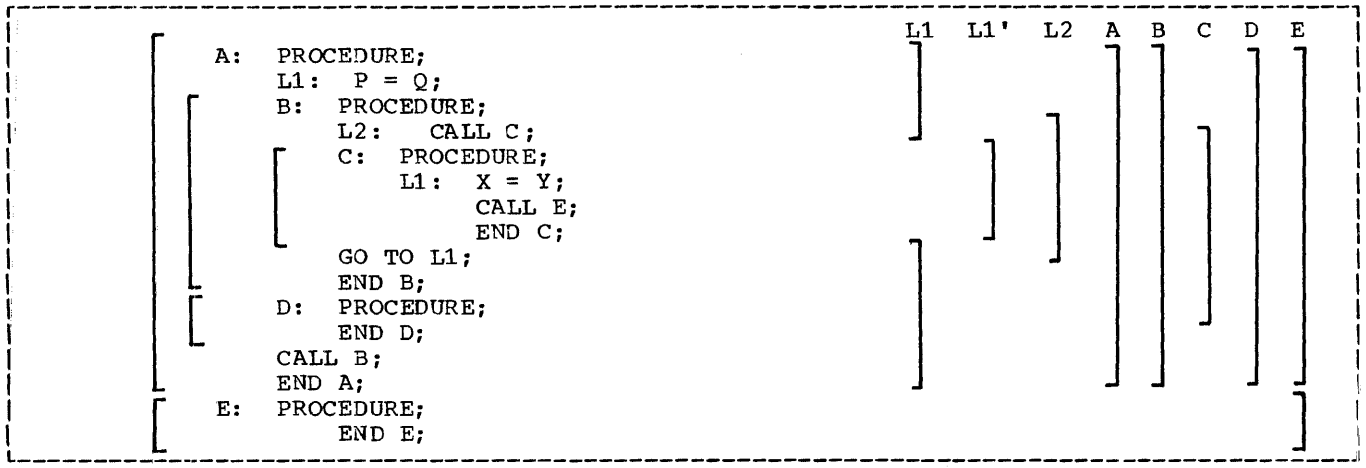


Figure 7-2. Scopes of Entry and Label Declarations

laration of E applies throughout block A and is linked to the explicit declaration of E that applies throughout block E. The scope of the name E is all of block A and all of block E. The scope of the name A is only all of the block A, and not E. However, it could appear in a CALL statement in E, since the CALL statement itself would provide a contextual declaration of A, which would then result in the scope of A being all of A and all of E.

The label L1 appears with statements internal to A and to C. Two separate declarations are therefore established; the first applies to all of block A except block C, the second applies to block C only. Therefore, when the GO TO statement in block B is executed, control is transferred to L1 in block A, and block B is terminated.

D and B are explicitly declared in block A and can be referred to anywhere within A; but since they are INTERNAL, they cannot be referred to in block E (unless passed as an argument to E).

C is explicitly declared in B and can be referred to from within B, but not from outside B.

L2 is declared in B and can be referred to in block B, including C, which is contained in B, but not from outside B.

APPLICATION OF DEFAULT ATTRIBUTES

The attributes associated with a name comprise those explicitly, contextually, or implicitly declared for that name, as well as those assumed by default. The default for each attribute is given in Part II, Section I, "Attributes."

THE INTERNAL AND EXTERNAL ATTRIBUTES

The scope of a name with the INTERNAL attribute is the same as the scope of its declaration. Any other explicit declaration of that name refers to a new object with a different, non-overlapping scope.

A name with the EXTERNAL attribute may be declared more than once in the same program, either in different external procedures or within blocks contained in external procedures. Each declaration of the name establishes a scope. These declarations are linked together and, within a program, all declarations of the same identifier with the EXTERNAL attribute refer to the same name. The scope of the name is the sum of the scopes of all the declarations of that name within the program.

Note: External names cannot be more than seven characters long for System/360 implementations.

Since these declarations all refer to the same thing, they must all result in the same set of attributes. It may be impossible for the compiler to check this, particularly if the names are declared in different procedures, so care should be taken to ensure that different declarations of the same name with the EXTERNAL attribute do have matching attributes. The attribute listing, which is available as optional output from the F Compiler, helps to check the use of names. The following example illustrates the above points in a program:

```
A: PROCEDURE;
   DECLARE S CHARACTER (20);
   CALL SET (3);
E: GET LIST (S,M,N);
   B:  BEGIN;
       DECLARE X(M,N), Y(M);
       GET LIST (X,Y);
       CALL C(X,Y);
       C:  PROCEDURE (P,Q);
           DECLARE P(*,*), Q(*),
             S BINARY FIXED EXTERNAL;
           S = 0;
           DO I = 1 TO M;
           IF SUM (P(I,*)) = Q(I)
              THEN GO TO B;
           S = S+1;
           IF S = 3 THEN CALL OUT (E);
           CALL D(I);
       B:  END;
           END C;
       D:  PROCEDURE (N);
           PUT LIST ('ERROR IN ROW ',
             N, 'TABLE NAME ', S);
           END D;
       END B;
   GO TO E;
   END A;
```

```
OUT:  PROCEDURE (R);
      DECLARE R LABEL,
        (M,L) STATIC INTERNAL
          INITIAL (0),
        S BINARY FIXED EXTERNAL,
        Z FIXED DECIMAL(1);
      M = M+1; S=0;
      IF M<L THEN STOP; ELSE GO TO R;
SET:  ENTRY (Z);
      L=Z;
      RETURN;
      END OUT;
```

A is an external procedure name; its scope is all of block A, plus any other blocks where A is declared (explicitly or contextually) as external.

S is explicitly declared in block A and block C. The character string declaration applies to all of block A except block C; the fixed binary declaration applies only within block C. Notice that although D is called from within block C, the reference to S in the PUT statement in D is to the character string S, and not to the S declared in block C.

N appears as a parameter in block D, but is also used outside the block. Its apearance as a parameter establishes an explicit declaration of N within D; the references outside D cause an implicit declaration of N in block A. These two declarations of the name N refer to different objects, although in this case, the objects have the same data attributes, which are, by default, FIXED (15,0), BINARY, and INTERNAL.

X and Y are known throughout B and could be referred to in block C or D within B, but not in that part of A outside B.

P and Q are parameters, and therefore their appearance in the parameter list is sufficient to constitute an explicit declaration. However, a separate DECLARE statement is required in order to specify that P and Q are arrays. Note that although the arguments X and Y are declared as arrays and are known in block C, it is still necessary to declare P and Q in a DECLARE statement to establish that they, too, are arrays. (The asterisk notation indicates that the bounds of the parameters are the same as the bounds of the arguments.)

I and M are not explicitly declared in the external procedure A; they are therefore implicitly declared and are known throughout A, even though I appears only within block C.

Within the external procedure A, OUT and SET are contextually declared as entry names, since they follow the keyword CALL.

They are therefore considered to be declared in A and are given the EXTERNAL attribute by default.

The second external procedure in the example has two entry names, SET and OUT. These are considered to be explicitly declared with the EXTERNAL attribute. The two entry names SET and OUT are therefore known throughout the two procedures.

The label B appears twice in the program, once as the label of a begin block, which is an explicit declaration, as a label in A. It is redeclared as a label within block C by its appearance as a prefix to the END statement. The reference to B in the GO TO statement within block C therefore refers to the label of the END statement within block C. Outside block C, any reference to B would be to the label of the begin block.

Note that C and D can be called from any point within B but not from that part of A outside B, nor from another external procedure. Similarly, since E is known throughout the external procedure A, a transfer to E may be made from any point within A. The label B within block C, however, can only be referred to from within C. Transfers out of a block by a GO TO statement can be made; but such transfers into a nested block generally cannot. An exception is shown in the external procedure OUT, where the label E from block A is passed as an argument to the label parameter R.

The statement GO TO R causes control to pass to the label E, even though E is declared within A, and not known within OUT.

The variables M and L are declared within the block OUT to be STATIC, so their values are preserved between calls to OUT.

In order to identify the S in the procedure OUT as the same S in the procedure C, both have been declared with the attribute EXTERNAL.

MULTIPLE DECLARATIONS AND AMBIGUOUS REFERENCES

Two or more declarations of the same identifier internal to the same block constitute a multiple declaration, unless at least one of the identifiers is declared within a structure in such a way that name qualification can be used to make the names unique.

Two or more declarations anywhere in a program of the same identifier as different names with the EXTERNAL attribute constitute a multiple declaration.

Multiple declarations are in error.

A name need have only enough qualification to make the name unique. Reference to a name is always taken to apply to the identifier declared in the innermost block containing the reference. An ambiguous reference is a name with insufficient qualification to make the name unique.

The following examples illustrate both multiple declarations and ambiguous references:

```
DECLARE 1 A, 2 C, 2 D, 3 E;
   BEGIN;
   DECLARE 1 A, 2 B, 3 C, 3 E;
   A.C = D.E;
```

In this example, A.C refers to C in the inner block; D.E refers to E in the outer block.

```
DECLARE 1 A, 2 B, 2 B, 2 C, 3 D, 2 D;
```

In this example, 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.

```
DECLARE 1 A, 2 B, 3 C, 2 D, 3 C;
```

In this example, A.C is ambiguous because neither C is completely qualified by this reference.

```
DECLARE 1 A, 2 A, 3 A;
```

In this example, A refers to the first A, A.A refers to the second A, and A.A.A refers to the third A.

```
DECLARE X;

DECLARE 1 Y, 2 X, 3 Z, 3 A,
             2 Y, 3 Z, 3 A;
```

In this example, X refers to the first DECLARE statement. A reference to Y.Z is ambiguous; Y.Y.Z refers to the second Z; and Y.X.Z refers to the first Z.

PL/I provides input and output statements that enable data to be transmitted between the internal and external storage devices of a computer. A collection of data external to a program is called a data set. Transmission of data from a data set to a program is called input, and transmission of data from a program to a data set is called output.

Data sets are stored on a variety of external storage media, such as punched cards, reels of magnetic tape, magnetic disks, magnetic drums, and punched paper tape. Despite their variety, external storage media have many common characteristics that permit standard methods of collecting, storing, and transmitting data. For convenience, thus, the general term volume is used to refer to a unit of external storage, such as a reel of magnetic tape or a disk pack, without regard to its specific physical composition.

The data items within a data set are arranged in distinct physical groupings called blocks. These blocks allow the data set to be transmitted and processed in portions rather than as a unit. For processing purposes, each block may consist of one or more logical subdivisions called records, each of which contains one or more data items.

A block is also called a physical record, because it is the unit of data that is physically transmitted to and from a volume. To avoid confusion between a physical record and its logical subdivisions, the logical subdivisions are called logical records.

When a block contains two or more records, the records are said to be blocked. Blocked records often permit more compact and efficient use of storage. Consider how data is stored on magnetic tape: the data between two successive interrecord gaps is one block, or physical record. If several logical records are contained within one block, the number of interblock gaps is reduced, and much more data can be stored on a full length of tape. For example, on a tape of density 800 characters/inch with an interrecord gap of 0.6 inches, a card image of 80 characters would take up 0.1 inches. If the records were unblocked, each record would require 0.1 inches, plus 0.6 inches for the interrecord gap, making a total of 0.7 inches. 100 records would therefore take up 70 inches of tape. If the records were blocked, however, at, say, 10 records to a block, each block of 10 records would take up 1 inch, plus 0.6 inches for the gap, making a total of 1.6 inches. Thus, 100 records would now take up only 16 inches of tape; this is less than 25 percent of the amount needed for unblocked records.

Most data processing applications are concerned with logical records rather than physical records. Therefore, the input and output statements of PL/I generally refer to logical records; this allows the programmer to concentrate on the data to be processed, without being directly concerned about its physical organization in external storage.

TYPES OF DATA TRANSMISSION

Two different types of data transmission can be used by a PL/I program, stream-oriented transmission and record-oriented transmission.

In stream-oriented transmission, the data in the data set is considered to be a continuous stream of data items in character form. Consequently, data conversion is implied in stream transmission, from character form to internal form on input, and from internal form to character form on output. The GET and PUT statements are the data transmission statements used in stream-oriented transmission. Variables, to which input data items are assigned, and expressions, from which output data items are transmitted, are generally specified in a data list with each GET or PUT statement.

Although data in the data set exists in record format, either unblocked or blocked, in stream transmission such organization is ignored within the program, and the data is treated as though it actually were a continuous stream of individual data items.

In record-oriented transmission, data in the data set is considered to be a collection of discrete logical records, recorded in any format acceptable to the computer. No data conversion is performed during record transmission; on input it is transmitted exactly as it is recorded in the data set; on output it is transmitted exactly as it is recorded internally.

The READ, REWRITE, LOCATE, and WRITE statements cause a single logical record to be transmitted to or from a data variable.

Note that although records may be blocked, in which case the physical record actually is transmitted to or from the data set as an entity, each data transmission statement in record I/O is concerned with a logical record. Blocked records are unblocked automatically.

The following discussion of files and file attributes should be of particular interest to a programmer using record-oriented transmission. File handling is simpler when using stream-oriented transmission, and, as can be noted, fewer attributes are applicable to stream files.

FILES

To allow a source program to deal primarily with the logical aspects of data rather than with its physical organization in a data set, PL/I employs a symbolic representation of a data set called a file. This symbolic representation determines how input and output statements access and process the associated data set. Unlike a data set, however, a file has significance only within the source program and does not exist as a physical entity external to the program.

PL/I requires a file name to be declared for a file and allows the characteristics of a file to be described with keywords called file attributes, which are specified for the file name.

FILE ATTRIBUTES

The following lists show file attributes that are applicable to each type of data transmission:

Stream Transmission
FILE
STREAM
INPUT
OUTPUT
PRINT
INTERNAL
EXTERNAL
ENVIRONMENT

Record Transmission
FILE
RECORD
INPUT
OUTPUT
UPDATE
INTERNAL
EXTERNAL
ENVIRONMENT
SEQUENTIAL
DIRECT
BUFFERED
UNBUFFERED
KEYED
BACKWARDS
EXCLUSIVE

A detailed description of each of these attributes appears in Part II, Section I, "Attributes." The discussions below give a brief description of each attribute and show how attributes are declared for a file.

The FILE Attribute

The FILE attribute indicates that the associated identifier is a file name. For example, the identifier MASTER is declared to be a file name in the following statement:

```
DECLARE MASTER FILE;
```

Alternative and Additive Attributes

The attributes associated with the FILE attribute fall into two categories: alternative attributes and additive attributes. An alternative attribute is one that is chosen from a group of attributes. If no explicit or implicit declaration is given for one of the alternative attributes in a group and if one of the alternatives is required, a default attribute is assumed.

An additive attribute is one that must be stated explicitly or is implied by another explicitly stated attribute or name. The additive attribute KEYED can be implied by the DIRECT attribute. The additive attribute PRINT can be implied by the standard output file name SYSPRINT. An additive attribute can never be applied by default.

Note: With the exception of the INTERNAL and EXTERNAL scope attributes, all the alternative and additive attributes imply the FILE attribute. Therefore, the FILE attribute need not be specified for a file that has at least one of the alternative or

additive attributes already specified explicitly. The FILE attribute must be specified explicitly, however, if only the INTERNAL or EXTERNAL attribute is specified; otherwise, the identifier will be assumed, by default, to be an arithmetic variable rather than a file name.

Alternative Attributes

PL/I provides five groups of alternative file attributes. Each group is discussed individually. Following is a list of the groups and the default for each:

Group Type	Alternative Attributes	Default Attribute
Usage	STREAM|RECORD	STREAM
Function	INPUT|OUTPUT|UPDATE	INPUT
Access	SEQUENTIAL|DIRECT	SEQUENTIAL
Buffering	BUFFERED|UNBUFFERED	BUFFERED
Scope	EXTERNAL|INTERNAL	EXTERNAL

The STREAM and RECORD Attributes

The STREAM and RECORD attributes describe the type of data transmission (stream-oriented or record-oriented) to be used in input and output operations for the file.

The STREAM attribute causes a file to be treated as a continuous stream of data items recorded only in character form.

The RECORD attribute causes a file to be treated as a sequence of logical records, each record consisting of one or more data items recorded in any internal form acceptable to the implementation.

```
DECLARE MASTER FILE RECORD,
        DETAIL FILE STREAM;
```

The INPUT, OUTPUT, and UPDATE Attributes

The function attributes determine the direction of data transmission permitted for a file. The INPUT attribute applies to files that are to be read only. The OUTPUT attribute applies to files that are to be created, and hence are to be written only. The UPDATE attribute describes a file that is to be used for both input and output; it allows records to be inserted into an existing file and other records already in that file to be altered or deleted.

```
DECLARE
   DETAIL FILE INPUT,
   REPORT FILE OUTPUT,
   MASTER FILE UPDATE;
```

The SEQUENTIAL and DIRECT Attributes

The access attributes apply only to a file with the RECORD attribute and describe how the records in the file are to be accessed.

The SEQUENTIAL attribute normally specifies that successive records in the file are to be accessed on the basis of their successive physical positions, such as they are on magnetic tape.

The DIRECT attribute specifies that a record in a file is to be accessed on the basis of its location in the file and not on the basis of its position relative to the record previously read or written. The location of the record is determined by a key; therefore, the DIRECT attribute implies the KEYED attribute. The associated data set must be in a direct-access volume.

The BUFFERED and UNBUFFERED Attributes

The buffering attributes apply only to a file that has the SEQUENTIAL and RECORD attributes. The BUFFERED attribute indicates that logical records transmitted to and from a file must pass through an intermediate internal-storage area. The size of a buffer usually corresponds to the size of the blocks (physical records) in the data set associated with the file (a discussion of block size and buffer allocation appears in this chapter in "ENVIRONMENT Attribute"). The use of buffers may help speed up processing by allowing an overlap of transmission and computing time. It further allows the automatic blocking and unblocking of records.

The UNBUFFERED attribute indicates that a logical record in a data set need not pass through a buffer but may be transmitted directly to and from the internal storage associated with a variable. The logical records and physical records are generally the same size in a data set that is associated with an UNBUFFFRED file.

Note: Specification of UNBUFFERED does not preclude the use of buffers. In some cases, "hidden buffers" are required. Those cases are listed in the discussion of the BUFFERED and UNBUFFERED attributes in Part II, Section I, "Attributes."

Additive Attributes

The additive attributes are:

PRINT

BACKWARDS

KEYED

EXCLUSIVE

ENVIRONMENT (option-list)

The PRINT Attribute

The PRINT attribute applies only to files with the STREAM and OUTPUT attributes. It indicates that the file is eventually to be printed, that is, the data associated with the file is to appear on printed pages, although it may first be written on some other medium. The PRINT attribute causes the associated record to be created with the initial byte reserved for a printer control character.

The BACKWARDS Attribute

The BACKWARDS attribute applies only to files with the SEQUENTIAL, RECORD, and INPUT attributes and only to data sets on magnetic tape. It indicates that a file is to be accessed in reverse order, beginning with the last record and proceeding through the file until the first record is accessed.

The KEYED Attribute

The KEYED attribute indicates that records in the file are to be accessed using one of the key options (KEY, KEYTO, or KEYFROM) of data transmission statements or of the DELETE statement. Note that the KEYED attribute does not necessarily indicate that the actual keys exist or are to be written in the data set. Consequently, it need not be specified unless one of the key options is to be used, even if keys actually exist in the associated data set. The STREAM and PRINT attributes cannot be applied to a file that has the KEYED attribute. The use of keys is discussed in detail in the sections "Environmental Considerations for Data Sets" and "Record-Oriented Transmission" in this chapter.

The EXCLUSIVE Attribute

The EXCLUSIVE attribute applies only to files with the RECORD, DIRECT, and UPDATE attributes. It specifies that any record in the file may be automatically locked by a task while it is operating on that record, to prevent interference by another concurrent task. It can be suppressed by the NOLOCK option on the READ statement.

For detailed information on the effects of operations on EXCLUSIVE files, see "The EXCLUSIVE Attribute," in Chapter 14.

The ENVIRONMENT Attribute

The ENVIRONMENT attribute specifies information about the physical organization of the data set associated with a file. These characteristics are indicated in a parenthesized option list in the ENVIRONMENT attribute specification and are dependent upon the implementation. The option list for the F Compiler is discussed in "Environmental Considerations for Data Sets."

OPENING AND CLOSING FILES

Before the data associated with a file can be transmitted by input or output statements, certain file preparation activities must occur, such as checking for the availability of external storage media, positioning the medium, and allocating appropriate programming support. Such activity is known as opening a file. Also, when processing is completed, the file must be closed. Closing a file involves releasing the facilities that were established during the opening of the file.

PL/I provides two statements, OPEN and CLOSE, to perform these functions. These statements, however, are optional. If an OPEN statement is not executed for a file, the file is opened automatically when the first data transmission statement for that

file is executed; in this case, the automatic file preparation consists of standard system procedures that use information about the file as specified in a DECLARE statement (or assumed from a contextual declaration). Similarly, if a file has not been closed before completion of a program, the file is closed automatically upon normal completion of the program.

The following discussions show the effect of OPEN and CLOSE statements upon files specified in these statements.

The OPEN Statement

Execution of an OPEN statement causes one or more files to be opened explicitly. The OPEN statement has the following basic format:

```
OPEN FILE(file-name) [option-list]
  [,FILE(file-name) [option-list]]...;
```

The option list of the OPEN statement can specify any of the alternative and additive attributes, except the INTERNAL, EXTERNAL, and ENVIRONMENT attributes. Attributes included as options in the OPEN statement are merged with those stated in a DECLARE statement. The same attributes need not be listed in both an OPEN statement and a DECLARE statement for the same file, and, of course, there can be no conflict. Other options that can appear in the OPEN statement are the TITLE option, used to associate the file name with the data set, and the PAGESIZE and LINESIZE options, used to specify layout of a data set. All of these options are discussed later in this chapter. The option list may precede the FILE (file name) specification.

For the F Compiler, the OPEN statement is executed by library routines that are loaded dynamically at the time the OPEN statement is executed. Consequently, execution time can be reduced if more than one file is specified in the same OPEN statement, since the routines need be loaded only once, regardless of the number of files being opened. Note, however, that such multiple opening may require considerably more storage than might otherwise be needed.

For a file to be opened explicitly, the OPEN statement must be executed before any of the input and output statements listed below in "Implicit Opening" are executed for the same file.

Implicit Opening

An implicit opening of a file occurs when one of the statements listed below is executed for a file for which an OPEN statement has not already been executed. The statement type determines which unspecified alternatives are applied to the file when it is opened.

The following list contains the statement identifiers and the attributes deduced from each:

Statement Identifier	Attributes Deduced
GET	STREAM, INPUT
PUT	STREAM, OUTPUT
READ	RECORD, INPUT (see Note)
WRITE	RECORD, OUTPUT (see Note)
LOCATE	RECORD, OUTPUT, SEQUENTIAL, BUFFERED
REWRITE	RECORD, UPDATE
DELETE	RECORD, UPDATE
UNLOCK	RECORD, DIRECT, UPDATE, EXCLUSIVE

An implicit opening caused by one of the above statements is equivalent to preceding the statement with an OPEN statement that specifies the deduced attributes.

Note: INPUT and OUTPUT are deduced from READ and WRITE only if UPDATE has not been explicitly declared.

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 the default attributes discussed earlier. 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 merged attributes and attributes that each implies after merging:

Merged Attributes	Implied Attributes
UPDATE	RECORD
SEQUENTIAL	RECORD
DIRECT	RECORD, KEYED
BUFFERED	RECORD, SEQUENTIAL
UNBUFFERED	RECORD, SEQUENTIAL
PRINT	OUTPUT, STREAM
BACKWARDS	RECORD, SEQUENTIAL, INPUT
KEYED	RECORD
EXCLUSIVE	RECORD, KEYED, DIRECT, UPDATE

The following two examples illustrate attribute merging for an explicit opening and for an implicit opening.

Explicit opening:

```
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.

Note: The attributes SEQUENTIAL or DIRECT and BUFFERED or UNBUFFERED do not apply to a file with the STREAM attribute.

Implicit opening:

```
DECLARE MASTER FILE KEYED INTERNAL;
        .
        .
        .
READ FILE (MASTER) INTO
   (MASTER RECORD) KEYTO(MASTER_KEY);
```

Attributes after merge due to the opening caused by execution of the READ statement are KEYED, INTERNAL, RECORD, and INPUT. Attributes after implication are KEYED, INTERNAL, RECORD, and INPUT. There are no additional attributes implied. Attributes after default application are KEYED, INTERNAL, RECORD, INPUT, SEQUENTIAL, and BUFFERED.

Associating Data Sets with Files

With the System/360 Operating System, the association of a file with a specific data set is accomplished using job control language, outside the PL/I program. At the time a file is opened, the PL/I file name is associated with the name (ddname) of a data definition statement (DD statement), which is, in turn, associated with the name of a specific data set (dsname). Note that the direct association is with the name of a DD statement, not with the name of the data set itself.

A ddname can be associated with a PL/I file either through the file name or through the character-string value of the expression in the TITLE option of the associated OPEN statement.

If a file is opened implicitly, or if no TITLE option is included in the OPEN statement that causes explicit opening of the file, the ddname is assumed to be the same as the file name. If the file name is longer than eight characters, the ddname is assumed to be composed of the first eight characters of the file name.

Note: Since external names are limited to seven characters for the F Compiler, an external file name of more than seven characters is shortened into a concatenation of the first four and the last three characters of the file name. Such a shortened name is not, however, the name used as the ddname in the associated DD statement.

Consider the following statements:

1. OPEN FILE(MASTER);
2. OPEN FILE(OLDMASTER);
3. READ FILE(DETAIL)...;

When statement number 1 is executed, the file name MASTER is taken to be the same as the ddname of a DD statement in the current job step. When statement number 2 is executed, the name OLDMASTE is taken to be the same as the ddname of a DD statement in the current job step. (The first eight characters of a file name form the ddname. Note, however, that if OLDMASTER is an external name, it will be shortened by the compiler to OLDMTER for use within the program.) If statement number 3 causes implicit opening of the file DETAIL, the name DETAIL is taken to be the same as the

ddname of a DD statement in the current job step.

In each of the above cases, a corresponding DD statement must appear in the job stream; otherwise, the UNDEFINEDFILE condition would be raised. The three DD statements would appear, in part, as follows:

```
1.  //MASTER   DD  DSNAME=...

2.  //OLDMASTE DD DSNAME=...

3.  //DETAIL   DD  DSNAME=,...
```

If a file is opened explicitly by an OPEN statement that includes a TITLE option, the ddname is taken from the TITLE option, and the file name is not used outside the program. The TITLE option appears in an OPEN statement as shown in the following format:

```
OPEN FILE(file-name) TITLE(expression);
```

The expression in the TITLE option is evaluated and converted to a character string, if necessary, that is assumed to be the ddname identifying the appropriate data set. If the character string is longer than eight characters, only the first eight characters are used. The following OPEN statement illustrates how the TITLE option might be used:

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

If this statement were executed, there must be a DD statement in the current job step in the job stream with DETAIL1 as its ddname. It might appear, in part, as follows:

```
//DETAIL1  DD  DSNAME=DETAILA,...
```

Thus, the data set DETAILA is associated with the file DETAIL through the ddname DETAIL1.

Although a data set name represents a specific collection of data, the file name can, at different times, represent entirely different data sets. Using the above example of the OPEN statement, whatever data set is named in the DSNAME parameter of the DETAIL1 DD statement is the one that is associated with DETAIL at the time it is opened.

Use of the TITLE option allows a programmer to choose dynamically, at open time, one among several data sets to be associated with a particular file name. Consider the following example:

```
DECLARE 1 INREC, 2 FIELD_1...,
            2 FILE_IDENT CHARACTER(8),
          DETAIL FILE INPUT...,
          MASTER FILE INPUT...;

OPEN FILE(DETAIL);

READ FILE(DETAIL) INTO (INREC);

OPEN FILE(MASTER) TITLE(FILE_IDENT);
```

Assume that the program containing these statements is used to process several different detail data sets, each of which has a different corresponding master data set. Assume, further, that the first record of each detail data set contains, as its last data item, a character string that identifies the appropriate master data set. The following DD statements might appear in the current job step:

```
//DETAIL    DD DSNAME=...

//MASTER1A  DD DSNAME=MASTER1A...

//MASTER1B  DD DSNAME=MASTER1B...

//MASTER1C  DD DSNAME=MASTER1C...
```

In this case, MASTER1A, MASTER1B, and MASTER1C represent three different master files. The first record of DETAIL would contain as its last item, either 'MASTER1A', 'MASTER1B', or 'MASTER1C', which is assigned to the character-string variable FILE_IDENT. When the OPEN statement is executed to open the file MASTER, the current value of FILE_IDENT would be taken to be the ddname, and the appropriate data set identified by that ddname would be associated with the file name MASTER.

Another similar use of the TITLE option is illustrated in the following statements:

```
DCL IDENT(3) CHAR(1)
    INITIAL('A', 'B', 'C');
DO  I = 1 TO 3;
    OPEN FILE(MASTER)
         TITLE('MASTER1'||IDENT(I));
     .
     .
     .
    CLOSE FILE(MASTER);
END;
```

In this example, IDENT is declared as a character-string array with three elements having as values the specific character strings 'A', 'B', and 'C'. When MASTER is opened during the first iteration of the DO-group, the character constant 'MASTER1' is concatenated with the value of the first element of IDENT, and the associated ddname is taken to be MASTER1A. After processing, the file is closed, dissociating the file name and the ddname. During the second

iteration of the group, MASTER is opened again. This time, however, the value of the second element of IDENT is taken, and MASTER is associated with the ddname MASTER1B. Similarly, during the final iteration of the group, MASTER is associated with the ddname MASTER1C.

Note: The character set of the job control language does not contain the break character (_). Consequently, this character cannot appear in ddnames. Care should thus be taken to avoid using break characters among the first eight characters of file names, unless the file is to be opened with a TITLE option with a valid ddname as its expression. The alphabetic extender characters $, @, and #, however, are valid for ddnames.

The CLOSE Statement

The basic form of the CLOSE statement is:

```
CLOSE FILE (file-name)
      [,FILE (file-name)]...;
```

Executing a CLOSE statement dissociates the specified file from the data set with which it became associated when the file was opened. The CLOSE statement also dissociates from the file all attributes established for it by the implicit or explicit opening process. If desired, new attributes may be specified for the file name in a subsequent OPEN statement. However, all attributes explicitly given to the file name in a DECLARE statement remain in effect.

As with the OPEN statement, closing more than one file with a single CLOSE statement can save execution time, but it may require the use of more storage than would otherwise be needed.

Note: Closing an already closed file or opening an already opened file has no effect.

LAYOUT OF STREAM FILES

When a stream data set is being created, its layout is governed by the LINESIZE option (and, if the associated file is a PRINT file, by the PAGESIZE option) on the OPEN statement that opens the file for output. The discussion below shows the effect of these options on a PRINT file; however, all stream data sets have a line size associated with them which is conceptually as described below, except that the line size might not necessarily refer to a line that is actually printed. If LINESIZE is not specified on the OPEN statement for a PRINT file, a default value is assumed (120 characters per line for the F Compiler). The line size for input is the line size with which the data set was created. If PAGESIZE is not specified on the OPEN statement for a PRINT file, a default value is assumed (60 lines per page for the F Compiler).

PAGE LAYOUT FOR PRINT FILES

The overall layout of a page in a file that has the PRINT attribute is controlled by means of the PAGESIZE and LINESIZE options of the OPEN statement. For example:

```
OPEN FILE(REPORT) OUTPUT STREAM PRINT
   PAGESIZE(55) LINESIZE(110);
```

This statement opens the REPORT file as a PRINT file. The specification PAGESIZE(55) indicates that each page should contain a maximum of 55 lines. An attempt to write on a page after 55 lines have already been written (or skipped) will raise the ENDPAGE condition. The standard system action for the ENDPAGE condition is to skip to a new page, but the programmer can establish his own action through use of the ON statement.

The ENDPAGE condition is raised only once per page. Consequently, printing can be continued beyond the specified PAGESIZE after the ENDPAGE condition has been raised the first time. This can be useful, for example, if a footing is to be written at the bottom of each page. Consider the following example:

```
ON ENDPAGE(REPORT) BEGIN;
   PUT FILE(REPORT) SKIP LIST
       (FOOTING);
   PUT FILE(REPORT) PAGE;
   N = N + 1;
   PUT FILE(REPORT) LIST
       ('PAGE '||N);
   PUT FILE(REPORT) SKIP (3);
   END;
```

Assume that REPORT has been opened with PAGESIZE(55), as shown in the previous example. When an attempt is made to write on line 56 (or to skip beyond line 55), the ENDPAGE condition will arise, and the begin block shown here will be executed. The first PUT statement specifies that a line is to be skipped, and the value of FOOTING, presumably a character string, is to be printed on line 57 (when ENDPAGE arises, the current line is always PAGESIZE+1).

The second PUT statement causes a skip to the next page and the ENDPAGE counter is automatically reset for the new page. The page number is incremented, and the character string 'PAGE ' is concatenated with the new page number and printed. The final PUT statement causes three lines to be skipped, so that the next printing will be on line 4. Control returns from the begin block to the PUT statement that caused the ENDPAGE condition, and the data is printed. Any SKIP option specified in that statement is ignored, however.

The specification LINESIZE(110) indicates that each line on the page can contain a maximum of 110 characters. An attempt to write a line greater than 110 characters will cause the excess characters to be placed on the next line.

The PAGESIZE option can be specified only for a file with the PRINT attribute (as stated earlier, the PRINT attribute implies the OUTPUT and STREAM attributes) and only in the OPEN statement. The LINESIZE option can be specified only for a file with the OUTPUT and STREAM attributes, with or without the PRINT attribute.

Further details of writing in PRINT files appear later in this chapter in "Data Transmission."

STANDARD FILES

Two standard system files are provided that can be used by any PL/I program. One is the standard system input file called SYSIN. The other is the standard system output file called SYSPRINT. These files need not be declared or opened explicitly; a standard set of attributes is applied automatically. For SYSIN, these attributes specify that it is a stream-oriented input file. For SYSPRINT, the standard attributes specify stream-oriented output that is to be printed. Both file names, SYSIN and SYSPRINT, are assumed to have the EXTERNAL attribute, even though SYSPRINT contains more than seven characters.

These file names need not be explicitly stated in GET and PUT statements when these files are to be used. GET and PUT I/O statements that do not name any file are equivalent to:

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

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

Any other references to SYSIN and SYSPRINT (such as in ON statements or in record-oriented statements) must be stated explicitly.

The identifiers SYSIN and SYSPRINT are not reserved for the specific purposes described above. These identifiers can be used for other purposes besides identifying standard system files. Other attributes can be applied to them, either explicitly or contextually, but the PRINT attribute is applied automatically to SYSPRINT when it is declared as a file name, unless the INTERNAL attribute is declared for it.

Note: Special care must be taken when SYSIN or SYSPRINT is declared by the programmer as anything other than a STREAM file. The F Compiler causes, in effect, the identifier SYSIN to be inserted into each GET statement in which no file name is explicitly stated and the identifier SYSPRINT to be inserted into each PUT statement in which no file name is explicitly stated. Consequently, the following would be in error:

```
DECLARE (SYSIN,SYSPRINT) FIXED
     DECIMAL (4,2);
          .
          .
          .
GET LIST (A,B,C);
PUT LIST (D,E,F);
```

The identifier SYSIN would be inserted into the GET statement, and SYSPRINT in the PUT statement. In this case, however, they would not refer to the standard files, but to the fixed-point variables declared in the block.

ENVIRONMENTAL CONSIDERATIONS FOR DATA SETS

The PL/I compiled program produced by the F Compiler is designed to be executed under control of the operating system for System/360. This operating system provides data management facilities that control the organization, location, storage, and retrieval of data sets. The PL/I program calls upon these facilities when it is being executed. The following discussions describe the relationship between the input and output statements of a PL/I program and the various data set organizations supported by the data management facilities of the operating system. A complete discussion of data management appears in the publication: IBM System/360 Operating System: Supervisor and Data Management Services, Form C28-6646.

DEVICE INDEPENDENCE OF INPUT AND OUTPUT STATEMENTS

The input and output statements of a PL/I source program are concerned with the logical organization of a data set and not with its physical characteristics. Much of the detailed information ultimately required by a PL/I program to process a data set (that is, information such as input/output device type, unit number, recording density, and buffering technique) need not be stated until the PL/I program is ready to be executed. Device independence of this type allows changes in this information without requiring changes to the PL/I program itself. The required information about specific input/output devices is supplied through data definition (DD) statements in the job step that calls for execution of the program. By changing the DD statements, different input/output devices or even different data sets may be specified for a file. Therefore, a PL/I program can be designed without specific knowledge of the input/output devices that will be used when the program is executed.

Some of the information specified in the ENVIRONMENT attribute can alternatively be specified in a DD statement.

The ENVIRONMENT Attribute

The ENVIRONMENT attribute provides information about the physical organization of the data set associated with a file. This information allows the compiler to determine the method of accessing the data set.

For the F Compiler, the ENVIRONMENT attribute has the following general format:

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

where "option list" is:

```
[F(block-size[,record-size])
 V{S|BS}(max-block-size
    [,max-record-size])
 U(max-block-size)]

[BUFFERS(n)]

[CONSECUTIVE
 INDEXED
 REGIONAL(1)
 REGIONAL(2)
 REGIONAL(3)]

[LEAVE
 REWIND]

[CTLASA
 CTL360]

[COBOL]

[INDEXAREA[(index-area-size)]]
[NOWRITE]

[GENKEY]
```

For ease of discussion, the options in the ENVIRONMENT attribute are divided into eight groups: record format, buffer allocation, data set organization, volume disposition, carriage control, data interchange, data management optimization, and key classification. The information supplied by the first three options can be alternatively specified in DD statements or by default. Options may appear in any order.

Record Format

Logical records can appear in the following formats: fixed-length (F-format), variable-length (V-format, VS-format, VBS-format), or undefined (U-format). The block size and record size are specified in number of bytes. Record format, block size, and record size must appear together either in the ENVIRONMENT attribute or in the DCB parameter of the DD statement.

Fixed-length records: For fixed-length records, the block size must always be stated. If the records are blocked, the record size must also be stated; if no record size is specified, the records are assumed to be unblocked (that is, each block contains only one record). The records are blocked and deblocked in accordance with the specified block size and record size. The block size must be an exact multiple of the record size.

Variable-length records: For variable-length V-format records, the maximum block size must always be stated. If the records are blocked, the maximum record size must also be stated; if maximum record size is not specified, the records are assumed to be unblocked. Four bytes are used in each block to specify the block length, and four bytes are used in each record to specify the record length; the programmer must therefore allow an additional four bytes when stating the block size and when stating the record size. The record size must never exceed the block size. For example, if the maximum number of bytes in a record is likely to be 120, the specified block size must not be less than 128 bytes whether the records are blocked or not, since unblocked records are considered to be in blocks of one record each; if the

records are blocked, the record size must not be less than 124 bytes, and must be at least four bytes less than the specified block size.

For variable-length VS-format records, the maximum block size must always be stated. VS-format records can exceed the block size; when they do, they are segmented and the segments placed in consecutive blocks. Each block can contain only one record or segment of a record. Deblocking depends on control information at the beginning of each block and at the beginning of each record: four bytes are used in each block to specify block length; another four bytes are used in each record or segment of a record to indicate record or segment length and to indicate whether a segment is the first, intermediate, last, or the only part of a record. For example, if the record format is specified as VS(80,200), a record that includes 180 bytes of data will appear in the data set as two blocks of 80 bytes (8 control bytes and 72 data bytes) and one block of 44 bytes (8 control bytes and 36 data bytes).

Variable-length VBS-format records are similar to VS-format records except that they are blocked, that is, each block contains as many records or segments as it can accommodate; each block is substantially the same size, although there can be a variation of up to four bytes, since each segment must contain at least one byte of data. For example, a block might contain the last segment of one record, one or more complete records, and the first segment of another record.

VS-format and VBS-format records are known as spanned records because they can start in one block and be continued in the next. However, segmentation does not affect the programmer as it occurs automatically and the records always appear as complete logical records. The use of spanned records allows the programmer to select a block size, independently of record size, that will combine optimum usage of external storage space with maximum efficiency of transmission.

Undefined-length records: For undefined-length records, all processing of records is the responsibility of the programmer. If a length specification is included in the record, the programmer must insert it himself, and he must retrieve the information himself.

Buffer Allocation

A buffer is an internal program-storage area that is used for intermediate storage of data transmitted to and from a data set. Allocating two or more buffers for a data set permits input and output activity to occur concurrently with internal processing.

The option BUFFERS (n) in the ENVIRONMENT attribute specifies the number (n) of buffers to be allocated for a data set. This number must not exceed 255. The BUFNO subparameter in the DD statement can be used, instead of the BUFFERS option in the ENVIRONMENT attribute, to specify the number of buffers. If the number of buffers is not specified, it is assumed to be either one or two, depending on the access method used. Note that a buffer specification for DIRECT files is ignored.

Data Set Organization

The organization of a data set determines how data is recorded in a data set volume and, once recorded, how data is subsequently retrieved so that it can be transmitted to the program. Logical records are stored in and retrieved from a data set, in either STREAM or RECORD SEQUENTIAL transmission, on the basis of successive physical positions or, in DIRECT RECORD transmission, on the basis of the values of keys specified in data transmission statements. These storage and retrieval methods provide PL/I with three general data set organizations: CONSECUTIVE, INDEXED, and REGIONAL. CONSECUTIVE organization is assumed by default.

Record Keys: Both INDEXED and REGIONAL data set organization allow the use of keys to identify specific records. There are two kinds of keys, recorded keys and source keys. A recorded key is a character string that actually appears in the data set, along with the record, as a positive identification of that record. It cannot exceed 255 bytes in length. A source key is a character string (or expression) that appears in a record-oriented data transmission statement to identify the record to which the statement refers.

The way keys are specified and used differs between INDEXED and REGIONAL, as well as among the three different kinds of REGIONAL organization. For data sets that contain recorded keys, a part or all of the source key must exactly match the recorded key in order to positively identify a record.

Whenever source keys are used in a program to access or create a data set (using the KEY or KEYFROM option) or whenever the KEYTO option is specified, the KEYED attribute must be specified for the file. In addition, for data sets that contain recorded keys, the KEYLEN subparameter of the DCB parameter of the associated DD statement must be used to specify the actual length, in bytes, of the recorded key.

The type and use of keys for each of the different data set organizations is explained in the discussions below. Keys are not used in the processing of CONSECUTIVE data sets.

CONSECUTIVE DATA SET ORGANIZATION: In a data set with CONSECUTIVE organization, the logical records are organized solely on the basis of their successive physical positions, such as they appear on magnetic tape. Such a data set does not use keys to determine the position of each record. Records are retrieved only in sequential order; therefore, the associated file must have the SEQUENTIAL attribute (or be a STREAM file). Records may be F-format or V-format, blocked or unblocked, or U-format.

Input/output devices permitted for CONSECUTIVE data sets include magnetic tape units, card readers and punches, printers, direct-access storage units, and paper tape readers.

Later discussions will show that both stream-oriented and record-oriented transmission statements can process data sets with CONSECUTIVE organizations. However, stream-oriented statements are restricted to this type of organization; record-oriented statements are not.

After a CONSECUTIVE data set is created, it may be opened only for input or update operations, unless DISP=MOD is specified in the DD statement, in which case it can be opened for output (records can then be added to the end of the data set). Reading of such a data set may be either forwards or backwards if the data set is recorded on magnetic tape. To read the data set backwards, the associated file must be opened with the BACKWARDS attribute. If a data set is first read or written forwards and then read backwards in the same program, the LEAVE option in the ENVIRONMENT attribute must be specified to prevent the normal rewind when the file is closed or when volume switching occurs with a multi-volume data set. V-format records cannot be read backwards.

Note the difference between the CONSECUTIVE option of the ENVIRONMENT attribute and the SEQUENTIAL attribute. CONSECUTIVE specifies the physical organization of a data set; SEQUENTIAL specifies how a file is to be processed. A data set with CONSECUTIVE organization must be associated with a SEQUENTIAL file; but a data set with INDEXED or REGIONAL organization can be associated with either a SEQUENTIAL or DIRECT file.

INDEXED DATA SET ORGANIZATION: A data set with INDEXED organization, which must be on a direct-access device, has its records arranged in logical sequence according to keys that are associated with every record. The key is a character string that usually represents an item within the record, such as a part number, a date, or a name. Logical records are arranged in ascending sequence on their keys, according to the collating sequence. There are indexes that specify the highest key for each track and for each cylinder.

Only fixed-length records (F-format), unblocked or blocked, can be used with the INDEXED organization.

Two subparameters of the DCB parameter must be specified for each INDEXED data set. They are:

```
DSORG=IS

KEYLEN=length-of-recorded-key
```

In addition, if the key is embedded within the record, the RKP subparameter must be included, giving the location of the key; the value specified in the RKP subparameter is one less than the byte number of the first character of the key; that is, if RKP=1, the key starts in the second byte of the record. The value assumed, if this subparameter is omitted, is RKP=0, which specifies that the key is not embedded in the record but is separate from it. Also if any records are to be deleted, or if already deleted records are to be ignored, the subparameter OPTCD=L must be specified. The OPTCD subparameter cannot be changed after the data set has been created.

Note: For unblocked records, the key, even if embedded, is always recorded in a position preceding the actual data. Consequently, the RKP subparameter need not be specified for unblocked records.

Unlike CONSECUTIVE organization, INDEXED organization does not require every record to be accessed in sequential fashion. Random retrieval, addition, deletion, and replacement of logical records are permitted in INDEXED data sets. However, if the programmer desires, either sequential access or direct access can be used with a data set that has INDEXED organization.

The key associated with a logical record in an INDEXED data set consists of a character string containing a maximum of 255 characters. It is always recorded in the data set and is usually a part of the data. The length of the recorded key must be specified in the DCB subparameter KEYLEN of the associated DD statement. Records in an INDEXED data set are accessed by means of record-oriented transmission statements. If access is direct, these statements employ a source key that identifies a particular record by the value of its recorded key.

Logical records within an INDEXED data set are either actual records containing valid data, or deleted or dummy records that can later be replaced by valid data. The programmer can create dummy records by marking the record as "deleted" (that is, by specifying the constant (8)'1'B as the first byte of the data section of the record).

An INDEXED data set can be created only sequentially. Once an INDEXED data set has been created, its associated file may have the INPUT or UPDATE attributes as well as the SEQUENTIAL or DIRECT attributes. When the file has the DIRECT attribute, records may be retrieved, added, deleted, and replaced at random.

SEQUENTIAL INDEXED Files: The file of an INDEXED data set accessed in SEQUENTIAL fashion may be opened with either the INPUT or the UPDATE attribute, but the data transmission statements need not include source keys, nor need the file have the KEYED attribute. Sequential access is in the order of ascending recorded-key values. Logical records are retrieved in this order and not necessarily in the order in which they were added to the data set.

When an INDEXED data set is accessed sequentially for either INPUT or UPDATE activity, it is possible to reposition the data set by using either a unique source key or a generic key in a READ statement. (The use of a generic key is discussed in the section "Key Classification" in this chapter.) For repositioning to occur, the associated file must have the KEYED attribute. Repositioning can occur in either a forward or backward direction to the specified record that is to be retrieved. Should a subsequent READ statement not contain a source key, the record with the next higher recorded key will be retrieved.

When the file of an INDEXED data set has the SEQUENTIAL and UPDATE attributes, the only I/O statements (other than OPEN and CLOSE) that can refer to the file are READ and REWRITE. Before a REWRITE statement can be executed, the specified record must have been retrieved by a READ statement. Every record that has been retrieved, however, need not be rewritten. Logical records cannot be added to an INDEXED data set that is being accessed in the SEQUENTIAL UPDATE mode.

DIRECT INDEXED Files: The file of an INDEXED data set accessed in DIRECT fashion may be opened with either the INPUT or the UPDATE attribute. For a file with the DIRECT and UPDATE attributes, logical records may be retrieved, added, deleted, and replaced according to the following conventions:

1. Retrieval: Deleted records are not made available by a READ statement.

2. Addition: If the key is unique, the logical record is inserted by a WRITE statement into the data set, replacing one marked as deleted, if it has the same key. If no space exists for the additional record, the KEY condition is raised. The KEY condition is also raised if the additional record has a key that is the same as the recorded key of a record already in the data set, but is not marked as deleted.

3. Deletion: The record specified by a source key in a DELETE statement is retrieved, marked as deleted, and rewritten into the data set. Deletion is possible only when the DD statement associated with the data set contains the DCB subparameter OPTCD=L. If the data set has blocked records and RKP=0, then records cannot be deleted.

4. Replacement: The record specified by a source key is replaced by the new record. Unblocked records may be replaced without being read. However, if the data set contains blocked records, the specified record must first be retrieved with a READ statement and then replaced with a REWRITE statement.

Note: The length specified in the DCB subparameter KEYLEN always gives the length of the recorded key and not necessarily the length of the source key. If the lengths of the source and recorded keys differ when an INDEXED data set is accessed, the lengths are made equal by truncating the source key on the right or by extending it on the right with blank characters.

The operating system permits recorded keys to be separate from or embedded within logical records. When the recorded keys are separate from the logical records, the DCB subparameter RKP=0 in the DD statement associated with the data set need not be

specified, since it is the default assumption.

Unblocked records always have a separate recorded key preceding each logical record, whether or not the key is also embedded within the record. Only the key of the last logical record in a block of records is recorded separately, preceding the block.

REGIONAL DATA SET ORGANIZATION: REGIONAL organization of a data set provides control of the physical placement of records in the data set. This type of control allows the programmer to optimize the record access time required by a particular application. Such optimization is not available with CONSECUTIVE and INDEXED organizations, in which successive records are written either in strict physical sequence or in logical sequence depending upon ascending key values. Neither of these methods takes advantage of the timing characteristics of direct-access storage devices. The input/output devices allowed for REGIONAL data sets are restricted to direct-access storage devices.

A data set with REGIONAL organization is divided into regions, each of which is identified by a region number and each of which may contain one or more records. The regions are numbered in succession, beginning with zero, and a record is accessed by specifying its region number in the source key of a record-oriented transmission statement. Two kinds of regional specifications are used, relative record and relative track. A relative record specification refers to a region of the data set by specifying the number of a particular record, relative to the first record in the data set, which is number zero. A relative track specification refers to a region of the data set by specifying the number of a particular track relative to the first track of the data set, which is track zero. In some cases, as is discussed later, either the relative record or the relative track indicates only the beginning of a region in which a record is to be written or from which it is to be accessed.

There are three types of REGIONAL organization, two of which, REGIONAL(2) and REGIONAL(3) permit recorded keys to appear physically in the data set with the logical records. Unlike the keys for INDEXED data sets, however, these recorded keys are never embedded within a record. When REGIONAL records are accessed by record-oriented statements, the source keys, specified in the statements, represent a region number and may also represent a recorded key.

Direct access of REGIONAL data sets employs the region number, specified in the source key, for direct access of the region. Once the region has been accessed, a sequential search may or may not be performed for a record that contains a recorded key identical to the source key. The search is performed from the located region onward through the other regions.

Sequential processing of REGIONAL data sets accesses records in ascending relative position, the initial position being region zero. The values of recorded keys do not affect this access sequence.

Each of the three REGIONAL types is described in the following discussions.

REGIONAL(1) Organization: A data set with REGIONAL(1) organization contains unblocked F-format records that do not have recorded keys. Each region in the data set contains only one logical record; therefore, each region number corresponds to a relative record position within the data set. The relative position of the first record is zero.

Since there are no recorded keys to be used for comparison, only a region number, which serves as the sole identification of a particular logical record, is meaningful in a source key. The character-string value of the source key must represent an unsigned decimal integer that does not exceed 16777215. Only the characters 0 through 9 and the blank character are valid in the source key (leading blanks are interpreted as zeros). If more than eight characters appear in the key, only the eight rightmost characters are used as the region number. If there are fewer than eight characters, blanks (interpreted as zeros) are inserted on the left.

REGIONAL(2) Organization: A data set with REGIONAL(2) organization contains unblocked F-format records that have recorded keys. As with REGIONAL(1) organization, each region in the data set contains only one logical record.

The recorded key associated with each logical record is a character string recorded in the data set and immediately preceding the record. The recorded key may or may not include the regional number as its rightmost eight characters. The source key (specified as a constant or some other expression) consists of a character-string value. It may be thought of as having two logical parts, the region specification and the specification of a comparison key to be compared, on input, with the recorded key, or to be written, on output, as the recorded key.

The rightmost eight characters of the source key make up the region specification, which states the region number. (Leading blanks in the region specification are interpreted as zeros.) A substring beginning at the left of the source key and containing the number of characters specified in the KEYLEN subparameter is the comparison key specification. To retrieve a record, this substring must exactly match the recorded key of the record. The comparison key can include the region specification if the region number is a part of the recorded key; in such a case, the source key and the comparison key specification are identical. In other cases, a portion of the source key may not be used. The comparison key is always equal to KEYLEN; if the source key is longer than KEYLEN+8, the characters in the source key between the comparison key and the region specification are ignored.

Consider the following examples of source keys (the character "b" represents a blank):

```
KEY  ('JOHNbDOEbbbbbb12363251')
```

The rightmost eight characters make up the region specification, the relative number of the record. Assume that the associated DD statement has the subparameter KEYLEN=14. In retrieving a record, the search will begin with the beginning of the track upon which record number 12363251 is recorded, and it will continue until a record is found having the recorded key of JOHNbDOEbbbbbb.

If the subparameter were KEYLEN=22, the search still would begin at the same place, but since the comparison specification and the source key are the same length, the search would be for a record having the recorded key 'JOHNbDOEbbbbbb12363251'.

```
KEY ('JOHNbDOEbbbbbbDIVISIONb423bbbb34627')
```

In this example, the rightmost eight characters contain blanks, which are interpreted as zeros. The search will begin at record number 00034627. If KEYLEN=14 is specified, the characters DIVISIONb423b will be ignored.

Assume that COUNTER is declared FIXED BINARY (21) and NAME is declared CHARACTER(15). The key might be specified as:

```
KEY (NAME || COUNTER)
```

The value of COUNTER will be converted to a character string of eleven characters (the rules for conversion specify that a binary value of this length, when converted to character, will result in a string of length 11, three blanks followed by eight decimal digits). The value of the rightmost eight characters of the converted string will be taken to be the region specification. Then if the keylength specification is KEYLEN=15, the value of NAME will be taken to be the comparison specification.

In any of these examples, if the operation were output, the search would begin at the beginning of the track of the region specified, and the recorded key and the record would be written in the first available space. In either input or output, the region specification indicates merely the beginning of the area where the search is to commence; it does not indicate a specific position where the record can be found or where it is to be written.

The closer a logical record is to the specified region, the more efficient it becomes to access the record, because the search continues through the entire data set, going from the region specified to the end of the data set, then from the beginning of the data set back to the specified region. The search can be limited, however, by the DD statement DCB subparameter LIMCT=n, where n is the number of regions to be searched. The search will continue only through the track containing the region whose number is given by r+n-1, where r is the region number specified in the source key.

The regional specification for REGIONAL(2) data sets cannot exceed 16,777,215 in value.

In a REGIONAL data set, a source key must be specified, but it need be no longer than one character. For example:

```
KEY('3')
```

The character '3' represents the region number. The comparison key, if shorter than the KEYLEN specification, is extended on the right with blanks. If KEYLEN=8 is specified, the comparison key will be considered to be '3bbbbbbb'.

REGIONAL(3) Organization: A data set with REGIONAL(3) organization contains unblocked logical records with F-, V-, or U-formats. Each record also has a recorded key, which is used like the recorded key in REGIONAL(2) organization.

Each region in a data set with REGIONAL(3) organization, differing from REGIONAL(2), corresponds to a track on the direct-access storage device; therefore, the source key for a REGIONAL(3) data set is the same as for REGIONAL(2), except that the region specification specifies a relative track number.

The search for matching source and recorded keys is the same as the search in REGIONAL(2) organization, except that the value of the region number (relative track number) must not exceed 32,767, and LIMCT, if specified in the DD statement, limits the number of tracks to be searched. Logical records are inserted into the first available space within, or following, the specified region.

Comparisons of REGIONAL Types: Records in a REGIONAL data set are either "actual," representing valid data, or "dummy," representing deleted records. Unlike INDEXED data sets, REGIONAL data sets do not require the DCB subparameter OPTCD=L in the DD statement to eliminate deleted records or to create dummy records. Dummy records, in F-format, are identical in REGIONAL(2) and REGIONAL(3) data sets. Since record lengths of U-format or V-format cannot be known beforehand, dummy records cannot be used for REGIONAL(3) data sets of U-format or V-format records. The system, however, maintains counts of space used and space available on each track in the capacity record for that track.

When a REGIONAL file is opened for DIRECT OUTPUT, the whole of the initial allocation of space is initialized. For F-format records, it is filled with dummy records; for U-format or V-format records, a capacity record is written for each track, which indicates that the track contains no records.

When a REGIONAL file is created by SEQUENTIAL OUTPUT with F-format records, regions that are incomplete when the region number is incremented, are filled out with dummy records. For U-format and V-format records, the capacity record of each track (i.e., region) is written when the region number is incremented. When the file is closed, the remainder of the current extent is initialized.

For retrieving records, a DIRECT file associated with a REGIONAL data set can have either INPUT or UPDATE attributes. For retrieving records, a SEQUENTIAL file associated with a REGIONAL data set must have the INPUT or UPDATE attribute. For REGIONAL(1) and REGIONAL(2) data sets, sequential access occurs in the order of ascending relative records. For REGIONAL(3), it occurs in the order of ascending relative tracks. The values of recorded keys do not affect the order of sequential access and the KEY option cannot be used. All records within a REGIONAL(1) data set, whether dummy or actual, are retrieved in sequence; therefore, the PL/I program should be prepared to recognize dummy records when they are in REGIONAL(1) data sets (the constant (8)'1'B in the initial byte). Dummy records in REGIONAL(2) and REGIONAL(3) data sets are not made available to the program when accessed sequentially.

When a REGIONAL data set is associated with a file that has the DIRECT attribute, records can be retrieved, added, deleted, and replaced according to the following conventions:

1. Retrieval

   REGIONAL(1): All records, whether dummy or actual, can be retrieved.

   REGIONAL(2): Dummy records cannot be retrieved.

   REGIONAL(3): Dummy records cannot be retrieved.

2. Addition

   REGIONAL(1): Addition involves the replacement of existing records, whether dummy or actual (no error condition is raised in either case).

   REGIONAL(2): Addition involves the replacement of dummy records within specified regions (or within subsequent regions if extended searches have been permitted by the DCB subparameter LIMCT in a DD statement).

   REGIONAL(3): For F-format records, addition is the same as that for REGIONAL(2). V- and U-format records are added to available space within specified tracks (or within subsequent tracks if extended searches have been permitted by the DCB subparameter LIMCT in a DD statement).

3. Deletion

   REGIONAL(1): The specified record is marked as a dummy record. The record space is available for re-use.

   REGIONAL(2): The specified record is marked as a dummy record. The recorded key is replaced with a dummy key. The record space is available for re-use.

   REGIONAL(3): For F-format records, deletion is the same as for REGIONAL(2). V- and U-format records are marked as dummy records. The recorded keys are replaced with dummy keys. The record space is not available for re-use.

4. Replacement

   REGIONAL(1) The specified record, whether dummy or actual, is rewritten.

   REGIONAL(2) A record with the specified key must exist. The record is rewritten.

   REGIONAL(3): Replacement is the same as for REGIONAL(2). All record formats are rewritten.

Volume Disposition

The volume disposition options allow the user to specify the action to be taken (1) when the end of a magnetic tape volume is

reached and (2) when a data set on a magnetic tape volume is closed normally or abnormally.

The action specified by the LEAVE option depends on the volume position.

1. If the end of the volume has been reached, no repositioning of the tape occurs and the channel is freed.

2. If a data set is closed normally or abnormally before the end of the volume, the tape is repositioned at the end of the data set (unless it is already there) or at the end of the current volume if a multivolume data set is being accessed. The channel is kept busy during repositioning.

The REWIND option allows the end-of-volume or data-set-closure tape action to be controlled by the DISP field of the associated DD statement. If DISP=(status,DELETE) is specified in the DD statement, the tape is rewound but not unloaded. If DISP=(status,KEEP|CATLG|UNCATLG) is specified, the tape is rewound and unloaded. If DISP=(status,PASS) is specified, the tape is wound on to the end of the data set, unless a BACKWARDS file is being used, in which case the tape is repositioned at the beginning of the data set. When DISP=(status,PASS) is specified, the channel is kept busy when positioning; in the other two cases the channel is freed when positioning.

If neither LEAVE nor REWIND is specified in the options list of the ENVIRONMENT attribute, the tape is repositioned at the beginning of the current data set on the current volume. The channel is kept busy while repositioning.

If both LEAVE and REWIND are specified as options of the ENVIRONMENT attribute, REWIND is ignored.

Carriage Control

The carriage control options in the ENVIRONMENT attribute specify that the first character of a record is to be interpreted as a carriage control character.

1. The CTLASA option specifies ASA standard control characters.

2. The CTL360 option specifies IBM System/360 machine code control characters.

Data Interchange

The COBOL option in the ENVIRONMENT attribute specifies that the file will contain structures mapped according to the COBOL (F) algorithm. This type of file is subject to the following restrictions:

1. The file can be used only for READ INTO and WRITE FROM statements.

2. The EVENT option cannot be used with the above statements.

3. If an ON-condition arises as a result of the READ INTO statement, the variable named in the INTO option cannot be used in the on-unit, and return from the on-unit must be normal if the completed variable is required.

4. The file name cannot be passed as an argument.

Data Management Optimization

The data management optimization options in the ENVIRONMENT attribute increase program efficiency, in certain circumstances, when DIRECT INDEXED data sets are to be accessed.

The INDEXAREA option improves the input/output speed of a DIRECT INPUT or DIRECT UPDATE file with INDEXED data set organization, by having the highest level of index placed in main storage. The "index area size," when specified, must be a decimal integer constant whose value lies within the range zero through 32,767. If an index area size is not specified, the highest level index is moved unconditionally into main storage. If an index area size is specified, the highest level index is held in main storage, provided that its size does not exceed that specified. If the specified size is less than zero or greater than 32,767, the compiler issues a warning message and ignores the option.

The NOWRITE option can be specified only for DIRECT UPDATE files with INDEXED data set organization. It informs the compiler that no records are to be added to the data set and that data management modules concerned solely with adding records are not required; it thus allows the size of the compiled program to be reduced.

Key Classification

The GENKEY (generic key) option applies only to INDEXED data sets. It enables the programmer to classify keys recorded in a data set and to use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.

A generic key is a character string that identifies a class of keys: all keys that begin with the string are members of that class. For example, the recorded keys 'ABCD', 'ABCE', and 'ABDF' are all members of the classes identified by the generic keys 'A' and 'AB', and the first two are also members of the class 'ABC'; and the three recorded keys can be considered to be unique members of the classes 'ABCD', 'ABCE', and 'ABDF', respectively.

The GENKEY option allows the programmer to start sequential reading or updating of an INDEXED data set from the first non-dummy record that has a key in a particular class; the class is identified by the inclusion of its generic key in the KEY option of a READ statement. Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

In the following example, an embedded key with a length of more than three bytes is assumed at the beginning of each record:

```
DCL IND FILE RECORD SEQUENTIAL KEYED
   UPDATE ENV (INDEXED GENKEY);
          .
          .
          .
READ FILE(IND) INTO(INFIELD) KEY('ABC');
IF SUBSTR(INFIELD,1,3)¬= 'ABC' THEN GO TO
   END;
          .
          .
          .
NEXT: READ FILE (IND) INTO (INFIELD);
          .
          .
          .
      GO TO NEXT;
```

The first READ statement causes the first non-dummy record in the data set whose key begins with 'ABC' to be read into INFIELD; each time the second READ statement is executed, the non-dummy record with the next higher key will be retrieved. Repeated execution of the second READ statement could result in reading records from higher key classes since no indication is given when the end of a key class is reached. It is the responsibility of the programmer to check each key if he does not wish to read beyond the key class. Any subsequent execution of the first read statement would reposition to the first record of the key class 'ABC'.

DATA TRANSMISSION

As discussed earlier in this chapter, PL/I provides two types of data transmission, stream-oriented and record-oriented.

With stream-oriented transmission, a data set is considered to be a continuous stream of data items in character form; internal bit-string representations and the internal formats of coded arithmetic data do not appear in the stream. Data items are assigned from the stream to program variables or from program variables (or expressions) into the stream, with appropriate conversion from or to character form. Stream-oriented transmission statements ignore the physical and logical boundaries between records.

With record-oriented transmission, a data set is treated as a collection of logical records, each of which consists of one or more data items. The data items can have any representation, internal or external, that is acceptable to the computer, and there is no data conversion. Each logical record is transmitted as a unit to or from a program variable.

Stream transmission uses only two input and output statements, GET and PUT, which get the next series of data items from the stream or put a specified set of data items into the stream. In record transmission, the corresponding statements are READ and WRITE, which read a logical record from the data set or write a specified logical record into the data set. A third statement, REWRITE, causes replacement of an existing record in an UPDATE file.

A fourth statement, LOCATE, is used for based variables only; it allocates storage for the variable in an output buffer, and writes out the record automatically at the next output statement on the file.

It is possible for the same data set to be processed at different times for either stream transmission or record transmission; however, all items in the data set would have to be in character form.

One of the attributes, STREAM or RECORD, specified for the file associated with a data set determines which transmission method is applicable to the file each time it is opened.

STREAM-ORIENTED TRANSMISSION

There are three modes of stream transmission: list-directed, data-directed, and edit-directed. All three modes use the same statements for input and output, the GET and PUT statements. These statements, in general, require the following information for each mode:

1. The name of the file associated with the data set from which data is to be obtained or to which data is to be assigned.

2. A list of program variables to which data items are to be assigned during input or from which data items are to be obtained during output. This list is called a data list. On output, the data list also can include constants and other expressions.

3. The format of each data item in the stream.

Under certain conditions all of this required information can be implied; in other cases, only a portion of it need be stated explicitly. If the file name is not specified, one of the standard files, SYSIN or SYSPRINT, is assumed; this applies to each of the three modes. In list-directed and data-directed transmission, the formats of data items are not specified in GET and PUT statements; and in data-directed transmission, even the data list need not be specified.

List-Directed Transmission

List-directed transmission permits the user to specify the variables to which data is assigned and to specify data to be transmitted without specifying the format.

Input: In general, the data items in the stream are character strings in the form of optionally signed valid constants or in the form of expressions that represent complex constants. The variables to which the data is to be assigned are specified by a data list. Items are separated by a comma and/or one or more blanks.

Output: The data values to be transmitted are specified by a variable, a constant, or an expression that represents a data item. Each data item placed in the stream is a character-string representation that reflects the attributes of the variable. Items are separated by a blank. Leading zeros of arithmetic data are suppressed. Binary fixed-point and floating-point items, however, are character strings that express the value in decimal representation.

For PRINT files, data items are automatically aligned on implementation-defined preset tab positions. For the F Compiler, these positions are 1, 25, 49, 73, 97, and 121, but provision is made for the programmer to alter these values (for information, see the publication IBM System/360 Operating System, PL/I (F) Programmer's Guide, Form C28-6594).

Data-Directed Transmission

Data-directed transmission permits the user to transmit self-identifying data.

Input: Each data item in the stream is in the form of an assignment statement that specifies both the value and the variable to which it is to be assigned. In general, values are in the form of valid constants. Items are separated by a comma and/or one or more blanks. A semicolon must end each group of items to be accessed by a single GET statement. A data list in the GET statement is optional, since the semicolon determines the number of items to be obtained from the stream.

Output: The data values to be transmitted may be specified by an optional data list. Each data item placed in the stream has the form of an assignment statement without a semicolon. Items are separated by a blank. The last item transmitted by each PUT statement is followed by a semicolon. Leading zeros of arithmetic data are suppressed. The character representation of each value reflects the attributes of the variable, except for fixed-point and floating-point binary items, which appear as values expressed in decimal notation.

If the data list is omitted, it is assumed to specify all variables that are known within the block containing the statement and are permitted in data-directed output.

For PRINT files, data items are automatically aligned on implementation-defined preset tab positions, referred to under "List-Directed Transmission."

Edit-Directed Transmission

Edit-directed transmission permits the user to specify the variables to which data is to be assigned or to specify data to be

transmitted. Edit-directed transmission allows a programmer to specify the format for each item on the external medium.

Input: Data in the stream is a continuous string of characters; different data items are not separated. The variables to which the data is to be assigned is specified by a data list. Format items in a format list in the GET statement specify the number of characters to be assigned to each variable and describe characteristics of the data (for example, the assumed location of a decimal point).

Output: The data values to be transmitted are defined by a data list. The format that the data is to have in the stream is defined by a format list.

DATA SPECIFICATIONS FOR STREAM TRANSMISSION

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:

(data-list)

where "data list" is defined as:

element [,element]...

Syntax rules:

The nature of the elements depends upon whether the data list is used for input or for output. The rules are as follows:

1. On input, a data-list element for edit-directed and list-directed transmission can be one of the following: an element, array, or structure variable, a pseudo-variable, or a repetitive specification (similar to a repetitive specification of a DO group) involving any of these elements. For a data-directed data specification, a data-list element can be an element, array, or structure variable. None of the names in a data-directed data list can be subscripted, but qualified names are allowed.

2. On output, a data-list element for edit-directed and list-directed data specifications can be one of the following: an element expression, an array expression, a structure expression, or a repetitive specification involving any of these elements. For a data-directed data specification, a data-list element can be an element, array, or structure variable, or a repetitive specification involving any of these elements. Subscripts are allowed for data-directed output.

3. The elements of a data list must be of arithmetic or string data type.

4. A data list must always be enclosed in parentheses.

Repetitive Specification

The general format of a repetitive specification is shown in Figure 8-1.

Syntax rules:

1. An element in the element list of the repetitive specification can be any of those allowed as data-list elements as listed above.

2. The expressions in the specification, which are the same as those in a DO statement, are described as follows:

   a. Each expression in the specification is an element 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. Expression 4 represents a second condition to control the number of repetitions. 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. Each repetitive specification must be enclosed in parentheses, as shown in the general format. Note that if a repetitive specification is the only element in a data list, two sets of outer parentheses are required, since the data list must have one set of parentheses and the repetitive specification must have a separate set.

4. As Figure 8-1 shows, the "specification" portion of a repetitive specification can be repeated a number of times, as in the following form:

   ```
   DO I = 1 TO 4, 6 TO 10
   ```

   Repetitive specifications can be nested; that is, an element of a repetitive specification can itself be a repetitive specification. Each DO portion must be delimited on the right with a right parenthesis (with its matching left parenthesis added to the beginning of the entire repetitive specification).

   When DO portions are nested, the rightmost DO is at the outer level of nesting. For example, consider the following statement:

   ```
   GET LIST (((A(I,J) DO I = 1 TO 2)
            DO J = 3 TO 4));
   ```

   Note the three sets of parentheses, in addition to the set used to delimit the subscript. The outermost set is the set required by the data list; the next is that required by the outer repetitive specification. The third

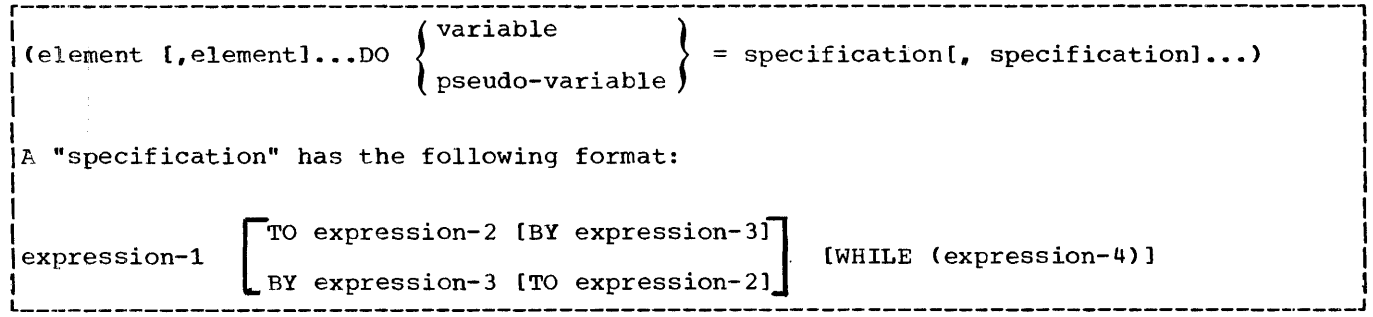


Figure 8-1. General Format for Repetitive Specifications.

set of parentheses is that required by the inner repetitive specification. This statement is equivalent to the following nested DO-groups:

```
DO J = 3 TO 4;
   DO I = 1 TO 2;
   GET LIST (A (I,J));
   END;
END;
```

It gives values to the elements of the array A in the following order:

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

Note: Although the DO keyword is used in the repetitive specification, a corresponding END statement is not allowed.

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 variable, 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 variable, the elements of the structure are transmitted in the order specified in the structure declaration.

For example, if a declaration is:

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

then the statement:

```
PUT FILE(X) LIST(A);
```

would result in the output being ordered as follows:

```
A.B(1) A.C(1) A.B(2) A.C(2) A.B(3)
   A.C(3)...etc.
```

If, however, the declaration had been:

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

then the same PUT statement would result in the output being ordered as follows:

```
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 for list-directed or edit-directed transmission, a variable is assigned a value, this new value is used if the variable appears in a later reference in the data list. For example:

```
GET LIST (N,(X(I) DO I=1 TO N), J, K,
   SUBSTR (NAME, J,K));
```

When this statement is executed, data is transmitted and assigned in the following order:

1. A new value is assigned to N.
2. Elements are assigned to the array X as specified in the repetitive specification in the order X(1),X(2),...X(N), with the new value of N used to specify the number of items to be assigned.
3. A new value is assigned to J.
4. A new value is assigned to K.

5. A substring of length K is assigned to the string variable NAME, beginning at the Jth character.

LIST-DIRECTED DATA SPECIFICATION

General format for a list-directed data specification, either input or output, is as follows:

```
LIST (data-list)
```

The data list is described in the preceding discussion. The keyword LIST must appear to specify the list-directed mode of transmission.

List-Directed Data in the Stream

Data in the stream, either input or output, is of character data type and has one of the following general forms:

```
[+|-] arithmetic-constant

      character-string-constant

      bit-string-constant

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

These forms correspond exactly to the forms used for writing optionally signed constants in a PL/I program. However, sterling constants cannot be used. A string constant must be one of the two permitted forms listed above; iteration and string repetition factors are not allowed. A blank must not precede the central + or - in complex expressions.

List-Directed Input Format

When the data named is an array, the data consists of constants, the first of which is assigned to the first element of the array, the second constant to the second element, etc., in row-major order.

A structure name in the data list represents a list of the contained element variables and arrays in the order specified in the structure description.

On input, 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 first non-blank character in the data set being a comma, or by two commas separated by an arbitrary number of blanks. A null field specifies that the value of the associated item in the data list 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 in the stream for the next GET statement is always at the character following the first blank or comma following the last data item transmitted. 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 GET statement will ignore intervening blanks and the comma (if present) and will access the next data item. However, if an edit-directed GET statement 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 removed, and the enclosed characters are interpreted as a character string.

If the data is a bit-string constant, enclosing quotation marks and the trailing character B are removed, and the enclosed characters are interpreted as a bit string.

If the data is an arithmetic constant or complex expression, it is converted to coded arithmetic form with the base, scale, mode, and precision implied by the constant.

Data type conversions follow the rules for conversion from character type, as listed in Part II, Section F, "Problem Data Conversion."

List-Directed Output Format

The values of the element variables and expressions in the data list are converted to character representations and transmitted to the data stream.

A blank separates successive data items transmitted. (For PRINT files, items are separated according to program tab settings.)

The length of the data field placed in the stream is a function of the attributes of the data item, including precision and

length. Detailed discussions of the conversion rules and their effect upon precision are listed in the sections covering conversion to character type in Part II, Section F, "Problem Data Conversion."

Fixed-point and floating-point binary data items are converted to decimal notation before being placed in the stream.

For numeric character values, the character-string value is transmitted.

Bit strings are converted to character representation of bit-string constants, consisting of the characters 0 and 1, enclosed in quotation marks, and followed by the letter B.

Character strings are written out. If the file does not have the attribute PRINT, enclosing quotation marks are supplied, and contained single quotation marks or apostrophes are replaced by two quotation marks. The field width is the current length of the string plus the number of added quotation marks. If the file has the attribute PRINT, enclosing quotation marks are not supplied, and contained single quotation marks or apostrophes are unmodified. The field width is the current length of the string.

Examples of list-directed data specifications:

```
LIST (CARD, RATE, DYNAMIC_FLOW)

LIST ((THICKNESS(DISTANCE)
   DO DISTANCE = 1 TO 1000))

LIST (P, Z, M, R)

LIST (A*B/C, (X+Y)**2)
```

The specification in the last example can be used only for output, since it contains operational expressions. Such expressions are evaluated when the statement is executed, and the result is placed in the stream.

DATA-DIRECTED DATA SPECIFICATION

General format for a data-directed data specification, either for input or output, is as follows:

Option 1: DATA

Option 2: DATA (data-list)

General rules:

1. The data list is described in "Data Lists" in this chapter. It cannot include parameters, defined variables, or based variables. For input, the data list cannot contain subscripted names. Names of structure elements in the data list need only have enough qualification to resolve any ambiguity; full qualification is not required.

2. Option 1 implies that a data list is assumed. This assumed data list contains all the names that are known to the block and are valid for data-directed transmission. On input, if the stream contains a name not known within the block, the NAME condition is raised. If the assumed data list contains a name that is not included in the stream, the value of the associated variable remains unchanged. On output, all items in the assumed data list are transmitted.

3. Recognition of a semicolon or an end of file in the stream on input causes transmission to cease, whether or not a data list is specified. On output, a semicolon is written into the stream after the last data item transmitted by each PUT statement.

Data-Directed Data in the Stream

The data in the stream associated with data-directed transmission is in the form of a list of element assignments having the following general format (the optionally signed constants, like the variable names and the equal signs, are in character form):

```
element-variable = constant
  [{b|,}element-variable = constant]...;
```

General rules:

1. The element variable may be a subscripted name. Subscripts must be optionally signed decimal integer constants.

2. On input, the element assignments may be separated by either a blank (b in the above format) or a comma. Redundant blanks are ignored. On output, the assignments are separated by a blank.

3. Each constant in the stream has one of the forms described for list-directed transmission.

Data-Directed Input Format

General rules for data-directed input:

1. If the data specification in option 1 is used, the names in the stream may be any names known at the point of transmission. Qualified names in the input stream must be fully qualified.

2. If option 2 is used, each element of the data list must be an element, array, or structure variable. Names cannot be subscripted, but qualified names are allowed in the data list. All names in the stream should appear in the data list; however, the order of the names need not be the same, and the data list may include names that do not appear in the stream. If a name appears in the stream but not in the data list, the NAME condition is raised.

   For example, consider the following data list, where A, B, C, and D are names of element variables:

   ```
   DATA (B, A, C, D)
   ```

   This data list may be associated with the following input data stream:

   ```
   A= 2.5, B= .0047, D= 125, Z= 'ABC';
   ```

   Note: C appears in the data list but not in the stream, and Z, not in the data list, will raise the NAME condition. The value of C will be unaltered.

3. If the data list in option 2 includes the name of an array, subscripted references to that array may appear in the stream although subscripted names cannot appear in the data list. The entire array need not appear in the stream; only those elements that actually appear in the stream will be assigned.

   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. In this case, only three elements are assigned; the remainder of the array is unchanged.

4. If the data list includes the names of structure elements, then fully qualified names must appear in the stream, although full qualification is not required in the data list. Consider the following structures:

   ```
   DECLARE 1 CARDIN, 2 PARTNO, 2 DESCRP,
           2 PRICE, 3 RETAIL, 3 WHSL,
           1 CARDOUT, 2 PARTNO, 2 DESCRP,
           2 PRICE, 3 RETAIL, 3 WHSL;
   ```

   If it is desired to read a value for CARDIN.PRICE.RETAIL, the data specification and input data stream could have the following forms:

   ```
   Data Specification   Input Data Stream
   DATA (CARDIN.RETAIL) CARDIN.PRICE.
                          RETAIL = 4.28;
   ```

5. Interleaved subscripts cannot appear in qualified names in the stream. All subscripts must be moved all the way to the right, following the last name of the qualified name. For example, assume that Y is declared as follows:

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

   An element name would have to appear in the stream as follows:

   ```
   Y.A.B(2,3,8)= 8.72
   ```

   The name in the data list, of course, could not contain the subscript.

Data-Directed Output Format

General rules for data-directed output:

1. An element of the data list may be an element, array, or structure variable, or a repetitive specification involving any of these elements or further repetitive specifications. Subscripted names can appear. The names appearing in the data list, together with their values, are transmitted in the form of a list of element assignments separated by blanks and terminated by a semicolon. (For PRINT files, items are separated according to program tab settings.)

2. Array variables in the data list are treated as a list of the contained

subscripted elements in row-major order.

Let X be an array declared as follows:

```
DECLARE X (2,4);
```

Let X appear in a data list as follows:

```
DATA (X)
```

Then, on output, the output data stream would be 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;
```

Note: In actual output, more than one blank would follow the equal sign. In conversion from coded arithmetic to character, leading zeros are converted to blanks, and up to three additional blanks may appear at the beginning of the field.

3. Subscript expressions that appear in a data list are evaluated and replaced by the value.

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 element or array elements, and any contained arrays are treated as above.

Consider the following structure:

```
1 A, 2 B, 2 C, 3 D
```

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

```
DATA (A)
```

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. In the following cases, data-directed output is not valid for subsequent data-directed input:

   a. When the precision attribute of a fixed-point variable is such that the assumed point is located outside the field with assumed zeros intervening; that is, if for precision (p,q) p is less than q, or q is less than zero. (In this case an exponent is transmitted, preceded by a letter F which is not valid for conversion to arithmetic type.)

   b. When the character-string value of a numeric character variable does not represent a valid optionally signed arithmetic constant. For example, this is always true for complex numeric character variables.

Length of Data-Directed Output Fields

The length of the data field on the external medium is a function of the attributes declared for the variable and, since the name is also included, the length of the fully qualified subscripted name. The field length for output items converted from coded arithmetic data, numeric character data, and bit-string data is the same as that for list-directed output data, and is governed by the rules for data conversion to character type as described in Part II, Section F, "Problem Data Conversion."

For character-string data, the contents of the character string are written out enclosed in quotation marks. Each quotation mark or apostrophe contained within the character string is represented by two successive quotation marks.

In the example shown in Figure 8-2, assume that A is declared as a one-dimensional array of six elements; B is a one-dimensional array of seven elements. The procedure calculates and writes out values for A(I) = B(I+1) + B(I).

EDIT-DIRECTED DATA SPECIFICATION

General format for an edit-directed data specification, either for input or output, is as follows:

```
EDIT (data-list) (format-list)
     [(data-list)(format-list)]...
```

```
AB:  PROCEDURE;
                                    Input Stream
     DECLARE A(6), B(7);
                                    B(1)=1, B(2)=2, B(3)=3,
     GET FILE (X) DATA (B);
                                    B(4)=1, B(5)=2, B(6)=3, B(7)=4;
     DO I = 1 TO 6;

     A (I) = B (I+1) + B (I);
                                    Output Stream
     END;
                                    A(1)= 3 A(2)= 5 A(3)= 4 A(4)= 3
     PUT FILE (Y) DATA (A);
                                    A(5)= 5 A(6)= 7;
     END AB;
```

Figure 8-2. Example of Data-Directed Transmission (Both Input and Output).

1. The data list, which must be enclosed in parentheses, is described above in "Data Lists." The format list, which also must be enclosed in parentheses, contains one or more format items. There are three types of format items: data format items, which describe data in the stream; control format items, which describe page, line, and spacing operations; and remote format items, which specify the label of a separate statement that contains the format list to be used. Format lists and format items are discussed in more detail in "Format Lists," below. Edit-directed transmission is the only mode that can be used for reading or writing sterling data, by use of a picture specification.

2. For input, data in the stream is considered to be a continuous string of characters not separated into individual data items. The number of characters for each data item is specified by a format item in the format list. The characters are treated according to the associated format item.

3. For output, the value of each item in the data list is converted to a format specified by the associated format item and placed in the stream in a field whose width also is specified by the format item.

4. For either input or output, the first data format item is associated with the first item in the data list, the second data format item with the second item in the data list, and so forth. If a format list contains fewer format items than there are items in the associated data list, the format list is re-used; if there are excessive format items, they are ignored. Suppose a format list contains five data format items and its associated data list specifies ten items to be transmitted. Then the sixth item in the data list will be associated with the first data format item, and so forth. Suppose a format list contains ten data format items and its associated data list specifies only five items. Then the sixth through the tenth format items will be ignored.

5. An array or structure variable in a data list is equivalent to n items in the data list, where n is the number of element items in the array or structure, each of which will be associated with a separate use of a data format item.

6. 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.

7. The specified transmission is complete when the last item in the data list has been processed using its corresponding format item. Subsequent format items, including control format items, are ignored.

8. On output, data items are not automatically separated, but arithmetic data items generally include leading blanks because of data conversion rules and zero suppression.

Examples:

```
GET EDIT (NAME, DATA, SALARY)
    (A(N), X(2), A(6), F(6,2));

PUT EDIT ('INVENTORY='||INUM,INVCODE)
    (A,F(5));
```

The first example specifies that the first N characters in the stream are to be treated as a character string and assigned to NAME; the next two characters are to be skipped; the next six are to be assigned to DATA in character format; and the next six characters are to be considered as an optionally signed decimal fixed-point constant and assigned to SALARY.

The second example specifies that the character string 'INVENTORY=' is to be concatenated with the value of INUM and placed in the stream in a field whose width is the length of the resultant string. Then the value of INVCODE is to be converted to character to represent an optionally signed decimal fixed-point integer constant and is then to be placed in the stream right-adjusted in a field with a width of five characters (leading characters may be blanks). Note that operational expressions and constants can appear in output data lists only.

Format Lists

Each edit-directed data specification requires its own format list.

General format:

(format-list)

where "format list" is defined as:

```
{ item                 }{ [, item                 ] }...
{ n item               }{ [, n item               ] }
{ n (format-list)      }{ [, n (format-list)      ] }
```

Syntax rules:

1. Each "item" represents a format item as described below.

2. The letter n represents an iteration factor, which is either an expression enclosed in parentheses or an unsigned decimal integer constant. If it is the latter, a blank must separate the constant and the following format item. The iteration factor specifies that the associated format item or format list is to be used n successive times. A zero or negative iteration factor specifies that the associated format item or format list 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 or format list is that item or list of items immediately to the right of the iteration factor.

General rule:

There are three types of format items: data format items, control format items, and the remote format item. Data format items specify the external forms that data fields are to take. Control format items specify the page, line, column, and spacing operations. The remote format item allows format items to be specified in a separate FORMAT statement elsewhere in the block.

Detailed discussions of the various types of format items appear in Part II, Section E, "Edit-Directed Format Items." The following discussions show how the format items are used in edit-directed data specifications.

Data Format Items

On input, each data format item specifies the number of characters to be associated with the data item and how to interpret the external data. The data item is assigned to the associated variable named in the data list, with necessary conversion to conform to the attributes of the variable. On output, the value of the associated element in the data list is converted to the character representation specified by the format item and is inserted into the data stream.

There are six data format items: fixed-point (F), floating-point (E), complex (C), picture (P), character-string (A), and bit-string (B). They are, in general, specified as follows:

```
F (w[,d[,p]])

E (w,d[,s])

C (real-format-item [,real-format-item])

P 'picture-specification'

A [(w)]

B [(w)]
```

In this list, the letter w represents an expression that specifies the number of characters in the field. The letter d specifies the number of digits to the right of a decimal point; it may be omitted for integers. The real format item of the complex format item represents the appear-

ance of either an F, E or P format item. The picture specification of the P format item can be either a numeric character specification or a character-string specification. On output, data associated with E and F format items is rounded if the internal precision exceeds the external precision.

A third specification (p) is allowed in the F format item; it is a scaling factor. A third specification (s) is allowed in the E format item to specify the number of digits that must be maintained in the first subfield of the floating-point number. These specifications are discussed in detail in Part II, Section E, "Edit-Directed Format Items."

Note: Fixed-point binary and floating-point binary data items must always be represented in the input stream with their values expressed in decimal digits. The F and E format items then are used to access them, and the values will be converted to binary representation upon assignment. On output, binary items are converted to decimal values and the associated F or E format items must state the field width and point placement in terms of the converted decimal number.

The following examples illustrate the use of format items:

1. GET FILE (INFILE) EDIT (ITEM) (A(20));

   This statement causes the next 20 characters in the file called INFILE to be assigned to ITEM. The value is automatically transformed from its character representation specified by the format item A(20), to the representation specified by the attributes declared for ITEM.

   Note: If the data list and format list were used for output, the length of a string item need not be specified in the format item if the field width is to be the same as the length of the string, that is, if no blanks are to follow the string.

2. PUT FILE (MASKFLE) EDIT (MASK) (B);

   Assume MASK has the attributes BIT (25); then the above statement writes the value of MASK in the file called MASKFLE as a string of 25 characters consisting of 0's and 1's. A field width specification can be given in the B format item. It must be stated for input.

3. PUT EDIT (TOTAL) (F(6,2));

   Assume TOTAL has the attributes FIXED (4,2); then the above statement specifies that the value of TOTAL is to be converted to the character representation of a fixed-point number and written into the standard output file SYSPRINT. A decimal point is to be inserted before the last two numeric characters, and the number will be right-adjusted in a field of six characters. Leading zeros will be changed to blanks, and, if necessary, a minus sign will be placed to the left of the first numeric character.

   In conversion from internal decimal fixed-point type to character type, the resultant string always is three characters longer than p, the number of digits in the precision specification of a decimal fixed-point variable. The extra characters may appear as blanks preceding the number in the converted string. And, since leading zeros are converted to blanks, additional blanks may precede the number. If a decimal point or a minus sign appears, either will cause one leading blank to be replaced.

   In edit-directed output, the field width specification in the format item (in this case, the 6 in the F(6,2) format item) can be used to truncate leading zeros. In this specification, one zero is truncated. TOTAL would be converted to a character string of length seven. If all four digits of the converted number are greater than zero, the number, with its inserted decimal point, will require five digit positions; if the number is negative, another digit position will be required for the minus sign. Consequently, the F(6,2) specification will always allow all digits, the point, and a possible sign to appear, but will remove the extra blank by truncation.

4. GET FILE(A) EDIT (ESTIMATE) (E(10,6));

   This statement obtains the next ten characters from the file called A and interprets them as a floating-point decimal number. A decimal point is assumed before the rightmost six digits of the mantissa. An actual point within the data can override this assumption. The value of the number is converted to the attributes of ESTIMATE and assigned to this variable.

5. GET EDIT (NAME, TOTAL)
   (P'AAAAA',P'9999');

   When this statement is executed, the standard input file SYSIN is assumed.

The first five characters must be alphabetic or blank and they are assigned to NAME. The next four characters must be nonblank numeric characters and they are assigned to TOTAL.

Control Format Items

The control format items are the spacing format item (X), and the COLUMN, LINE, PAGE, and SKIP format items. The spacing format item specifies relative spacing in the data stream. The PAGE and LINE format items can be used only with PRINT files and, consequently, can only appear in PUT statements. All but PAGE generally include expressions. LINE, PAGE, and SKIP can also appear separately as options in the PUT statement; SKIP can appear as an option in the GET statement.

The following examples illustrate the use of the control format items:

1.
```
GET EDIT (NUMBER, REBATE)
         (A(5), X(5), A(5));
```

This statement treats the next 15 characters from the standard input file, SYSIN, as follows: the first five characters are assigned to NUMBER, the next five characters are spaced over and ignored, and the remaining five characters are assigned to REBATE.

2.
```
GET FILE(IN) EDIT(MAN,OVERTIME)
  (SKIP(1), A(6), COLUMN(60), F(4,2));
```

This statement positions the data set associated with file IN to a new line; the first six characters on the line are assigned to MAN, and the four characters beginning at character position 60 are assigned to OVERTIME.

3.
```
PUT FILE(OUT) EDIT (PART, COUNT)
         (A(4), X(2), F(5));
```

This statement places in the file named OUT four characters that represent the value of PART, then two blank characters, and finally five characters that represent the fixed-point value of COUNT.

4. The following examples show the use of the COLUMN, LINE, PAGE, and SKIP format items in combination with one other.

```
PUT EDIT ('QUARTERLY STATEMENT')
    (PAGE, LINE(2), A(19));
```

```
PUT EDIT
    (ACCT#, BOUGHT, SOLD,
     PAYMENT, BALANCE)
    (SKIP(3), A(6), COLUMN(14),
     F(7,2), COLUMN(30), F(7,2),
     COLUMN(45), F(7,2),
     COLUMN(60), F(7,2));
```

The first PUT statement specifies that the heading QUARTERLY STATEMENT is to be written on line two of a new page in the standard output file SYSPRINT. The second statement specifies that two lines are to be skipped (that is, "skip to the third following line") and the value of ACCT# is to be written, beginning at the first character of the fifth line; the value of BOUGHT, beginning at character position 14; the value of SOLD, beginning at character position 30; the value of PAYMENT, beginning at character position 45; and the value of BALANCE at character position 60.

Note: Control format items are executed at the time they are encountered in the format list. Any control format list that appears after the data list is exhausted will have no effect.

Remote Format Item

The remote format item (R) specifies the label of a FORMAT statement (or a label variable whose value is the label of a FORMAT statement) located elsewhere; the FORMAT statement and the GET or PUT statement specifying the remote format item must be internal to the same block. The FORMAT statement contains the remotely situated format items. This facility permits the choice of different format specifications at execution time, as illustrated by the following example:

```
DECLARE SWITCH LABEL;
GET FILE(IN) LIST(CODE);
IF CODE = 1
    THEN SWITCH = L1;
    ELSE SWITCH = L2;
GET FILE(IN) EDIT (W,X,Y,Z)
    (R(SWITCH));
L1: FORMAT (4 F(8,3));
L2: FORMAT (4 E(12,6));
```

SWITCH has been declared to be a label variable; the second GET statement can be made to operate with either of the two FORMAT statements.

Expressions in Format Items

The w, p, d, and s specifications in data format items, as well as the specifications in control format items, need not be decimal integer constants. Expressions are allowed. They may be variables or other expressions.

On input, a value read into a variable can be used in a format item associated with another variable later in the data list.

```
PUT EDIT (NAME,NUMBER,CITY)
    (A(N),A(N-4),A(10));

GET EDIT (M,STRING_A,I,STRING_B)
    (F(2),A(M),X(M),F(2),A(I));
```

In the first example, the value of NAME is inserted in the stream as a character string left-adjusted in a field of N characters; NUMBER is left-adjusted in a field of N-4 characters; and CITY is left-adjusted in a field of 10 characters. In the second example, the first two characters are assigned to M. The value of M is then taken to specify the number of characters to be assigned to STRING_A and also to specify the number of characters to be ignored before two characters are assigned to I, whose value then is used to specify the number of characters to be assigned to STRING_B.

STREAM-ORIENTED DATA TRANSMISSION STATEMENTS

The following provides a summary of the STREAM data transmission statements, along with their options, according to file attributes (the statements are discussed individually in detail in Part II, Section J, "Statements").

STREAM INPUT:

```
GET [FILE (file-name)]
    data-specification [COPY]
    [SKIP [(expression)]];
```

STREAM OUTPUT:

```
PUT [FILE (file-name)]
    data-specification
    [SKIP [(expression)]];
```

STREAM OUTPUT PRINT:

```
PUT [FILE (file-name)]
    [data-specification]
    [PAGE [LINE(expression)]
     SKIP[(expression)]
     LINE  (expression)     ];
```

Note: The data specification can be omitted for STREAM OUTPUT PRINT files only if one of the control options appears.

In all of the above, 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)]...
```

The COPY option for STREAM INPUT specifies that each data item is to be written, exactly as read, into the standard output file SYSPRINT.

Format lists may use any of the following format items:

A,B,C,E,F, P,R,X, SKIP [(w)] COLUMN (w)	which may be used with any STREAM file
PAGE LINE (w)	which can be used with STREAM OUTPUT PRINT files only

Note that for non-PRINT files, the expression w in the SKIP option must have a value that is greater than zero when converted to an integer. If it has not, the F Compiler substitutes a value of 1.

RECORD-ORIENTED TRANSMISSION

Data sets that contain discrete records, or which are to be created as collections of discrete records, may be manipulated with record-oriented operation statements. These statements are READ, WRITE, REWRITE, LOCATE, and DELETE. A general description of each of these statements is contained in this chapter; they are described in detail in Part II, Section J, "Statements." Each record obtained from a data set or transmitted to a data set is defined in terms of the data attributes of a variable (usually a structure). For input operations, the record is obtained from the data set and assigned, without conversion, to the variable. For output operations, the data is transmitted without conversion into the data set.

The variables involved in record transmission must be unsubscripted, of level 1 (element and array variables not contained in structures are of level 1 by default), and may be of any storage class. The variables cannot be parameters or defined variables. They may be label, pointer, or

event variables, but such data may lose its validity in transmission.

RECORD-ORIENTED DATA TRANSMISSION STATEMENTS

There are four statements that actually cause transmission of records to or from external storage. They are READ, WRITE, LOCATE, and REWRITE. A fifth statement, the DELETE statement, is used to delete records from an UPDATE file. The attributes of the file determine which statements can be used.

The READ statement can be used with any INPUT or UPDATE file. It causes a record to be transmitted from the data set to the program, either directly to a variable or to a buffer. In the case of blocked records, the READ statement causes a logical record to be transferred from a buffer to the variable. For blocked records, consequently, every READ statement may not cause physical input.

The WRITE statement can be used with any OUTPUT file, and with DIRECT UPDATE, but not with SEQUENTIAL UPDATE. It causes a record to be transmitted from the program to the data set. For unblocked records, the transmission may be directly from a variable or from a buffer. For blocked records, the WRITE statement causes a logical record to be placed into a buffer. Only when the blocking of the record is complete is there actual physical output.

The REWRITE statement causes a record to be replaced in an UPDATE file. For SEQUENTIAL UPDATE files, the REWRITE statement specifies that the last record read from the file is to be rewritten; consequently a record must be read before it can be rewritten. For DIRECT UPDATE files, the REWRITE statement must specify a key; consequently, any record can be rewritten whether or not it has first been read.

The LOCATE statement can be used only with an OUTPUT SEQUENTIAL BUFFERED file. It allocates storage within an output buffer for a based variable, setting a pointer to the location in the buffer as it does so. This pointer can then be used to refer to the allocation so that data can be moved into the buffer. The record is written out automatically, immediately before execution of the next WRITE or LOCATE statement for the file, or before the file is closed. See also Chapter 14, "Based Storage and List Processing."

The DELETE statement specifies that a record in an UPDATE file be marked as deleted. For a DIRECT UPDATE file, the DELETE statement must specify a key; consequently, any record can be deleted.

Options of Record-Oriented Transmission Statements

Options that are allowed for record-oriented data transmission statements differ according to the attributes of the associated file and the purpose of the statement. A list of all of the allowed combinations for each type of file is given later in this chapter.

Each option consists of a keyword followed by a value, which is a file name, a variable, or an expression. This value always must be enclosed in parentheses. In any statement, the options may appear in any order.

The FILE Option

The FILE option must appear in every record-oriented statement. It specifies the name of the file upon which the operation is to take place. It consists of the keyword FILE followed by the file name enclosed in parentheses. An example of the FILE option is shown in each of the statements in this section.

The INTO Option

The INTO option can be used in the READ statement for any type of INPUT or UPDATE file. The INTO option specifies a variable to which the logical record is to be assigned.

```
READ FILE (DETAIL) INTO (RECORD_1);
```

This specifies that the next sequential record is to be assigned to the variable RECORD_1.

Note that the INTO option can name an element string variable of varying length; thus it is possible to read a record whose length is unknown and cannot be determined from the data. The current length of the string is set to the length of the record. The LENGTH built-in function can be used to find the length of the record.

The FROM Option

The FROM option must be used in the WRITE statement for any OUTPUT file and for a DIRECT UPDATE file. It also can be used in the REWRITE statement for any UPDATE file. The FROM option specifies the variable from which the record is to be written. If this variable is a string of varying length, the current length of the string determines the size of the record.

For files other than DIRECT UPDATE or SEQUENTIAL UNBUFFERED UPDATE files, the FROM option can be omitted. If the last record was read by a READ statement with the INTO option, REWRITE without FROM 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, in the buffer, by whatever assignments were made.

```
WRITE FILE (MASTER) FROM (MAS_REC);
```

```
REWRITE FILE (MASTER) FROM (MAS_REC);
```

Both statements specify that the value of the variable MAS_REC is to be written into the file MASTER. In the case of the WRITE statement, it specifies a new record in a SEQUENTIAL OUTPUT file.

The REWRITE statement specifies that MAS_REC is to replace the last record read from a SEQUENTIAL UPDATE file.

The SET Option

The SET option can be used with a READ statement or a LOCATE statement. It specifies that a named pointer variable is to be set to point to the location in the buffer into which data has been moved during the READ operation, or which has been allocated by the LOCATE statement.

For detailed information, see Chapter 14, "Based Storage and List Processing."

The IGNORE Option

The IGNORE option can be used in a READ statement for any SEQUENTIAL INPUT or UPDATE file. It includes an expression whose integral value specifies a number of records to be skipped over and ignored.

```
READ FILE (IN) IGNORE (3);
```

This statement specifies that the next three records in the file are to be skipped.

The KEY Option

The KEY option applies only to files associated with data sets of REGIONAL or INDEXED organization. It can be used in the READ statement for files with the INPUT or UPDATE attributes. (If the file has the SEQUENTIAL attribute, the associated data set must be of INDEXED organization.) The KEY option also is used in the REWRITE and DELETE statements for DIRECT UPDATE files. Any file for which the KEY option is used must also have the KEYED attribute.

The KEY option consists of the keyword KEY followed by a parenthesized expression, which is a source key that identifies a particular record. The expression may be a character-string constant, a variable, or any other element expression. An expression is evaluated and converted to a character string.

Following is a summary of what the character string is and what it represents for each of the data set organizations to which it is applicable:

REGIONAL (1)	A string of characters consisting of digits or blanks, the rightmost eight of which specify the relative record number of the desired record.
REGIONAL (2)	A string of characters, the rightmost eight of which must consist of digits or blanks. These rightmost eight characters specify a relative record number that is the beginning of a region to be searched. The record to be accessed is identified by a recorded key that exactly matches that portion of the source key character string whose length, beginning from the left, extends the number of characters specified in the KEYLEN subparameter of the associated DD statement. This string may or may not include the rightmost eight characters.

REGIONAL (3)	Same as REGIONAL (2), except that the rightmost eight characters specify a relative track that is the beginning of the region to be searched.
INDEXED	A string of characters, the first n of which exactly match the recorded key of the record (where n is the number specified by KEYLEN). If the recorded key is embedded in the record and the record is blocked, its location must be specified in the RKP subparameter of the DD statement.

If, for a REGIONAL(1) data set, the source key is longer than eight characters, other characters to the left are ignored. If the source key character string of a REGIONAL (2), REGIONAL (3), or INDEXED data set is shorter than the length specified by KEYLEN, it is extended on the right with blanks before a comparison is made. If longer, it is truncated on the right before comparison.

```
READ FILE (MASTER) INTO (MAS_REC) KEY
   ('00003253');

DELETE FILE (FILEX) KEY (NAME||AREA#);

REWRITE FILE (FILEZ) FROM (PAY_REC)
   KEY (NAME);
```

The first statement specifies that record number 3253 in the REGIONAL (1) data set associated with the file MASTER is to be read and assigned to the variable MAS_REC.

The second statement, which would be appropriate for either a REGIONAL (2) or REGIONAL (3) data set, specifies that a record is to be deleted from the DIRECT UPDATE file FILEX. The record is to be found in a region identified by the value of AREA#, which, in this case, must be a number of at least eight digits. The specific record is to be recognized by a recorded key that matches all or the first portion of the concatenated string of the length specified by KEYLEN.

The third statement, which would be appropriate for an INDEXED data set, specifies that the value of the variable PAY_REC is to be written in the DIRECT UPDATE file FILEZ. It is to replace the record which has a recorded key that matches the value of NAME. NAME could be an element of PAY_REC, assuming that PAY_REC has been declared to be a structure.

Note: The fact that certain data set organizations are mentioned in connection with the above examples does not mean that an example implies a specific organization; what is meant is that the example is typical of that organization.

The KEYFROM Option

The KEYFROM option must be specified in a WRITE statement used to create a REGIONAL or INDEXED data set or in a WRITE statement used to add new records to an INDEXED or REGIONAL data set. It cannot be used with CONSECUTIVE organization. Therefore, it can appear in a WRITE statement for a SEQUENTIAL OUTPUT file, either BUFFERED or UNBUFFERED, or for a DIRECT OUTPUT or DIRECT UPDATE file. It can be used with a LOCATE statement. Any file for which the KEYFROM option is specified must have the KEYED attribute.

The KEYFROM option specifies the location, within the data set, where the record is to be written. For REGIONAL(1) data sets, it specifies only the region number. For REGIONAL(2) and REGIONAL(3) data sets, it also specifies a character string to be written as a recorded key. For INDEXED data sets, it specifies a recorded key, whose value is used to determine the location. It is written with the keyword KEYFROM followed by a parenthesized expression. The expression can be a constant, a variable, or any other expression. The value is always converted to a character string. For all but REGIONAL(1), the KEYLEN DD statement subparameter must specify the length of the recorded key to be written.

```
WRITE FILE (PAYROLL) FROM (PAY_REC)
   KEYFROM (NAME||COUNTER+1);

WRITE FILE (LOANS) KEYFROM (LOAN#)
   FROM (LOAN_REC);
```

The first statement, which could be appropriate for a REGIONAL (2) data set, specifies that the value of PAY_REC is to be written as the next sequential record in the file PAYROLL. The value of COUNTER+1 specifies the region immediately following that in which the last record was written. The source key is to be a concatenation of the value of NAME and the value of the expression COUNTER+1, with the first n characters to be written as the recorded key (the value of n must be specified by KEYLEN).

The second statement specifies that the value of LOAN_REC is to be written as the

next record in the file LOANS, with the value of LOAN# to be used as the key.

The KEYTO Option

The KEYTO option can be used in the READ statement for a SEQUENTIAL INPUT or UPDATE file that is associated with a REGIONAL or INDEXED data set. It is specified by writing the keyword KEYTO, followed by the parenthesized name of a character-string variable.

For REGIONAL(1) data sets, the KEYTO option specifies that the region number of the record being read is to be assigned to the specified variable. For REGIONAL(2), REGIONAL(3), or INDEXED data sets, it specifies that only the recorded key is to be assigned. Any file for which the KEYTO option is specified must have the KEYED attribute. For example:

```
READ FILE(DETAIL) INTO (INVTRY)
   KEYTO(KEY_CHK);
```

The first statement specifies that the next record in the file DETAIL is to be assigned to INVTRY and the key of the record is to be assigned to KEY_CHK.

The EVENT Option

The EVENT option can be specified in any READ, WRITE, REWRITE, or DELETE statement for an UNBUFFERED file, either SEQUENTIAL or DIRECT. The option specifies asynchronous processing, with input/output operations proceeding while other processing continues.

The EVENT option is specified with the keyword EVENT followed by a parenthesized name of an event variable. The appearance of an event variable in the EVENT option constitutes a contextual declaration; consequently, the scope of an event variable is throughout the external block.

The EVENT option specifies that the input or output operation is to take place asynchronously and that record I/O interrupts (except for UNDEFINEDFILE) are not to occur until a WAIT statement, specifying the same event variable, is executed by the same task. For example:

```
READ FILE (MASTER) INTO (REC_VAR)
     EVENT (RECORD_1);
        .
        .
        .
WAIT (RECORD_1);
```

When the READ statement is executed, the input operation is started. As soon as the input operation is commenced, in-line processing continues. No I/O interrupt for RECORD, TRANSMIT, KEY, or ENDFILE conditions will take place until the WAIT statement is executed. If, when the WAIT statement is executed, the input operation is not complete, and if none of the four conditions is raised, in-line processing stops, but the operation continues. When the operation is successfully completed, processing continues with the next statement following the WAIT statement. If any of the four conditions arise during execution of the READ statement, an interrupt will occur when the WAIT statement is executed. On-units will be entered in the order in which the conditions arose. Then, upon normal return from all of the affected on-units, processing continues with the next statement following the WAIT statement.

Note that although the EVENT option specifies asynchronous processing, it also specifies synchronous interrupts; none of the four conditions can cause an interrupt until they are synchronized with processing by the WAIT statement.

Other interrupts can occur, however. Any condition that arises during the in-line processing will, of course, cause an interrupt if it is enabled. In addition, if the I/O statement containing the EVENT option should cause implicit opening of the file, and if the UNDEFINEDFILE condition should arise because of that implicit opening, the interrupt will occur at the time the UNDEFINEDFILE condition is raised. Only the four conditions TRANSMIT, KEY, RECORD, and ENDFILE can be synchronized by the WAIT statement.

Once a statement containing an EVENT option is executed, the event variable named in the option is considered to be active. It cannot be specified again in an EVENT option until after its corresponding WAIT statement has been executed.

With the F Compiler, an input/output event should be waited for only by the task that initiated the input/output operation.

(The EVENT option is also used with the CALL statement to specify asynchronous execution of procedures. See Chapter 15, "Multitasking.")

Record-Oriented Transmission Statement Formats

This section provides a summary of the allowed RECORD transmission statements, along with their options, according to file

attributes. Options can appear in any order.

SEQUENTIAL BUFFERED INPUT:

```
READ FILE (file-name)
   INTO (variable)
   [KEYTO (character-string-variable)];

READ FILE (file-name)
   [IGNORE (expression)];

READ FILE (file-name)
   INTO (variable)
   KEY (expression);

READ FILE (file-name)
   SET (pointer-variable)
   [KEYTO (character-string-variable)];

READ FILE (file-name)
   SET (pointer-variable)
   KEY (expression);
```

SEQUENTIAL BUFFERED OUTPUT:

```
WRITE FILE (file-name)
   FROM (variable)
   [KEYFROM (expression)];

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

SEQUENTIAL BUFFERED UPDATE:

```
READ FILE (file-name)
   INTO (variable)
   [KEYTO (character-string-variable)];

REWRITE FILE (file-name);

REWRITE FILE (file-name)
   FROM (variable);

READ FILE (file-name)
   [IGNORE (expression)];

READ FILE (file-name)
   INTO (variable)
   KEY (expression);

READ FILE (file-name)
   SET (pointer-variable)
   [KEYTO (character-string-variable)];

READ FILE (file-name)
   SET (pointer-variable)
   KEY (expression);

DELETE FILE (file-name);
```

SEQUENTIAL UNBUFFERED INPUT:

```
READ FILE (file-name)
   INTO (variable)
   [KEYTO (character-string-variable)]
   [EVENT (event-variable)];

READ FILE (file-name)
   [IGNORE (expression)]
   [EVENT (event-variable)];

READ FILE (file-name)
   INTO (variable)
   KEY (expression)
   [EVENT (event-variable)];
```

SEQUENTIAL UNBUFFERED OUTPUT:

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

SEQUENTIAL UNBUFFERED UPDATE:

```
READ FILE (file-name)
   INTO (variable)
   [KEYTO (character-string-variable)]
   [EVENT (event-variable)];

REWRITE FILE (file-name)
   FROM (variable)
   [EVENT (event-variable)];

READ FILE (file-name)
   [IGNORE (expression)]
   [EVENT (event-variable)];

READ FILE (file-name)
   INTO (variable)
   KEY (expression)
   [EVENT (event-variable)];

DELETE FILE (file-name)
   [EVENT (event-variable)];
```

DIRECT INPUT:

```
READ FILE (file-name)
   INTO (variable)
   KEY (expression)
   [EVENT (event-variable)];
```

DIRECT OUTPUT:

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

DIRECT UPDATE:

```
READ FILE (file-name)
   INTO (variable)
   KEY (expression)
   [EVENT (event-variable)];

REWRITE FILE (file-name)
   FROM (variable)
   KEY (expression)
   [EVENT (event-variable)];
```

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

DELETE FILE (file-name)
   KEY (expression)
   [EVENT (event-variable)];
```

DIRECT UPDATE EXCLUSIVE

```
READ FILE (file-name)
   INTO (variable)
   KEY (expression) [NOLOCK]
   [EVENT (event-variable)];

REWRITE FILE (file-name)
   FROM (variable)
   KEY (expression)
   [EVENT (event-variable)];

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

DELETE FILE (file-name)
   KEY (expression)
   [EVENT (event-variable)];

UNLOCK FILE (file-name)
   KEY (expression);
```

Summary of Record-Oriented Transmission

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 (or from the last to the first if the BACKWARDS attribute has been specified).

2. A DIRECT file specifies that the accessing, creation, or modification of the data set records may be performed in random order. The particular record of the data set to be operated upon is identified by a specified key.

3. A data set that is accessed, created, or modified in the SEQUENTIAL access method may or may not have recorded keys. If it does, 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 KEYFROM and KEYTO options. The most efficient way to create a data set containing recorded keys is as a SEQUENTIAL OUTPUT file. It then can be accessed 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 has INDEXED organization and if the file has the KEYED attribute.

5. Existing records of a data set in a SEQUENTIAL UPDATE file can be rewritten, modified, ignored, or deleted. The DELETE statement used with this type of file specifies that the last record read is to be deleted.[1] Operation with a DIRECT UPDATE file, however, can specify which record is to be deleted by means of a key; also, records can be added to the data set by means of the WRITE statement. An existing record in an UPDATE file can be replaced through use of a REWRITE statement.

6. The FROM option in a REWRITE statement for a SEQUENTIAL UPDATE must specifically name the variable into which the data has been read if that data is to be rewritten. For the F Compiler, a REWRITE statement without a FROM option is treated as a currently null statement.

7. 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 the data set. Consequently, record protection is provided only if all tasks refer to the data set through use of

---

[1] If the DELETE statement is used with a SEQUENTIAL file, the data set must have INDEXED organization.

the same file; if they refer to the same data set using different files, the protection does not apply. In addition, the data set to which reference is made by more than one task through the same file must be opened by a parent of all these tasks. Note that a reference to a file parameter and its associated argument are references to the same file.

8. A WRITE statement adds a record to a data set, while a REWRITE statement replaces a record. Thus, a WRITE statement may be used with OUTPUT files, and DIRECT UPDATE files, but a REWRITE statement may be used with UPDATE files only. Moreover, for DIRECT files, a REWRITE statement uses the KEY option to identify the existing record to be replaced; a WRITE statement uses the KEYFROM option, which not only specifies where the record is to be written in the data set, but also specifies, except for REGIONAL (1), an identifying key to be recorded in the data set.

9. Records of a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file can be skipped over and ignored by use of the IGNORE option of a READ statement. The expression of the IGNORE option specifies the number of records to be skipped. A READ statement in which only the FILE option appears indicates that one record is to be skipped.

EXAMPLES OF DECLARATIONS FOR RECORD FILES

Following are examples of declarations of files, including the ENVIRONMENT attribute:

```
DECLARE FILE#3 INPUT DIRECT
   ENVIRONMENT(V(328) REGIONAL(3));
```

This declaration specifies only three file attributes: INPUT, DIRECT, and ENVIRONMENT. Other implied attributes are FILE (implied by any of the attributes) and RECORD and KEYED (implied by DIRECT). Scope is EXTERNAL, by default. The ENVIRONMENT attribute specifies that the data set is of the REGIONAL(3) organization and contains unblocked varying-length records with a maximum length of 328 bytes. Note that a maximum length record will contain only 320 bytes of data to be used by the program, because 8 bytes are required for control information in such V-format records. The KEY option must be specified in each READ statement that refers to this file.

```
DECLARE INVNTRY UPDATE BUFFERED
 ENVIRONMENT(F(100) INDEXED BUFFERS(4));
```

This declaration also specifies only three file attributes: UPDATE, BUFFERED, and ENVIRONMENT. Implied attributes are FILE, RECORD, and SEQUENTIAL (the last two attributes are implied by BUFFERED). Scope is EXTERNAL, by default. The data set is of INDEXED organization, and it contains fixed-length records of 100 bytes each. Four buffers are to be allocated for use in accessing the data set. Note that although the data set actually contains recorded keys, the KEYTO option cannot be specified in a READ statement, since the KEYED attribute has not been specified.

Note that for both of the above declarations, all necessary attributes are either stated or implied in the DECLARE statement. None of the attributes can be changed in an OPEN statement or in a DD statement. The second declaration might have been written:

```
DECLARE INVNTRY
   ENVIRONMENT(F(100) INDEXED);
```

With such a declaration, INVNTRY can be opened for different purposes. It could, for example, be opened as follows:

```
OPEN FILE(INVNTRY)
   UPDATE SEQUENTIAL BUFFERED;
```

With this OPEN statement, the file attributes would be the same as those specified (or implied) in the DECLARE statement in the second example above (the number of buffers would have to be stated in the associated DD statement). The file might be opened in this way, then closed, and then later opened with a different set of attributes, for example:

```
OPEN FILE(INVNTRY)
   INPUT SEQUENTIAL KEYED;
```

This OPEN statement allows records to be read with either the KEYTO or the KEYED option. Because the file is SEQUENTIAL and the data set is INDEXED, the data set may be accessed in a purely sequential manner; or, by means of a READ statement with a KEY option, it may be accessed randomly. A KEY option in a READ statement with a file of this description causes a specified record to be obtained. Subsequent READ statements without a KEY option access records sequentially, beginning with the next record.

The data manipulation performed by the arithmetic, comparison, and bit-string operators are extended in PL/I by a variety of string-handling and editing features. These features are specified by data attributes, statement options, built-in functions, and pseudo-variables.

The following discussions give general descriptions of each feature, along with illustrative examples.

EDITING BY ASSIGNMENT

The most fundamental form of editing performed by the assignment statement involves converting the data type of the value on the right side of the assignment symbol to conform to the attributes of the receiving variable. Because the assigned value is made to conform to the attributes of the receiving field, the precision or length of the assigned value may be altered. Such alteration can involve the addition of digits or characters to and the deletion of digits or characters from the converted item. The rules for data conversion are discussed in Chapter 4, "Expressions," and in Part II, Section F, "Problem Data Conversion."

ALTERING THE LENGTH OF STRING DATA

When a value is assigned to a string variable, it is converted, if necessary, to the same string type (character or bit) as the receiving string and also, if necessary, is truncated or extended on the right to conform to the declared length of the receiving string. For example, assume SUBJECT has the attributes CHARACTER (10), indicating a fixed-length character string of ten characters. Consider the following statement:

```
SUBJECT = 'TRANSFORMATIONS';
```

The length of the string on the right is fifteen characters; therefore, five characters will be truncated from the right end of the string when it is assigned to SUBJECT. This is equivalent to executing:

```
SUBJECT = 'TRANSFORMA';
```

If the assigned string is shorter than the length declared for the receiving string variable, the assigned string is extended on the right either with blanks, in the case of a character-string variable, or with zeros, in the case of a bit-string variable. Assume SUBJECT still has the attributes CHARACTER (10). Then the following two statements assign equivalent values to SUBJECT:

```
SUBJECT = 'PHYSICS';

SUBJECT = 'PHYSICSbbb';
```

The letter b indicates a blank character.

Let CODE be a bit-string variable with the attributes BIT(10). Then the following two statements assign equivalent values to CODE:

```
CODE = '110011'B;

CODE = '1100110000'B;
```

Note, however, that the following statements do not assign equivalent values to SUBJECT if it has the attributes CHARACTER (10):

```
SUBJECT = '110011'B;

SUBJECT = '1100110000'B;
```

When the first statement is executed, the bit-string constant on the right is first converted to a character string and is then extended on the right with blank characters rather than zero bits. This statement is equivalent to:

```
SUBJECT = '110011bbbb';
```

The second of the two statements requires only a conversion from bit-string to character-string type and is equivalent to:

```
SUBJECT = '1100110000';
```

A string value, however, is not extended with blank characters or zero bits when it is assigned to a string variable that has the VARYING attribute. Instead, the length specification of the receiving string variable is effectively adjusted to describe the length of each assigned string. Truncation will occur, though, if the length of the assigned string exceeds the maximum length declared for the varying-length string variable.

OTHER FORMS OF ASSIGNMENT

In addition to the assignment statement, PL/I provides other ways of assigning values to variables. Among these are two methods that involve input and output statements: one in which actual input and output operations are performed, and one in which data movement is entirely internal.

Input and Output Operations

Although the assignment statement is concerned with the transmission of data between storage locations internal to a computer, input and output operations can also be treated as related forms of assignment in which transmission occurs between the internal and external storage facilities of the computer.

Record-oriented operations, however, do not cause any data conversion of items in a logical record when it is transmitted. Required editing of the record must be performed within internal storage either before the record is written or after it is read.

Stream-oriented operations, on the other hand, do provide a variety of editing functions that can be applied when data items are read or written. These editing functions are similar to those provided by the assignment statement, except that any data conversion always involves character type, conversion from character type on input, and conversion to character type on output.

The STRING Option in GET and PUT Statements

The STRING option in GET and PUT statements allows the statements to be used to transmit data between internal storage locations rather than between the internal and external storage facilities. In both GET and PUT statements, the FILE option, specified by FILE (file-name), is replaced by the STRING option, as shown in the following formats:

```
GET STRING (character-string-variable)
    data-specification;

PUT STRING (character-string-variable)
    data-specification;
```

The GET statement specifies that data items to be assigned to variables in the data list are to be obtained from the specified character string. The PUT statement specifies that data items of the data list are to be assigned to the specified character-string variable. The "data-specification" is the same as described for input and output. In general, it takes one of the following forms:

```
DATA [(data-list)]

LIST (data-list)

EDIT (data-list) (format-list)
```

Although the STRING option can be used with each of the three modes of stream-oriented transmission, it is most useful with edit-directed transmission, which considers the input stream to be a continuous string of characters. For list-directed and data-directed GET statements, individual items in the character string must be separated by commas or blanks; for data-directed GET statements, the string must also include the transmission-terminating semicolon, and each data item must appear in the form of an assignment statement. Edit-directed transmission provides editing facility by means of the format list.

The STRING option permits data gathering or scattering operations to be performed with a single statement, and it allows stream-oriented processing of character strings that are transmitted by record-oriented statements.

Consider the following statement:

```
PUT STRING (RECORD) EDIT
   (NAME, PAY#, HOURS*RATE)
   (A(12), A(7), P'$999V.99');
```

This statement specifies that the character-string value of NAME is to be assigned to the first (leftmost) 12 character positions of the string named RECORD, and that the character-string value of PAY# is to be assigned to the next seven character positions of RECORD. The value of HOURS is then to be multiplied by the value of RATE, and the product is to be edited into the next seven character positions, according to the picture specification.

Frequently, it is necessary to read records of different formats, each of which gives an indication of its format within the record by the value of a data item. The STRING option provides an easy way to handle such records; for example:

```
READ FILE (INPUTR) INTO (TEMP);
GET STRING (TEMP) EDIT (CODE) (F(1));
IF CODE ¬= 1 THEN GO TO OTHER_TYPE;
GET STRING (TEMP) EDIT (X,Y,Z)
   (X(1), 3 F(10,4));
```

The READ statement reads a record from the input file INPUTR. The first GET statement uses the STRING option to extract the code from the first byte of the record and to assign it to CODE. The code is tested to determine the format of the record. If the code is 1, the second GET statement then uses the STRING option to assign the items in the record to X,Y, and Z. Note that the second GET statement specifies that the first character in the string TEMP is to be ignored (the X(1) format item in the format list). Each GET statement with the STRING option always specifies that the scanning is to begin at the first character of the string. Thus, the character that is ignored in the second GET statement is the same character that is assigned to CODE by the first GET statement.

In a similar way, the PUT statement with a STRING option can be used to create a record within internal storage. In the following example, assume that the file OUTPRT is eventually to be printed.

```
PUT STRING (RECORD) EDIT
   (NAME, PAY#, HOURS*RATE)
   (X(1), A(12), X(10), A(7), X(10),
    P'$999V.99');

WRITE FILE (OUTPRT) FROM (RECORD);
```

The PUT statement specifies, by the X(1) spacing format item, that the first character assigned to the character-string variable is to be a single blank, the ASA carriage-control code that specifies a single space before printing. Following that, the values of the variables NAME and PAY# and of the expression HOURS*RATE are assigned. The format list specifies that ten blank characters are to be inserted between NAME and PAY# and between PAY# and the expression value. The WRITE statement specifies that record transmission is to be used to write the record into the file OUTPRT.

THE PICTURE SPECIFICATION

The editing capabilities associated with data assignment, namely, conversion to a specified data type with accompanying truncation and/or padding, can be extended by use of the picture specification. A picture specification consists of a sequence of character codes (picture characters) that specify editing operations to be performed on a character string. (A detailed discussion of each picture character, together with examples of its use, appears in Part II, Section D, "Picture Specification Characters." The following discussions are concerned with general principles that govern the use of the picture specification.)

A picture specification can be used to describe ordinary character-string data, or it can be used to describe numeric character data, which is data that represents a numeric value.

A picture specification is always enclosed in quotation marks and is used either with a PICTURE attribute in a DECLARE statement or with a P format item in an edit-directed GET, PUT, or FORMAT statement:

```
DECLARE CODE PICTURE 'XXXXX';

GET FILE (IN) EDIT (CODE) (P'XXXXX');

PUT FILE (OUT) EDIT (CODE) (P'XXXXX');
```

Character-String Picture Specifications

A character-string picture specification describes a fixed-length character string; the number of picture characters in the specification determines the length of the string. For example, the PICTURE attribute in the above DECLARE statement describes CODE as a character string of length five and is equivalent to the attribute CHARACTER (5). The picture character X also specifies that any character recognized by the computer can occur in the corresponding position of the character string.

Any value assigned to CODE will be converted, if necessary, to a character string and will be truncated or extended on the right as required, to meet the five-character length of CODE. Consider the following examples:

```
CODE = 'A2B9C8';

CODE = '4F';
```

In the first assignment, one character is truncated from the right end of the assigned character string. In the second assignment, three blank characters are appended to the right end of the assigned character string.

The format item P 'XXXXX' in the above GET and PUT statements describes a character string of length five in external storage and is equivalent to the format item A(5).

Additional character-string picture characters, other than X, can restrict the characters in the corresponding positions of the character string to a specific type

of character. For example, the picture characters A and 9 specify, respectively, alphabetic or blank and decimal numeric or blank characters. Consider the following PICTURE attributes:

```
PICTURE 'AAAAA'

PICTURE 'X9999'
```

Both of these attribute specifications describe character strings of length five. The first, however, requires all characters in the string to be alphabetic or blank. The second requires all but the first character to be numeric or blank. Any attempt to assign to the string a character with a type different from that specified by the corresponding picture character will raise the CONVERSION error condition.

Only the picture characters X, A, and 9 can appear in a character-string specification, and all can appear in the same picture specification:

```
DECLARE TAG PICTURE 'XX99AA';
```

This statement declares TAG to be a character-string variable representing a string of length six, in which the two leftmost positions can contain any characters, the two middle characters must contain numeric or blank characters, and the two rightmost characters must contain alphabetic or blank characters. The following assignment statement illustrates a correct use of TAG:

```
TAG = '*906RZ';
```

The following statement,however, is incorrect:

```
TAG = 'ABCDEF';
```

In this assignment, the two middle characters are not numeric; consequently, the CONVERSION error condition will be raised.

Any picture specification that contains at least one X or A describes a character-string field. If the picture specification contains neither X nor A, it is said to describe a numeric character field, because the associated character string can be given a numeric interpretation.

Numeric Character Picture Specifications

In addition to the picture character 9, numeric character specifications can contain other picture characters that are used to edit numeric character data. These additional characters apply only to numeric character data and, therefore, cannot be used in any specification that contains either an X or an A. The general functions performed by the additional picture characters are described in "Editing Numeric Character Data" below.

Note: The picture character 9 in a character-string picture specification indicates that the associated character can be either a digit or a blank. The same character in a numeric character specification, however, indicates that the associated character can be a digit only.

Assignment to character-string variables is always from left to right; padding and truncation are on the right. Assignment to a numeric character variable, however, depends upon the location of an assumed decimal point (specified by the picture character V). Values assigned to numeric character fields are always point aligned.

Values of Numeric Character Variables

The value of a numeric character variable can be interpreted in two ways, either as an arithmetic value or as a character-string value.

For a numeric character variable described with a picture specification that contains only one or more occurrences of the character 9, the arithmetic value is the value expressed by the character string, that is, a decimal integer.

If, however, editing characters are included in the picture specification, the arithmetic value and the character-string value generally would be different. Editing characters are actually stored internally in the specified positions of the data item. The editing characters then are considered to be part of the character-string value of the variable. The editing characters are not, however, a part of the variable's arithmetic value, which involves only decimal digits, the assumed location of a decimal point, and a sign (if one is present).

If the value of a numeric character variable is assigned to another numeric character variable or to a coded arithmetic variable, only the arithmetic value is assigned. In the assignment to a coded arithmetic variable (or in the appearance of a numeric character variable in an arithmetic expression operation), conversion to coded arithmetic is performed.

If the value of a numeric character variable is assigned to a character-string

variable, no actual conversion is necessary, and any specified editing characters are included in the assignment.

An ordinary character-string variable (specified with the CHARACTER attribute) can be defined on a numeric character variable, using the DEFINED attribute specification. Any reference to the character-string variable is a reference to the character-string value of the numeric character variable. For example:

```
DECLARE A PICTURE '$999V.99',
        B CHARACTER(7) DEFINED A,
        C DECIMAL FIXED (5,2);

A = 128.76;

C = A;
```

After the constant is assigned to A, its arithmetic value is 128.76. This is the value that is assigned to C (after conversion to internal coded arithmetic). The character-string value of A, however, is $128.76; if it were assigned to a character-string variable with a length of 7 or greater, this is the value that would be assigned. The same value, $128.76, is the value of B, since a character string defined on a numeric character variable represents the character-string value of the numeric character variable. Note that the assignment of B to C would raise the CONVERSION condition, since the character-string value of A represents an invalid arithmetic constant.

No arithmetic variable (except another numeric character variable) can be defined on a numeric character variable without causing an error.

Editing Numeric Character Data

Because the picture specification of a numeric character field cannot contain the characters X and A, the value of a numeric character data item can always be given a numeric interpretation. Consider the following declaration:

```
DECLARE COUNT PICTURE '99999';
```

Although COUNT is a string of five characters, it can only contain numeric digits; therefore, it is a numeric character variable whose value can be interpreted as a five-digit fixed-point decimal integer. Unless specified otherwise (with the picture character V), a decimal point is always assumed to be at the right end of a numeric character data item. For example, let COUNT, as declared above, appear in the following assignment statement:

```
COUNT = 111.01B;
```

Because COUNT is a numeric character variable, the binary constant 111.01B is first converted to decimal with the value 7.25. When the assignment is performed, the decimal point is aligned on the assumed point declared by the numeric character variable, and the two rightmost digits are truncated. Four zero digits are then appended on the left end. The effect of the above assignment therefore, is equivalent to the following statement:

```
COUNT = 00007;
```

The picture character V allows an assumed decimal point to be specified anywhere in a numeric data item, and not just at the right end:

```
DECLARE TOTAL PICTURE '999V99';
```

Here the value of TOTAL is interpreted as a string of five characters representing a five-digit, unsigned fixed-point decimal number with two fractional places. The decimal point of a value assigned to TOTAL will be aligned between the third and fourth digits as specified by the picture character V. Consequently, the following two assignment statements are equivalent:

```
TOTAL = 123;

TOTAL = 123.00;
```

Note, however, that TOTAL contains only five characters. The picture character V does not specify an actual character position in the numeric character field; it is used only to align decimal points. And if TOTAL were printed, no decimal point would appear in the printed field; its character-string value does not include a decimal point.

A decimal point picture character(.) can appear in a numeric picture specification. It merely indicates that a point is to be included in the character representation of the value. Therefore, the decimal point is a part of its character-string value. The decimal point picture character does <u>not</u> cause decimal point alignment during assignment; it is not a part of the variable's arithmetic value. Only the character V causes alignment of decimal points. For example:

```
DECLARE SUM PICTURE '999V.99';
```

SUM is a numeric character variable representing numbers of five digits with a decimal point assumed between the third and fourth digits. The actual point specified by the decimal point insertion character is not a part of the arithmetic value; it is, however, part of its character-string value. (The decimal point picture character can appear on either side of the character V. See Part II, Section D, "Picture Specification Characters.") The following two statements assign the same character string to SUM:

```
SUM = 123;

SUM = 123.00;
```

In the first statement, two zero digits are added to the right of the digits 123.

Note the effect of the following declaration:

```
DECLARE RATE PICTURE '9V99.99";
```

Let RATE be used as follows:

```
RATE = 7.62;
```

When this statement is executed, decimal point alignment occurs on the character V and not on the decimal point picture character that appears in the picture specification for RATE. If RATE were printed, it would appear as '762.00', but its arithmetic value would be 7.6200.

Unlike the character V, which can appear only once in a picture specification, the decimal point picture character can appear more than once; this allows digit groups within the numeric character data item to be separated by points, as is common in Dewey decimal notation and in the numeric notations of some European countries.

Because a decimal point picture character causes a period character to be inserted into the character-string value of a numeric character data item, it is called an insertion character. PL/I provides three other insertion characters: comma (,), slash(/), and blank(B), which are used in the same way as the decimal point picture character except that a comma, slash, or blank is inserted into the character string. Consider the following statements:

```
DECLARE RESULT PICTURE '9.999.999,V99';

RESULT = 1234567;
```

The character-string value of RESULT would be '1.234.567,00'. Note that decimal point alignment occurs before the two rightmost digit positions, as specified by the character V. If RESULT were assigned to a coded arithmetic field, the value of the data converted to arithmetic would be 1234567.00.

Besides supplying insertion characters, PL/I also provides replacement characters that allow zeros in specified positions to be replaced by blanks or asterisks. One such picture character is the character Z, which is used to replace leading (leftmost) zeros with blanks:

```
DECLARE TALLY PICTURE 'ZZZ9';

TALLY = 0012;
```

The character-string value of TALLY is equivalent to the character-string constant 'bb12' (where the letter b indicates a blank character).

Other picture characters control the appearance of signs and the currency symbol ($) in specified positions of numeric character data items. For example, a dollar sign can be appended to the left of a numeric character item, as indicated in the following statements:

```
DECLARE PRICE PICTURE '$99V.99';

PRICE = 12.45;
```

The character-string value of PRICE is equivalent to the character-string constant '$12.45'. Its arithmetic value, however, is 12.45.

The picture specification can also specify floating-point and British sterling formats, as well as scaling factors for fixed-point values. These formats are discussed in Part II, Section D, "Picture Specification Characters."

Using Numeric Character Data

One purpose of a numeric character picture specification is to edit data that is to be printed. For example, in a payroll application, the digits representing an employee's salary might be 0017250. These digits would be much more meaningful on a paycheck in an edited form, such as $**172.50; the asterisks would also discourage an attempt to alter the amount. This could be done, for example, with the specification '$****9.99'.

PL/I, however, does not restrict the use of numeric character data to output purposes. Numeric character variables can be used wherever arithmetic expressions are permitted. Consider the following example:

```
DECLARE RESULT PICTURE 'XXXXXX', COST
     PICTURE '$9V.99';

COST = 7.15;

RESULT = COST;
```

In this example, the arithmetic value of COST would be 7.15. When COST is assigned to RESULT, however, the insertion characters ($ and .) appear as part of the character string, and the value of RESULT is '$7.15b'. Note that in the assignment of numeric character data to character-string variables, leading blanks are not inserted as they are in conversion from arithmetic type to character type. The only differences between the numeric character data and the character-string data is that the character-string value is left-adjusted and the insertion characters are actually a part of the data, while with a numeric character variable, data is point aligned and insertion characters, though actually present, are not considered to be a part of the arithmetic value.

If specified in an arithmetic expression, the value of a numeric character data item is converted to coded arithmetic. Note, however, that this conversion will always require the compiler to insert extra coding. Note also, that any editing characters in the picture specification will be lost in the conversion.

Assume that RESULT and COST in the following statements are declared as above:

```
COST = 1.10;

RESULT = 2*COST;
```

The character-string value of COST is $1.10. The editing characters ($ and .) are present in the item. However, when the expression 2*COST is evaluated, the arithmetic value of COST is converted to coded arithmetic. When the value of the expression is assigned to RESULT, the value of RESULT will be 'bb2.20' (conversion of an arithmetic item to character type causes insertion of three leading blanks, one of which is deleted when the field expands to allow for the decimal point). Note that if RESULT had been declared as PICTURE 'XXXXX', the value of RESULT, after assignment, would have been 'bb2.2'. Since a character string is assigned from left to right, truncation is always on the right; and the inserted leading blanks would force truncation.

BIT-STRING HANDLING

The following examples illustrate some of the facilities of PL/I that can be used in bit-string manipulations.

```
DECLARE 1 PERSONNEL_RECORD,
          2 NAME,
            3 LAST CHARACTER(15),
            3 FIRST CHARACTER(10),
            3 MIDDLE CHARACTER(1),
          2 CODE_STRING,
            3 MALE BIT(1),
            3 SECRETARIAL BIT(1),
            3 AGE,
              4 (UNDER_20,
                 TWENTY_TO_30,
                 OVER_30) BIT(1),
            3 HEIGHT,
              4 (OVER_6,
                 FIVE_SIX_TO_6,
                 UNDER_5_6) BIT(1),
            3 WEIGHT,
              4 (OVER_180,
                 ONE_TEN_TO_180,
                 UNDER_110) BIT(1),
            3 EYES,
              4 (BLUE,
                 BROWN,
                 HAZEL,
                 GREY,
                 OTHER) BIT(1),
            3 HAIR,
              4 (BROWN,
                 BLACK,
                 BLOND,
                 RED,
                 GREY,
                 BALD) BIT(1),
            3 EDUCATION,
              4 (COLLEGE,
                 HIGH_SCHOOL,
                 GRAMMAR_SCHOOL) BIT(1);
```

This structure contains NAME, a minor structure of character-strings, and CODE_STRING, a minor structure of bit-strings. By default, the elements of PERSONNEL-RECORD have the UNALIGNED attribute. Consequently, CODE_STRING is mapped with eight elements per byte, that is, in the same way as a bit-string of length 25.

Each of the first two bits of the string represents only two alternatives: MALE or ¬MALE and SECRETARIAL or ¬SECRETARIAL. The other categories (at level 3) list several alternatives each. (Note that the level number 4 and the attributes BIT(1) are factored for each category.)

The following portion of a program might be used with PERSONNEL_RECORD:

```
INREC: READ FILE(PERSONNEL)
           INTO (PERSONNEL_RECORD);

       IF (¬MALE & SECRETARIAL
           & UNDER_20
           & UNDER_5_6
           & UNDER_110
           & BLUE
           & (HAIR.BROWN|BLOND)
           & HIGH_SCHOOL)
           | (MALE & ¬SECRETARIAL
           & OVER_30
           & OVER_6
           & OVER_180
           & EYES.GREY
           & BALD
           & COLLEGE)

           THEN PUT LIST (NAME);

           GO TO INREC;
```

Another way to program the same information retrieval operation, as shown in the following coding, would result in considerably shorter execution time:

```
     DECLARE PERS_STRING BIT(25) DEFINED
          CODE_STRING;

     IF PERS_STRING
          = '0110000100110000100000010'B
          THEN GO TO OUTP;

     IF PERS_STRING
          = '0110000100110000001000010'B
          THEN GO TO OUTP;

     IF PERS_STRING
          = '1000110010000010000001100'B
          THEN GO TO OUTP;

     GO TO INREC;

OUTP:  PUT LIST (NAME);

     GO TO INREC;
```

In this example, the bit string PERS_STRING is defined on the minor structure CODE_STRING. Bit-string constants are constructed to represent the values of the information being sought. The bit string then is compared, in turn, with each of the bit-string constants. Note that the first and second constants are identical except that the first tests for brown hair and the second tests for blond hair. These two variations are specified in the first example by (HAIR.BROWN|BLOND).

Note that the second method of testing PERSONNEL_RECORD could not be used if the structure were ALIGNED (the base identifier for overlay defining must be UNALIGNED). The first method, if it were used, would be more efficient with an ALIGNED structure.

The tests might also be made with a series of IF statements, either nested or unnested, in which each bit would be tested with a single IF statement. It would require a greater amount of coding, but it would be faster at execution time than an IF statement containing many bit-string operators.

CHARACTER-STRING AND BIT-STRING BUILT-IN FUNCTIONS

PL/I provides a number of built-in functions, some of which also can be used as pseudo-variables, to add power to the string-handling facilities of the language. Following are brief descriptions of these functions (more detailed descriptions appear in Part II, Section G, "Built-In Functions and Pseudo-Variables"):

The BIT built-in function specifies that a data item is to be converted to a bit string. The built-in function allows a programmer to specify the length of the converted string, overriding the length that would result from the standard rules of data conversion.

The CHAR built-in function is exactly the same as the BIT built-in function, except that the conversion is to a character string of a specified length.

The SUBSTR built-in function, which can also serve as a pseudo-variable in a receiving field, allows a specific substring to be extracted from (or assigned to, in the case of a pseudo-variable) a specified string value.

The INDEX built-in function allows a string, either a character string or a bit string, to be searched for the first occurrence of a specified substring, which can be a single character or bit. The value returned is the location of the first character or bit of the substring, relative to the beginning of the string. The value is expressed as a binary integer. If the substring does not occur in the specified string, the value returned is zero.

The LENGTH built-in function gives the current length of a character string or bit string. It is particularly useful with strings that have the VARYING attribute.

The HIGH built-in function provides a string of a specified length that consists of repeated occurrences of the highest character in the collating sequence. For

System/360 implementations, the character is hexadecimal FF.

The LOW built-in function provides a string of a specified length that consists of repeated occurrences of the lowest character in the collating sequence. For System/360 implementations, the character is hexadecimal 00.

The REPEAT built-in function permits a string to be formed from repeated occurrences of a specified substring. It is used to create string patterns.

The STRING built-in function concatenates all the elements in an aggregate variable into a single string element.

The BOOL built-in function allows up to 16 different Boolean operations to be applied to two specified bit strings.

The UNSPEC built-in function, which can also be used as a pseudo-variable, specifies that the internal coded representation of a value is to be regarded as a bit string with no conversion.

CHAPTER 10: SUBROUTINES AND FUNCTIONS

ARGUMENTS AND PARAMETERS

Data can be made known in an invoked procedure by extending the scope of the names identifying that data to include the invoked procedure. This extension of scope is accomplished by nesting procedures or by specifying the EXTERNAL attribute for the names.

There is yet another way in which data can be made known in an invoked procedure, and that is to specify the names as arguments in a list in the invoking statement. Each argument in the list is an expression, a file name, a statement label constant or variable, or an entry name that is to be passed to the invoked procedure.

Since arguments are passed to it, the invoked procedure must have some way of accepting them. This is done by the explicit declaration of one or more parameters in a list in the PROCEDURE or ENTRY statement that is the entry point at which the procedure is invoked. A parameter is a name used within the invoked procedure to represent another name (or expression) that is passed to the procedure as an argument. Each parameter in the parameter list of the invoked procedure has a corresponding argument in the argument list of the invoking statement. This correspondence is taken from left-to-right; the first argument corresponds to the first parameter, the second argument corresponds to the second parameter, and so forth. In general, any reference to a parameter within the invoked procedure is treated as a reference to the corresponding argument. The number of arguments and parameters must be the same.

The example below illustrates how parameters and arguments may be used:

```
PRMAIN: PROCEDURE;
        DECLARE NAME CHARACTER (20),
          ITEM BIT(5);
            .
            .
            .
        CALL OUTSUB (NAME, ITEM);
            .
            .
            .
        END PRMAIN;

OUTSUB: PROCEDURE (A,B);
        DECLARE A CHARACTER (20),
          B BIT(5);
            .
            .
            .
        PUT LIST (A,B);
            .
            .
            .
        END OUTSUB;
```

In procedure PRMAIN, NAME is declared as a character string, and ITEM as a bit string. The CALL statement in PRMAIN invokes the procedure called OUTSUB, and the parenthesized list included in this procedure reference contains the two arguments being passed to OUTSUB. The PROCEDURE statement defining OUTSUB declares two parameters, A and B. When OUTSUB is invoked, NAME is associated with A and ITEM is associated with B. Each reference to A in OUTSUB is treated as a reference to NAME and each reference to B is treated as a reference to ITEM. Therefore, the PUT LIST (A,B) statement causes the values of NAME and ITEM to be written into the standard system output file, SYSPRINT.

Note that the passing of arguments usually involves the passing of names and not merely the values represented by these names. (In general, the name that is passed is usually the address of the value or an address that can be used to retrieve the value.) As a result, storage allocated for a variable before it is passed as an argument is not duplicated when the procedure is invoked. Any change of value specified for a parameter actually is a change in the value of the argument. Such changes are in effect when control is returned to the invoking block.

A parameter can be thought of as indirectly representing the value that is directly represented by an argument. Thus, since both the argument and the parameter represent the same value, the attributes of a parameter and its corresponding argument must agree. For example, an obvious error exists if a parameter has the attribute FILE and its corresponding argument has the attribute FLOAT. However, there are cases in which such an error may not be so obvious, for example, when an argument is a constant. Certain inconsistencies between the attributes of an argument and its associated parameter can be resolved by specifying, in an invoking procedure, the ENTRY attribute for an entry name to be

invoked. The ENTRY attribute specification provides the facility to specify that the compiler is to generate coding to convert one or more arguments to conform with the attributes of the associated parameters. This topic is discussed later in this chapter in the sections "The ENTRY Attribute" and "Dummy Arguments."

A name is explicitly declared to be a parameter by its appearance in the parameter list of a PROCEDURE or ENTRY statement. However, its attributes, unless defaults apply, must be explicitly stated within that procedure in a DECLARE statement.

Parameters, therefore, provide the means for generalizing procedures so that data whose names may not be known within such procedures can, nevertheless, be operated upon. There are two types of generalized procedures that can be written in PL/I: subroutine procedures (called simply, subroutines) and function procedures (functions).

SUBROUTINES

(The discussion in this section applies to synchronous operation and does not completely cover asynchronous operation, although the rules apply generally to all subroutines, whether or not the CALL statement contains one of the multitasking options. Multitasking is discussed in Chapter 15, "Multitasking.")

A subroutine is a procedure that usually requires arguments to be passed to it in an invoking CALL statement. It can be either an external or internal procedure. A reference to such a procedure is known as a subroutine reference. The general format of a subroutine reference is as follows:

```
CALL entry-name [(argument[,argument]...)];
```

Note that a subroutine can also be invoked through the CALL option of an INITIAL attribute specification.

Whenever a subroutine is invoked, the arguments of the invoking statement are associated with the parameters of the entry point, and control is then passed to that entry point. The subroutine is thus activated, and execution begins.

Upon termination of a subroutine, control normally is returned to the invoking block. A subroutine can be terminated normally in any of the following ways:

1. Control reaches the final END statement of the subroutine. Execution of this statement causes control to be returned to the first executable statement logically following the statement that originally invoked the subroutine. There is an exception, however: return of control from a subroutine invoked by the CALL option is to the statement containing the CALL option at the point immediately following that option. Either of these is considered to be a normal return.

2. Control reaches a RETURN statement in the subroutine. This causes the same normal return caused by the END statement.

3. Control reaches a GO TO statement that transfers control out of the subroutine. (This is not permitted if the subroutine is invoked by the CALL option.) The GO TO statement may specify a label in a containing block (the label must be known within the subroutine), or it may specify a parameter that has been associated with a label argument passed to the subroutine. Although this is considered to be normal termination of the subroutine, it is not normal return of control, as effected by an END or RETURN statement.

With synchronous operation, a STOP or EXIT statement encountered in a subroutine abnormally terminates execution of that subroutine and of the entire program associated with the procedure that invoked it.

The following example illustrates how a subroutine interacts with the procedure that invokes it:

```
A:  PROCEDURE;
    DECLARE RATE FLOAT (10), TIME FLOAT(5),
         DISTANCE FLOAT(15), MASTER FILE;
              .
              .
              .
    CALL READCM (RATE, TIME, DISTANCE,
         MASTER);
              .
              .
              .
    END A;
```

```
READCM: PROCEDURE (W,X,Y,Z);
        DECLARE W FLOAT (10), X FLOAT(5),
                Y FLOAT(15), Z FILE;
             .
             .
             .
        GET FILE (Z) LIST (W,X,Y);
             .
             .
             .
        Y = W*X;
        IF Y > 0 THEN RETURN;
             ELSE PUT LIST('ERROR READCM');
        END READCM;
```

The arguments RATE, TIME, DISTANCE, and MASTER are passed to the parameters W, X, Y, and Z. Consequently, in the subroutine, a reference to W is the same as a reference to RATE, X the same as TIME, Y the same as DISTANCE, and Z the same as MASTER.

FUNCTIONS

A function is a procedure that usually requires arguments to be passed to it when it is invoked. It cannot be executed asynchronously with the invoking procedure. Unlike a subroutine, which is invoked by a CALL statement or CALL option, a function is invoked by the appearance of the function name (and associated arguments) in an expression. Such an appearance is called a function reference. Like a subroutine, a function can operate upon the arguments passed to it and upon other known data. But unlike a subroutine, a function is written to compute a single value which is returned, with control, to the point of invocation, the function reference. This single value can be of arithmetic, string (including picture data), locator, or area type. An example of a function reference is contained in the following procedure:

```
MAINP: PROCEDURE;
          .
          .
          .
       GET LIST (A, B, C, Y);
          .
          .
          .
       X = Y**3+SPROD(A,B,C);
          .
          .
          .
       END MAINP;
```

In the above procedure, the assignment statement

```
X = Y**3+SPROD(A,B,C);
```

contains a reference to a function called SPROD. The parenthesized list following the function name contains the arguments that are being passed to SPROD. Assume that SPROD has been defined as follows:

```
SPROD: PROCEDURE (U,V,W);
          .
          .
          .
       IF U > V + W
             THEN RETURN (0);
             ELSE RETURN (U*V*W);
          .
          .
          .
       END SPROD;
```

When SPROD is invoked by MAINP, the arguments A, B, and C are associated with the parameters U, V, and W, respectively. Since attributes have not been explicitly declared for the arguments and parameters, default attributes of FLOAT DECIMAL (6) are applied to each argument and parameter. (The default precision is that defined for System/360 implementations.) Hence, the attributes are consistent, and the association of the arguments with the parameters produces no error.

During the execution of SPROD, the IF statement is encountered and a test is made. If U is greater than V + W, the statement associated with the THEN clause is executed; otherwise, the statement associated with the ELSE clause is executed. In either case, the executed statement is a RETURN statement.

The RETURN statement is the usual way by which a function is terminated and control is returned to the invoking procedure. Its use in a function differs somewhat from its use in a subroutine; in a function, not only does it return control but it also returns a value to the point of invocation. The general form of the RETURN statement, when it is used in a function, is as follows:

```
RETURN (element-expression);
```

The expression must be present and must represent a single value; i.e., it cannot be an array or structure expression. It is this value that is returned to the invoking procedure at the point of invocation. Thus, for the above example, SPROD returns either 0 or the value represented by U*V*W, along with control to the invoking expression in MAINP. The returned value then effectively replaces the function reference, and evaluation of the invoking expression continues.

A function can also be terminated by execution of a GO TO statement. If this method is used, evaluation of the expres-

sion that invoked the function will not be completed, and control will go to the designated statement. As in a subroutine, the transfer point specified in a GO TO statement may be a parameter that has been associated with a label argument. For example, assume that MAINP and SPROD have been defined as follows:

```
MAINP: PROCEDURE;
         .
         .
         .
       GET LIST (A,B,C,Y);
       X = Y**3+SPROD(A,B,C,LAB1);
         .
         .
         .
LAB1:  CALL ERRT;
         .
         .
         .
       END MAINP;

SPROD: PROCEDURE (U,V,W,Z);
       DECLARE Z LABEL;
         .
         .
         .
       IF U > V + W
             THEN GO TO Z;
             ELSE RETURN (U*V*W);
         .
         .
         .
      END SPROD;
```

In MAINP, LAB1 is explicitly declared to be a statement label constant by its appearance as a label for the CALL ERRT statement. When SPROD is invoked, LAB1 is associated with parameter Z. Since the attributes of A must agree with those of LAB1, Z is declared to have the LABEL attribute. When the IF statement in SPROD is executed, a test is made. If U is greater than V + W, the THEN clause is executed, control returns to MAINP at the statement labeled LAB1, and evaluation of the expression that invoked SPROD is discontinued. If U is not greater than V + W, the ELSE clause is executed and a return to MAINP is made in the normal fashion. Additional information about the use of label arguments and label parameters is contained in the section "Relationship of Arguments and Parameters" in this chapter.

Note: In some instances, a function may be so defined that it does not require arguments. In such cases, the appearance of the function name within an expression will be recognized as a function reference only if the function name has been explicitly or contextually declared to be an entry name. See "The ENTRY Attribute" in this chapter for additional information.

Attributes of Returned Values

The attributes of the value returned by a function may be declared in two ways:

1. They may be declared by default according to the first letter of the function name.
2. They may be explicitly declared following the parameter list in the function PROCEDURE (or ENTRY) statement.

Note that the value of the expression in the RETURN statement is converted within the function, whenever necessary, to conform to the attributes specified by one of the two methods above.

In the previous examples of MAINP and SPROD, the PROCEDURE statement of SPROD contains no attributes declared for the value it returns. Thus, these attributes must be determined from the first letter of its name, S. The attributes of the returned value are therefore FLOAT and DECIMAL. Since these are the attributes that the returned value is expected to have, no conflict exists.

Note: Unless the invoking procedure provides the compiler with information to the contrary, the attributes of the value returned by a function to the invoking procedure are always determined from the first letter of the function name.

The way in which attributes can be declared for the returned value in the PROCEDURE or ENTRY statement is illustrated in the following example. Assume that the PROCEDURE statement for SPROD has been specified as follows:

```
SPROD: PROCEDURE (U,V,W,Z) FIXED BINARY;
```

With this declaration, the value returned by SPROD will have the attributes FIXED and BINARY. However, since these attributes differ from those that would be determined from the first letter of the function name, this difference must be stated in the invoking procedure to avoid a possible error. The PL/I programmer communicates this information to the compiler with the RETURNS attribute specified in the invoking procedure.

The RETURNS attribute is specified in a DECLARE statement for an entry name. It specifies the attributes of the value returned by that function. It further specifies, by implication, the ENTRY attribute for the name; consequently, it is an entry name attribute specification. Unless default attributes for the entry name

apply, any invocation of a function must appear within the scope of a RETURNS attribute declaration for the entry name. For an internal function, the RETURNS attribute can be specified only in a DECLARE statement that is internal to the same block as the function procedure.

The general format of the RETURNS attribute is:

RETURNS (attribute-list)

A RETURNS attribute specifies that within the invoking procedure the value returned from the named entry point is to be treated as though it had the attributes given in the attribute list. The word treated is used because no conversion is performed in an invoking block upon any value returned to it. Therefore, if the attributes of the returned value do not agree with those in the attribute list of the RETURNS attribute, an error will probably result.

In order to specify to the compiler that coding for MAINP is to handle the FIXED BINARY value being returned by SPROD, the following declaration must be given within MAINP:

```
DECLARE SPROD RETURNS (FIXED BINARY);
```

It is important to note some of the things that are implied in the above discussion. Principally, it should be remembered that during compilation of the invoking block, there is no way for the compiler to check a function procedure to determine the attributes of the value it returns. In the absence of explicit information in a RETURNS attribute specification, the compiler can only assume that the attributes will be consistent with the attributes implied by the first letter of the function name. This is true even if the function procedure is contained in the invoking procedure. If the returned value does not have the attributes that the invoking procedure is prepared to receive, no conversion can be performed. The RETURNS attribute must be declared for a function that returns any value with attributes not consistent with default attributes for the function name.

uilt-In Functions

Similar to function procedures that a programmer can define for himself is a comprehensive set of pre-defined functions called 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 other necessary or useful functions related to language facilities, such as functions for manipulating strings and arrays.

Built-in functions are invoked in the same way that programmer-defined functions are invoked. However, many built-in functions can return array or structure values, whereas a programmer-defined function can return only an element value.

Note: Some built-in functions may actually be compiled as in-line code rather than as procedure invocations. All are referred to in a PL/I source program, however, by function references, whether or not they result in an actual procedure invocation.

Neither the ENTRY attribute nor the RETURNS attribute can be specified for any built-in function name. The use of the name in a function reference is recognized without need for any further identification; attributes of values returned by built-in functions are known by the compiler.

But since built-in function names are PL/I keywords, they are not reserved; the same identifiers can be used as programmer-defined names. Consequently, ambiguity might occur if a built-in function reference were to be used in a block that is contained in another block in which the same identifier is declared for some other purpose. To avoid this ambiguity, the BUILTIN attribute can be declared for a built-in function name in any block that has inherited, from a containing block, some other declaration of the identifier. Consider the following example.

```
A: PROCEDURE;
     .
     .
     .
   B: BEGIN;
      DECLARE SQRT FLOAT BINARY;
        .
        .
        .
      C: BEGIN;
         DECLARE SQRT BUILTIN;
           .
           .
           .
         END C;
        .
        .
        .
      END B;
     .
     .
     .
   END A;
```

Assume that in external procedure A, SQRT is neither explicitly nor contextually declared for some other use. Consequently, any reference to SQRT would refer to the built-in function of that name. In B, however, SQRT is declared to be a floating-point binary variable, and it cannot be used in any other way. Finally, in C, SQRT is declared with the BUILTIN attribute so that any reference to SQRT will be recognized as a reference to the built-in function and not to the floating-point binary variable declared in B.

Note that a variable having the same identifier as a built-in function can be contextually declared by its appearance on the left-hand side of an assignment symbol (in an assignment statement, a DO statement, or a repetitive specification) or in the data list of a GET statement, provided that it is neither enclosed within nor immediately followed by an argument list. (This does not apply to the names ONCHAR, ONSOURCE, and PRIORITY which are pseudo-variables that do not require arguments.) For example, if the statement SQRT = 1 had appeared in procedure B instead of the explicit declaration, SQRT would have been contextually declared as a floating-point decimal variable.

A programmer can even use a built-in function name as the entry name of a programmer-written function and, in the same program, use both the built-in function and the programmer-written function. This can be accomplished by use of the BUILTIN attribute and the ENTRY attribute. (The ENTRY attribute, which is used in a DECLARE statement to specify that the associated identifier is an entry name, is discussed in a later section of this chapter.)

The following example illustrates use of the ENTRY attribute in conjunction with the BUILTIN attribute.

```
SQRT: PROCEDURE (PARAM) FIXED (6,2);
      DECLARE PARAM FIXED (12);
        .
        .
        .
      END SQRT;

A: PROCEDURE;
   DECLARE SQRT ENTRY RETURNS
        (FIXED(6,2)), Y FIXED(12);
        .
        .
        .
   X = SQRT(Y);
        .
        .
        .
   B: BEGIN;
      DECLARE SQRT BUILTIN;
        .
        .
        .
      Z = SQRT (P);
        .
        .
        .
      END B;
        .
        .
        .
   END A;
```

The use of SQRT as the label of the first PROCEDURE statement is an explicit declaration of the identifier as an entry name. Since, in this case, SQRT is not the built-in function, the entry name must be explicitly declared in A (and the RETURNS attribute is specified because the attributes of the returned value are not apparent in the function name). The function reference in the assignment statement in A thus refers to the programmer-written SQRT function. In the begin block, the identifier SQRT is declared with the BUILTIN attribute. Consequently, the function reference in the assignment statement in B refers to the built-in SQRT function.

If a programmer-written function using the name of a built-in function is external, any procedure containing a reference to that function name must also contain an entry declaration of that name; otherwise a reference to the identifier would be a reference to the built-in function. In the above example, if the PROCEDURE B were not contained in A, there would be no need to specify the BUILTIN attribute; so long as the identifier SQRT is not known as some other name, the identifier would refer to the built-in function.

If a programmer-written function using the name of a built-in function is internal, any reference to the identifier in the containing block would be a reference to the programmer-written function, provided that its name is known in the block in which the reference is made. No entry name attributes would have to be specified if attributes to the returned value could be inferred from the entry name.

RELATIONSHIP OF ARGUMENTS AND PARAMETERS

When a function or subroutine is invoked, a relationship is established between the arguments of the invoking statement or expression and the parameters of the invoked entry point. This relationship is dependent upon whether or not dummy arguments are created.

DUMMY ARGUMENTS

In the introductory discussion of arguments and parameters, it is pointed out that the name of an argument, not its value, is passed to a subroutine or function. However, there are times when an argument has no name. A constant, for example, has no name; nor does an operational expression. But the mechanism that associates arguments with parameters cannot handle such values directly. Therefore, the compiler must provide storage for such values and assign an internal name for each. These internal names are called dummy arguments. They are not accessible to the PL/I programmer, but he should be aware of their existence because any change to a parameter will be reflected only in the value of the dummy argument and not in the value of the original argument from which it was constructed.

A dummy argument is always created for any of the following cases:

1. If an argument is a constant

2. If an argument is an expression involving operators

3. If an argument is an expression in parentheses

4. If an argument is a variable whose data attributes are different from the data attributes declared for the parameter in an entry name attribute specification appearing in the invoking block

5. If an argument is itself a function reference containing arguments

6. If, for the F Compiler, an argument is a controlled array or string associated with a simple parameter, unless the asterisk notation is used.

In all other cases, the argument name is passed directly. The parameter becomes identical with the passed argument; thus, changes to the value of a parameter will be reflected in the value of the original argument only if a dummy argument is not passed.

A task variable cannot be passed as an argument if this would cause a dummy argument to be created.

THE ENTRY ATTRIBUTE

There is no way during compilation of a subroutine or function that the compiler can know the attributes of arguments that will be passed to a parameter. The compiler must assume that the attributes of each argument will agree with the attributes of its associated parameter. Wherever there is disagreement, the program must provide, in the invoking procedure, an ENTRY attribute declaration for the entry name of the subroutine or function being invoked. The general form of the ENTRY attribute is as follows:

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

Note that the above format allows the keyword ENTRY to be specified without accompanying parameter attribute lists, as it might be used to identify a function entry name that does not require arguments.

Each parameter attribute list in the ENTRY attribute specification corresponds to one parameter of the subroutine or function involved and specifies the attributes of that parameter. In general, if the attributes of an argument do not agree with those of its corresponding parameter (as specified in a parameter attribute list), a dummy argument is constructed for that argument if conversion is possible. The dummy argument contains the value of the original argument converted to conform with the attributes of the corresponding parameter. Thus, when the subroutine or function is invoked, it is the dummy argument that is passed to it.

If an ENTRY attribute with parameter attribute lists is not used, the compiler assumes that the arguments are compatible and acts according to the default attributes of the parameters. If the argument attributes do not agree with the attributes of the corresponding parameter, no conversion occurs, and an error probably results. For example, if a fixed decimal argument, which should be byte aligned, is passed to a procedure which expects a fixed binary argument, then a specification interrupt probably occurs when the argument is treated as fullword binary.

When the above form of the ENTRY attribute is used, each parameter of the subroutine or function must be accounted for. If there is no need to specify the attributes of a particular parameter, its place must be kept by a comma. For example, the statement:

```
DECLARE SUBR ENTRY (FIXED,,FLOAT);
```

specifies that SUBR is an entry name that has three parameters: the first and third have the attributes FIXED and FLOAT, respectively, while the attributes of the second are presumably the same as those of the argument being passed. Since the attributes of the second parameter are not stated, no assumptions are made and no conversions are performed.

As mentioned earlier, the ENTRY attribute may be specified without parameter attribute lists. It is used in this way to indicate that the associated identifier is an entry name. Such an indication is necessary if an identifier is not otherwise recognizable as an entry name, that is, if it is not explicitly or contextually declared to be an entry name in one of the following ways:

1. By its appearance as a label of a PROCEDURE or ENTRY statement (explicit)

2. By its appearance immediately following the keyword CALL (contextual)

3. By its appearance as the function name in a function reference that contains an argument list (contextual)

Therefore, if a reference is made to an entry name in a block in which it does not appear in one of these three ways, the identifier must be given the ENTRY attribute explicitly, or by implication (see "Note" below), in a DECLARE statement within the block. For example, assume that the following has been specified:

```
A: PROCEDURE;
     .
     .
     .
   PUT LIST (RANDOM);
     .
     .
     .
   END A;
```

Assume also that A is an external procedure and RANDOM is an external function that requires no arguments and returns a random number. As the procedure is shown above, RANDOM is not recognizable within A as an entry name, and the result of the PUT statement therefore is undefined. In order for RANDOM to be recognized within A as an entry name, it must be declared to have the ENTRY attribute. For example:

```
A: PROCEDURE;
     DECLARE RANDOM ENTRY;
       .
       .
       .
     PUT LIST (RANDOM);
       .
       .
       .
     END A;
```

Now, RANDOM is recognized as an entry name, and the appearance of RANDOM in the PUT statement cannot be interpreted as anything but a function reference. Therefore, the PUT statement results in the output transmission of the random number returned by RANDOM.

Note: The ENTRY attribute is implied -- and therefore need not be stated explicitly -- for an identifier that is declared in a DECLARE statement to have one of the entry name attributes RETURNS, REDUCIBLE, IRREDUCIBLE, USES, or SETS.

Entry Names as Arguments

When an entry name is specified as an argument of a function or subroutine reference, one of the following applies:

1. If the entry name argument, call it M, is specified with an argument list of its own, it is recognized as a function reference; M is invoked, and the value returned by M effectively replaces M and its argument list in the containing argument list.

2. If the entry name argument appears without an argument list, but within an operational expression or within parentheses, then it is taken to be a function reference with no arguments. For example:

   ```
   CALL A((B));
   ```

   This passes, as the argument to procedure A, the value returned by the function procedure B.

3. If the entry name argument appears without an argument list and neither within an operational expression nor within parentheses, the entry name itself is passed to the function or subroutine being invoked. In such cases, the entry name is not taken to be a function reference, even if it is the name of a function that does not require arguments. For example:

   ```
   CALL A(B);
   ```

This passes the entry name B as an argument to procedure A.

There is an exception to this rule, however: if an identifier is known as an entry name and appears as an argument and if the parameter attribute list for that argument specifies an attribute other than ENTRY, the entry name will be invoked and its returned value passed. For example:

```
A: PROCEDURE;
   DECLARE B ENTRY,
           C ENTRY(FLOAT);
      .
      .
      .
   X = C(B);
      .
      .
      .
   END A;
```

In this case, B is invoked and its returned value is passed to C.

Consider the following example:

```
CALLP: PROCEDURE;
       DECLARE RREAD ENTRY,
               SUBR ENTRY (ENTRY, FLOAT,
                  FIXED BINARY, LABEL);
         .
         .
         .
       GET LIST (R,S);
         .
         .
         .
       CALL SUBR (RREAD, SQRT(R), S,
          LAB1);
         .
         .
         .
 LAB1: CALL ERRT(S);
         .
         .
         .
       END CALLP;

 SUBR: PROCEDURE(NAME, X, J, TRANPT);
       DECLARE NAME ENTRY, TRANPT LABEL;
         .
         .
         .
       IF X > J THEN CALL NAME(J);
                ELSE GO TO TRANPT;
         .
         .
         .
       END SUBR;
```

In this example, assume that CALLP, SUBR, and RREAD are external. In CALLP, both RREAD and SUBR are explicitly declared to have the ENTRY attribute. (Actually, the explicit declaration for SUBR is used principally to provide information about the characteristics of the parameters of SUBR.) Four arguments are specified in the CALL SUBR statement. These arguments are interpreted as follows:

1. The first argument, RREAD, is recognized as an entry name (because of the ENTRY attribute declaration). This argument is not in conflict with the first parameter as specified in the parameter attribute list in the ENTRY attribute declaration for SUBR in CALLP. Therefore, since RREAD is recognized as an entry name and not as a function reference, the entry name is passed at invocation.

2. The second argument, SQRT(R), is recognized as a function reference because of the argument list accompanying the entry name. SQRT is invoked, and the value returned by SQRT is assigned to a dummy argument, which effectively replaces the reference to SQRT. The attributes of the dummy argument agree with those of the second parameter, as specified in the parameter attribute list declaration. When SUBR is invoked, the dummy argument is passed to it.

3. The third argument, S, is simply a decimal floating-point element variable. However, since its attributes do not agree with those of the third parameter, as specified in the parameter attribute list declaration, a dummy argument is created containing the value of S converted to the attributes of the third parameter. When SUBR is invoked, the dummy argument is passed.

4. The fourth argument, LAB1, is a statement-label constant. Its attributes agree with those of the fourth parameter. But since it is a constant, a dummy argument is created for it. When SUBR is invoked, the dummy argument is passed.

In SUBR, four parameters are explicitly declared in the PROCEDURE statement. If no further explicit declarations were given for these parameters, arithmetic default attributes would be supplied for each. Therefore, since NAME must represent an entry name, it is explicitly declared with the ENTRY attribute, and since TRANPT must represent a statement label, it is explicitly declared with the LABEL attribute. X and J are arithmetic, so the defaults are allowed to apply.

Note that the appearance of NAME in the CALL statement does not constitute a contextual declaration of NAME as an entry

name. Such a contextual declaration can be made only if no explicit declaration applies, and the appearance of NAME in the PROCEDURE statement of SUBR constitutes an explicit declaration of NAME as a parameter. If the attributes of a parameter are not explicitly declared in a complementary DECLARE statement, arithmetic defaults apply. Consequently, NAME must be explicitly declared to have the ENTRY attribute; otherwise, it would be assumed to be a binary fixed-point variable, and its use in the CALL statement would result in an error.

ALLOCATION OF PARAMETERS

A parameter cannot be declared to have any of the storage class attributes STATIC, AUTOMATIC, or BASED. It can, however, be declared to have the CONTROLLED attribute. Thus, there are two classes of parameters, as far as storage allocation is concerned: those that have no storage class, i.e., simple parameters, and those that have the CONTROLLED attribute, i.e., controlled parameters.

A simple parameter may be associated with an argument of any storage class. However, if more than one generation of the argument exists, the parameter is associated only with that generation existing at the time of invocation.

A controlled parameter must always have a corresponding controlled argument. Such an argument cannot be subscripted, cannot be an element of a structure, and cannot cause a dummy to be created. If more than one generation of the argument exists at the time of invocation, the parameter corresponds to the entire stack of these generations. Thus, at the time of invocation, a controlled parameter represents the current generation of the corresponding argument. A controlled parameter may be allocated and freed in the invoked procedure, thus allowing the manipulation of the allocation stack of the associated argument. A simple parameter cannot be specified in an ALLOCATE or FREE statement.

Parameter Bounds and Lengths

If an argument is a string or an array, the length of the string or the bounds of the array must be declared for the corresponding parameter. The number of dimensions and the bounds of an array parameter or the length of a string parameter must be the same as that for the current generation of the corresponding argument. Usually, this can be assured simply by specifying actual numbers for the bounds or length of the parameter. However, the actual bounds or length may not always be known at the time that the subroutine or function is written. Whenever this is the case, bounds or length for a simple parameter may be specified by asterisks; bounds or length for a controlled parameter may be specified either by asterisks or by expressions.

Simple Parameter Bounds and Lengths

When the actual length or bounds of a simple parameter are not known, they can be specified in a DECLARE statement by asterisks. When an asterisk is used, the length or bounds are taken from the current generation of the corresponding argument; if no current generation exists, any reference to the variable is an error. If an asterisk is used to represent the bounds of one dimension of an array parameter, the bounds of all other dimensions of that parameter must be specified by asterisks.

Controlled Parameter Bounds and Lengths

The bounds or length of a controlled parameter can be represented in a DECLARE statement either by asterisks or by element expressions.

Asterisk Notation: When asterisks are used, length or bounds of the controlled parameter are taken from the current generation of the corresponding argument. Any subsequent allocation of the controlled parameter uses these same bounds or length, unless they are overridden by a different length or bounds specification in the ALLOCATE statement. If no current generation of the argument exists, the asterisks only determine the dimensionality of the parameter, and an ALLOCATE statement in the invoked procedure must specify bounds or length for the controlled parameter before other references to the parameter can be made.

Expression Notation: The bounds or length of a controlled parameter can also be specified by element expressions. These expressions are evaluated at the time of allocation. Each time the parameter is allocated, the expressions are re-evaluated to give current bounds or length for the new allocation. However, such expressions in a DECLARE statement can be overridden by a bounds or length specification in the ALLOCATE statement itself.

If a current generation of the argument exists at the time of invocation, the expressions evaluated at invocation must give the same bounds or length as the argument. If a current generation does not exist, then no requirements are made on the values of these expressions. They are evaluated each time the parameter is allocated, except in those cases where the expressions are overridden by a bounds or length specification in the ALLOCATE statement itself. For example:

```
MAIN: PROCEDURE OPTIONS(MAIN);
      DECLARE (A(20), B(30), C(100),
               D(100))CONTROLLED,
               NAME CHARACTER (20),
               I FIXED(3,0);
         .
         .
         .
      ALLOCATE A,B;
      CALL SUB1(A,B);
         .
         .
      FREE A,B;
         .
         .
         .
      FREE A,B;
      GET LIST (NAME,I);
      CALL SUB2 (C,D,NAME,I);
         .
         .
         .
      FREE C,D;
         .
         .
         .
      END MAIN;

SUB1: PROCEDURE (U,V);
      DECLARE (U(*), V(*)) CONTROLLED;
         .
         .
         .
      ALLOCATE U(30), V(40);
         .
         .
         .
      RETURN;
      END SUB1;

SUB2: PROCEDURE (X,Y,NAMEA,N);
      DECLARE (X(N),Y(N))CONTROLLED,
               NAMEA CHARACTER (*),
               N FIXED(3,0);
         .
         .
         .
      ALLOCATE X,Y;
         .
         .
         .
      RETURN;
      END SUB2;
```

In the procedure MAIN, the arrays A, B, C, and D are declared with the CONTROLLED storage class attribute; NAME and I are AUTOMATIC by default.

When SUB1 is invoked, A and B, which have been allocated as declared, are passed. SUB1 declares its parameters with the asterisk notation. The ALLOCATE statement, however, specifies bounds for the arrays; consequently, the allocated arrays, which are actually a second generation of A and B, have bounds different from the first generation (if no bounds were specified in the ALLOCATE statement, the bounds of the new generation would be identical to those of the first generation).

After control returns to MAIN, the first FREE statement frees the second generation of A and B (allocated in SUB1 as parameters), and the second FREE statement frees the first generation (allocated in MAIN).

When SUB2 is invoked, C and D are passed to X and Y, NAME is passed to NAMEA, and I is passed to N. In SUB2, X and Y are declared with bounds that depend upon the value of I (passed to N). When X and Y are allocated, this value determines the bounds of the allocated array.

Although NAME (corresponding to NAMEA) is not controlled, the asterisk notation for the length of NAMEA indicates that the length is to be picked up from the declaration of the argument (NAME).

ARGUMENT AND PARAMETER TYPES

In general, an argument and its corresponding parameter may be of any data organization and type. For example, an argument may be a statement label, provided that the corresponding parameter is declared with the LABEL attribute; it may be an entry name, provided that the corresponding parameter is an entry name, and so on. However, not all parameter/argument relationships are so clear-cut. Some need further definition and clarification. Such cases are given below.

If a parameter is an element, i.e., a variable that is neither a structure nor an array, the argument must be an element expression. If the argument is a subscripted variable, the subscripts are evaluated before the subroutine or function is invoked and the name of the specified element is passed. If the argument is a constant, the attributes of the corresponding parameter must agree with the attributes indicated by the constant, unless the

ENTRY attribute is specified for the entry name.

If a parameter is an array, the argument must be an array expression or an element expression. If the argument is an element expression, the corresponding parameter attribute list must specify the bounds of the array parameter. (Note, however, that in this case the bounds in the parameter attribute list cannot be asterisks.) This causes the construction of a dummy array argument, whose bounds are those of the array parameter. The value of the element expression then becomes the value of each element of the dummy array argument.

If a parameter is a structure, the argument must be a structure expression or an element expression. If the argument is an element expression, the corresponding parameter attribute list must specify the structure description of the structure parameter (only level numbers need be used -- see the discussion of the ENTRY attribute in Part II, Section I, "Attributes," for details). This causes the construction of a dummy structure argument, whose description matches that of the structure parameter. The value of the element expression then becomes the value of each element of the dummy structure argument. The relative structuring of the argument and the parameter must be the same; the level numbers need not be identical. The element value must be one that can be converted to conform with the attributes of all the elementary names of the structure.

If a parameter is an element label variable, the argument must be either an element label variable or a label constant. If the argument is a label constant, a dummy argument is constructed.

If the parameter is an array label variable, the argument must be an array label variable, an element label variable, or a label constant. If the argument is either of the latter two, the corresponding parameter attribute list must specify that the parameter is a label array, giving the bounds of that array. This causes the construction of a dummy array label argument, whose bounds are those of the label array parameter.

If a parameter is an entry name, the argument must be an entry name. Note that the name of a mathematical built-in function can be passed as an argument, but no other built-in function names can be passed.

If a parameter is a file name, the argument must be a file name. The attributes of the file name parameter are always ignored.

If a parameter is a fixed-length string variable, the argument should be a fixed-length string. If the argument is of varying length, a parameter attribute list describing the parameter as a fixed-length string must be given in the invoking procedure. Similarly, if a parameter is a varying-length string variable, the argument should be a varying-length string. If the argument is of fixed length, a parameter attribute list describing the parameter as a varying-length string must be given in the invoking procedure. Whenever a varying-length string argument is passed to a non-varying string parameter whose length is undefined (i.e. specified by an asterisk), the maximum length of the argument is passed to the invoked procedure. This is true even when the argument is an element; the object of passing the maximum length rather than the current length is to maintain a consistent rule for both element and array arguments. (If the argument were a varying-length string array passed to a non-varying undefined-length parameter, only one length could be passed, and this would naturally be the maximum length.)

Example:

```
DECLARE A CHARACTER(50) VARYING,
        PROC1 ENTRY (CHARACTER(*));

A='123';
CALL PROC1(A);

PROC1: PROCEDURE (B);
DECLARE B CHARACTER(*),
        C CHARACTER(5);

C=B || '45';
/* C='123bb' NOT '12345' */

.
.
.
```

In this example, to pass A, a dummy of length 50 (i.e., the maximum length of A) is created. In the concatenation operation, '45' is concatenated at the right of the character string of length 50 (which contains '123' followed by 47 blanks). The result is then truncated to fit into C, which has length 5, so that C='123bb'.

If a parameter is a locator variable of either pointer or offset type, the argument must be a locator variable of either type. If the types differ, a dummy argument is created. (See also Chapter 14, "Based Storage and List Processing.")

GENERIC NAMES AND REFERENCES

A generic name represents a family of procedure entry points, each member of which can be invoked by a generic reference, that is, a procedure reference using the generic name in place of the actual entry name. The member invoked is determined according to the number and attributes of the arguments specified in he generic reference; it is that member whose parameters match the arguments in number and attributes.

A generic name must be declared with the GENERIC attribute. The general format of this attribute is as follows:

```
generic-name GENERIC (member-declaration
  [,member-declaration]...)
```

Each member declaration corresponds to one procedure entry point in the family. It specifies the entry name of the member, followed by the ENTRY attribute and its associated parameter attribute list; this list gives the number and attributes of the parameters for that entry name. For example, consider the following statement:

```
DECLARE CALC GENERIC
        (FXDCAL ENTRY(FIXED,FIXED),
        FLOCAL ENTRY(FLOAT,FLOAT),
        MIXED ENTRY (FLOAT,FIXED));
```

This statement defines CALC as a generic name having three members, FXDCAL, FLOCAL, and MIXED. One of these three function procedures will be invoked by a generic reference to CALC, depending on the characteristics of the two arguments in that reference. For example, consider the following statement:

```
Z= X + CALC(X,Y);
```

If X and Y are floating-point and fixed-point, respectively, MIXED will be invoked.

When a PL/I program is executed, a large number of exceptional conditions are monitored by the system and their occurrences are automatically detected whenever they arise. These exceptional conditions may be errors, such as overflow or an input/output transmission error, or they may be conditions that are expected but infrequent, such as the end of a file or the end of a page when output is being printed. When checking out a program, a programmer can also get a selective flow trace and dumps by specifying that the occurrence of any one of a list of identifiers be treated as an exceptional condition.

Each of the conditions for which a test may be made has been given a name, and these names are used by the programmer to control the handling of exceptional conditions. The list of condition names is part of the PL/I language. For keyword names and descriptions of each of the conditions, see Part II, Section H, "ON-Conditions."

ENABLED CONDITIONS AND ESTABLISHED ACTION

A condition that is being monitored, and the occurrence of which will cause an interrupt, is said to be enabled. Any action specified to take place when an occurrence of the condition causes an interrupt, is said to be established.

Most conditions are checked for automatically, and when they occur, the system will take control and perform some standard action specified for the condition. These conditions are enabled by default, and the standard system action is established for them.

The most common system action is to raise the ERROR condition. This provides a common condition that may be used to check for a number of different types of errors, rather than checking each error type individually. Standard system action for the ERROR condition is:

1. If the condition is raised in a major task, the FINISH condition is raised and, subsequently, the major task is terminated.
2. If the condition is raised in any other task, that task is terminated.

The programmer may specify whether or not some conditions are to be enabled, that is, are to be checked for so that they will cause an interrupt when they arise. If a condition is disabled, an occurrence of the condition will not cause an interrupt.

All input/output conditions and the ERROR and FINISH conditions are always enabled and cannot be disabled. All of the computational conditions and the program checkout conditions may be enabled or disabled. The program checkout conditions and the SIZE condition must be explicitly enabled if they are to cause an interrupt; all other conditions are enabled by default and must be explicitly disabled if they are not to cause an interrupt when they occur.

Condition Prefixes

Enabling and disabling can be specified for certain conditions by a condition prefix. A condition prefix is a list of one or more condition names, enclosed in parentheses and separated by commas, and connected to a statement (or a statement label) by a colon. The prefix always precedes the statement and any statement labels. A condition name in a prefix list indicates that the corresponding condition is enabled within the scope of the prefix. Some condition names can be preceded by the word NO, without a separating blank or connector, to indicate that the corresponding condition is disabled.

Scope of the Condition Prefix

The scope of the prefix, that is, the part of the program throughout which it applies, is usually the statement to which the prefix is attached. The prefix does not apply to any functions or subroutines that may be invoked in the execution of the statement.

A condition prefix to an IF statement applies only to the evaluation of the expression following the IF; it does not apply to the statements in the THEN or ELSE clauses, although these may themselves have prefixes. Similarly, a prefix to the ON statement has no effect on the statements in the on-unit. A condition prefix to a DO statement applies only to the evaluation of

any expressions in the DO statement itself and not to any other statement in the DO-group.

Condition prefixes to the PROCEDURE statement and the BEGIN statement are special (though commonly used) cases. A condition prefix attached to a PROCEDURE or BEGIN statement applies to all the statements up to and including the corresponding END statement. This includes other PROCEDURE or BEGIN statements nested within that block. It does not apply to any procedures lying outside that block, which may be invoked during execution of the program.

The enabling or disabling of a condition may be redefined within a block by attaching a prefix to statements within the block, including PROCEDURE and BEGIN statements (thus redefining the enabling or disabling of the condition within nested blocks). Such a redefinition applies only to the execution of the statement to which the prefix is attached. In the case of a nested PROCEDURE or BEGIN statement, it applies only to the block the statement defines, as well as any blocks contained within that block. When control passes out of the scope of the redefining prefix, the redefinition no longer applies. A condition prefix can be attached to any statement except a DECLARE or ENTRY statement.

The ON Statement

A system action exists for every condition, and if an interrupt occurs, the system action will be performed unless the programmer has specified an alternate action in an ON statement for that condition. The purpose of the ON statement is to establish the action to be taken when an interrupt results from an exceptional condition that has been enabled, either by default or by a condition prefix.

Note: The action specified in an ON statement will not be executed during any portion of a program throughout which the condition has been disabled.

The form of the ON statement is:

```
ON condition-name [SNAP] on-unit
                         SYSTEM;
```

(See Part II, Section J, "Statements" for a full description.)

The keyword SYSTEM followed by a semicolon specifies standard system action whenever an interrupt occurs. It re-establishes system action for a condition for which some other action has been established.

The on-unit is used by the programmer to specify an alternate action to be taken whenever an interrupt occurs.

The SNAP option specifies that when an interrupt occurs, debugging information will be written in a debugging file. The form and content of the information depends upon the implementation. For the F Compiler, it is a list of all active procedures. The information is written in the standard system file SYSPRINT. If SNAP is specified, the action of the SNAP option precedes the action of the on-unit. If SNAP SYSTEM is specified, the system action message is followed immediately by a list of active procedures.

The on-unit must be either a single, unlabeled, simple statement or an unlabeled begin block. The single statement cannot be a RETURN, FORMAT, or DECLARE statement. It cannot be either of the two compound statements, IF and ON, or a DO-group. (PROCEDURE, BEGIN, END, and DO statements can never appear as single statements.) The begin block, if it appears, can contain any statement except RETURN, although the RETURN statement can appear within a procedure nested in the begin block.

The single statement on-unit, or the begin block on-unit, is executed as though it were a procedure (without parameters) that was called at the point in the program at which the interrupt occurred. If the on-unit is a single statement it behaves exactly as though it were enclosed by PROCEDURE and END statements; when execution reaches the END statement of the unit, control returns to the point from which the block was invoked. Just as with a procedure, control may be transferred out of an on-unit by a GO TO statement; in this case, control is transferred to the point specified in the GO TO, and a normal return does not occur.

Note: The specific point to which control returns from an on-unit varies for different conditions. In some cases, it returns to the point that immediately follows the action in which the condition arose. In other cases, control returns to the actual point of interrupt, and the action is reattempted. An example of the latter case is the return from the on-unit of an ON CONVERSION statement. When an interrupt occurs as the result of a conversion error, control returns from the on-unit to reattempt conversion of the character that caused the error (on the assumption that the invalid character has been changed during execution of the on-unit). If the invalid character is not changed, the ERROR condition is raised.

The Null On-Unit

A special case of an on-unit is the null statement. The effect of this is to say

"When an interrupt occurs as a result of this condition, do nothing."

Use of the null on-unit is not the same as disabling, for two reasons: first, a null on-unit may be specified for any condition, but not all conditions can be disabled; and, second, disabling a condition, if possible, may save time by avoiding any checking for this condition. If a null on-unit is specified, the system must still check for occurrence of the condition, transfer control to the on-unit whenever an interrupt occurs, and then, after doing nothing, return from the on-unit.

Note: With the F Compiler, a null on-unit for the CONVERSION condition will not cause a permanent loop if a conversion error occurs, because no conversion is re-attempted unless the invalid character is changed in the on-unit. If it is not changed, the ERROR condition is raised.

Scope of the ON Statement

The execution of an ON statement associates an action specification with the named condition. Once this association is established, it remains until it is overridden or until termination of the block in which the ON statement is executed.

An established interrupt action passes from a block to any block it activates, and the action remains in force for all subsequently activated blocks unless it is overridden by the execution of another ON statement for the same condition. If it is overridden, the new action remains in force only until that block is terminated. When control returns to the activating block, all established interrupt actions that existed at that point are re-established. This makes it impossible for a subroutine to alter the interrupt action established for the block that invoked the subroutine.

If more than one ON statement for the same condition appears in the same block, each subsequently executed ON statement permanently overrides the previously established condition. No re-establishment is possible, except through execution of another ON statement with an identical action specification (or re-execution, through some transfer of control, of an overridden ON statement).

The REVERT Statement

The REVERT statement is used to cancel the effect of one or more previously executed ON statements. It can affect only ON statements that are internal to the block in which the REVERT statement occurs and which have been executed in the same invocation of that block. The effect of the REVERT statement is to cancel the effect of any ON statement for the named condition that has been executed in the same block in which the REVERT statement is executed. It then re-establishes the action that was in force at that time of activation of that block. This statement has the form:

```
REVERT condition-name;
```

A REVERT statement that is executed in a block in which no on-unit has been established for the named condition is treated as a null statement.

The SIGNAL Statement

The programmer may simulate the occurrence of an ON condition by means of the SIGNAL statement. An interrupt will occur unless the named condition is disabled. This statement has the form:

```
SIGNAL condition-name;
```

The SIGNAL statement causes execution of the interrupt action currently established for the specified condition. The principal use of this statement is in program checking, to test the action of an on-unit, and to determine that the correct action is associated with the condition.

If the signaled condition is not enabled, the SIGNAL statement is treated as a null statement.

The CONDITION Condition

The ON-condition of the form:

```
CONDITION (identifier)
```

allows a programmer to establish an on-unit that will be executed whenever a SIGNAL statement is executed specifying CONDITION and that identifier.

As a debugging aid, this condition can be used to establish an on-unit whose execution results in printing information that shows the current status of the pro-

gram. The advantage of using this technique is that the statements of the on-unit need be written only once. They can be executed from any point in the program through placement of a SIGNAL statement.

Following is an example of how the CONDITION condition might be included in a program:

```
ON CONDITION (TEST) BEGIN;
                      .
                      .
                      .
                    END;
```

Execution of the begin block would be caused wherever the following statement appears:

```
SIGNAL CONDITION (TEST);
```

The CONDITION condition always is enabled, but it can be raised only by the SIGNAL statement.

The CHECK Condition

The CHECK condition is an important tool provided in PL/I for program testing. The keyword CHECK in a prefix list is followed by a parenthesized name list. The names in the list may be statement label constants, entry names, and variables, including array and structure variables, label variables, task variables, event variables, area variables, and locator variables. Subscripted names are not allowed but qualified names can be used. Parameters, and variables with the DEFINED or BASED attributes cannot be checked.

The CHECK prefix may be attached only to PROCEDURE or BEGIN statements, and therefore, it always applies to an entire block.

An interrupt will generally occur immediately after the execution of a statement in which the value of a variable in a check list may have been altered. Exceptions are as follows:

1. With the F Compiler, during data-directed input, the interrupt occurs after the first checked variable receives its value.

2. With statement labels and entry names, the interrupt occurs immediately before the execution of the statement or the invocation of the entry name.

The system action for the CHECK condition is to print the identifier causing the interrupt and, if it is a variable (other than a program control variable), to print its new value in the form of data-directed output. For label variables and other program control variables, only the variable is printed; no value is included.

Thus, by preceding a block with a CHECK prefix, as shown in the example in Figure 11-1, the programmer can obtain selective dumps in a readable format by specifying variables; he can obtain a flow trace by specifying labels and entry names.

The CHECK condition may also be specified in an ON statement, if other than system action is required. This gives the user all the facilities of PL/I, including the simplicity of data-directed output for controlling and editing his debugging information.

The SUBSCRIPTRANGE Condition

Another ON condition that is used principally for program checkout, but that may also be used in production, is the SUBSCRIPTRANGE condition. This condition must be enabled in a condition prefix. When it is enabled, each subscript in an array reference is checked every time it is evaluated to see that it lies within the specified bounds. The condition is raised if any subscript is too high or too low.

Since this checking involves a substantial overhead, it usually is used only in program testing, and is removed for production programs.

The STRINGRANGE Condition

The STRINGRANGE condition is not enabled unless it appears in a condition prefix. It is raised by an invalid reference to the SUBSTR built-in function and pseudo-variable, the arguments to which must lie within certain bounds (see "SUBSTR String Built-in Function," in Section G). It allows execution to continue with a SUBSTR reference that has been revised either automatically (that is, by standard system action) or by the programmer using an on-unit.

Condition Built-In Functions and Condition Codes

When an on-unit is invoked, it is as if it were a procedure without arguments. It

is therefore impossible to pass to the on-unit any information about the interrupt (other than the name of the condition). To assist the programmer in making use of on-units, some special functions are provided that may be used to inquire about the cause of an interrupt and possibly to attempt to correct the error.

These condition built-in functions can be used only in on-units or in blocks invoked by on-units. They are listed in Part II, Section G, "Built-In Functions and Pseudo-Variables."

The condition built-in functions provide information such as the name of the procedure in which the interrupt occurred, the character or character string that caused a conversion interrupt, the value of the key used in the last record transmitted, and so on. Some can be used as pseudo-variables for error correction.

The ONCODE function provides a binary integer whose value depends on the cause of the last interrupt. This function, used in conjunction with the ERROR condition, allows the programmer to deal with errors that may be detected by the implementation, but that are not represented by condition names in the language.

EXAMPLE OF USE OF ON-CONDITIONS

The routine shown in Figure 11-1 illustrates the use of the ON statement, the SIGNAL and REVERT statements, and condition prefixes. The routine reads batches of cards containing test readings. Each batch has a header card with a sample number, called SNO, of zero and a trailer card with SNO equal to 9999. If a conversion error is found, one retry is attempted with the error character set to zero. Two data fields are used to calculate a subscript; if the subscript is out of range, the sample is ignored. If there is more than one subscript error or more than one conversion error in a batch, that batch is then ignored.

The numbers to the right of each line are card sequence numbers, which are not part of the program itself.

The CHECK prefixes in cards 1 and 25 are included to help with debugging; in a production program, they would be removed. The prefix specifies that just before the statements HEADER, NEWBATCH, and BADBATCH are executed, and just before the procedure INPUT is invoked, an interrupt will occur. Since no ON statement has been executed for the CHECK condition, system action is performed. This will result in the appropriate name being written on SYSPRINT.

Since the labels used within the internal procedure INPUT are not known in DIST, they cannot be specified in a CHECK list for DIST. A separate CHECK prefix is therefore inserted just before the procedure statement heading INPUT. This check list specifies the labels in INPUT, and the array TABLE.

It is worth noting again that the CHECK condition prefix can be applied only to PROCEDURE and BEGIN blocks, and not to individual statements. The first statement executed is the ON ENDFILE statement in card 9. This specifies that the external procedure SUMMARY is to be called when an ENDFILE interrupt occurs. This action applies within DIST and within INPUT and within all other procedures called by DIST, unless they establish their own action for ENDFILE.

Throughout the procedure, any conditions except SIZE, SUBSCRIPTRANGE, STRINGRANGE, and CHECK are enabled by default; and for all conditions except those mentioned explicitly in ON statements, the system action applies. This system action, in most cases, is to generate a message and then to raise the ERROR condition. The action specified for the ERROR condition in card 13 is to display the contents of the card being processed. When the ERROR action is completed, the FINISH condition is raised, and execution of the program is terminated.

The statement in card 10 specifies action to be taken whenever a CONVERSION interrupt occurs. Since this action consists of more than one statement, it is bracketed by BEGIN and END statements.

The main loop of the program starts with the statement HEADER. Since the CHECK condition is enabled for HEADER, an interrupt will occur before HEADER is executed. The READ statement with the INTO option will not cause a CHECK condition to be raised for the variable specified in the INTO option; consequently, SAMPLE does not appear in a CHECK list. Instead, PUT DATA statements are used to list the input.

The card read is assumed to be a header card. If it is not, the SIGNAL CONVERSION statement causes execution of the BEGIN block, which in turn calls a procedure (not shown here) that reads on, ignoring cards until it reaches a header card. The begin block ends with a GO TO statement that terminates the on-unit.

```
(CHECK(HEADER,NEWBATCH,INPUT,BADBATCH)):     /*DEBUG*/                    01
    DIST: PROCEDURE;                                                      02
          DECLARE 1 SAMPLE EXTERNAL,                                      03
                    2 BATCH CHARACTER(6),                                 04
                    2 SNO PICTURE '9999',                                 05
                    2 READINGS CHARACTER(70),                             06
                  TABLE(15,15) EXTERNAL;                                  07
   /* ESTABLISH INTERRUPT ACTIONS FOR ENDFILE & CONVERSION */             08
          ON ENDFILE (PDATA) CALL SUMMARY;                                09
          ON CONVERSION BEGIN;  CALL SKIPBCH;                             10
                                GO TO NEWBATCH;                           11
                            END;                                          12
          ON ERROR DISPLAY(BATCH||SNO||READINGS);                         13
   /* MAIN LOOP TO PROCESS HEADER & TABLE */                              14
  HEADER: READ INTO (SAMPLE) FILE (PDATA);                                15
          PUT DATA (SAMPLE);                    /*DEBUG*/                 16
          IF SNO ¬= 0 THEN SIGNAL CONVERSION;                             17
NEWBATCH: GET LIST (OMIN,OINT,AMIN,AINT) STRING (READINGS);               18
          TABLE = 0;                                                      19
          CALL INPUT;                                                     20
          CALL PROCESS;                                                   21
          GO TO HEADER;                                                   22
   /* ERROR RETURN FROM INPUT */                                          23
BADBATCH: SIGNAL CONVERSION;                                              24
(CHECK(IN1,IN2,ERR2,ERR1,TABLE)):           /*DEBUG*/                     25
           INPUT: PROCEDURE;                                              26
           /* ESTABLISH INTERRUPT ACTIONS FOR CONVERSION & SUBRG */       27
                 ON CONVERSION BEGIN;                                     28
                               IF ONCODE = 624 & ONCHAR = ' '             29
                                  THEN DO; ONCHAR = '0';                  30
                                           GO TO ERR1;                    31
                                       END;                               32
                                  ELSE GO TO BADBATCH;                    33
                               END;                                       34
                 ON SUBSCRIPTRANGE GO TO ERR2;                            35
           /* LOOP TO READ SAMPLE DATA AND ENTER IN TABLE */              36
             IN1: READ INTO (SAMPLE) FILE (PDATA);                        37
                  IF SNO = 9999 THEN RETURN;          /*TRAILER CARD*/   38
             IN2: GET EDIT (R,OMEGA,ALPHA)(3 P'999')                      39
                      STRING (READINGS);                                  40
(SUBSCRIPTRANGE): TABLE((OMEGA-OMIN)/OINT,(ALPHA-AMIN)/AINT) = R;         41
                  GO TO IN1;                                              42
           /* FIRST CONVERSION & SUBSCRIPTRANGE ERROR IN THIS BATCH */    43
            ERR2: ON SUBSCRIPTRANGE GO TO BADBATCH;  /*FOR NEXT ERROR*/   44
                  CALL ERRMESS(SAMPLE,02);                                45
                  GO TO IN1;                                              46
            ERR1: REVERT CONVERSION;       /*SWITCH FOR NEXT ERROR*/      47
                  CALL ERRMESS(SAMPLE,01);                                48
                  GO TO IN2;                                              49
                  END INPUT;                                              50
          END DIST;                                                       51
```

Figure 11-1. A Program Checkout Routine

The GET statement labeled NEWBATCH uses the STRING option to get the different test numbers that have been read into the character string READINGS, which is an element of SAMPLE. Since the variables named in the data list are not explicitly declared, their appearance causes implicit declaration with the attributes FLOAT DECIMAL (6).

The array TABLE is initialized to zero before the procedure INPUT is called. This procedure inherits the on-units already established in DIST, but it can override them.

The first statement of INPUT establishes a new action for CONVERSION interrupts. Whenever an interrupt occurs, the ONCODE is tested to check that the interrupt is due to an illegal P format input character and that the illegal character is a blank. If the illegal character is a blank, it is replaced by a zero, and control is transferred to ERR1.

ERR1 is internal to the procedure INPUT. The statement, REVERT CONVERSION, nullifies the ON CONVERSION statement executed in INPUT and restores the action specified for conversion interrupts in DIST (which causes the batch to be ignored).

After a routine is called to write an error message, control goes to IN2, which retries the conversion. If another conversion error occurs, the interrupt action is that specified in cards 10 and 11.

The second ON statement in INPUT establishes the action for a SUBSCRIPTRANGE interrupt. This condition must be explicitly enabled by a SUBSCRIPTRANGE prefix for an interrupt to occur. If an interrupt does occur, the on-unit causes a transfer to ERR2, which establishes a new on-unit for SUBSCRIPTRANGE interrupts, overriding the action specified in the ON statement in card 35. Any subsequent subscript errors in this batch will, therefore, cause control to go to BADBATCH, which signals the CONVERSION condition as it existed in the procedure DIST. Note that on leaving INPUT, the on-action reverts to that established in DIST, which in this case calls SKIPBCH to get to the next header card.

After establishment of a new on-unit, a message is printed, and a new sample card is read.

The statement labeled IN1 reads an 80-column card image into the structure SAMPLE. A READ statement does not cause input data to be checked, so the CONVERSION condition cannot arise.

The statement IN2 checks and edits the data in card columns 11 through 19 according to the picture format item. A non-numeric character (including blank) in these columns will cause a conversion interrupt, with the results discussed above.

The next statement (card 41) has a SUBSCRIPTRANGE prefix. The data just read is used to calculate a double subscript. If either subscript falls outside the bounds declared for TABLE, an interrupt occurs. If both fall outside the range, two interrupts occur.

CHAPTER 12: 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 does have some control over his source program during compile time. His source program can contain special statements (identified by a leading %) that can cause parts of the source program to be altered in various ways:

1. Any identifier appearing in the source program can be changed.

2. If conditional compilation is desired, the programmer can indicate which sections of his program are to be compiled.

3. Strings of text residing in a user library or a system library can be incorporated into the source program.

PL/I makes source program alteration at compile time possible by a somewhat different approach to compile time processing. Compile time as defined in PL/I has two stages:

1. The Preprocessor Stage -- During this stage, the user's source program is scanned for preprocessor statements, special statements that cause the preprocessor to alter the text being scanned. These statements are considered part of the source program, and appear freely intermixed with the statements and other text of the source program. The altered source program, resulting from the action of the preprocessor statements, then serves as input to the second stage. Note that the preprocessor stage is optional; the publication IBM System/360 Operating System PL/I(F) Programmer's Guide, Form C28-6594, describes how this stage can be used or avoided for the F-level PL/I Compiler.

2. The Processor Stage -- During this stage, the output from the first stage is compiled into an executable object program.

This chapter is concerned with the first stage; the actual compilation of a program is not discussed.

PREPROCESSOR INPUT AND OUTPUT

The preprocessor interprets preprocessor statements and acts upon the source program accordingly. Input to the preprocessor is a sequence of characters that is the user's source program. It contains preprocessor statements freely intermixed with the rest of the user's source program. Preprocessor statements are identified by a leading percent symbol (%) and are executed as they are encountered by the preprocessor (with the exception of preprocessor procedures, which must be invoked in order to be executed). One or more blanks may separate the percent symbol from the statement.

While checking the preprocessor statements for correct format and such, the preprocessor also checks the rest of the source program text to insure that there are no unmatched comment or character-string delimiters. A percent symbol appearing within a comment or character string is considered to be part of that comment or string. This is the extent of the checking done at this stage on all text other than preprocessor statements.

Output from the preprocessor consists of a new character string called the preprocessed text, which consists of the altered source program and which serves as input to the processor stage. Note that preprocessor statements are replaced by blanks in the preprocessed text.

PREPROCESSOR SCAN

The preprocessor starts its scan of the input text at the beginning of the string and scans each character sequentially. As long as a preprocessor statement is not encountered, the characters are placed into the preprocessed text in the same order and general form in which they were scanned. However, when a preprocessor statement is encountered, it is executed. This execution can cause the scanning of the source program and the subsequent formation of preprocessed text to be altered in either of two ways:

1. The executed statement may cause the preprocessor to continue the scan from a different point in the program. This new point may very well be one that has already been scanned.

2. The executed statement may initiate replacement activity. That is, it may cause an identifier not appearing in a preprocessor statement to be replaced when that identifier is subsequently encountered in the scan. The replacement value will then be written into the preprocessed text in place of the old identifier (see "Rescanning and Replacement" below for details).

The scan is terminated when an attempt is made to scan beyond the last character in the source program. The preprocessed text is completed and the second stage of compilation can then begin.

Rescanning and Replacement

For an identifier to be replaced by a new value, the identifier must first be activated for replacement. Initially, an identifier is activated by its appearance in a preprocessor DECLARE statement (i.e., a % DECLARE statement). (It can be deactivated by appearing in a % DEACTIVATE statement and it can be reactivated by appearing in a % ACTIVATE statement.) After it has been activated initially, it must be given a replacement value. This is usually done via the execution of a preprocessor assignment statement. Once an identifier has been activated and been given a value, any occurrence of that identifier in text other than preprocessor statements is replaced by that value, provided that the identifier is still active when it is encountered by the scan. The new value is not immediately inserted into the preprocessed text, however; it must be checked to see whether or not it, or any part of it, is subject to replacement by still another value (a rescan is made to determine this). If it cannot be replaced, it is inserted into the preprocessed text; if it can be replaced, replacement activity continues until no further replacements can be made. Thus, insertion of a value into preprocessed text takes place only after all possible replacements have been made. Note that the deactivation of an identifier causes it to lose its replacement capability but not its value. Hence, the subsequent reactivation of such an identifier need not be accompanied by the assignment of a replacement value.

For example, if the source program contained the following sequence of statements:

```
%DECLARE A CHARACTER, B FIXED;
%A = 'B+C';
%B = 2;
X = A;
```

then the following would be inserted into the preprocessed text in place of the above sequence:

```
X = 2+C;
```

In this example, the first statement is a preprocessor DECLARE statement that activates A and B and also establishes them as preprocessor variables. (An identifier must be established as a preprocessor variable before it can be assigned a value in a preprocessor statement; it can be so established only through a preprocessor DECLARE statement.) The second and third statements are preprocessor assignment statements; the second assigns the character string 'B+C' to A, and the third assigns the constant 2 to B. The fourth statement is a nonpreprocessor statement[1] and, therefore, is not executed at this stage. However, because this statement contains A, and A is a preprocessor variable that has been activated for replacement, the current value of A will replace it in that statement. Thus, the string 'B+C' replaces A in the statement. But this string contains the preprocessor variable B. Upon checking B, the preprocessor finds that it has been activated and that it has been assigned a value of 2. Hence, the value 2 replaces B in the string. Further checking shows that 2 cannot be replaced; scanning resumes with +C which, again, cannot be replaced. Thus, the chain of replacements comes to an end and the resulting statement is inserted into the preprocessed text.

Note that the preprocessor variable B has a default precision of (5,0) and, therefore, actually contains 2 preceded by four zeros. When this value replaces B in the string 'B+C' it is converted to a character string and becomes 2 preceded by seven blanks (the rules for conversion of decimal fixed-point values to character

---

[1]For the purpose of this discussion, a nonpreprocessor statement is any statement or set of one or more identifiers that appears in the source program but is not contained in a preprocessor statement, nor in a preprocessor procedure, nor in a comment.

string are followed). See the section "Preprocessor Expressions" for details.

Also note that each time a replacement occurs, a blank is appended to each end of the replacement value. Hence, in the above example, the first replacement results in a blank being appended to each end of the string 'B+C', and the second replacement results in another blank being appended to each side of the 2 that replaces the B. The result, therefore, will have nine additional blanks immediately before the 2, one additional blank immediately after the 2, and one additional blank immediately after the C.

Replacement values must not contain percent symbols, unmatched quotation marks, or unmatched comment delimiters.

The following example illustrates how compile-time facilities can be used to speed up the execution of a DO-loop.

A programmer might include the following loop in his program:

```
DO I=1 TO 10;
Z(I)=X(I)+Y(I);
END;
```

The following sequence would accomplish the same thing, but without the requirements of incrementing and testing during execution of the compiled program:

```
%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 first statement activates I and establishes it as a preprocessor variable. The second statement assigns the value 1 to I. This means that subsequent encounters of the identifier I in non-preprocessor statements will be replaced by 1 (provided that I remains activated). The third statement is a preprocessor null statement that is used as the transfer target for the preprocessor GO TO statement appearing later.

The fourth statement, not being a preprocessor statement, is only scanned for replacement activity; it is not executed. The first time that this statement is scanned, I has the value 1 and has been activated. Therefore, each occurrence of I in this statement is replaced by 1 and the following is inserted into the preprocessed text being formed:

```
Z( 1 )=X( 1 )+Y( 1 );
```

Note that each 1 is actually preceded by seven blanks of its own in addition to the one replacement blank shown.

The fifth statement increments the value of I by 1 and the sixth statement, a preprocessor IF statement, tests the value of I. If I is not greater than 10, the scan is resumed at the statement labeled LAB; otherwise, the scan continues with the text immediately following the %GO TO statement. Hence, for each increment of I, up to and including 10, the assignment statement is rescanned and each occurrence of I is replaced by its current value. As a result, the following statements are inserted into the preprocessed text:

```
Z( 1 )=X( 1 )+Y( 1 );

Z( 2 )=X( 2 )+Y( 2 );
      .
      .
      .
Z( 10 )=X( 10 )+Y( 10 );
```

As before, each number from 1 through 9 is preceded by seven blanks in addition to the replacement blank shown; 10 is preceded by six blanks in addition to the replacement blank shown.

When the value of I reaches 11, control falls through to the %DEACTIVATE statement. This statement is interpreted as follows: subsequent encounters of the identifier I in source program text are not to be replaced by the value 11 in the preprocessed text being formed; each I will be left unmodified, either for the remainder of the scan or at least until I is reactivated by a %ACTIVATE statement. If I is again activated, it will still have the value 11 (unless an intervening preprocessor assignment statement has established a new value for I).

PREPROCESSOR VARIABLES

A preprocessor variable is an identifier that has been specified in a %DECLARE statement with either the FIXED or CHARACTER attribute. No other attributes can be declared for a preprocessor variable. Defaults are applied, however. A preprocessor variable declared with the FIXED attribute is also given the attributes DECIMAL and, for the F Compiler, precision (5,0) by default; a CHARACTER preprocessor variable is given the VARYING attribute with no maximum length. No contextual or implicit declaration of identifiers is allowed in preprocessor statements.

The scope of a preprocessor variable encompasses all text except those preprocessor procedures that have redeclared that variable. The scope of a preprocessor variable that has been declared in a preprocessor procedure is the entire procedure (there is no nesting of preprocessor procedures).

When a preprocessor variable has been given a value, that value replaces all occurrences of the corresponding identifier in text other than preprocessor statements

during the time that the variable is active. If the preprocessor variable is inactive (or if it has no value), replacement activity cannot occur for the corresponding identifier.

A preprocessor variable is activated initially by its appearance in the %DECLARE statement. It can be deactivated and subsequently reactivated by its appearance in %DEACTIVATE and %ACTIVATE statements, respectively. Deactivation of a preprocessor variable does not strip it of its value; in other words, an inactive preprocessor variable retains the value it had while it was active and can be altered by a preprocessor statement or procedure if so desired.

PREPROCESSOR EXPRESSIONS

Preprocessor expressions are written and evaluated in the same way as source program expressions, with the following exceptions:

1. The operands of a preprocessor expression can consist only of preprocessor variables, references to preprocessor procedures, decimal integer constants, bit-string constants, character-string constants, and references to the built-in function SUBSTR. Repetition factors are not allowed with the string constants and the arguments of a reference to SUBSTR must be preprocessor expressions.

2. The exponentiation symbol (**) cannot be used as an arithmetic operator.

3. For arithmetic operations, only decimal integer arithmetic of precision (5,0) is performed; that is, each operand is converted to a decimal fixed-point value of precision (5,0) before the operation is performed, and the decimal fixed-point result is converted to precision (5,0) also. Any character string being converted to an arithmetic value must be in the form of an optionally signed decimal integer constant. Note that the properties of the division operator are affected. For example, the expression 3/5 evaluates to 0, rather than to 0.6.

4. The conversion of a fixed-point decimal number to a character string always results in a string of length 8. (Leading zeros in the number are replaced by blanks and an additional three blanks are appended to the left end of the number, one of which is replaced by a minus sign if the number is negative.)

A character string in an expression being assigned to a preprocessor variable may include preprocessor variables, references to preprocessor procedures, constants, and operators; preprocessor statements cannot be included in such strings. Note that if the programmer desires to insert a multiple character operator such as ¬= into preprocessed text, the operator must appear in the source program as an entity. For example, one cannot have a ¬A in the source program and expect a %A='¬=' statement to generate the operator ¬= in the preprocessed text. The reason is that all replacements cause a blank to be appended to each end of the replacement value. Thus, the hypothetical case cited would result in ¬b=b (where each b represents a blank) being inserted into the preprocessed text.

PREPROCESSOR PROCEDURES

A preprocessor procedure is an internal function procedure that can be executed only at the preprocessor stage. Its syntax differs from other function procedures in that its PROCEDURE and END statements must each have a leading percent symbol. The format of a preprocessor procedure is as follows:

```
%label: [label:]... PROCEDURE [(identifier
        [,identifier] ...)]
        {CHARACTER|FIXED};
              .
              .
              .
[label:]...RETURN
          (preprocessor-expression);
              .
              .
              .
% [label:] ...   END [label];
```

More than one RETURN statement may appear. The general rules governing the statements that can appear within a preprocessor procedure are given in the description of the %PROCEDURE statement in Part II, Section J, "Statements." One thing should be noted, however: no statement appearing within a preprocessor procedure can have a leading percent symbol.

INVOCATION OF PREPROCESSOR PROCEDURES

A preprocessor procedure is invoked by a function reference in the usual sense; i.e., by the appearance of the entry name and its associated argument list (if any) in an expression. The function reference

can appear in a preprocessor statement or in a nonpreprocessor statement. However, at least one condition must be met for the function to be invoked: regardless of where the reference appears, the function can be invoked if and only if the entry name used in that reference has been explicitly declared with the ENTRY and RETURNS attributes in a %DECLARE statement. This, and not its appearance as a label of a %PROCEDURE statement, is what establishes it as an entry name; in fact, it is not even necessary for the preprocessor procedure to have been scanned before the reference is encountered (the procedure has only to be in the source program somewhere -- anywhere -- when the reference is encountered). This is the only condition that must be met for a preprocessor procedure to be invoked by a reference in a preprocessor statement.

A second condition must be met if the reference to the preprocessor procedure is made in a nonpreprocessor statement: the entry name used in the reference must be active at the time the reference is encountered. Entry names of preprocessor functions are the same as preprocessor variables as far as activation and deactivation is concerned; i.e., they must be activated initially by a %DECLARE statement and they can be deactivated and reactivated thereafter by %DEACTIVATE and %ACTIVATE statements. Thus, since the first condition requires that the entry name appear in a %DECLARE statement, this second condition would be restrictive only if the entry name had later appeared in a %DEACTIVATE statement.

The value returned by a preprocessor function (i.e., the value of the preprocessor expression in the RETURN statement) always replaces the function reference and its associated argument list. Note that for a reference made in a preprocessor statement, the replacement is only for that particular execution of the statement; a subsequent scanning of the statement would again result in the invocation of the function.

ARGUMENTS AND PARAMETERS FOR PREPROCESSOR FUNCTIONS

The number of arguments in a preprocessor function reference must always agree with the number of parameters accounted for in the ENTRY attribute specified for that function in a %DECLARE statement. If parameters are not accounted for, the preprocessor assumes that the corresponding procedure has none and no arguments are passed. If, however, parameters are accounted for, the preprocessor expects to find a parenthesized list of arguments, separated by commas and equal in number to the parameters accounted for in the procedure reference. The number of parameters accounted for in the ENTRY attribute and the actual number of parameters in the %PROCEDURE statement, however, need not be the same. The arguments are interpreted according to the type of statement (preprocessor or nonpreprocessor) in which the function reference appears. The arguments in the argument list are evaluated before any match is made with the parameter list. If there are more arguments than parameters, the excess arguments on the right are ignored. (Note that for a function reference argument, the function is invoked and executed, even if the argument is ignored later.) If there are fewer arguments than parameters, the excess parameters on the right are given values of zero, for FIXED parameters, or the null string, for CHARACTER parameters. The usual rules concerning the creation of dummy arguments apply if the function reference is in a preprocessor statement, but dummy arguments are always created if the function reference occurs in a nonpreprocessor statement.

If the function reference appears in a nonpreprocessor statement, the arguments are interpreted as character strings and are delimited by the appearance of a comma or a right parenthesis occurring outside of balanced parentheses. For example, the argument list (A(B,C),D) has two arguments, namely, the string A(B,C) and the string D. Each argument is then scanned for possible replacement activity. Both the procedure name and its argument list must be found at one replacement level. Thus, only the commas and parentheses seen in the text being scanned when the procedure name is encountered are considered in this context. After all replacements have been made, each resulting argument is converted to the type indicated by the corresponding parameter attribute in the ENTRY attribute declaration for the function entry name (i.e., the ENTRY attribute declaration in the %DECLARE statement). No conversion is performed if a corresponding parameter attribute is not given in the ENTRY declaration. (The ENTRY attribute is discussed under "The %DECLARE Statement" in Part II, Section J, "Statements.")

If the function reference appears in a preprocessor statement, the arguments are associated with the parameters in the normal fashion. If there is a disagreement, the arguments are converted to the attributes of the corresponding parameters as specified in the ENTRY attribute of the %DECLARE statement for the entry name. Only preprocessor variables, character-

string constants, and fixed-point decimal constants can be passed to a preprocessor function invoked by a preprocessor statement.

Returned Value

The value returned by a preprocessor function to the point of invocation is represented by the preprocessor expression in the RETURN statement of that function. Before being returned, this value is converted (if necessary) to the attribute (CHARACTER or FIXED) specified in the function's %PROCEDURE statement. The attribute of the returned value must be consistent with the attribute specified with the RETURNS attribute in the ENTRY attribute specification of the %DECLARE statement for the entry name. If the point of invocation is in a nonpreprocessor statement, the value is scanned for replacement activity after it has replaced the function reference. Note that the replacement of a function reference in a nonpreprocessor statement involves surrounding the replacement value by blanks (one blank on each end) in the same way that it does for the replacement of an identifier by the value of the preprocessor variable. Following are examples of preprocessor functions.

In the statements below, VALUE is a preprocessor function procedure that returns a character string of the form arg1(arg2), where arg1 and arg2 represent the arguments that have been passed to the function.

Assume that the source program contains the following sequence:

```
%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);
```

When the scan encounters the last statement, A is active and is thus eligible for replacement. Since VALUE is also active, the reference to it in the last statement causes the preprocessor to invoke the preprocessor function procedure of that name. However, before the arguments A and 3 are passed to VALUE, A is replaced by its value Z (assigned to A in a previous assignment statement), and 3 is converted to fixed-point to conform to the attribute of its corresponding parameter. VALUE then performs a concatenation of these arguments and the parentheses and returns the concatenated value, that is, the string Z (3), to the point of invocation. The returned value replaces the function reference and the result is inserted into the preprocessed text. Thus, the preprocessed text generated by the above sequence is as follows (replacement blanks are not shown):

```
DECLARE (Z(10),Q) FIXED;
Q = 6+Z(3);
```

The preprocessor function procedure GEN defined in the following example can generate GENERIC declarations for up to 99 parameters. Only four are generated in this example, however.

Assume that the source program contains the following sequence:

```
%DCL GEN ENTRY (CHAR, FIXED, FIXED,
         CHAR) RETURNS (CHAR);
   .
   .
   .
DCL A GEN (A,2,5,FIXED),...;
   .
   .
   .
%GEN: PROC (NAME,LOW,HIGH,ATTR)
          CHAR;
  DCL (NAME, SUFFIX, ATTR,  STRING)
    CHAR, (LOW, HIGH, I, J) FIXED;
  STRING='GENERIC(';
  DO I=LOW TO HIGH; /* ENTRY DCL
                       LOOP */
    IF I>9
      THEN  SUFFIX=SUBSTR(I, 7, 2);
        /* 2 DIGIT*/
      ELSE SUFFIX=SUBSTR(I, 8,  1);
        /*1 DIGIT*/
    STRING=STRING||NAME||SUFFIX||
      ' ENTRY (';
    DO J=1 TO I; /* PAR ATTR LIST*/
      STRING=STRING||ATTR;
      IF J<I /* PARAM ATTR
                SEPARATOR */
        THEN STRING=STRING||',';
        ELSE STRING=STRING||')';
    END;
    IF I<HIGH /* ENTRY DCL
                 SEPARATOR*/
        THEN STRING=STRING||',';
        ELSE STRING=STRING||')';
  END;
  RETURN (STRING);
% END;
```

The following is generated into preprocessed text:

```
DCL A GENERIC(A2 ENTRY (FIXED,FIXED),
              A3 ENTRY (FIXED, FIXED,
                FIXED),
              A4 ENTRY (FIXED, FIXED,
                FIXED, FIXED),
              A5 ENTRY (FIXED, FIXED,
                FIXED, FIXED, FIXED)),
```

Note that the above example refers to the built-in function SUBSTR. It is the only built-in function that can be invoked at the preprocessor stage. It can be invoked by a reference in either a preprocessor or a nonpreprocessor statement.

Use of the SUBSTR Built-In Function

A reference to SUBSTR in a nonpreprocessor statement is executed by the preprocessor only if the name SUBSTR is active. The built-in function SUBSTR can be activated only by a %ACTIVATE statement. If the identifier SUBSTR is given the ENTRY attribute in a %DECLARE statement, it is assumed to refer to a user-defined preprocessor procedure of that name. The arguments in a nonpreprocessor statement reference to the built-in function SUBSTR are interpreted in the same way that arguments in any nonpreprocessor statement reference to a preprocessor function are interpreted, that is, as character strings.

A preprocessor statement reference to SUBSTR is always valid.

THE PREPROCESSOR DO-GROUP

The preprocessor DO-group can provide iterative execution of the preprocessor statements contained within the group. The format of the preprocessor DO-group is as follows:

```
%[label:]... DO [i=m1[TO m2[BY m3]]
                   [BY m3[TO m2]]];
               .
               .
               .
%[label:]... END[label];
```

In the above format, i must be a preprocessor variable and m1, m2, and m3 must be preprocessor expressions. The label that can follow the keyword END must be one of the labels preceding the keyword DO. Preprocessor DO-groups may be nested and multiple closure is allowed.

Control cannot be transferred into a preprocessor DO-group specifying iteration, except by way of a return from a preprocessor procedure invoked from within the group.

Both preprocessor statements and text other than preprocessor statements can appear within a preprocessor DO-group. However, only the preprocessor statements are executed; nonpreprocessor statements are scanned but only for possible replacement activity.

Noniterative preprocessor DO-groups are useful as THEN or ELSE clauses of %IF statements.

The expansion of a preprocessor DO-group is the same as that shown under the nonpreprocessor DO statement in Part II, Section J, "Statements."

The example below results in the same expansion generated for the example of preprocessor loop expansion in the section "Rescanning and Replacement" in this chapter:

```
%DECLARE I FIXED;
%DO I=1 TO 10;
Z(I)=X(I)+Y(I);
%END;
%DEACTIVATE I;
```

The second example under "Returned Value" shows how preprocessor DO-groups can be used within a preprocessor procedure (percent symbols must be omitted, of course).

INCLUSION OF EXTERNAL TEXT

Strings of external text can be incorporated into the source program at the preprocessor stage by use of the %INCLUDE statement. Such text, once incorporated, is called included text and may consist of both preprocessor and nonpreprocessor statements. Hence, included text can contribute to the preprocessed text being formed.

The general format and the rules governing the use of the %INCLUDE statement are presented in Part II, Section J, "Statements."

The text specified by a %INCLUDE statement is incorporated into the source program immediately after the point at which the statement is executed. The scan therefore continues with the first character in the included text. All preprocessor statements in this text are executed and replacements are made where required.

Preprocessor procedures whose declarations appear in external text can be invoked only after that external text becomes included text. The result of a preprocessor procedure reference encountered before that procedure has been incorporated into the source program is undefined.

Assume that PAYRL is a member of the data set SYSLIB and 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),
     3 REGLR FIXED DECIMAL (8,2),
     3 OVERTIM FIXED DECIMAL (8,2),
   2 RATE,
     3 REGLAR FIXED DECIMAL (8,2),
     3 OVERTIME FIXED DECIMAL (8,2);
```

Then the following sequence of preprocessor statements:

```
%DECLARE PAYROLL CHARACTER;
%PAYROLL='CUM_PAY';
%INCLUDE PAYRL;
%DEACTIVATE PAYROLL;
%INCLUDE PAYRL;
```

will generate two identical structure declarations into the preprocessed text, the only difference being their names, CUM_PAY and PAYROLL. Execution of the first %INCLUDE statement causes the text in PAYRL to be incorporated into the source program. When the scan encounters the identifier PAYROLL in this included text, it replaces it by the current value of the active preprocessor variable PAYROLL, namely, CUM_PAY. Further scanning of the included text results in no additional replacements. The scan then encounters the %DEACTIVATE statement. Execution of this statement deactivates the preprocessor variable PAYROLL and makes the identifier ineligible for replacement. When the second %INCLUDE statement is executed, the text in PAYRL once again is incorporated into the source program. This time, however, scanning of the included text results in no replacements whatsoever, because none of the identifiers in the included text are active. Thus, two structure declarations, differing in name only, are inserted into preprocessed text.

PREPROCESSOR STATEMENTS

This section lists those statements that can be used at the preprocessor stage and briefly discusses those preprocessor statements that have not yet been explained in this chapter. All of the preprocessor statements, their formats, and the rules governing their use are described in the section "Preprocessor Statements" in Part II, Section J, "Statements."

But first, some unrelated comments pertaining to preprocessor statements in general should be made:

1. Some keywords appearing in preprocessor statements can be abbreviated as shown in Part II, Section C, "Keywords and Abbreviations."

2. Comments can appear within preprocessor statements wherever blanks can appear; however, such comments are never inserted into preprocessed text.

3. All preprocessor statements can be labeled. Such labels must appear immediately following the % (only blanks can intervene). All labels must be unsubscripted statement label constants. (Labels on %DECLARE statements are ignored.)

The functions performed by the following preprocessor statements have already been discussed in this chapter:

```
% ACTIVATE
% DEACTIVATE
% DECLARE
% DO
% END
% INCLUDE
% PROCEDURE
RETURN
```

Note that the preprocessor RETURN statement cannot have a leading % because it can be used only within a preprocessor procedure, and all percent symbols must be omitted therein.

Four other statements can be executed at the preprocessor stage:

```
% assignment
% GO TO
% IF
% null
```

The preprocessor assignment statement is used to evaluate preprocessor expressions and to assign the result to a preprocessor variable. All of the examples shown in this section make use of this statement.

The % GO TO statement causes the preprocessor to interrupt its sequential scanning and continue it elsewhere in the source program, specifically at the label speci-

fied in the % GO TO. Thus, it can be useful for rescanning or avoiding text.

The % IF statement can be used to control the sequence of the scan according to the value of a preprocessor expression. It must have a THEN clause and it can have an ELSE clause. Each clause, as well as each preprocessor statement within the clause, must be preceded by a %. Nesting of %IF statements is allowed and must follow the same rules that apply for the nesting of nonpreprocessor IF statements.

The preprocessor null statement is the same as a nonpreprocessor null statement (except for the %). It can be used to provide transfer targets for %GO TO statements or it can be used in nested %IF statements to balance the %ELSE clauses. For example, %ELSE%; is a null ELSE clause.

In PL/I there are often several ways of producing a given effect, but one method is not necessarily as efficient, from a particular point of view, as another. For example, efficient use of storage sometimes affects the efficiency of the object code, and vice versa. The efficiency of the object code is also dependent on such considerations as the number of conversions required and the program structure.

Other factors that can improve performance are the use of based storage and multitasking facilities. These subjects are discussed separately in Chapters 14 and 15.

EFFICIENT PERFORMANCE AND DATA CONVERSION

One of the features of PL/I is the variety of conversions permitted within expressions. It must, however, be appreciated that whenever conversions occur at execution time, there is liable to be some loss of efficiency. The amount of time taken to perform any particular conversion will obviously vary widely and will depend upon the implementation.

Some general questions that a programmer should ask when concerned with the efficiency of a particular statement are:

1. Is a conversion implied? For example, a decimal constant subscript implies conversion to binary before it is used.
2. Will the conversion be performed at compile time?
3. Could the conversion be avoided?
4. Could it be moved outside a loop?

The answers to the first two questions depend upon the implementation. For further details, see IBM System/360 Operating System: PL/I(F) Programmer's Guide, Form C28-6594.

ADJUSTABLE BOUNDS AND STRING LENGTHS

While the use of expressions for bounds of array dimensions and lengths of strings may result in much more effective use of storage, it will also usually result in rather slower object code, since there is less opportunity for performing address calculations at compile time. The degree to which this affects the speed of the program will, of course, depend upon the program and the implementation.

VARYING STRING LENGTHS

VARYING strings do not result in an economy of storage in the F-level implementation, since storage is always allocated to the maximum length specified for a string. They should, therefore, be used only when required by the logic of the program.

BLOCKS AND GROUPS

A DO-group has a much simpler function than a begin block, since it has no effect upon the scope of names, the allocation of storage, and the handling of interrupts. Therefore, DO-groups should be used in preference to begin blocks whenever possible. In the F-level implementation, a begin block requires extra coding for a prologue and an epilogue, while a noniterative DO-group does not usually require extra coding to be generated.

THE ALIGNED AND UNALIGNED ATTRIBUTES

When a programmer is dealing with many data elements, he will usually have to choose between economy of storage and speed of execution. The ALIGNED and UNALIGNED attributes allow him to specify his choice for element or aggregate variables.

The ALIGNED attribute specifies that the implementation is free to choose the boundaries on which data items are to be aligned. This makes it possible for the implementation to speed up the execution of the program at some cost in data storage. In System/360 implementations, ALIGNED bit strings begin on byte boundaries. Since System/360 has character-handling instructions, there is no need to align character strings, and so the attribute is effectively ignored (except that it still prohibits overlay defining).

The effect of the UNALIGNED attribute in System/360 implementations is that data elements are stored in contiguous positions. Character-string items and word and doubleword items are mapped on the next available byte boundary; bit-string items are mapped on the next available bit. This has two main consequences; the most efficient use is made of storage, and the effect of defining a structure on a character- or bit-string element variable is predictable.

Either attribute applied to an array or structure affects the contained members, except those members or elements that are explicitly declared otherwise. Application of either attribute to a contained array or structure overrides an ALIGNED or UNALIGNED attribute that has been declared for the containing structure.

THE USE OF THE PICTURE ATTRIBUTE

A numeric character data item is an entirely different thing from a coded arithmetic item. It is always stored in character form, and it may contain editing characters. Arithmetic operations with numeric character fields always imply conversion, and the conversion may not be trivial.

The principal use of picture characters in PL/I is for editing, and not for computation. If the values are required more than once for computation, it is usually advisable to assign them to a temporary coded arithmetic variable.

The purpose of this chapter is to describe the PL/I based storage facilities currently implemented by the F Compiler, and to give some indication of their use.

INTRODUCTION

Storage allocation is the association of the requisite amount of storage with a variable; it is effectively a two-way process: the storage is associated with a variable, and the variable is associated with the storage. Allocation will be made either statically (that is, before the program is executed), or dynamically (that is, during execution). A statically allocated variable remains allocated for the duration of the program, but a dynamically allocated variable may relinquish its storage before the program has finished.

The storage class attributes determine which kind of allocation is to apply to a given variable. STATIC specifies that allocation will be made statically; AUTOMATIC, CONTROLLED, and BASED each specify a type of dynamic allocation. Automatic storage is allocated automatically on entry to the block in which the variable is declared, and freed automatically when the block is terminated; once freed, the value of the variable is lost. Controlled storage allocation is under the direct control of the programmer, using the ALLOCATE and FREE statements. Based storage allocation is also under the direct control of the programmer, but with some essential differences from controlled allocation.

When the programmer reallocates a controlled variable without first freeing it, the value of the earlier allocation is not lost. All values are held, but in such a way that only one value is available for use at a given time. Effectively, the values are stacked. On the other hand, when a based variable is reallocated without first being freed, all the values are not only held, but are also available for use at any time.

Whenever a based variable is allocated, a pointer variable is set to a value relating to the address of the allocation; by including this pointer variable in a reference to the based variable, the programmer can distinguish between different allocations of one based variable. In other words, reference to the based variable can be qualified by a pointer value. The pointer variable is one of two types of locator variable. The other type, the offset variable, is discussed later.

The based variable can be a structure containing a locator for another allocation, which in turn can contain a locator for yet another allocation, and so on. This is the fundamental concept underlying PL/I list processing; different allocations can be chained together in a specific order. In fact, they can be chained together in several different orders at once by using several different sets of locators. Thus, for example, it is possible to sort a list without duplicating the list items or moving them around; any sequence can be specified by a set of locators. This facility can also be used to chain like items together without necessarily implying a particular order.

A list or chain of associated based variables could be scattered over a large area of storage, linked only by pointers. However, to facilitate input/output and assignment, the based variables can be collected together into a reserved area. The relative locations of the items can then be established. The reason for this provision is that the value of a pointer is absolute and refers to only one allocation of a variable; for example, if a list of associated based structures containing pointers were written out and later read in again, this would constitute a reallocation, within which the pointer values would be meaningless because the addresses would be different. However, another kind of locator variable, called an offset variable, is available, which establishes the location of an item relative to the start of an area. Because it is relative, the value of an offset variable retains its meaning across input/output and assignment.

As well as providing a list processing facility, based storage allows the programmer to make more efficient use of record-oriented input/output. This type of input/output normally involves the use of intermediate buffers and work areas; but a based variable can be virtually overlaid on a buffer, and processing can take place within the buffer. Several separate based variables can be effectively overlaid on the same buffer at once; this allows easy handling of files containing different types of record. (The type of record would be designated within the record itself; the correct based variable could then be

determined from a test made after the record has been read into the buffer.) This type of input/output using based variables is the PL/I form of locate mode input/output.

BASED VARIABLES AND POINTER VARIABLES

A based variable is a variable that can be allocated in more than one location in storage, thus simultaneously representing a number of values, any of which can be retrieved by specifying a pointer variable associated with the relevant storage location.

When a based variable is declared, it is associated with a pointer variable. The form of the declaration is:

```
identifier BASED (pointer-variable)
```

Example:

```
DECLARE X BASED (P);
```

This declaration also contextually declares P to be a pointer variable unless an explicit declaration for P exists. Pointer variables can be declared explicitly, with the following format:

```
identifier POINTER
```

When an unqualified reference is made to the based variable, the value of the pointer variable included in the declaration will be used to determine which allocation is concerned. For example:

```
X = X + 1;
```

In this statement, the pointer variable used to determine the location of X will in both cases be P; that is, the references to X are implicitly qualified by the pointer P. Note, however, that X could have been explicitly qualified by other pointer variables. Explicit pointer qualification is discussed below.

POINTER QUALIFICATION

Reference to a based variable can be explicitly qualified by means of the following format:

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

The pointer variable must be neither subscripted nor based; a qualified name is allowed. For example:

```
P -> X = Q -> X;
```

This statement means simply that the value of one allocation of X is to be assigned to another allocation of X; the X allocated in the location associated with P is to be made equal to the X allocated in the location associated with Q. The appearance of P and Q in the statement contextually declares them as pointer variables, unless explicit declarations exist for P and Q.

The arrow, or pointer qualifier, is a composite symbol made up of a minus sign followed by a greater-than sign. Its equivalent in the 48-character set is PT. It does not signify an operation; its function is similar to that of the period symbol in an ordinary qualified name.

RULES AND RESTRICTIONS

Full details of the rules governing based variables and pointer variables are given under the respective attributes in Section I, Attributes. However, the following points should be carefully noted:

1. Based variables may not have the EXTERNAL, VARYING, or INITIAL attributes.

2. Based label arrays cannot be initialized by subscripted label prefixes.

3. Based variables cannot be checked by means of a CHECK condition prefix.

4. Based variables cannot be transmitted using data-directed input/output.

5. The pointer variable qualifying a based variable (whether implicitly or explicitly) cannot itself be based, nor can it be subscripted; it must be an element variable, or an element of a structure; a qualified name is allowed. (Arrays of pointer variables are allowed, but the value of an element of such an array would have to be assigned to an element pointer variable before it could be used to qualify a based reference.)

6. Pointer variables cannot be operands of any operators except the comparison operators = and ¬=. The value of a pointer variable can be compared with that of any other locator variable, or with a locator value returned by a function.

7. Assignment of a pointer variable value can be made only to another locator variable.

8. Pointer variables cannot be transmitted using STREAM input/output.

9. The pointer variable declared with a based variable is not given the value of the NULL built-in function by the declaration.

10. Only the INITIAL CALL form of the INITIAL attribute is allowed in pointer declarations.

Note: The allocation of a based variable will always take at least eight bytes of storage, even if the based variable is a bit-string variable of length 1.

Pointer Defining

A pointer variable can be defined on another pointer variable using overlay or correspondence defining.

SELF-DEFINING DATA

A self-defining record is one which contains, within itself, information about its own fields, such as the length of a string. PL/I allows the programmer to declare a based structure in a way that is designed to help manipulate such data. The F Compiler supports this to a limited extent: a based structure can be declared to have either one adjustable array bound or one adjustable string length, governed by a variable contained within the structure itself. This variable is given a value when the structure is allocated; the value is assigned from a variable outside the structure. Note that the variable outside the structure is used only on allocation (either by an ALLOCATE statement or by a LOCATE statement); for any other reference to the structure, the variable inside the structure will apply.

The REFER Option

The REFER option is used in the declaration of a based structure to specify that, on allocation of the structure, the value of a variable outside the structure is to be assigned to an element of the structure, and that this value will be the length or bound of another element of the same allocation of the structure.

The REFER option has the following general form:

element-variable REFER (element-variable)

The element variables must be unsubscripted fixed binary variables of default precision (15,0). The variable on the left-hand side of the keyword must not belong to the structure; it can be qualified or pointer-qualified. The variable on the right-hand side must belong to the structure.

For example:

```
DCL 1 STR BASED (P),
      2 Y FIXED BINARY,
      2 Z (B.X REFER (Y));
```

This declaration specifies that the based structure STR will consist of an array Z and an element Y; when STR is allocated, the upper bound of Z is set equal to the current value of B.X, and this value is assigned to Y. For any other reference to the variable, the bound value is taken from Y.

Note that this option can be used only once in a declaration. If it is used to specify an array bound, the bound must be the upper bound of the leading dimension of the element with which it is used, and the dimension must belong to the last element in the structure declaration, or to a minor structure containing the last element.

For example:

```
DCL 1 STR BASED (P),
      2 A,
        3 B FIXED BINARY,
        3 C CHARACTER (20),
      2 D,
        3 E FIXED BINARY,
        3 F (0:X REFER (E), 0:9);
```

In this declaration, the REFER option is used to specify an adjustable upper bound for the array F; in this case, it could not have appeared in any place other than that shown.

Note: Since the adjustable bound must be part of the leading dimension of the element with which it is declared, it is

LOCATE WITH AND WITHOUT SET

The LOCATE statement has the following basic format:

```
LOCATE based-variable FILE (file-name)
  [SET (pointer-variable)];
```

The pointer variable can be any variable that represents a single pointer value.

This statement allocates storage, in an output buffer, for a based variable. The action is similar to that of the READ and SET, in that the based variable is, in effect, overlaid on the buffer. In this case, however, data is moved (by subsequent statements) into the output buffer in such a way that the fields of the record are located relative to the elements of the based variable; the record is automatically written onto the specified file immediately before execution of the next WRITE, LOCATE, or CLOSE statement (or implicit close operation) for the file. This means that the programmer must assign values to the variable after allocation and before the next input/output operation on the file.

Again, a pointer variable is set to point to the buffer. This pointer variable will be that specified in the SET option, if the option appears; if the option is omitted, the pointer variable that was declared with the specified based variable is set.

not possible for that element to inherit a dimension from a higher level. (Inherited dimensions would automatically become the leading dimensions of the lower-level member.)

For example:

```
DCL 1 STR BASED (P),
      2 D (10),
        3 E (50),
        3 F (50);
```

In this declaration, both E and F would have implied bounds of 1:10, inherited from D; the REFER option could not have been used with them but could have been used with D (in place of 10).

If the REFER option is used to specify a string length, that string must be an element variable, and it must be the last element variable in the structure declaration.

If the element variable on the right-hand side of REFER varies during the program then:

1. The structure must not be freed until the element variable is restored to the value it had when allocated;

2. The structure must not be written out while the element variable has a value greater than the value with which it was allocated.

3. The structure may be written out when the element variable has a value equal to or less than the value it had when allocated. The number of elements or the string length actually written will be that indicated by the current value of the variable.

For example:

```
DCL 1 REC BASED (P),
      2 N,
      2 A (M REFER(N)),
    M INITIAL (100);
        .
        .
        .
ALLOCATE REC;

N = 86;

WRITE FILE (X) FROM (REC);
```

In this example, 86 elements of REC are written. It would be an error to attempt to free REC at this point, since N must be restored to the value it had when allocated (i.e. 100). If N was assigned a value greater than 100, an error would occur when the WRITE statement was encountered.

POINTER SETTING, BASED STORAGE ALLOCATION, AND INPUT/OUTPUT

Before a reference can be made to a based variable, the qualifying pointer variable must have a value. This value can be set in any of five different ways:

1. With the SET option of a READ statement;

2. By a LOCATE statement;

3. By an ALLOCATE statement;

4. By assignment of the value of another locator variable, or a locator value returned by a user-defined function;

5. By assignment of an ADDR built-in function value.

Note that the actual value is in all cases set by the implementation. The programmer has no direct control over addressing; he cannot, for example, assign a constant to a pointer variable.

A special form of assignment to a pointer variable is made using the NULL built-in function. This assigns a special value to the pointer, that cannot be related to any address; its purpose is to give a positive indication that the pointer does not currently identify any allocation of a variable.

READ WITH SET

The READ statement with the SET option has the following basic format:

```
READ FILE (file-name)
  SET (pointer-variable);
```

The pointer variable can be any variable that represents a single pointer value. This form of the READ statement causes a record to be read into a buffer and the specified pointer variable to be set to point to the buffer. A based variable reference, qualified by the same pointer, will then relate to the fields of the record.

A based variable used to describe a record in a buffer is effectively overlaid on the buffer. The result of a reference to an element of the based variable is the same as it would be if the record had been read directly into the structure described.

ALLOCATE WITH AND WITHOUT SET

The ALLOCATE statement, as used with based variables, has the following basic format:

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

The effect of this statement is similar to that of the LOCATE statement, in that it allocates storage for the based variable and sets a pointer to point to the allocation. In this case, however, no output is implied; the storage is not allocated in a buffer. If the SET option appears, the specified pointer variable is set; if the option is omitted, the pointer variable that was declared with the specified based variable is set.

The IN option, if included, specifies that the allocation is to be made within the reserved area of storage named. Areas are discussed in detail later in this chapter. The area variable can be any variable that represents a single area; the pointer variable can be any variable that represents a single pointer value.

POINTER ASSIGNMENT

The value of a pointer variable may be assigned to another pointer variable in a simple assignment statement. Assume that P and Q are pointer variables and that P has a valid pointer value.

```
Q = P;
```

This statement specifies that Q is to be set to point to the same location that P points to. A reference to a based variable qualified by Q will then be effectively identical to a reference to the same based variable qualified by P. For example (assuming that X is a based variable associated with the pointer P by declaration), the references X, P -> X, and Q -> X will be identical in effect.

The ADDR Built-in Function

The value returned by the ADDR built-in function is a valid pointer value that specifies the location of a data variable named as the argument of the function reference. For example:

```
P = ADDR (X);
```

Execution of this statement will give the pointer variable P a value so that it points to the location of the data variable X. The value of an ADDR function reference can be assigned to a locator variable only.

The argument can be a variable that represents an element, an array, a structure, an area, an element of an array, a minor structure, or an element of a structure. The value returned is always a pointer value. Note that if a based variable has not been allocated, its ADDR is undefined; however, the ADDR value of an unallocated controlled variable is null.

The ADDR of an element of an array or structure returns a value that relates to the address of the element. However, a pointer qualifying a subscripted or qualified variable is assumed to point to the array or structure in which the element is contained, not to the element itself. For example:

```
DCL A (10,10) CHARACTER (20) BASED (P),
    B CHARACTER (20) BASED (Q),
    C (10,10) CHARACTER (20);
```

Given this declaration, if ADDR (C) is assigned to P, then A (1,1) will refer to the first element of C. If ADDR (C(2,3)) is assigned to Q, then B is effectively overlaid on the third element in the second row of C. (This technique, like the other overlaying techniques made possible by the use of based variables and pointers, is extremely powerful; however, such techniques should be used only with the understanding that the compiler has no means of recognizing incompatibilities between the attributes of the based variable and the attributes of the variable being effectively overlaid.)

Since ADDR returns a single value only, the elements of an array or structure argument must occupy successive locations in storage. For example:

```
DCL A(10,10);
```

For the array declared above, ADDR would not be permitted for the cross-section A(*,10), because each element in the cross-section would belong to a different row, and would be separated from its column neighbor by other elements in its row. ADDR would, however, be permitted for the cross-section A(10,*); this cross-section consists of one entire row whose elements occupy successive locations in storage.

Note also that since the F Compiler implementation of based storage does not support bit addressing, the argument to the ADDR built-in function must be aligned on a byte (or higher) boundary. In the case of

bit strings belonging to unaligned arrays and structures, therefore, ADDR should be used only for the level 1 name or for minor structures that are not composed entirely of bit strings.

The NULL Built-in Function

The NULL built-in function requires no arguments; it returns a null pointer value (that is, a special pointer value that cannct relate to any address in storage). Its purpose is to provide a positive indication that a pointer does not currently identify any allocation of a variable. Examples of its use include the following:

1. If NULL is assigned to a pointer at the start of a program, a later test of the pointer against NULL will show whether a based variable qualified by the pointer has been allocated or not.

2. A terminal pointer in a chain can be set to the value of NULL so that the beginning or end of the chain can be recognized.

FREEING BASED STORAGE

The storage that has been associated with a based variable by one of the allocation methods described above can be freed explicitly or, in certain cases, implicitly, for possible re-use. Once the storage for a based variable has been freed, a reference to the associated pointer becomes invalid.

THE FREE STATEMENT

The FREE statement, as applied to based variables, has the following basic format:

```
FREE qualified-reference
  [IN(area-variable)]
  [,qualified-reference
  [IN(area-variable)]]...;
```

where "qualified-reference" is defined as:

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

This statement frees the storage associated with one or more allocations of one or more specified based variables. The allocations are identified by the current values of the specified pointers. If a pointer is omitted, it is assumed to be that declared with the based variable concerned.

IN (area-variable) must be specified if the allocation was made within an area; otherwise, it must be omitted. Areas are discussed later in this chapter.

The amount of storage freed depends on the attributes of the based variable, including the current value of any adjustable bound or length specification. The programmer is responsible for ensuring that the amount freed coincides with the amount originally allocated. For example:

```
DECLARE 1 S BASED (P),
          2 N,
          2 X(M REFER (N));
.
.
.
M = 50;
ALLOCATE S;
/*X HAS 50 ELEMENTS AND THE VALUE OF N IS
SET TO 50*/
.
.
.
M = 80; /*THIS HAS NO EFFECT ON THE
CURRENT ALLOCATION OF S*/
P -> N = 10;
FREE S;
/*THIS IS IN ERROR BECAUSE STORAGE EQUI-
VALENT TO 40 ELEMENTS OF X IS LEFT
UNFREED*/
```

Based storage allocated in a task cannot be freed by a FREE statement in a descendant task, unless it has been allocated within an area belonging to the descendant task (that is, an area that was allocated in the descendant task).

It is an error to attempt to free based variables that have not been allocated.

IMPLICIT FREEING

In certain circumstances, based storage is freed without the use of an explicit FREE statement, as follows:

1. Storage that has been allocated by the LOCATE statement is freed after the variable is written out.

2. Storage that has been effectively allocated by a READ statement with the SET option is freed by the next read or close operation on the file.

3. All storage is freed at the end of the task in which it was allocated, unless it was allocated within an area

belonging to another task. (Storage allocated within an area is freed on termination of the task in which the area was allocated.)

AREAS AND OFFSETS

Based variables can be allocated within an area of storage that has been reserved by allocation of an area variable. This has the effect of grouping the data items together so that they can be easily transmitted or assigned as a single unit while still retaining their individual identities. The items stay in their relative locations, which can be identified by offsets from a pointer that identifies the start of the area. This does not mean that pointers cannot be used within areas; however, offsets have the advantage of remaining valid during area transmission and assignment.

Offsets, like pointers, can be used to build chains of data; however, they cannot be used directly as based variable qualifiers nor can they appear in a SET option. Assignment from pointer to offset implies conversion to offset; similarly, assignment from offset to pointer implies conversion to pointer. Hence, an offset variable can be given a value by assigning a pointer value to it; and, in order to use an offset as a qualifier, its value is assigned to a nonbased pointer.

AREA VARIABLES

The AREA attribute defines an area of storage that is to be reserved for the allocation of based variables. It has the following general format:

```
AREA [(expression)]
```

The number of bytes of storage is specified by the integral value of the expression, if present; otherwise, an implementation defined value of 1000 bytes is assumed. This value is the size of the area.

The implementation defined maximum size of an area is 32,767 bytes.

The size of an area is the amount of storage that is reserved by the area allocation for the allocation of based variables. The amount of the reserved storage that is actually in use is known as the extent of the area; it is defined as the amount of storage between the start of the reserved area and the end of that unfreed allocation which is furthest from the start of the area. In addition to the declared size, the implementation requires an extra 16 bytes of control information, giving such details as the size, and the length of the current extent. These 16 bytes are allocated immediately before the start of the reserved area, and are added to the area size to obtain the length of the area (that is, the actual amount of storage needed for the area allocation). The distinction between area size and length is important to the discussion of area input/output later in this chapter.

Example:

```
DECLARE A STATIC AREA(2000),
        B AREA,
        C AREA (N);
```

This statement specifies that:

1. A is a static area variable reserving 2000 bytes of storage. (The size of an area of static storage class, if specified, must be specified as an unsigned fixed decimal integer constant.)
2. B is an automatic area variable reserving 1000 bytes of storage.
3. C is an automatic area variable whose size depends on the value of N current at the time of entry to the block.

Rules and Restrictions

The following rules apply to area variables:

1. Data of the area type cannot be converted to any other data type; an area can be assigned to an area variable only.
2. No operators can be applied to area variables.
3. Only the INITIAL CALL form of the INITIAL attribute is allowed with area variables.
4. When an area is allocated, it is automatically given the EMPTY state (see "The EMPTY Built-in Function" in this chapter, for explanation of EMPTY).

OFFSET VARIABLES

Declaration of an offset variable must be explicit. The OFFSET attribute has the following form:

```
OFFSET (variable)
```

The variable within the parentheses must be an unsubscripted level 1 based area variable.

The function of an offset variable is to provide a locator value that points to the location of a based variable relative to the start of a based area. If the containing area is transmitted or assigned, the offsets will still point to the correct locations of the components.

Example:

```
DECLARE A AREA BASED(P),
        O OFFSET(A),
        X BASED(Q);
```

This declaration specifies that A is a based area variable, that the value of O will point to a location relative to the start of A, and that X is a based variable. If X were now allocated within A, the value of its pointer could be assigned to O to establish the location of X relative to the start of A. If A and O were written out and then read back in again, O would still point to X relative to the start of A, although the pointer for A itself would have been reset.

Rules and Restrictions

The following rules apply to offset variables:

1. Offset variables cannot be used to qualify a based reference.
2. Assignment of an offset value can be made only to a locator variable. When an offset value is assigned to an offset variable, the area variables named in the OFFSET attributes are ignored. A pointer value can be assigned to an offset variable, with implicit conversion.
3. Offset variables cannot be operands of any operators except the comparison operators = and ¬=. The value of an offset variable can be compared with that of any other locator variable, or with a locator value returned by any function.
4. Offset variables cannot be transmitted using STREAM input/output.
5. Offset variables cannot appear in any SET option.

ALLOCATION WITHIN AN AREA

Based variables are allocated within an area by means of an ALLOCATE statement with the IN option. This sets a pointer which can be converted to offset by assignment to an offset variable. For example:

```
DECLARE A AREA BASED(V),
        1 B BASED(P),
          2 O OFFSET(A),
          2 VALUE,
        Q POINTER;

ALLOCATE A;
.
.
.
ALLOCATE B IN (A);
.
.
.
ALLOCATE B IN (A) SET (Q);
O=Q;
```

The first ALLOCATE statement causes the area A to be allocated, reserving 1000 bytes of storage for allocation of based variables, and sets V.

The second ALLOCATE statement causes B to be allocated within the area V -> A, and sets P.

The third ALLOCATE statement causes another allocation of B (different from P -> B) to be made within the area V -> A, and sets Q.

The assignment statement causes the value of Q to be converted to offset (relative to the pointer V) and assigned to P -> O. Thus, the first allocation of the structure B contains an offset value that points to the second allocation of B. The setting of offset values is discussed below.

SETTING OFFSET VALUES

An offset variable can be given a value by assignment only, since it cannot appear in a SET option, nor is any implicit setting possible. In the above example, P -> O was given its value by assignment from Q. Note, however, that the reference

O -> VALUE, for example, would be invalid, since offsets cannot be used as qualifiers.

The NULLO Built-in Function

The NULLO built-in function is the offset equivalent of the NULL built-in function as used with pointers. It requires no arguments, and returns a null value that can be assigned to offset variables only.

Note: A null offset value cannot be converted to a null pointer value, nor can a null pointer value be converted to a null offset value. Therefore, the value of the NULLO built-in function cannot be assigned, even indirectly, to a pointer variable; nor can the value of the NULL built-in function be assigned to an offset variable. For example:

```
DECLARE O OFFSET(A),
        P POINTER;
.
.
.
P=NULL;
O=P;
O=NULLO;
P=O;
```

The second and fourth assignments in the above example would be invalid. They could be made valid by inserting IF statements, such as the following:

```
IF P=NULL THEN O=NULLO;
   ELSE O=P;
```

AREA ASSIGNMENT AND INPUT/OUTPUT

The value of an area expression can be assigned to one or more area variables by an assignment statement. Area-to-area assignment has the effect of freeing all allocations in the target area and then assigning the extent of the source area to the target area, in such a way that all allocations in the source area maintain their locations relative to each other; that is, any gaps left by freeing operations in the source area are maintained during the assignment (such a gap might have been left, for example, if the second of three contiguous allocations had been freed; if the gaps were automatically closed up, some offset values might lose their meaning).

If a source area containing no allocations is assigned to a target area, the effect is merely to free all allocations in the target area.

A possible use for area assignment is to allow for expansion of a list of based variables beyond the bounds of its original area. When an attempt is made to allocate a based variable within an area that contains insufficient free storage to accommodate it, the AREA condition is raised (see below). The on-unit for this condition could be to change the value of a pointer qualifying the reference to the inadequate area, so that it pointed to a different area; on return from the on-unit, the allocation would be attempted again, within the new area. Alternatively, the on-unit could write out the area and reset it to EMPTY.

The EMPTY Built-in Function

The EMPTY built-in function requires no arguments and returns an area of zero size and extent. The effect is to free all allocations in the target area.

Example:

```
DECLARE A AREA,
        I BASED(P),
        J BASED(Q);
.
.
.
ALLOCATE I IN(A), J IN (A);
.
.
.
A=EMPTY;
/*EQUIVALENT TO:
  FREE I IN (A), J IN (A); */
```

The AREA ON-Condition

The AREA condition is raised in any of the following circumstances:

1. When an attempt is made to allocate a based variable within an area that contains insufficient free storage for the allocation to be made.
2. When an attempt is made to perform an area assignment, and the target area contains insufficient storage to accomodate the extent of the source area.
3. When a SIGNAL AREA statement is executed.

The ONCODE built-in function can be used to determine whether the condition was raised by an allocation, an assignment, or a SIGNAL statement.

On normal return from the on-unit, the action is as follows:

1. If the condition was raised by an allocation, the allocation is re-attempted. If the on-unit has changed the value of a pointer qualifying the reference to the inadequate area so that it points to another area, the allocation is reattempted within the new area. Note that if the on-unit does not effectively correct the fault, a loop may result.

2. If the condition was raised by an area assignment, or by a SIGNAL statement, execution continues at the point of interrupt.

If no on-unit is specified, the system will comment and raise the ERROR condition.

Input and Output

The area facility is designed to allow easy input and output of complete lists of based variables as one unit, to and from RECORD files. The control information is transmitted with the area. Consequently, the record length required is governed by the area length (i.e., area size + 16): the RECORD condition is raised if the length of an area named in the INTO option of a READ statement, or in the FROM option of a WRITE statement, differs from the relevant record length. Note that even though the RECORD condition is raised, incorrect control information will be transmitted; when an area is written out, it must not be read back into an area of different size.

In the case of READ with SET, the length of the input area in the buffer is equal to the length of the area used to create the record.

AREA AND OFFSET DEFINING

An offset can be defined on an offset, using overlay or correspondence defining. In the declarations of the defined offset and the base offset, the variables named in the two OFFSET atttributes need not be the same.

Similarly, an area can be defined on an area, using overlay or correspondence defining. The base area must have a size equal to that of the defined area.

COMMUNICATION BETWEEN PROCEDURES

Similarly to variables of other data types, locator and area variables in one procedure can be related to those in another procedure by means of arguments and parameters, and the general rules are as described in Chapter 10, "Subroutines and Functions." There are necessarily some restrictions, which will be explained in the following discussion; but a general rule is that where it is possible to assign the value of one variable to another variable, it is also possible to relate the two variables by an argument and a parameter.

ARGUMENTS AND PARAMETERS

A locator argument of either pointer or offset type can be passed to a locator parameter only. The parameter can be of either type, but if the argument type differs from the parameter type, a dummy argument is created by the compiler, and a change in the value of the parameter will not be reflected in the value of the original argument.

Pointer to Pointer

No conversion is necessary when a pointer argument is passed to a pointer parameter; normally, therefore, no dummy argument is created, and a change in the value of the parameter will be reflected in the value of the argument. Note, however, that this reflected change could be avoided, if necessary, by passing the argument as an expression in parentheses: this causes a dummy argument to be created. For example:

```
PROC1:  PROCEDURE;
        DECLARE (P,Q) POINTER;
        .
        .
        .
        CALL PROC2((P),Q);
PROC2:  PROCEDURE(R,S);
        DECLARE (R,S) POINTER;
        .
        .
        .
END PROC1;
```

In this example, a change in the value of S will be reflected in the value of Q, but a change in the value of R will have no effect on P.

Offset to Pointer

Passing an offset argument to a pointer parameter implies conversion to a dummy pointer argument, which is then passed to the entry point. The entry must be declared with the POINTER attribute in the parameter attribute list. For example:

```
PROC3:  PROCEDURE;
        DECLARE    PROC4    ENTRY
        (POINTER),
                O OFFSET(A),
                A AREA BASED(P);
        .
        .
        .
        CALL PROC4(O);
PROC4:  PROCEDURE(Q);
        DECLARE Q POINTER;
        .
        .
        .
END PROC3;
```

In this example, the values of P and O are used to obtain the value of the dummy pointer argument to be passed to PROC4.

Offset to Offset

When an offset argument is passed to an offset parameter, variables named in the OFFSET attribute of the offset declarations are ignored, just as they are ignored for offset assignment; if they differ, the fact that they differ does not imply conversion to a dummy argument. For example:

```
PROC5:  PROCEDURE;
        DECLARE OA OFFSET(A),
                A AREA BASED(P),
                B AREA BASED(Q),...
        .
        .
        .
        CALL PROC6(OA);
PROC6:  PROCEDURE(OB);
        DECLARE OB OFFSET(B),...
        .
        .
        .
END PROC5;
```

In this example, OA would be passed directly to OB.

Pointer to Offset

Passing a pointer argument to an offset parameter implies conversion to a dummy offset argument, which is then passed to the entry point. The entry must be declared with the OFFSET attribute in the parameter attribute list, and the two OFFSET attribute specifications must name the same variable. For example:

```
PROC7:  PROCEDURE;
        DECLARE PROC8 ENTRY (OFFSET(A)),
                P POINTER,
                A AREA BASED(Q);
        .
        .
        .
        CALL PROC8(P);
PROC8:  PROCEDURE(O);
        DECLARE O OFFSET(A);
        .
        .
        .
END PROC7;
```

In this example, the values of P and Q are used to obtain the value of the dummy offset argument to be passed to PROC8.

The variable following the keyword OFFSET is not considered during selection of a generic entry point.

Area to Area

An area argument can be passed only to an area parameter. If the size of the argument differs from the size appearing in the parameter attribute list of the relevant entry declaration, the argument is first assigned to a dummy area argument with the size specified in the entry declaration; the dummy argument is then passed to the entry point.

The size of an area argument is not considered during selection of a generic entry point.

RETURNS FROM ENTRY POINTS

An entry point can return a locator value or an area; hence, the PROCEDURE and ENTRY statements and the RETURNS attribute may specify the POINTER, OFFSET, or AREA attributes.

Locator Returns

Either type of locator variable can appear in a RETURN statement in a procedure that returns a locator value. If the

procedure is to return an offset value but the RETURN statement specifies a pointer, there is implicit conversion to offset, and vice versa. For example:

```
PROCA:  PROCEDURE POINTER;
        DECLARE A AREA BASED(P),
                O OFFSET(A);
        .
        .
        .
        RETURN (O);
END PROCA;
```

The values of O and P are used to obtain the pointer value to be returned.

```
PROCB:  PROCEDURE OFFSET(B);
        DECLARE B AREA BASED(Q),
                R POINTER;
        .
        .
        .
        RETURN(R);
END PROCB;
```

The values of Q and R are used to obtain the offset value to be returned. Note that the OFFSET attribute is specified in the PROCEDURE statement complete with the name of the relevant area variable; the keyword OFFSET alone is not sufficient.

Similarly, a locator value returned by a function may undergo implicit conversion. For example:

```
DECLARE O ENTRY RETURNS(OFFSET(A)),
        A AREA BASED(P),
        Q POINTER;
.
.
.
Q=O;
```

The value of P and the value returned by O are used to obtain the pointer value to be assigned to Q.

Area Returns

If a return statement identifies an area that has an extent different from that specified in the relevant PROCEDURE or ENTRY statement, assignment is made to a dummy area with the correct extent, thus effectively performing a conversion.

VARIABLE LENGTH PARAMETER LISTS

In PL/I, a procedure can have only a fixed number of parameters, all of which must be specified. However, by passing an array of pointers as a single argument, it is possible to simulate a variable length parameter list, since the array can have adjustable bounds.

The following procedure sorts a variable number of based character-string variables according to their values in relation to the collating sequence. The pointers qualifying these based variables are passed as an array argument to the procedure.

Assume that the calling procedure contains an array of pointers, KEYPOINTS, with one dimension, which is named as an argument in the CALL statement, and whose elements each point to a based character-string variable.

```
SORT:  PROCEDURE(P);
       DECLARE
         P(*) POINTER,
         (H,L) FIXED BINARY,
         LISTEL BASED (POINTER1)
         CHARACTER(60),
         POINTER2 POINTER;

       H=HBOUND(P,1);
       L=LBOUND(P,1);
       /*THE HBOUND AND LBOUND BUILT-IN
       FUNCTIONS RETURN THE UPPER AND
       LOWER BOUNDS OF THE SPECIFIED
       DIMENSION (IN THIS CASE, THE FIRST
       AND ONLY DIMENSION).  THESE VALUES
       ARE USED IN SETTING THE CONTROL
       VARIABLES OF THE FOLLOWING
       DO-GROUPS SO THAT THE NUMBER OF
       ITERATIONS IS CORRECT FOR THE
       NUMBER OF PARAMETERS*/

  I1:  DO I=L TO H-1;
         POINTER1=P(I);
         /*THE VARIABLE LISTEL NOW HAS A
         VALUE*/

         DO J=I+1 TO H;
           POINTER2=P(J);
           /*THIS IS NECESSARY, SINCE THE
           IMPLEMENTATION DOES NOT
           SUPPORT SUBSCRIPTED POINTER
           QUALIFIERS*/

         IF LISTEL /*IMPLICITLY QUALIFIED
                   BY POINTER1*/
           >POINTER2->LISTEL
         THEN DO;
           /*REORDER ARRAY ELEMENTS*/
           P(I)=P(J);
           P(J)=POINTER1;
           POINTER1=P(I);

END I1;
END SORT;
```

After execution of this procedure, the elements of KEYPOINTS will have been rearranged so that the first element points

to the based variable with the lowest value according to the collating sequence, the second element points to the based variable with the next lowest value, and so on. Thus, the based variables will have been logically sorted without changing the physical order of the data.

EXAMPLES OF LIST PROCESSING TECHNIQUE

The following examples illustrate the use of based storage, locator variables, and areas, for list processing and input/output.

Example 1

This procedure builds a two-directional chain through items that are allocated in the calling procedure and identified in turn by passing a pointer parameter. Each item consists of an allocation of a basic structure that contains two pointers and a data value (in this case, a character string). One pointer identifies the preceding item, and the other identifies the following item. The ends of the chain are recognized by a null value for a contained pointer (for example, the backwards pointer in the first item is null). The locations of the ends of the chain are identified by a head pointer and a tail pointer. Figure 14-1 shows a diagrammatic representation of the chain.

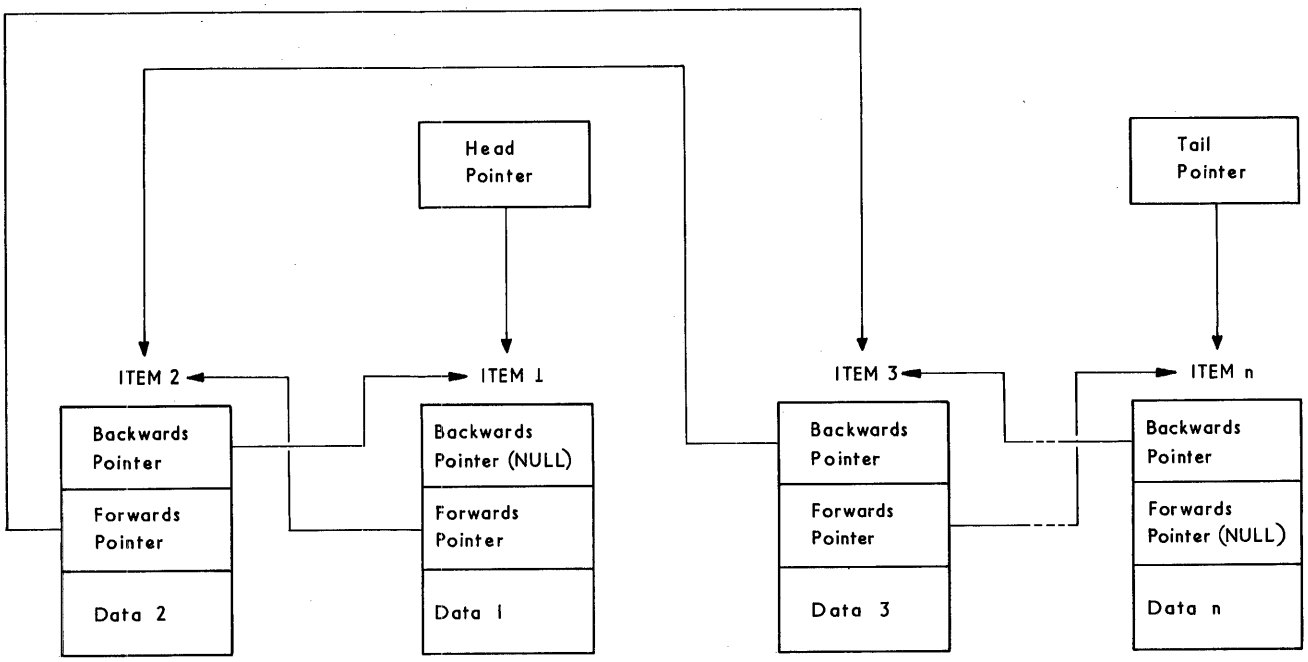


Figure 14-1. Example of Two-Directional Chain

```
/*EXAMPLE 1*/
BUILD_CHAIN: PROCEDURE(ELEMPTR);
  DECLARE
    1  ELEMENT BASED(ELEMPTR),
         2  BACK_CHAIN POINTER,
         2  FWD_CHAIN POINTER,
         2  DATA CHARACTER(50),
    ELEMPTR POINTER,
    (HEAD, TAIL) POINTER STATIC EXTERNAL;

  /*ASSUME THAT HEAD AND TAIL ARE
  INITIALLY ASSIGNED THE VALUE OF THE
  NULL BUILT-IN FUNCTION IN THE PROCEDURE
  THAT CALLS BUILD_CHAIN*/

  IF HEAD=NULL
    THEN /*FIRST ELEMENT*/
      HEAD=ELEMPTR; /*SET HEAD POINTER*/

    ELSE /*NOT FIRST ELEMENT*/
      TAIL->FWD_CHAIN=ELEMPTR;
      /*UPDATE FWD CHAIN*/

  BACK_CHAIN=TAIL;
  /*UPDATE BACK CHAIN*/

  TAIL=ELEMPTR; /*UPDATE TAIL POINTER*/

  FWD_CHAIN=NULL; /*SET END INDICATOR OF
  FWD CHAIN*/
  END BUILD_CHAIN;
```

Note that the parameter ELEMPTR may identify a nonbased structure, provided that this structure has the same structuring and attributes as ELEMENT.

Example 2

This procedure deletes an item from the chain created by the procedure in example 1. The item to be deleted is identified by a pointer parameter.

```
/* EXAMPLE 2 */
ALTER_CHAIN: PROCEDURE(ELEMPTR);
  DECLARE
    1 ELEMENT BASED(ELEMPTR),
        2 BACK_CHAIN POINTER,
        2 FWD_CHAIN POINTER,
        2 DATA CHARACTER(50),
    ELEMPTR POINTER,
    (HEAD, TAIL) POINTER STATIC EXTERNAL,
    (PRED, SUCC) POINTER STATIC;

/*SET POINTERS TO PREDECESSOR AND
  SUCCESSOR OF ELEMENT BEING DELETED.
  PRED AND SUCC ARE USED BECAUSE
  BACK_CHAIN AND FWD_CHAIN, BEING BASED,
  CANNOT BE USED AS QUALIFIERS*/

PRED=BACK_CHAIN;
SUCC=FWD_CHAIN;

/*UPDATE FORWARD CHAIN*/
IF PRED=NULL
    THEN HEAD=SUCC; /*DELETE HEAD*/
    ELSE PRED->FWD_CHAIN=SUCC;

/*UPDATE BACKWARD CHAIN*/
IF SUCC=NULL
    THEN TAIL=PRED; /*DELETE TAIL*/
    ELSE SUCC->BACK_CHAIN=PRED;
END ALTER_CHAIN;
```

Example 3

This procedure builds a sequential list through several allocations of an area variable. Within each area allocation, the procedure builds a chain of structure allocations, each of which contains an offset identifying the following item in the chain, a character string value, and a value (passed from the calling procedure) indicating the length of the string. The location of the first item in the chain is indicated by an offset attached to the area. This offset is part of a structure containing the first offset and the area; consequently, the area is a level 2 variable. Since a level 2 variable cannot be named in the OFFSET attribute, a dummy level 1 area variable is effectively overlaid on the level 2 area, and this dummy is named in the OFFSET attributes.

The procedure sets pointers to the start of the area and to each item in the area. These pointers are external, and are therefore known to the calling procedure.

Each area allocation is in output buffer space, and when filled, is written onto a data set, using locate-mode output. This output process is controlled by an on-unit for the AREA condition. The items in the area are chained by offsets to ensure that the chain is not invalidated by input/output operations on the list. It is assumed that the output file is opened and closed by the calling procedure.

```
/*EXAMPLE 3*/
BUILD_LIST: PROCEDURE(N);
  DECLARE
    N FIXED BINARY,
    1 LIST BASED(LISTPTR),
      2 FIRST OFFSET(DUMMY),
      2 BODY AREA,
    1 ELEM BASED(ELEMPTR),
      2 CHAIN OFFSET(DUMMY),
      2 STRING,
          3 LENGTH FIXED BINARY,
          3 DATA CHARACTER(N REFER
          (LENGTH)),
    (ELEMPTR, LISTPTR) POINTER STATIC
    EXTERNAL, /*THESE POINTERS ARE
    INITIALIZED TO NULL BY THE CALLING
    PROCEDURE*/
    LFILE FILE RECORD SEQUENTIAL
    EXTERNAL,
    LASTELEM POINTER STATIC,
    DUMMY AREA BASED(DPTR);
```

```
    ON AREA
      BEGIN; /*ALLOCATE OUTPUT BUFFER
      SPACE*/
          LOCATE LIST FILE(LFILE) SET
          (LISTPTR);
          DPTR=ADDR(BODY);
          LASTELEM=NULL; /*INDICATES NEW
          AREA*/
          END;

    IF LISTPTR=NULL
    THEN SIGNAL AREA; /*CREATE FIRST AREA*/
    ALLOCATE ELEM IN (BODY); /*ELEMPTR IS
    SET AUTOMATICALLY*/
    IF LASTELEM=NULL /*SET FORWARD CHAIN*/
      THEN FIRST=ELEMPTR; /*FIRST ELEMENT
      OF AREA*/
      ELSE LASTELEM->CHAIN=ELEMPTR; /*OTHER
      ELEMENTS*/
    CHAIN=NULLO; /*SET END-OF-CHAIN
    INDICATOR*/
    LASTELEM=ELEMPTR; /*SAVE POINTER TO NEW
    ELEMENT*/
  END BUILD_LIST;
```

Note that LFILE in examples 3 and 4 should have a record length of 1020 to accommodate the records created by allocations of the structure LIST. This is made up of 1000 bytes (default size for an area) plus 16 bytes of area control information, plus 4 bytes for the offset variable FIRST.

Example 4

This procedure sequentially retrieves the list items created by the procedure in example 3. The procedure sets a pointer to the next item in the list, or if the item has been retrieved, sets the pointer to null.

```
  /*EXAMPLE 4*/
  GET_ELEMENT: PROCEDURE;
  /*ASSUME THE SAME DECLARATIONS AS IN
  EXAMPLE 3, AND ASSUME THAT LISTPTR IS
  INITIALIZED TO NULL BY THE CALLING
  PROCEDURE*/

    ON ENDFILE(LFILE)
      BEGIN;
        ELEMPTR=NULL; /*ALL ELEMENTS
        RETRIEVED*/
        CLOSE FILE(LFILE);
        GO TO EXIT;
      END;

    IF LISTPTR=NULL /*FIRST ELEMENT TEST*/
      THEN DO;
        OPEN FILE(LFILE);
        GO TO READ_AREA;
      END;

    IF LASTELEM->CHAIN=NULLO /*END-OF-AREA
    TEST*/
      THEN READ_AREA: /*READ RECORD INTO
      BUFFER*/
        DO;
        READ FILE(LFILE) SET (LISTPTR);
        DPTR=ADDR(BODY);
        ELEMPTR=FIRST; /*SET PTR TO FIRST
        ELEMENT*/
        END;
      ELSE ELEMPTR=LASTELEM->CHAIN; /*SET
      POINTER TO FOLLOWING ELEMENT*/
    LASTELEM=ELEMPTR; /*SAVE POINTER TO NEW
    ELEMENT*/
  EXIT: END GET_ELEMENT;
```

CHAPTER 15: MULTITASKING

The use of a computing system to execute a number of operations concurrently is broadly termed multiprogramming. The PL/I programmer can make use of the multiprogramming capability of the system by means of the multitasking facilities described in this chapter.

INTRODUCTION

A PL/I program is a set of one or more procedures, each of which consists of a set of PL/I statements. The execution of these statements constitutes one or more tasks, each of which can be identified by a different task name. A task is dynamic; it exists only while the program is being executed. This distinction between the program and its execution is essential to the discussion of multitasking. One set of statements could be executed several times in different tasks.

When the multitasking facilities are not used, the execution of a program constitutes a single task, with a single flow of control; when a procedure invokes another procedure, control is passed to the invoked procedure, and execution of the invoking procedure is suspended until the invoked procedure passes control back to it. This serial type of operation is said to be synchronous; when the programmer is concerned only with synchronous operations, the distinction between program and task is relatively unimportant.

With multitasking, the invoking procedure does not relinquish control to the invoked procedure. Instead, an additional flow of control is established, so that both procedures can be executed (in effect) concurrently. This process is known as attaching a task. The attached task is a subtask of the attaching task. Any task can attach a number of subtasks. The task that has control at the outset is called the major task. This parallel type of operation is said to be asynchronous.

The diagram shown in Figure 15-1 illustrates the difference between synchronous and asynchronous operations. The arrowed lines represent the control flows. Procedures A and B are executed synchronously; C and D are executed asynchronously.

When several procedures are executed as asychronous tasks, individual statements are not necessarily executed simultaneously by different tasks; whether this occurs depends on the state and resources of the system. Hence, at any given time, it may be necessary for the system to select its next action from a number of different tasks. Each task has a priority value associated with it, which governs this selection process. The programmer can control the priority of the task, within limits, if he wishes to do so; otherwise, the priority value is set automatically.

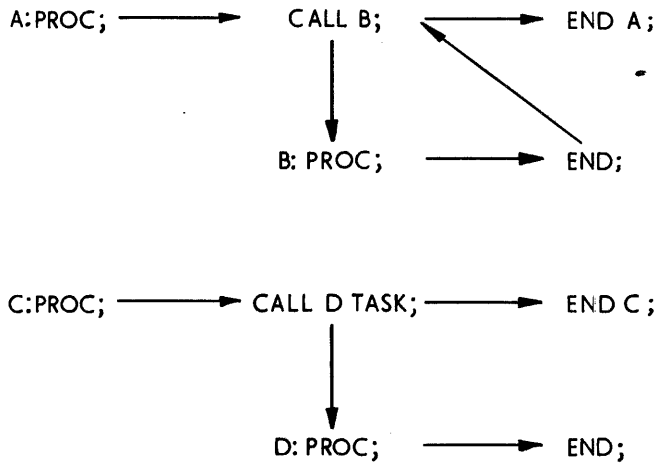


Figure 15-1. Synchronous and Asynchronous Operation

It may be that one task is to run independently of other concurrent tasks for some time, but then become dependent on some other task (for example, one task may require the result of another task before it can be completed). To allow for this, provision has been made for one task to await the completion of an operation at any stage of another task before carrying on. This process is known as task synchronization. Information about the state of an operation can be held by an event variable, to which an event name refers. By specifying an event name in a WAIT statement, the programmer can cause the task to wait for completion of the associated operation before proceeding.

The programmer can apply the EVENT option to tasks and certain input/output operations, in which case the value of the event variable is set automatically as a result of the operation concerned; or he can set the value explicitly.

The EVENT option allows an input/output operation to proceed asynchronously with the task that initiated it; at any time subsequent to the initiation of the input/output operation, the task can await its completion. For example, a task can display a message to the operator and, instead of waiting for a reply, can immediately proceed, pausing later to deal with the reply.

In general, the rules associated with the synchronous invocation of procedures apply equally to the asynchronous attachment of tasks. For example, on-units established prior to attachment of a subtask are inherited by the subtask, just as if the initial block of the subtask had been synchronously invoked. However, asynchronous operation introduces some extra considerations, such as the fact that a number of concurrent tasks can independently refer to one variable. This necessitates some extra rules, which are described in this chapter.

Multitasking also requires some extra rules and provisions for input/output. For example, without special provision, there would be nothing to prevent one task from operating on a record in a DIRECT UPDATE file while another task was operating on the same record; to cope with this, the EXCLUSIVE file attribute is provided.

Tasks can be terminated in a number of different ways. Normal termination occurs when control for the task reaches a RETURN or END statement. The EXIT statement specifies abnormal termination of the task and its subtasks, while the STOP statement specifies abnormal termination of the major task (even if STOP is executed in a subtask). When a task is terminated, any of its subtasks that are still active are abnormally terminated.

Multitasking may allow the central processing unit and input/output channels to be used more efficiently, by reducing the amount of waiting time. It does not necessarily follow that an asynchronous program will be more efficient than an equivalent synchronous program (although it may be easier to write). It depends on the amount of overlap possible between operations with varying amounts of input/output; if the overlap is slight, multitasking could be the less efficient method, because of the increased system overheads.

CREATION OF TASKS

The programmer specifies the creation of an individual task by using one or more of the multitasking options with a CALL statement. Once a procedure has been activated by execution of such a CALL statement, all blocks synchroncusly activated as a result of its execution become part of the created task, and all tasks attached as a result of its execution become subtasks of the created task. The created task itself is a subtask of the task executing the CALL statement. All programmer-created tasks are subtasks of the major task.

THE CALL STATEMENT

The CALL statement for asynchronous operation has the same form as that for synchronous operation, except for the addition of one (or any combination) of the multitasking options, TASK, EVENT, or PRIORITY. These options, in addition to their individual meanings (listed below), all specify that the invoked procedure is to be executed concurrently with the invoking procedure.

The CALL statement for asynchronous operation can specify arguments to be passed to the invoked procedure, just as it could if the operation were to be synchronous.

The TASK Option

The TASK option has the following format:

```
TASK [(element-task-name)]
```

The task name can be subscripted and/or qualified. Without the task name, the option merely specifies asynchronous operation. If the task is to have a name, the option must appear complete with the task name, which is thus contextually declared to have the TASK attribute, unless an explicit declaration exists. This is the only way in which a task can acquire a name. (Explicit declaration of a task variable does not associate the task name with any task.) The name can be used to control the priority of the task at some other point, by means of the PRIORITY pseudo-variable and built-in function. The task name has no other use to the PL/I programmer.

The EVENT Option

The EVENT option has the following format:

```
EVENT (element-event-name)
```

The event name can be subscripted and/or qualified. When this option is used, the event name is contextually declared to have the EVENT attribute (unless an explicit declaration exists) and is associated with the completion of the task created by the CALL statement. Another task can then be made to wait for completion of this task by specifying the event name in a WAIT statement of the other task.

An event variable has two separate values: a completion value that indicates whether or not the event is complete, and a status value that indicates whether the event has been abnormally completed. The completion value is a single bit, and the status value is a fixed binary number of default precision ((15,0) for the F Compiler). When the CALL statement is executed, the completion value of the event variable is set to '0'B (for "incomplete") and the status value to zero (for "not abnormally completed"). On termination of the created task, the completion value is set to '1'B, and, in the case of abnormal termination, the status value is set to 1 (if it is still zero).

The EVENT option can also be specified on the READ, WRITE, REWRITE, and DELETE statements, and on the DISPLAY statement with the REPLY option (see Chapter 8, "Input and Output"). In these cases, it allows other processing to continue while the input/output operation is being executed.

The PRIORITY Option

When a number of tasks simultaneously require attention, a choice has to be made by the system. Under the operating system, this choice is based on the relative importance of the various tasks: a task that has a higher priority value than the others will receive attention first. Note that tasks other than those executing the user's program may require attention from the system, and may have a higher priority than any of the user's tasks.

The PRIORITY option has the following format:

```
PRIORITY (expression)
```

If this option appears in the CALL statement, the expression is evaluated to a binary integer m, of precision (n,0), where n is implementation-defined (15 for the F Compiler). The priority of the created task is then made m relative to the task executing the CALL statement. With the F Compiler the lowest absolute priority possible is 0; the highest absolute priority possible is 234. (See "Priority of Tasks," in this chapter.)

If the option does not appear, the priority of the attached task is equated to that of the task variable named in the TASK option, if any, or else equated to the priority of the attaching task.

Examples

```
1. CALL PROCA TASK(T1);

2. CALL PROCA TASK(T2) EVENT(ET2);

3. CALL PROCA TASK(T3) EVENT(ET3)
   PRIORITY(-2);

4. CALL PROCA PRIORITY(1);
```

The CALL statements in the above examples create four different tasks that execute one procedure, PROCA. In example 3, the subtask T3 has a lower priority than the attaching task, while in example 4, the unnamed subtask has a higher priority than the attaching task.

Priority of Tasks

A priority specified in a PL/I source program is a relative value; the actual value depends on factors outside the source program.

Under the IBM System/360 Operating System, the priority associated with each job step is provided by the programmer, using the PRTY parameter in the JOB statement. This priority can have any number from 0 through 14: the higher the number, the higher the priority. The priority of the major task of the PL/I program when it is first entered is given by

```
Priority=(16*(job step priority))+10
```

This is the maximum priority for the program; that is, the highest priority that any task of the PL/I program can have. If an attempt is made to create a subtask with a higher priority than the maximum priority, the subtask will be executed at the maximum priority. Priority can be reduced to zero, but not below (a priority of less than zero will be treated as zero

priority). A task can change only its own priority or that of its immediate subtasks.

These conventions must be interpreted carefully when the PRIORITY built-in function or pseudo-variable is used. The effect of the statement

```
PRIORITY(T)=N;
```

is to set the priority of the task T equal to the priority of the current task plus the integral value of the expression N. If the priority thus calculated would be higher than the maximum priority or less than zero, the implementation ensures that the priority is set to the maximum, or zero, respectively.

The PRIORITY built-in function returns the relative priority of the named task (that is, the difference between the actual priority of the named task and the actual priority of the current task). Consider a task, T1, that attaches a subtask, T2, that itself attaches a subtask, T3. If task T2 executes the sequence of statements

```
PRIORITY(T3)=3;
X=PRIORITY(T3);
```

X will not necessarily have the value 3. If, for example, task T2 had an actual priority of 24, and the maximum priority were 26, then execution of the first statement would result in task T3 having a priority of 26, not 27. Relative to task T2, task T3 would have a priority of 2; hence, after execution of the second statement, X would have a value of 2.

Between execution of the two statements, control could pass to task T1, which could change the priority of task T2, in which case the value of X would depend on the new priority. For example, given the same original priorities as before, task T3 would have a priority of 26 after execution of the first statement. If the priority of task T2 were now changed to 20 by its attaching task, T1, execution of the second statement would result in X having a value of 6.

COORDINATION AND SYNCHRONIZATION OF TASKS

The rules for scope of names apply to blocks in the same way whether or not they are invoked as, or by, subtasks; thus, data and files can be shared between asynchronously executing tasks. Hence, a high degree of cooperation is possible between tasks, but this necessitates some coordination. Certain additional rules are introduced to deal with sharing of data and files between tasks, and the WAIT statement is provided to allow task synchronization.

SHARING DATA BETWEEN TASKS

It is the programmer's reponsibility to ensure that two references to the same variable cannot be in effect at one instant. He can do so by including an appropriate WAIT statement at a suitable point in his source program to force temporary synchronization of the tasks involved. Subject to this qualification, and the normal rules of scope, the following additional rules apply:

1. Static variables can be referred to in any task in which they are known.

2. Regardless of task boundaries, an automatic variable can be referred to in any block in which it is known, to which it is passed as an argument, or in which it is referred to using a valid locator variable.

3. Controlled variables can be referred to in any task in which they are known. However, not all allocations are known in each task. When a task is initiated, only the latest allocation, if any, of each controlled variable is known to the attached task. Both tasks may refer to this allocation. Subsequent allocations in the attached task are known only within the attached task; subsequent allocations within the attaching task are known only within the attaching task. A task can free only its own allocations; an attempt to free allocations made by another task will have no effect. No allocations of the controlled variable need exist at the time of attaching. It is not permissible for a task to free a controlled allocation shared with a subtask if the subtask will later refer to the allocation. When a task is terminated, all allocations of controlled storage made within that task are freed.

4. Based variables allocated within an area are freed when the area is freed; unless contained in an area allocated by another task, all based variable allocations (including areas) are freed on termination of the task that allocated them.

5. Any allocation of any variable of any storage class can be referred to in any task by means of an appropriate based variable reference. The pro-

grammer must ensure that the required variable has been allocated at the time of reference.

SHARING FILES BETWEEN TASKS

A file is shared between a task and its subtask if the file is open at the time the subtask is attached. The rules concerning such shared files are given below, first as applied to the subtask, and then as applied to the attaching task.

1. If a subtask shares a file with its attaching task, the subtask must not close the file. A subtask must not access a shared file after its attaching task has closed the file, even if the attaching task reopens the file beforehand.

2. If a task shares a file with one of its subtasks, it may close the shared file, provided that the subtask will make no subsequent attempt to access the file.

If a file name is known to a task and its subtask, and the associated file was not open when the subtask was attached, then the file is not shared; the effect is as if the task and its subtask were separate tasks to which the file name were known. That is, each task may separately open, access, and close the file. This type of operation is guaranteed only for files that are DIRECT in both tasks. Note that if one task opens a file, no other task can provide the corresponding close operation.

It is possible for two or more tasks to operate simultaneously on the same record in a DIRECT UPDATE file; this can be avoided by use of the EXCLUSIVE file attribute.

The EXCLUSIVE Attribute

When access to a record is restricted to one task, the record is said to be locked by that task. The EXCLUSIVE attribute, which can be specified for DIRECT UPDATE files only, provides a temporary locking mechanism to prevent one task from interfering with an operation by another task. Table 15-1 shows the effects of various operations on an EXCLUSIVE file.

Table 15-1. Effect of Operations on EXCLUSIVE Files

Attempted Operation	Current State of Addressed Record		
	Unlocked	Locked by this task	Locked by another task
READ NOLOCK	Proceed	Proceed	Wait for unlock
READ	1. Lock record 2. Proceed	Proceed	Wait for unlock
DELETE/REWRITE	1. Lock record 2. Proceed 3. Unlock[1] record	1. Proceed 2. Unlock[1] record	Wait for unlock
UNLOCK	No effect	Unlock record	No effect
CLOSE FILE	Unlock all locked records, and proceed with closing operation		
Terminate Task	Unlock all records locked by task. Close file, if opened in this task		

[1]The unlocking occurs at the end of the operation, on normal return from any on-units entered because of the operation (that is, at the corresponding WAIT statement when the EVENT option has been specified). If an abnormal return is made from such an on-unit, it is the programmer's responsibility to ensure that the record is unlocked. If the EVENT option has been specified with a READ statement, the operation is not completed until the corresponding WAIT statement is reached; in the meantime, no attempt to delete or rewrite the record should be made.

THE WAIT STATEMENT

The WAIT statement has the following format:

```
WAIT (event-name [,event-name]...)
     [(element-expression)];
```

Full details of the WAIT statement are given in Part II, Section J, "Statements"; the following is a shorter description, providing background to the present discussion.

The WAIT statement specifies that the task executing it will go into a waiting state (that is, execution of the WAIT statement will be extended) until such time as some or all of the named events have been completed. An event is complete when its completion value is '1'B. Note that the WAIT statement must specify event names, not task names.

The number of events to be awaited is given by the integral value of the expression, if present; otherwise all the named events have to be complete before the task can continue.

An event variable named in the list may be associated with an input/output operation that has been initiated by the task executing the WAIT statement. In this case, execution of the WAIT statement has the following effect:

1. If transmission ends (or has ended) normally, the event variable is set complete.

2. If the transmission ends (or has ended) requiring input/output conditions to be raised, the event variable is set abnormal (i.e., its status value is set to 1) and all the required conditions are raised. The event variable is set complete on return from the last on-unit. The order in which conditions are raised does not depend on the order in which the event names appear in the list.

If an abnormal return is made from an on-unit entered from the WAIT operation, the associated event variable is set complete, the WAIT operation is terminated, and control for the task passes to the point specified by the abnormal return.

Example

```
P1: PROCEDURE;
      .
      .
      .
    CALL P2 EVENT(EP2);
    CALL P3 EVENT(EP3);
    WAIT (EP2, EP3) (1);
      .
      .
      .
    END P1;
```

In this example, the task executing P1 will proceed until it reaches the WAIT statement; it will then await the completion of either the task executing P2, or that executing P3, before continuing.

TESTING AND SETTING EVENT VARIABLES

The two values, completion and status, of an event variable can be retrieved by the built-in functions COMPLETION and STATUS.

The COMPLETION function returns the current completion value of the event variable named in the argument. This value is '0'B if the event is incomplete, or '1'B if the event is complete.

The STATUS function returns the current status value of the event variable named in the argument. This value is nonzero if the event variable has been set abnormal, or 0 if it is normal.

These two built-in functions can also be used as pseudo-variables; thus, either of the two values of an event variable can be set independently. Alternatively, it is possible to assign the composite value of one event variable to another by specifying the event variables in an assignment statement. Thus, the setting of an event variable can be controlled by the programmer. By this means, he can mark the stages of a task; and, by using a WAIT statement in one task and an event assignment (from the COMPLETION built-in function or another event variable) in another task, he can synchronize any stage of one task with any stage of another.

The programmer should not attempt to assign a completion value to an event variable currently associated with an entire task or with an input/output event. An input/output event is never complete until the associated WAIT statement is executed.

Other ways in which an event variable can be set have already been discussed (such as specifying the event name in the EVENT option of a CALL statement). Full details of event variables will be found under "The EVENT Attribute" in Part II, Section I, "Attributes." See also "The EVENT Option," under "Record-Oriented Transmission," in Chapter 8, "Input and Output."

Note

When tasks are being synchronized, the following points should be kept in mind:

1. Under the operating system, an event must not be waited for by two or more different tasks.

2. With the F Compiler, an input/output event can be awaited only by the task that initiated it.

3. The following example shows one way in which two tasks, T1 and T2, could enter an infinite waiting state:

```
Task T1                    Task T2 (Event E2)

.                          COMPLETION(EV)='0'B;
.                          .
.                          .
WAIT (E2);                 .
.                          WAIT (EV);
.                          .
.                          .
COMPLETION(EV)='1'B;       .
.                          RETURN;
```

THE DELAY STATEMENT

The DELAY statement (see Part II, Section J, "Statements") allows a task to wait for a specified period, without reference to an event variable.

TERMINATION OF TASKS

A task is terminated by the occurrence of one of the following:

1. Control for the task reaches a RETURN or END statement for the initial procedure of the task.

2. Control for the task reaches an EXIT statement.

3. Control for the task, or for any other task, reaches a STOP statement.

4. The block in which the task was attached is terminated (either normally or abnormally).

5. The attaching task itself is terminated.

Termination is normal only if item (1) of the above list applies. In all other cases, termination is abnormal.

To avoid unintentional abnormal termination of a subtask, an attaching task should always wait for completion of the subtask before the task itself is allowed to be terminated.

When a task is terminated, the following actions are performed:

1. All input/output events that have been initiated in the task and are not yet complete are set complete, and their status values are set to 1; the results of the input/output operations are not defined.

2. All files that have been opened during the task and have not yet been closed are closed; all input/output conditions are disabled while this action is taking place.

3. All allocations of controlled variables made by the task are freed.

4. All allocations of based variables made by the task are freed, except those it has allocated within an area allocated by another task (these are freed when the area is freed).

5. All active blocks (including all active subtasks) in the task are terminated.

6. If the EVENT option was specified when the task was attached, the completion value of the associated event variable is set to '1'B. If the status value is still zero, and termination is abnormal, the status value is set to 1. Note, however, that termination of a subtask that has active subtasks has no effect on the completion values of event variables associated with these active subtasks.

7. All records locked by the task are unlocked.

Note: If a task is terminated while it is assigning a value to a variable, the value of the variable is undefined after termination. Similarly, if a task is terminated while it is creating or updating an OUTPUT or UPDATE file, the effect on the associated data set is undefined after termination.

It is the responsibility of the programmer to ensure that assignment and transmission are properly completed before termination of the task performing these operations.

PROGRAMMING EXAMPLE

This example shows an application of multitasking to a banking system. The program is divided into a batch section and a real-time section. Each section constitutes a subtask of the major task; each subtask has other subtasks attached to it that perform the various data processing routines necessary in each section. The use of several subtasks increases the program efficiency by permitting overlap between the input/output operations and the operations performed by the central processing unit.

The batch section of the program processes batches of cards that contain account information (such as cheques cashed, deposits made, or loan account details) and, after a certain number of transactions, produces a statement.

The real-time section of the program provides a means of communication between itself and the operator, using the DISPLAY statement with the REPLY option. This facility permits the user to issue commands to the program through the operator's console. These commands can:

1. Cause management or credit information, bank statements, or similar information to be made immediately available.

2. Initiate or terminate processing. Thus the user can initiate the processing of card batches, terminate a section of processing, terminate the entire program, or reply to a call for clarification of mispunched data.

The functions of the various tasks that make up the program, and their relationship to each other, are shown in Figure 15-2. Suggested coding for the ONLINE and PROCESS procedures is given below. These procedures are internal to the BANKER procedure, as are all the procedures in the program in this case. If they had been external procedures, the PROCEDURE statements would have needed the OPTIONS (TASK) option.

```
ONLINE:  PROCEDURE;
         DECLARE COMMAND CHARACTER(30) VARYING,
                 COMTYPE(8) CHARACTER(30) VARYING,
                 COUNT(8) FIXED BINARY INITIAL ((8)0),
                 ID CHARACTER (72) VARYING,
                 XL(8) LABEL,
                 ENDBEVT EVENT EXTERNAL;
         COMTYPE(1) = 'CREDIT';
         COMTYPE(2) = 'STATEMENT';
         COMTYPE(3) = 'INFORMATION';
         COMTYPE(4) = 'CALL BATCH';
         COMTYPE(5) = 'END BATCH';
         .
         .
         .
         COMTYPE(8) = 'END PROGRAM';

START:   DISPLAY ('NEXT COMMAND') REPLY (COMMAND);
         /*TASK IS IN WAITING STATE UNTIL REPLY IS RECEIVED*/
X:       DO I = 1 TO 8;
            IF COMMAND = COMTYPE (I)
              THEN GO TO XL(I);
         END;
         DISPLAY ('UNRECOGNIZABLE COMMAND, REPEAT')
            REPLY (COMMAND);
         GO TO X;

XL(1):   DISPLAY ('ACCOUNT ID') REPLY (ID);
         COUNT(1) = COUNT(1) + 1;
         CALL CREDIT (ID) PRIORITY (-1); /*ATTACH CREDIT TASK*/
         GO TO START;

XL(2):   .
         .
         .

XL(5):   COMPLETION (ENDBEVT) = '1'B;
         /*SETS EVENT COMPLETE IN BATCH. BATCH
         WILL TERMINATE WHEN ALL CARDS READ IN*/
         GO TO START;
         .
         .
         .
         END ONLINE;
```

```
PROCESS: PROCEDURE;
         DECLARE ANS CHARACTER (30) VARYING,
                 (READEVT, ENDEVT, TEVREAD,
                 TEVUPDT, TEVRED) EVENT EXTERNAL;

  WS:    WAIT (READEVT, ENDBEVT) (1);
         IF COMPLETION(READEVT)='1'B THEN GO TO READIN;
         WAIT (TEVREAD, TEVUPDT, TEVRED) (3);

  EXS:   EXIT;
         /*IF 'END BATCH' COMMAND WAIT FOR ASSOCIATED
         TASKS BEFORE BATCH IS TERMINATED*/

READIN:  COMPLETION (READEVT) = '0'B;
         CALL READER TASK (PR1) PRIORITY (-1) EVENT (TEVREAD);
         CALL UPDATE TASK (PR2) PRIORITY (-2) EVENT (TEVUPDT);
         CALL RED TASK (PR4) PRIORITY (-3) EVENT (TEVRED);
         WAIT (TEVREAD, TEVUPDT, TEVRED) (3);
         DISPLAY ('CARDS PROCESSED') REPLY (ANS);
         IF ANS = 'WAIT' THEN GO TO WS; /*WAIT FOR COMMAND*/
         IF ANS = 'READ' THEN GO TO READIN; /*PROCESS NEXT BATCH*/

END PROCESS;
```

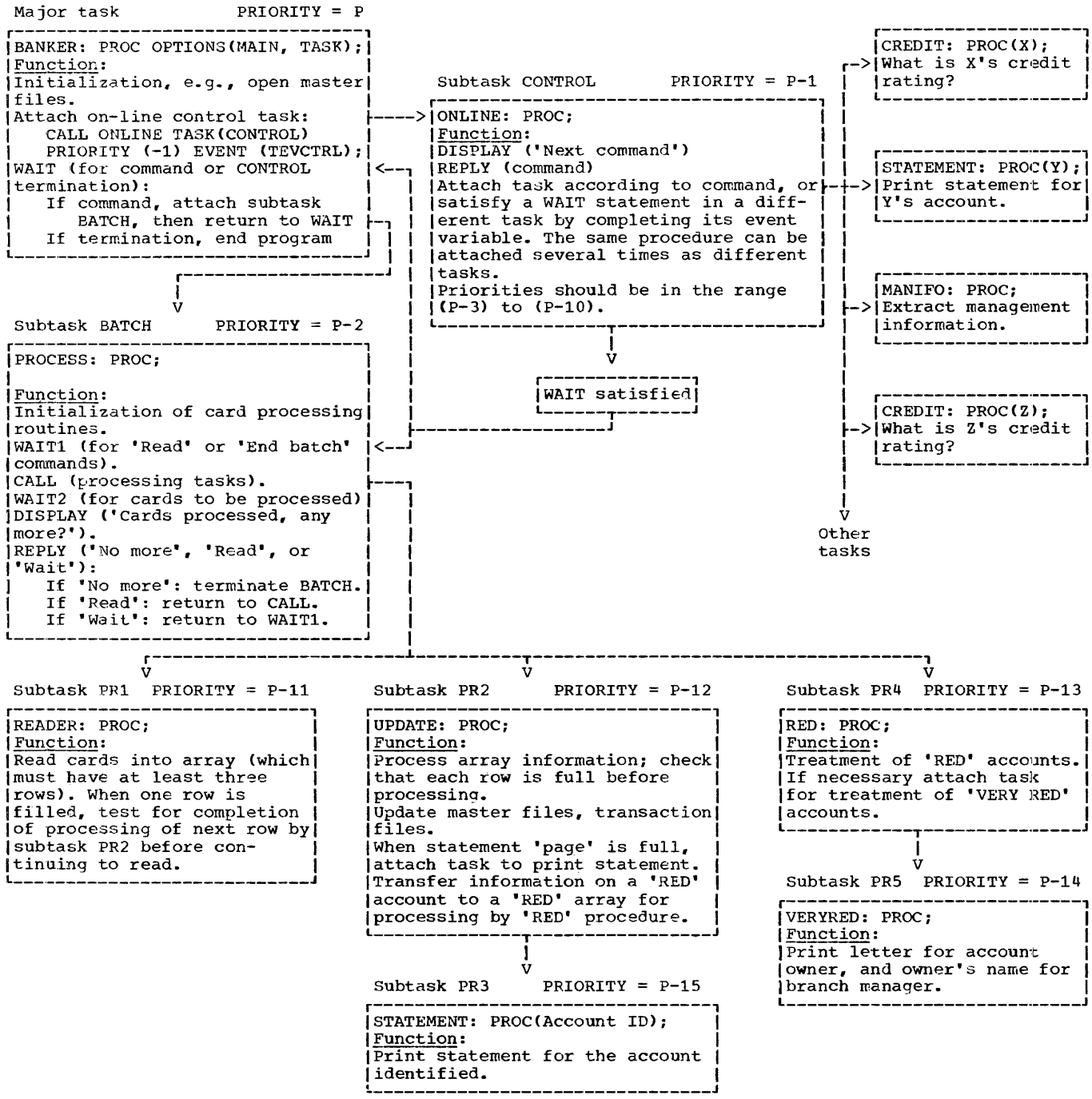


Figure 15-2. Flow Diagram for Programming Example of Multitasking

```
C71A3:  PROCEDURE OPTIONS (MAIN);                                       01
        DECLARE 1 CARDIN,                                               02
                  2 CODE CHARACTER (1),                                 03
                  2 ITEM CHARACTER (8),                                 04
                  2 QTY PICTURE 'ZZZ9',                                 05
                  2 REST CHARACTER (67);                                06
    /* INSERT DECLARATION FOR STOCK FILE AND STOCK RECORD */            07
        %INCLUDE F71SF,F71SR;                                           08
         DECLARE LOSS FIXED DECIMAL (10,2);                             09
         DECLARE PAGE_NO FIXED DECIMAL INITIAL (0);                     10
    /* SET UP HEADING AND PAGE CONTROL */                               11
         OPEN FILE (EXCP) PRINT PAGESIZE (50);                          12
         ON ENDPAGE (EXCP) BEGIN;                                       13
              PAGE_NO = PAGE_NO + 1;                                    14
              PUT FILE (EXCP) PAGE LINE(3)                              15
                EDIT ('PAGE',PAGE_NO,'EXCEPTIONAL ADJUSTMENTS')         16
                     (X(10),A(4),F(5),X(20),A(25));                     17
              PUT FILE (EXCP) LINE (6)                                  18
                EDIT ('ITEM NO','DESCRIPTION','VALUE','TYPE')           19
                     (COL(10),A(10),COL(25),A(12),COL(54),2 A(10));     20
              END;                                                      21
    /* PRINT HEADINGS FOR FIRST PAGE */                                 22
         SIGNAL ENDPAGE (EXCP);                                         23
    /* SET UP ERROR CONTROL */                                          24
         ON ERROR BEGIN;                                                25
    /* TEST FOR KEY OR CONVERSION ERROR OR SIGNALLED ERROR */           26
         IF ONCODE>49&ONCODE<58|ONCODE>599&ONCODE<630|ONCODE=9          27
         THEN DO;                                                       28
              WRITE FILE (ERRORS) FROM (CARDIN);                        29
              GO TO NEWCARD;                                            30
              END;                                                      31
    /* IF NOT KEY OR CONVERSION ERROR THEN DISPLAY INPUT AND END JOB*/  32
         DISPLAY(CODE||ITEM||QTY||ONCODE);                              33
         END;                                                           34
    /* SET UP NORMAL TERMINATION CONTROL */                             35
         ON ENDFILE(INFILE) GO TO TERM;                                 36
    /* MAIN UPDATING LOOP */                                            37
NEWCARD:  READ INTO (CARDIN) FILE (INFILE);                             38
          READ FILE (STOCK) INTO (STREC) KEY (ITEM);                    39
          IF CODE = 'A' THEN                                            40
    /* ADJUSTMENTS TO STOCK LEVEL */                                    41
          DO;                                                           42
               LOSS = (SQTY-QTY)*SPRICE;                                43
               IF LOSS>1000 THEN CALL EXCPT (LOSS,'S');                 44
               SQTY = QTY;                                              45
               REWRITE FILE (STOCK) FROM (STREC) KEY (ITEM);            46
               GO TO NEWCARD;                                           47
          END;                                                          48
          IF CODE ¬='D' THEN SIGNAL ERROR; /* ILLEGAL CODE */           49
    /* DELETIONS */                                                     50
          LOSS = SQTY * SPRICE;                                         51
          IF LOSS>1000 THEN CALL EXCPT (LOSS,'O');                      52
          DELETE FILE (STOCK) KEY (ITEM);                               53
          GO TO NEWCARD;                                                54
EXCPT:    PROCEDURE (L,TYPE);                                           55
          DECLARE L FIXED DECIMAL (10,2),                               56
                  TYPE CHARACTER (1);                                   57
          PUT FILE(EXCP) SKIP                                           58
              EDIT (ITEM,SDESC,L,TYPE)                                  59
                   (COL(10),A(8),COL(25),A(25),F(12,2),X(3),A(1));      60
          END EXCPT;                                                    61
    /* NORMAL TERMINATION */                                            62
TERM:  PUT FILE(EXCP) SKIP(2) LIST('END OF JOB');                       63
END C71A3;                                                              64
```

This example illustrates the use of PL/I for some common operations. The program reads cards and tests a card code that specifies either adjustment to stock levels or deletions of stock items from the file. In either case, if the transaction involves a decrease in stock value of more than $1,000, a subroutine is called to print a line of an exception report. An ON statement establishes an on-unit to control the page headings. An ERROR on-unit is used to check for unmatched keys and conversion errors; these errors cause the input card to be copied into an error file, and a new input card to be read. Other errors cause the input card to be displayed to the operator and the job terminated.

The pattern of indention illustrates the free format allowed by PL/I. An F Compiler option allows a programmer to specify the delimiting of the margins for scanning source statements (default for System/360 implementations specifies that columns 2 through 72 are to be scanned). So long as the specified margins are used, statements may begin or end at any place. Statements may be continued from card to card without any continuation notation, as are DECLARE statements and PUT statements in this example. Even constants can be continued from card to card if the last character on the first card is in the rightmost specified column and the first character on the next card is in the leftmost specified column. The card sequence numbers, as shown in the example, can be punched in columns outside the specified area, for example, with the default option, in columns 73 through 80.

The PROCEDURE statement in card 1 names the procedure; the MAIN option is an implementation-defined option which defines the entry point for the program (which may consist of more than one external procedure) when it is executed.

The DECLARE statement in cards 2 through 6 describes the input transaction card. The input transaction card has a one-character code punched in column 1, an item number punched in columns 2 through 9, and a four-digit quantity right adjusted in columns 10 through 13. The remainder of the card is declared, but not used.

Card 7 inserts a comment into the program. Comments may appear freely throughout a program. They do not affect execution; they merely provide documentation. As can be noted here, and in other comments inserted (cards 11, 22, 24, etc.), comments are delimited by the character pairs /* and */.

The %INCLUDE statement in card 8 is a preprocessor statement. This is indicated by the percent sign that is the first character of the statement keyword. This statement causes the compile-time preprocessor to obtain, from an installation standard library, declarations and other program text, including the DECLARE statements that declare the file STOCK and the structure STREC, which describes records in the file. These declarations are identified by the names F71SF and F71SR. Assume that the file declaration specifies a direct-access file whose records can be located by use of keys and that the structure declaration includes the following fields:

```
SQTY   FIXED DECIMAL (5)
SDESC  CHARACTER (25)
SPRICE FIXED DECIMAL (7,2)
```

The declaration in card 9 specifies that LOSS is a fixed-point decimal number (with a total of 10 digits and a sign) which will be treated as having 2 fractional digits. In System/360 implementations, this field will be stored in packed decimal.

PAGE_NO is given the attributes FIXED and DECIMAL. Note that since no precision is specified, PAGE_NO is given default precision, which is 5 digits. The INITIAL attribute sets the value of PAGE_NO to zero each time the procedure is entered.

The first statement executed is the OPEN statement in card 12. This statement adds the attribute PRINT to any other attributes that may have been specified in a DD statement with the ddname EXCP. When the file is opened, the page size is fixed at 50; that is, an interrupt will occur if an attempt is made to space or write past line 50. This OPEN statement causes contextual declaration of the file EXCP. Note that the READ statement in card 38 causes implicit opening, as well as contextual declaration, of the file INFILE.

The ON ENDPAGE statement in card 13 associates the on-unit, the begin block in cards 13 through 21, with any ENDPAGE interrupt for file EXCP. The effect of executing the ON statement is to associate an action with an interrupt, not to cause the action to be performed. In this case, the next statement executed is the SIGNAL statement in card 23, which simulates an interrupt and causes the begin block to be executed.

The begin block starting in card 13 increments PAGE_NO so that the first page is numbered 1. The first PUT statement in the block (cards 15, 16, 17) starts a new page and then sets the current line to 3. The page heading is printed on the third line. The action of the data list and format list is as follows:

Space ten columns; assign the constant 'PAGE' to a four-character field; convert the value of PAGE_NO to an integer and place it, right-adjusted, in a five-character field; space 20 columns along the line; assign the constant 'EXCEPTIONAL ADJUSTMENTS' to a 25-character field which starts in column 40.

The second PUT statement (cards 18, 19, 20) prints the column headings. In this statement, the LINE option specifies that printing is to begin on the sixth line; then the COL format item is used to position the fields, so that the first heading starts in column 10, the second heading in column 25, the third heading in column 54. The fourth heading follows immediately after the third, but since the third field is specified as 10 characters in width, there will be five spaces between the E of VALUE and the T of TYPE.

The END statement then causes return of control to the point following the interrupt, which in the first execution of the on-unit is the ON statement in card 25.

The ON ERROR statement specifies the action to be performed for any execution-time error detected by the PL/I error-handling mechanism. These errors are not necessarily limited to those for which ON conditions are specified in the language; for example, there could be ERROR interrupts for errors in mathematical routines such as SQRT and EXP.

When a detectable error occurs, whatever its cause, the BEGIN block will be executed as the on-unit. The first statement tests the value of ONCODE to see if it belongs to a particular class of errors. The ONCODE function, which can be used only in an on-unit, permits the programmer to determine the cause of the interrupt. Since ON-codes are implementation-defined, it is possible for an implementation to allow the programmer to make a finer distinction between causes of an error than the language alone can give. It also allows the programmer to determine errors not defined by ON conditions. In the case of a KEY interrupt, for example, eight different causes can be identified. There are 30 different codes for CONVERSION interrupts. The IF statement tests for KEY errors by checking the ONCODE to see if it lies in the range 50 to 57. Similarly, the range 600 to 629 is tested for conversion errors, and the ONCODE value 9 is tested to determine if the interrupt was caused by a SIGNAL statement.

If the ONCODE indicates that the error is due to a key or conversion error, or if it was signalled, the DO group in cards 28 through 31 is executed, and control is transferred out of the on-unit to the statement labeled NEWCARD.

If the error is not one of those listed above, a message is displayed to the operator. Although only a single expression is allowed in a DISPLAY statement, it is possible to display the contents of several fields by concatenating them into a single character string.

Unless control is specifically transferred out of the ERROR on-unit when it is completed (in this case, when the END statement is executed) and the ERROR condition is raised in a major task, the FINISH condition is raised and, subsequently, the major task is terminated. (Standard system action for the FINISH condition in the absence of an on-unit is for no action to be taken; that is, execution of the interrupted statement is resumed.) If the ERROR condition is raised in any other task, that task is terminated. Since no ON FINISH statement is included in this program, standard system action will be taken if the FINISH condition is raised.

The ON statement in card 36 establishes normal termination, that is, the action to be taken when the last item has been read from INFILE. It specifies that when the ENDFILE condition is raised for the file, a message is to be printed out (card 63). The END statement in card 64 then causes termination of execution.

The main loop of the program starts with the statement labeled NEWCARD in card 38. This READ statement specifies that a record is to be copied from the input file INFILE into the 80-character structure CARDIN.

The next READ statement (card 39) uses the value of ITEM (secured by the preceding READ operation) as a key to identify the record to be copied from the STOCK file into the structure STREC (whose declaration was incorporated into the program by the %INCLUDE statement). A key is always considered to be a character string whose length is determined by the system (from the KEYLEN subparameter of the DCB parameter in the DD statement defining that data set). In this example, ITEM is declared as a character string, so no conversion need be performed.

The IF statement (card 40) compares the correct value of CODE with the constant 'A'. If they are equal, the whole of the DO group in cards 42 to 48 is executed. This group computes the value of LOSS and, if it is greater than 1000, calls a subroutine EXCPT. This subroutine has two parameters, and the CALL statement must specify

two arguments with exactly the same attributes as the parameters. In this case, the second argument is a one-character constant that matches the parameter TYPE in the procedure. Note that an incorrectly written constant, for example, ' A' (with an additional blank), would be an error.

Whether or not LOSS is greater than 1000, the field SQTY is altered, the record is rewritten, and control is returned to NEWCARD to repeat the loop. Note that the REWRITE statement is used in rewriting the record; the WRITE statement would attempt to create a new record with the same key, and would cause an error. If CODE is not 'A', the next IF statement (card 49) compares the value of CODE with 'D' (the only other valid code). If the code is invalid (if CODE is not 'D'), the SIGNAL statement causes the ERROR on-unit to be executed. The system sets the ONCODE to 9, which is tested for in the ERROR on-unit to determine the type of error.

If the code is 'D', the remaining statements are executed. LOSS is computed, and if it is greater than 1000, the EXCPT subroutine is called. After return from the subroutine, or immediately if LOSS is not greater than 1000, the record is deleted and control is returned to NEWCARD.

The subroutine EXCPT prints a line of the exception report. This subroutine is nested in the main procedure, so it can access any of the variables declared in the outer procedure. In this case the only names declared in the inner procedure are the parameters. Parameter names are always treated as if they were declared within the procedure in which they are specified. If they are to have any attributes other than the usual default attributes, they must be declared explicitly.

The SKIP option on the PUT statement causes spacing to a new line before the EDIT list is printed. The EDIT list uses COL format items to line up the data fields under the headings described in the ENDPAGE on-unit.

PART II

Throughout this publication, 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, the punctuation that is required, and the 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 syntactically, using this notation, or are defined semantically.

   Examples:

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

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

   c. DO-statement. This denotes the occurrence of a DO statement. The upper-case letters are used to indicate a language keyword.

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

   Example:

   ```
   DECLARE identifier FIXED;
   ```

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

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

   a. A single notation variable or notation constant.

   b. Any collection of notation variables, notation constants, syntax-language symbols, and keywords surrounded by braces or brackets.

4. Braces {} are used to denote grouping of more than one element into a syntactic unit.

   Example:

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

   The vertical stacking of syntactic 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. Brackets can serve the additional purpose of delimiting a syntactic unit.

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

This denotes that the two occurrences of the variable "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.

60-CHARACTER SET

Character	Card-Punch	8-Bit Code
blank	no punches	0100 0000
.	12-8-3	0100 1011
<	12-8-4	0100 1100
(	12-8-5	0100 1101
+	12-8-6	0100 1110
|	12-8-7	0100 1111
&	12	0101 0000
$	11-8-3	0101 1011
*	11-8-4	0101 1100
)	11-8-5	0101 1101
;	11-8-6	0101 1110
¬	11-8-7	0101 1111
-	11	0110 0000
/	0-1	0110 0001
,	0-8-3	0110 1011
%	0-8-4	0110 1100
_	0-8-5	0110 1101
>	0-8-6	0110 1110
?	0-8-7	0110 1111
:	8-2	0111 1010
#	8-3	0111 1011
@	8-4	0111 1100
'	8-5	0111 1101
=	8-6	0111 1110
A	12-1	1100 0001
B	12-2	1100 0010
C	12-3	1100 0011
D	12-4	1100 0100
E	12-5	1100 0101
F	12-6	1100 0110
G	12-7	1100 0111
H	12-8	1100 1000
I	12-9	1100 1001
J	11-1	1101 0001
K	11-2	1101 0010
L	11-3	1101 0011
M	11-4	1101 0100
N	11-5	1101 0101
O	11-6	1101 0110
P	11-7	1101 0111
Q	11-8	1101 1000
R	11-9	1101 1001
S	0-2	1110 0010
T	0-3	1110 0011
U	0-4	1110 0100
V	0-5	1110 0101
W	0-6	1110 0110
X	0-7	1110 0111
Y	0-8	1110 1000
Z	0-9	1110 1001
0	0	1111 0000
1	1	1111 0001
2	2	1111 0010
3	3	1111 0011
4	4	1111 0100
5	5	1111 0101
6	6	1111 0110
7	7	1111 0111
8	8	1111 1000
9	9	1111 1001

Composite Symbols	Card-Punch
<=	12-8-4, 8-6
||	12-8-7, 12-8-7
**	11-8-4, 11-8-4
¬<	11-8-7, 12-8-4
¬>	11-8-7, 0-8-6
¬=	11-8-7, 8-6
>=	0-8-6, 8-6
/*	0-1, 11-8-4
*/	11-8-4, 0-1
->	11, 0-8-6

48-CHARACTER SET

Character	Card-Punch	8-Bit Code
blank	no punches	0100 0000
.	12-8-3	0100 1011
(	12-8-5	0100 1101
+	12-8-6	0100 1110
$	11-8-3	0101 1011
*	11-8-4	0101 1100
)	11-8-5	0101 1101
-	11	0110 0000
/	0-1	0110 0001
,	0-8-3	0110 1011
'	8-5	0111 1101
=	8-6	0111 1110
A	12-1	1100 0001
B	12-2	1100 0010
C	12-3	1100 0011
D	12-4	1100 0100
E	12-5	1100 0101
F	12-6	1100 0110
G	12-7	1100 0111
H	12-8	1100 1000
I	12-9	1100 1001
J	11-1	1101 0001
K	11-2	1101 0010
L	11-3	1101 0011
M	11-4	1101 0100
N	11-5	1101 0101
O	11-6	1101 0110
P	11-7	1101 0111
Q	11-8	1101 1000
R	11-9	1101 1001
S	0-2	1110 0010
T	0-3	1110 0011
U	0-4	1110 0100
V	0-5	1110 0101
W	0-6	1110 0110
X	0-7	1110 0111
Y	0-8	1110 1000
Z	0-9	1110 1001
0	0	1111 0000
1	1	1111 0001
2	2	1111 0010
3	3	1111 0011
4	4	1111 0100
5	5	1111 0101
6	6	1111 0110
7	7	1111 0111
8	8	1111 1000
9	9	1111 1001

Composite Symbols	Card Punch	60-Character Set Equivalent
..	12-8-3, 12-8-3	:
LE	11-3, 12-5	<=
CAT	12-3, 12-1, 0-3	||
**	11-8-4, 11-8-4	**
NL	11-5, 11-3	¬<
NG	11-5, 12-7	¬>
NE	11-5, 12-5	¬=
//	0-1, 0-1	%
,.	0-8-3, 12-8-3	;
AND	12-1, 11-5, 12-4	&
GE	12-7, 12-5	>=
GT	12-7, 0-3	>
LT	11-3, 0-3	<
NOT	11-5, 11-6, 0-3	¬
OR	11-6, 11-9	|
/*	0-1, 11-8-4	/*
*/	11-8-4, 0-1	*/
PT	11-7, 0-3	->

Note: When using the 48-character set, the following rules should be observed:

1. The two periods that replace the colon must be immediately preceded by a blank if the preceding character is a period.

2. 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.

3. 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.

Keyword	Abbreviation	Use of Keyword
ABNORMAL	ABNL	attribute
ABS(x)		built-in function
%ACTIVATE	%ACT	preprocessor statement
ADD(x,y,p[,q])		built-in function
ADDR(x)		built-in function
ALIGNED		attribute
ALL(x)		built-in function
ALLOCATE		statement
ALLOCATION(x)		built-in function
ANY(x)		built-in function
AREA		condition
AREA[(size)]		attribute
ATAN(x[,y])		built-in function
ATAND(x[,y])		built-in function
ATANH(x)		built-in function
AUTOMATIC	AUTO	attribute
BACKWARDS		attribute, option of OPEN statement
BASED(pointer-variable)		attribute
BEGIN		statement
BINARY	BIN	attribute
BINARY(x[,p[,q]])	BIN(x[,p[,q]])	built-in function
BIT(length)		attribute
BIT(expression[,size])		built-in function
BOOL(x,y,w)		built-in function
BUFFERED		attribute
BUFFERS(n)		option of ENVIRONMENT attribute
BUILTIN		attribute
BY		clause of DO statement
BY NAME		option of the assignment statement
CALL entry-name		statement or option of INITIAL attribute
CEIL(x)		built-in function
CHAR(expression[,size])		built-in function
CHARACTER(length)	CHAR(length)	attribute
CHECK (name-list)		condition
CLOSE		statement
COBOL		option of ENVIRONMENT attribute
COLUMN(w)	COL(w)	format item
COMPLETION(event-name)		built-in function, pseudo-variable
COMPLEX	CPLX	data attribute
COMPLEX(a,b)	CPLX(a,b)	built-in function, pseudo-variable
CONDITION(name)		condition
CONJG(x)		built-in function
CONSECUTIVE		option of ENVIRONMENT attribute
CONTROLLED	CTL	attribute
CONVERSION	CONV	condition
COPY		option of GET statement
COS(x)		built-in function
COSD(x)		built-in function
COSH(x)		built-in function
COUNT(file-name)		built-in function
CTLASA		option of ENVIRONMENT attribute
CTL360		option of ENVIRONMENT attribute
DATA		STREAM I/O transmission mode
DATAFIELD		built-in function
DATE		built-in function
%DEACTIVATE	%DEACT	preprocessor statement
DECIMAL	DEC	attribute
DECIMAL(x[,p[,q]])	DEC(x[,p[,q]])	built-in function
DECLARE	DCL	statement
%DECLARE	%DCL	preprocessor statement
DEFINED	DEF	attribute
DELAY(n)		statement
DELETE		statement
DIM(x,n)		built-in function

Keyword	Abbreviation	Use of Keyword
DIRECT		attribute
DISPLAY		statement
DIVIDE(x,y,p[,q])		built-in function
DO		statement
%DO		preprocessor statement
EDIT		STREAM I/O transmission mode
ELSE		clause of IF statement
%ELSE		clause of %IF statement
EMPTY		built-in function
END		statement
%END		preprocessor statement
ENDFILE(file-name)		condition
ENDPAGE(file-name)		condition
ENTRY		attribute or statement
ENVIRONMENT	ENV	attribute
ERF(x)		built-in function
ERFC(x)		built-in function
ERROR		condition
EVENT		option of CALL, READ, WRITE, REWRITE, and DELETE statements, attribute
EXCLUSIVE		attribute
EXIT		statement
EXP(x)		built-in function
EXTERNAL	EXT	attribute
F(block-size[,record-size])		option of ENVIRONMENT attribute
FILE		attribute
FILE(file-name)		option of GET and PUT statements, specification of RECORD I/O statements
FINISH		condition
FIXED		attribute
FIXED(x[,p[,q]])		built-in function
FIXEDOVERFLOW	FOFL	condition
FLOAT		attribute
FLOAT(x[,p])		built-in function
FLOOR(x)		built-in function
FORMAT(format-list)		statement
FREE		statement
FROM		option of WRITE or REWRITE statements
GENERIC		attribute
GENKEY		option of ENVIRONMENT attribute
GET		statement
GO TO	GOTO	statement
%GO TO	%GOTO	preprocessor statement
HBOUND(x,h)		built-in function
HIGH(i)		built-in function
IF		statement
%IF		preprocessor statement
IGNORE(n)		option of READ statement
IMAG(x)		built-in function, pseudo-variable
IN		option of ALLOCATE and FREE statements
%INCLUDE		preprocessor statement
INDEX(string,config)		built-in function
INDEXAREA [(index-area-size)]		option of ENVIRONMENT attribute
INDEXED		option of ENVIRONMENT attribute
INITIAL	INIT	attribute
INPUT		attribute, option of the OPEN statement
INTERNAL	INT	attribute
INTO(variable)		option of READ statement
IRREDUCIBLE	IRRED	attribute
KEY(file-name)		condition
KEY(x)		option of READ, DELETE, and REWRITE statements
KEYED		attribute, option of OPEN statement
KEYFROM(x)		option of WRITE statement
KEYTO(variable)		option of READ statement
LABEL		attribute

Keyword	Abbreviation	Use of Keyword
LENGTH(string)		built-in function
LBOUND(x,n)		built-in function
LEAVE		option of ENVIRONMENT attribute
LIKE		attribute
LINE(w)		format item, option of PUT statement
LINENO(file-name)		built-in function
LINESIZE		option of OPEN statement
LIST		STREAM I/O transmission mode
LOCATE		statement
LOG(x)		built-in function
LOG2(x)		built-in function
LOG10(x)		built-in function
LOW(i)		built-in function
MAIN		option of PROCEDURE statement
MAX(x_1,x_2...x_n)		built-in function
MIN(x_1,x_2...x_n)		built-in function
MOD(x_1,x_2)		built-in function
MULTIPLY(x_1,x_2,p[,q])		built-in function
NAME(file-name)		condition
NOCHECK		condition prefix identifier (disables CHECK)
NOCONVERSION	NOCONV	condition prefix identifier (disables CONVERSION)
NOFIXEDOVERFLOW	NOFOFL	condition prefix identifier (disables FIXEDOVERFLOW)
NOLOCK		option of READ statement
NOOVERFLOW	NOOFL	condition prefix identifier (disables OVERFLOW)
NOSIZE		condition prefix identifier (disables SIZE)
NOSTRINGRANGE	NOSTRG	condition prefix identifier (disables STRINGRANGE)
NOSUBSCRIPTRANGE	NOSUBRG	condition prefix identifier (disables SUBSCRIPTRANGE)
NOUNDERFLOW	NOUFL	condition prefix identifier (disables UNDERFLOW)
NOWRITE		option of ENVIRONMENT attribute
NOZERODIVIDE	NOZDIV	condition prefix identifier (disables ZERODIVIDE)
NORMAL		attribute
NULL		built-in function
NULLO		built-in function
OFFSET(area-name)		attribute
ON		statement
ONCHAR		built-in function, pseudo-variable
ONCOUNT		built-in function
ONCODE		built-in function
ONFILE		built-in function
ONKEY		built-in function
ONLOC		built-in function
ONSOURCE		built-in function, pseudo-variable
OPEN		statement
OPTIONS(list)		option of PROCEDURE statement
OUTPUT		attribute, option of the OPEN statement
OVERFLOW	OFL	condition
PACKED		attribute
PAGE		format item, option of PUT statement
PAGESIZE(w)		option of the OPEN statement
PICTURE	PIC	attribute
POINTER	PTR	attribute
POLY(a,x)		built-in function
POSITION(i)	POS(i)	attribute
PRECISION(x,p[,q])	PREC(x,p[,q])	built-in function
PRINT		attribute, option of OPEN statement
PRIORITY(x)		option of CALL statement
PRIORITY[(task-name)]		built-in function, pseudo-variable
PROCEDURE	PROC	statement

Keyword	Abbreviation	Use of Keyword
%PROCEDURE	%PROC	preprocessor statement
PROD(x)		built-in function
PUT		statement
READ		statement
REAL		attribute
REAL(x)		built-in function, pseudo-variable
RECORD		attribute, option of OPEN statement
RECURSIVE		option of PROCEDURE statement
REDUCIBLE	RED	attribute
REENTRANT		option of PROCEDURE statement
REFER		option of BASED attribute
REGIONAL(1|2|3)		option of ENVIRONMENT attribute
REPEAT(string,i)		built-in function
REPLY(c)		option of DISPLAY statement
RETURN		statement
RETURNS		attribute
REVERT		statement
REWIND		option of ENVIRONMENT attribute
REWRITE		statement
ROUND(x,n)		built-in function
SEQUENTIAL		attribute
SET(pointer-variable)		option of ALLOCATE, LOCATE, and READ statements
SETS		attribute
SIGN(x)		built-in function
SIGNAL		statement
SIN(x)		built-in function
SIND(x)		built-in function
SINH(x)		built-in function
SIZE		condition
SKIP[(x)]		format item, option of GET and PUT statements
SNAP		option of ON statement
SQRT(x)		built-in function
STATIC		attribute
STATUS(event-name)		built-in function, pseudo-variable
STOP		statement
STREAM		attribute, option of OPEN statement
STRING(x)		built-in function
STRINGRANGE	STRG	condition
STRING(string-name)		option of GET and PUT statements
iSUB		dummy variable of DEFINED attribute
SUBSCRIPTRANGE	SUBRG	condition
SUBSTR(string,i[,j])		built-in function, pseudo-variable
SUM(x)		built-in function
SYSIN		name of standard system input file
SYSPRINT		name of standard system output file
SYSTEM		option of the ON statement
TAN(x)		built-in function
TAND(x)		built-in function
TANH(x)		built-in function
TASK		attribute, option of PROCEDURE statement
TASK[(task-name)]		option of CALL statement
THEN		clause of IF statement
%THEN		cluase of %IF statement
TIME		built-in function
TO		clause of DO statement
TITLE(x)		option of OPEN statement
TRANSMIT		condition
TRUNC(x)		built-in function
U(max-block-size)		option of ENVIRONMENT attribute
UNALIGNED	UNAL	attribute
UNBUFFERED		attribute, option of OPEN statement
UNDEFINEDFILE(file-name)	UNDF(file-name)	condition
UNDERFLOW	UFL	condition
UNLOCK		statement

Keyword	Abbreviation	Use of Keyword
UNSPEC(x)		built-in function, pseudo-variable
UPDATE		attribute, option of OPEN statement
USES		attribute
V(max-block-size[,max-record-size])		option of ENVIRONMENT attribute
VARYING	VAR	attribute
VBS(max-block-size[,max-record-size])		option of ENVIRONMENT attribute
VS(max-block-size[,max-record-size])		option of ENVIRONMENT attribute
WAIT		statement
WHILE		clause of DO statement
WRITE		statement
ZERODIVIDE	ZDIV	condition

SECTION D: PICTURE SPECIFICATION CHARACTERS

Picture specification characters appear in either the PICTURE attribute or the P format item for edit-directed input and output. In either case, an individual character has the same meaning. A discussion of the concepts of picture specifications appears in Part I, Chapter 9, "Editing and String Handling."

Picture characters are used to describe the attributes of the associated data item, whether it is the value of a variable or a data item to be transmitted between the program and external storage.

A picture specification always describes a character representation that is either a character-string data item or a numeric character data item. A character-string pictured item is one that can consist of alphabetic characters, decimal digits, and other special characters. A numeric character pictured item is one in which the data itself can consist only of decimal digits, a decimal point and, optionally, a plus or minus sign. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified, but they are not a part of the arithmetic value of the numeric character variable, although the characters are stored with the digits and are considered to be part of the character-string value of the variable.

Arithmetic data assigned to a numeric character variable is converted to character representation. Editing, such as zero suppression and the insertion of other characters, can be specified for a numeric character data item. Editing cannot be specified for pictured character-string data.

Data assigned to a variable declared with a numeric picture specification (or data to be written with a numeric picture format item) must be either internal coded arithmetic data or data that can be converted to coded arithmetic. Thus, assigned data can contain only digits and, optionally, a decimal point and a sign. It should not contain any other character, even though that character (for example, a currency symbol) is specified in the picture specification and is to be inserted into the data as part of its character-string value; if it does, the CONVERSION condition is raised.

Numeric character data to be read using the P format item must conform to the specification contained in the P format item, including editing characters. If the indicated character does not appear in the input stream, the CONVERSION condition is raised.

Data assigned to a variable declared with a character-string picture specification (or data to be written with a character-string picture format item) should conform, character by character (or be convertible, character by character) to the picture specification; if it does not, the CONVERSION condition is raised.

Figures in this section illustrate how different picture specifications affect the representation of values when assigned to a pictured variable or when printed using the P format item. Each figure shows the original value of the data, the attributes of the variable from which it is assigned (or written), the picture specification, and the character-string value of the numeric character or pictured character-string variable.

PICTURE CHARACTERS FOR CHARACTER-STRING DATA

Only three picture characters can be used in character-string picture specifications:

X specifies that the associated position can contain any character whose internal bit configuration can be recognized by the computer in use.

A specifies that the associated position can contain any alphabetic character or a blank character.

9 specifies that the associated position can contain any decimal digit or a blank character.

No insertion characters can be specified. At least one A or X must appear.

Figure D-1 gives examples of character-string picture specifications. In the figure, the letter b indicates a blank character. Note that assignments are left-adjusted, and any necessary padding with blanks is on the right.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
CHARACTER(5)	'9B/2L'	XXXXX	9B/2L
CHARACTER(5)	'9B/2L'	XXX	9B/
CHARACTER(5)	'9B/2L'	XXXXXXX	9B/2Lbb
CHARACTER(5)	'ABCDE'	AAAAA	ABCDE
CHARACTER(5)	'ABCDE'	AAAAAA	ABCDEb
CHARACTER(5)	'ABCDE'	AAA	ABC
CHARACTER(5)	'12/34'	99X99	12/34
CHARACTER(5)	'L26.7'	A99X9	L26.7

[1]A variable declared with a character-string picture specification has a character-string value only.

Figure D-1. Pictured Character-String Examples

PICTURE CHARACTERS FOR NUMERIC CHARACTER DATA

Numeric character data must represent numeric values; therefore, the associated picture specification cannot contain the characters X or A. The picture characters for numeric character data can specify detailed editing of the data.

A numeric character variable can be considered to have two different kinds of value, depending upon its use. They are (1) its arithmetic value and (2) its character-string value.

The arithmetic value is the value expressed by the decimal digits of the data item, the assumed location of a decimal point, and possibly a sign. The arithmetic value of a numeric character variable is used whenever the variable appears in an expression that results in a coded arithmetic value or whenever the variable is assigned to a coded arithmetic, numeric character, or bit-string variable. In such cases, the arithmetic value of the numeric character variable is converted to internal coded arithmetic representation.

The character-string value is the value expressed by the decimal digits of the data item, as well as all of the editing and insertion characters appearing in the picture specification. The character-string value does not, however, include the assumed location of a decimal point, as specified by the picture character V. The character-string value of a numeric character variable is used whenever the variable appears in a character-string expression operation or in an assignment to a character-string variable, whenever the data is printed using list-directed or data-directed output, or whenever a reference is made to a character-string variable that is defined on the numeric character variable. In such cases, no data conversion is necessary.

The picture characters for numeric character specifications may be grouped into the following categories:

- Digit and Decimal-Point Specifiers
- Zero Suppression Characters
- Insertion Characters
- Signs and Currency Symbol
- Credit, Debit, and Overpunched Signs
- Exponent Specifiers
- Scaling Factor
- Sterling Pictures

The picture characters in these groups may be used in various combinations. Consequently, a numeric character specification can consist of two or more parts such as a sign specification, an integer subfield, a fractional subfield and, for floating-point, an exponent field. A sterling picture specification contains separate fields for pounds, shillings, and pence; the pence field can have an integer subfield and a fractional subfield.

A major requirement of the picture specification for numeric character data is that each field must contain at least one picture character that specifies a digit position. This picture character, however, need not be the digit character 9. Other picture characters, such as the zero suppression characters (Z or * or Y), also specify digit positions. At least one of these characters must be used to define a numeric character specification.

DIGIT AND DECIMAL-POINT SPECIFIERS

The picture characters 9 and V are used in the simplest form of numeric character specifications that represent fixed-point decimal values.

Figure D-2 gives examples of numeric character specifications.

9 specifies that the associated position in the data item is to contain a decimal digit.

V specifies that a decimal point is assumed at this position in the associated data item. However, it does not specify that an actual decimal point is to be inserted. The integer and fractional parts of the assigned value are aligned on the V character; therefore, an assigned value may be truncated or extended with zero digits at either end. (Note that if significant digits are truncated on the left, the result is undefined and a SIZE interrupt will occur, if SIZE is enabled.) If no V character appears in the picture specification of a fixed-point decimal value (or in the first field of a picture specification of a floating-point decimal value), a V is assumed at the right end of the field specification. This can cause the assigned value to be truncated, if necessary, to an integer. The V character cannot appear more than once in a picture specification. The V is considered to be a subfield delimiter in the picture specification; that is, the portion preceding the V and the portion following it (if any) are each a subfield of the specification.

ZERO SUPPRESSION CHARACTERS

The zero suppression picture characters specify conditional digit positions in the character-string value and may cause leading zeros to be replaced by asterisks or blanks and nonleading zeros to be replaced by blanks. Leading zeros are those that occur in the leftmost digit positions of

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(5)	12345	99999	12345
FIXED(5)	12345	99999V	12345
FIXED(5)	12345	999V99	34500[2]
FIXED(5)	12345	V99999	00000[2]
FIXED(7)	1234567	99999	34567[2]
FIXED(3)	123	99999	00123
FIXED(5,2)	123.45	999V99	12345
FIXED(7,2)	12345.67	9V9	56[2]
FIXED(5,2)	123.45	99999	00123

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.
[2]In this case, PL/I does not define the result since significant digits have been truncated on the left. The result shown, however, is that given for System/360 implementations.

Figure D-2. Pictured Numeric Character Examples

fixed-point numbers or in the leftmost digit positions of the two parts of floating-point numbers, that are to the left of the assumed position of a decimal point, and that are not preceded by any of the digits 1 through 9. The leftmost nonzero digit in a number and all digits, zeros or not, to the right of it represent significant digits. Note that a floating-point number can also have a leading zero in the exponent field.

Figure D-3 gives examples of the use of zero suppression characters. In the figure, the letter b indicates a blank character.

Z specifies a conditional digit position and causes a leading zero in the associated data position to be replaced by a blank character. When the associated data position does not contain a leading zero, the digit in the position is not replaced by a blank character. The picture character Z cannot appear in the same subfield as the picture character *, nor can it appear to the right of a drifting picture character or any of the picture characters 9, T, I, or R in a field.

* specifies a conditional digit position and is used the way the picture character Z is used, except that leading zeros are replaced by asterisks. The picture character * cannot appear with the picture character Z in the same subfield, nor can it appear to the right of a drifting picture character or any of the picture characters 9, T, I, or R in a field.

Y specifies a conditional digit position and causes a zero digit, leading or nonleading, in the associated position to be replaced by a blank character. When the associated position does not contain a zero digit, the digit in the position is not replaced by a blank character.

Note: If one of the picture characters Z or * appears to the right of the picture character V, then all fractional digit

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(5)	12345	ZZZ99	12345
FIXED(5)	00100	ZZZ99	bb100
FIXED(5)	00000	ZZZ99	bbb00
FIXED(5)	00100	ZZZZZ	bb100
FIXED(5)	00000	ZZZZZ	bbbbb
FIXED(5,2)	123.45	ZZZ99	bb123
FIXED(5,2)	001.23	ZZZV99	bb123
FIXED(5)	12345	ZZZV99	34500[2]
FIXED(5)	00000	ZZZVZZ	bbbbb
FIXED(5)	00100	*****	**100
FIXED(5)	00000	*****	*****
FIXED(5,2)	000.01	***V**	***01
FIXED(5)	00100	YYYYY	bb1bb
FIXED(5)	10203	9Y9Y9	1b2b3

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.
[2]In this case, PL/I does not define the result since significant digits have been truncated on the left. The result shown, however, is that given for System/360 implementations.

Figure D-3. Examples of Zero Suppression

positions in the specification, as well as all integer digit positions, must employ the Z or * picture character, respectively. When all digit positions to the right of the picture character V contain zero suppression picture characters, fractional zeros of the value are suppressed only if all positions in the fractional part contain zeros and all integer positions have been suppressed. The entire character-string value of the data item will then consist of blanks or asterisks. No digits in the fractional part are replaced by blanks or asterisks if the fractional part contains any significant digit.

INSERTION CHARACTERS

The picture characters comma (,), point (.), slash (/), and blank (B) are insertion characters; they cause the specified character to be inserted into the associated position of the numeric character data. They do not indicate digit positions, but are inserted between digits. Each does, however, actually represent a character position in the character-string value, whether or not the character is suppressed. The comma, point, and slash are conditional insertion characters; within a string of zero suppression characters, they, too, may be suppressed. The blank (B) is an unconditional insertion character; it always specifies that a blank is to appear in the associated position.

Note: Insertion characters are applicable only to the character-string value. They specify nothing about the arithmetic value of the data item.

Figure D-4 gives examples of the use of insertion characters. In the figure, the letter b indicates a blank character.

, causes a comma to be inserted into the associated position of the numeric character data when no zero suppression occurs. If zero suppression does occur, the comma is inserted only when an unsuppressed digit appears to the left of the comma position, or when a V appears immediately to the left of it and the fractional part contains any significant digits.[1] In all other cases where zero suppression occurs, one of three possible characters is inserted in place of the comma. The choice of character to replace the comma depends upon the first picture character that both precedes the comma position and specifies a digit position:

- If this character position is an asterisk, the comma position is assigned an asterisk.
- If this character position is a drifting sign or a drifting currency symbol (discussed later), the drifting string is assumed to include the comma position, which is assigned the drifting character.
- If this character position is not an asterisk or a drifting character, the comma position is assigned a blank character.

. is used the same way the comma picture character is used, except that a point (.) is assigned to the associated position. This character never causes point alignment in the picture specifications of a fixed-point decimal number and is not a part of the arithmetic value of the data item. That function is served solely by the picture character V. Unless the V actually appears, it is assumed to be to the right of the rightmost digit position in the field, and point alignment is handled accordingly, even if the point insertion character appears elsewhere. The point (or the comma or slash) can be used in conjunction with the V to cause insertion of the point (or comma or slash) in the position that delimits the end of the integer portion and the beginning of the fractional portion of a fixed-point (or floating-point) number, as might be desired in printing, since the V does not cause printing of a point. The point must immediately precede or immediately follow the V. If the point precedes the V, it will be inserted only if a significant digit appears to the left of the V, even if all fractional digits are significant. If the point immediately follows the V, it will be suppressed if all digits to the right of the V are suppressed, but it will appear if there are any fractional digits (along with any intervening zeros).

---

[1] In the special case of a conditional insertion character that is preceded either by nothing or only by characters that do not specify digit positions, the conditional position will always contain the conditional insertion character.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(4)	1234	9,999	1,234
FIXED(6,2)	1234.56	9,999V.99	1,234.56
FIXED(4,2)	12.34	ZZ.VZZ	12.34
FIXED(4,2)	00.03	ZZ.VZZ	bbb03
FIXED(4,2)	00.03	ZZV.ZZ	bb.03
FIXED(4,2)	12.34	ZZV.ZZ	12.34
FIXED(4,2)	00.00	ZZV.ZZ	bbbbb
FIXED(9,2)	1234567.89	9,999,999.V99	1,234,567.89
FIXED(7,2)	12345.67	**,999V.99	12,345.67
FIXED(7,2)	00123.45	**,999V.99	***123.45
FIXED(9,2)	1234567.89	9.999.999V,99	1.234.567,89
FIXED(6)	123456	99/99/99	12/34/56
FIXED(6)	123456	99.9/99.9	12.3/45.6
FIXED(6)	001234	ZZ/ZZ/ZZ	bbb12/34
FIXED(6)	000012	ZZ/ZZ/ZZ	bbbbbb12
FIXED(6)	000000	ZZ/ZZ/ZZ	bbbbbbbb
FIXED(6)	000000	**/**/**	********
FIXED(6)	123456	99B99B99	12b34b56
FIXED(3)	123	9BB9BB9	1bb2bb3
FIXED(2)	12	9BB/9BB	1bb/2bb

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D-4. Examples of Insertion Characters

/ is used the same way the comma picture character is used, except that a slash (/) is inserted in the associated position.

B specifies that a blank character always be inserted into the associated position of the character-string value of the numeric character data.

SIGNS AND CURRENCY SYMBOL

The picture characters S, +, and - specify signs in numeric character data. The picture character $ specifies a currency symbol in the character-string value of numeric character data.

These picture characters may be used in either a static or a drifting manner. A drifting character is similar to a zero suppression character in that it can cause zero suppression. However, the character specified by the drifting string is always inserted in the position specified by the end of the drifting string or in the position immediately to the left of the first significant digit.

The static use of these characters specifies that a sign, a currency symbol, or a blank always appears in the associated position. The drifting use specifies that

leading zeros are to be suppressed. In this case, the rightmost suppressed position associated with the picture character will contain a sign, a blank, or a currency symbol.

A drifting character is specified by multiple use of that character in a picture field. Thus, if a field contains one currency symbol ($), it is interpreted as static; if it contains more than one, it is interpreted as drifting. The drifting character must be specified in each digit position through which it may drift.

Drifting characters must appear in strings. A string is a sequence of the same drifting character, optionally containing a V and one of the insertion characters comma, point, slash, or B. Any of the insertion characters slash, comma, point, or B following the last drifting symbol of the string is considered part of the drifting string. However, a following V terminates the drifting string and is not part of it. A field of a picture specification can contain only one drifting string. A drifting string cannot be preceded by a digit position. The picture characters * and Z cannot appear to the right of a drifting string in a field.

Figure D-5 gives examples of the use of drifting picture characters. In the figure, the letter b indicates a blank character.

The position in the data associated with the characters slash, comma, point, and B appearing in a string of drifting characters will contain one of the following:

- slash, comma, point, or blank if a significant digit has appeared to the left

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(5,2)	123.45	$999V.99	$123.45
FIXED(5,2)	001.23	$ZZZV.99	$bb1.23
FIXED(5,2)	000.00	$ZZZV.ZZ	$bbbbbb
FIXED(1)	0	$$$.$$	bbbbb$
FIXED(5,2)	123.45	$$$9V.99	$123.45
FIXED(5,2)	001.23	$$$9V.99	bb$1.23
FIXED(5,2)	012.00	99$	12$
FIXED(2)	12	$$$,999	bbb$012
FIXED(4)	1234	$$$,999	b$1,234
FIXED(5,2)	123.45	S999V.99	+123.45
FIXED(5,2)	-123.45	S999V.99	-123.45
FIXED(5,2)	-123.45	+999V.99	b123.45
FIXED(5,2)	123.45	-999V.99	b123.45
FIXED(5,2)	123.45	999V.99S	123.45+
FIXED(5,2)	001.23	++B+9V.99	bbb+1.23
FIXED(5,2)	001.23	---9V.99	bbb1.23
FIXED(5,2)	-001.23	SSS9V.99	bb-1.23

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D-5. Examples of Drifting Picture Characters

- the drifting symbol, if the next position to the right contains the leftmost significant digit of the field

- blank, if the leftmost significant digit of the field is more than one position to the right

If a drifting string contains the drifting character n times, then the string is associated with n-1 conditional digit positions. The position associated with the leftmost drifting character can contain only the drifting character or blank, never a digit. If a drifting string is specified for a field, the other potentially drifting characters can appear only once in the field, i.e., the other character represents a static sign or currency symbol.

If a drifting string contains a V within it, the V delimits the preceding portion as a subfield, and all digit positions of the subfield following the V must also be part of the drifting string that commences the second subfield.

Only one type of sign character can appear in each field. An S, +, or - used as a static character can appear to the left of all digits in the mantissa and exponent fields of a floating-point specification, and either to the right or left of all digit positions of a fixed-point specification.

In the case in which all digit positions after the V contain drifting characters, suppression in the subfield will occur only if all of the integer and fractional digits are zero. The resulting edited data item will then be all blanks, except for the rightmost digit position, which will contain the drifting character. If there are any significant fractional digits, the entire fractional portion will appear unsuppressed.

$ specifies the currency symbol. If this character appears more than once, it is a drifting character; otherwise it is a static character. The static character specifies that the character is to be placed in the associated position. The static character must appear either to the left of all digit positions in a field of a specification or to the right of all digit positions in a specification. See details above for the drifting use of the character.

S specifies the plus sign character (+) if the data value is ≥0, otherwise it specifies the minus sign character (-). The character may be drifting or static. The rules are identical to those for the currency symbol.

+ specifies the plus sign character (+) if the data value is ≥0, otherwise it specifies a blank. The character may be drifting or static. The rules are identical to those for the currency symbol.

- specifies the minus sign character (-) if the data value is <0, otherwise it specifies a blank. The character may be drifting or static. The rules are identical to those for the currency symbol.

CREDIT, DEBIT, AND OVERPUNCHED SIGNS

The character pairs CR (credit) and DB (debit) specify the signs of real numeric character data items and usually appear in business report forms.

Any of the picture characters T, I, or R specifies an overpunched sign in the associated digit position of numeric character data. An overpunched sign is a 12-punch (for plus) or an 11-punch (for minus) punched into the same column as a digit. It indicates the sign of the arithmetic data item. Only one overpunched sign can appear in a specification for a fixed-point number. A floating-point specification can contain two, one in the mantissa field and one in the exponent field. The overpunch character can, however, be specified for any digit position within a field. The overpunched number then will appear in the specified digit position.

Note: When an overpunch character occurs in a P format item for edit-directed input, the corresponding character in the input stream may contain an overpunched sign.

Figure D-6 gives examples of the CR, DB, and overpunch characters. In the figure, the letter b indicates a blank character.

CR specifies that the associated positions will contain the letters CR if the value of the data is less than zero. Otherwise, the positions will contain two blanks. The characters CR can appear only to the right of all digit positions of a field.

DB is used the same way that CR is used except that the letters DB appear in the associated positions.

T specifies that the associated position, on input, will contain a digit overpunched with the sign of the data. It also specifies that an overpunch is to be indicated in the character-string value.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(3)	-123	$Z.99CR	$1.23CR
FIXED(4,2)	12.34	$ZZV.99CR	$12.34bb
FIXED(4,2)	-12.34	$ZZV.99DB	$12.34DB
FIXED(4,2)	12.34	$ZZV.99DB	$12.34bb
FIXED(4)	1021	999I	102A
FIXED(4)	-1021	Z99R	102J
FIXED(4)	1021	99T9	10B1

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D-6. Examples of CR, DB, T, I, and R Picture Characters

I specifies that the associated position, on input, will contain a digit overpunched with + if the value is ≥0; otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character-string value if the data value is ≥0.

R specifies that the associated position, on input, will contain a digit overpunched with - if the value is <0; otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character-string value if the data value is <0.

Note: The picture characters CR, DB, T, I, and R cannot be used with any other sign characters in the same field.

EXPONENT SPECIFIERS

The picture characters K and E delimit the exponent field of a numeric character specification that describes floating-point decimal numbers. The exponent field is always the last field of a numeric character floating-point picture specification. The picture characters K and E cannot appear in the same specification.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FLOAT(5)	.12345E06	V.99999E99	.12345E06
FLOAT(5)	.12345E-06	V.99999ES99	.12345E-06
FLOAT(5)	.12345E+06	V.99999KS99	.12345+06
FLOAT(5)	-123.45E+12	S999V.99ES99	-123.45E+12
FLOAT(5)	001.23E-01	SSS9.V99ESS9	+123.00Eb-3
FLOAT(5)	001.23E+04	ZZZV.99KS99	123.00+02
FLOAT(5)	001.23E+04	SZ99V.99ES99	+123.00E+02
FLOAT(5)	001.23E+04	SSSSV.99E-99	+123.00Eb02

[1]The arithmetic value is the value expressed by the mantissa, multiplied by 10 to the power indicated in the exponent field.

Figure D-7. Examples of Floating-Point Picture Specifications

Figure D-7 gives examples of the use of exponent delimiters. In the figure, the letter b indicates a blank character.

K specifies that the exponent field appears to the right of the associated position. It does not specify a character in the numeric character data item.

E specifies that the associated position contains the letter E, which indicates the start of the exponent field.

The value of the exponent is adjusted in the character-string value so that the first significant digit of the first field (the mantissa) appears in the position associated with the first digit specifier of the specification (even if it is a zero suppression character).

SCALING FACTOR

The picture character F specifies a scaling factor for fixed-point decimal numbers. It appears at the right end of the picture specification and is used in the following format:

F ([+|-] decimal-integer-constant)

F specifies that the optionally signed decimal integer constant enclosed in parentheses is the scaling factor. The scaling factor specifies that the decimal point in the arithmetic value of the variable is that number of places to the right (if the scaling factor is positive) or to the left (if negative) of its assumed position in the character-string value.

For System/360 implementations, the scaling factor cannot specify a fixed-point number that contains more than 15 digits.

Figure D-8 shows examples of the use of the scaling factor picture character.

STERLING PICTURES

The following picture characters are used in picture specifications for sterling data:

8 specifies the position of a shilling digit in BSI single-character representation. Ten shillings is represented by a 12-punch (&) and eleven through nineteen shillings are represented by the characters A through I, respectively.

7 specifies the position of a pence digit in BSI single-character representation. Ten pence is represented by a 12-punch (&) and eleven pence is represented by an 11-punch (-).

6 specifies the position of a pence digit in IBM single-character representation. Ten pence is represented by an 11-punch (-) and eleven pence is represented by a 12-punch (&).

P specifies that the associated position contains the pence character D.

G specifies the start of a sterling picture. It does not specify a character in the numeric character data item.

H specifies that the associated position contains the shilling character S.

M specifies the start of a field. It does not specify a character in the numeric character data item.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(4,0)	1200	99F(2)	12
FIXED(7,0)	-1234500	S999V99F(4)	-12345
FIXED(5,5)	.00012	99F(-5)	12
FIXED(6,6)	.012345	999V99F(-4)	12345

[1]The arithmetic value is the same as the character-string value, multiplied by 10 to the power of the scaling factor.

Figure D-8. Examples of Scaling Factor Picture Characters

Figure D-9 gives examples of the use of sterling picture specifications.

Sterling data items are considered to be real fixed-point decimal data. When involved in arithmetic operations, they are converted to a value representing fixed-point pence. Sterling pictures have the general form:

```
PICTURE

    'G [editing-character-1] ...

    M pounds-field

    M [separator-1] ...
          shillings-field

    M [separator-2] ...
          pence-field

      [editing-character-2] ...'
```

"Editing character 1" can be one or more of the following static picture characters:

$ + - S

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

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

The last four characters ($ + - S) must be drifting characters. The comma can be used as an insertion character.

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

/ . B

The "shillings field" can be:

{99 | YY | ZZ | Y9 | Z9 | YZ | 8}

One of the nines can be replaced by T, I, or R, if no other sign indicator appears in any of the fields of the specification.

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

/ . B H

The "pence field" takes the form:

{99|YY|ZZ|Y9|7|Z9|YZ|6}

[[V|V.|.V] 9|Z|Y]...

One of the nines can be replaced by T, I, or R, if no other sign indicator appears in any of the fields of the specification.

"Editing character 2" can be one or more of the static picture characters $, +, -, or S and one or more of B, P, CR, or DB. A sign character or CR or DB can appear only if no other sign indicator appears in any of the fields of the specification.

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

Zero suppression in sterling pictures is performed on the total data item, not separately on each of the pounds, shillings, and pence fields. The Z picture character is not allowed to the right of a 6, 7, 8, or 9 picture character in a sterling specification. In sterling pictures, the field separator characters slash (/), point (.), B, and H are never suppressed.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(4)	0534	GMZ9M.8M.99V.9CR	b2.4.06.0bb
FIXED(4)	0019	GMZZM.ZZM.ZZP	bb.b1.07D

[1]The arithmetic value of a numeric character variable declared with a sterling picture specification is its value expressed as a valid sterling fixed-point constant, which for arithmetic operations is always converted to its value expressed in pence.

Figure D-9. Examples of Sterling Picture Specifications

SECTION E: EDIT-DIRECTED FORMAT ITEMS

This section describes each of the edit-directed format items that can appear in the format list of a GET or PUT statement.

There are three categories of format items: data format items, control format items, and the remote format item.

In this section, the three categories are discussed separately and the format items are listed under each category. The remainder of the section contains detailed discussions of each of the format items, with the discussions appearing in alphabetic order.

DATA FORMAT ITEMS

A data format item describes the external format of a single data item.

For input, the data in the stream is considered to be a continuous string of characters; all blanks are treated as characters in the stream, as are quotation marks. Each data format item in a GET statement specifies the number of characters to be obtained from the stream and describes the way those characters are to be interpreted. Strings should not be enclosed in quotation marks, nor should the letter B be used to identify bit strings. If the characters in the stream cannot be interpreted in the manner specified, the CONVERSION condition is raised.

For output, the data in the stream takes the form specified by the format list. Each data format item in a PUT statement specifies the width of a field into which the associated data item in character form is to be placed and describes the format that the value is to take. Enclosing quotation marks are not inserted, nor is the letter B to identify bit strings.

Leading blanks are not inserted automatically to separate data items in the output stream. String data is left-adjusted in the field, whose width is specified. Arithmetic data is right-adjusted. Because of the rules for conversion of arithmetic data to character type, which can cause up to three leading blanks to be inserted (in addition to any blanks that replace leading zeros), there generally will be at least one blank preceding an arithmetic item in the converted field. Leading blanks will not appear in the stream, however, unless the specified field width allows for them. Truncation, due to inadequate field-width specification is on the left for arithmetic items, on the right for string items.

Note that the value of binary data both on input and output is always represented in decimal form for edit-directed transmission.

Following is a list of data format items:

Fixed-point format item	F(specification)
Floating-point format item	E(specification)
Complex format item	C(specification)
Picture format item	P'picture-specification'
Bit-string format item	B(specification)
Character-string format item	A(specification)

CONTROL FORMAT ITEMS

The control format items specify the layout of the data set associated with a file. The following is a list of control format items:

Paging format item	PAGE
Line skipping format item	SKIP[(specification)]
Line position format item	LINE (specification)
Column position	COLUMN(specification)
Spacing format item	X(specification)

A control format item has no effect unless it is encountered before the data list is exhausted.

The PAGE and LINE format items apply only to output and only to files with the

PRINT attribute. The SKIP and COLUMN format items apply to both input and output.

The PAGE, SKIP, and LINE format items have the same effect as the corresponding options of the PUT statement (and of the GET statement, in the case of SKIP), except that the format items take effect only when they are encountered in the format list, while the options take effect before any data is transmitted.

The COLUMN format item positions the file to the specified character position in the current line.

The spacing format item specifies relative horizontal spacing. On input, it specifies a number of characters in the stream to be skipped over and ignored; on output, it specifies a number of blanks to be inserted into the stream.

REMOTE FORMAT ITEM

The remote format item specifies the label of a FORMAT statement that contains a format list which is to be taken to replace the remote format item.

The remote format item is:

```
R(statement-label-designator)
```

The "statement label designator" is a label constant or an element label variable.

USE OF FORMAT ITEMS

The "specification" that is listed above for all but the picture, PAGE, and remote format items can contain one or more expressions. Such expressions can be specified as decimal integer constants, as element variables, or as other element expressions. The value assigned to a variable during an input operation can be used in an expression in a format item that is associated with a later data item. An expression is evaluated and converted to an integer each time the format item is used.

ALPHABETIC LIST OF FORMAT ITEMS

The A Format Item

The A format item is:

```
A [(field-width)]
```

The character-string format item describes the external representation of a string of characters.

General rules:

1. The "field width" is an expression that is evaluated and converted to an integer each time the format item is used. It specifies the number of character positions in the data stream that contain (or will contain) the string.

2. On input, the specified number of characters is obtained from the data stream and assigned, with any necessary conversion, truncation, or padding, to the associated element in the data list. The field width is always required on input, and if it has a value less than or equal to zero, a null string is assumed. If quotation marks appear in the stream, they are treated as characters in the string.

3. On output, the associated element in the data list is converted, if necessary, to a string of characters and is truncated or extended with blanks on the right to the specified field width before being placed into the data stream. If the field width is less than or equal to zero, the format item and its associated element in the data list are skipped, and no characters are placed into the data stream. Enclosing quotation marks are never inserted. If the field width is not specified, it is assumed to be equal to the character-string length of the element named in the data list (after conversion, if necessary, according to the rules given in Section F, "Problem Data Conversion").

The B Format Item

The B format item is:

```
B [(field-width)]
```

The bit-string format item describes the external representation of a bit string. Each bit is represented by the character 0 or 1.

General rules:

1. The "field width" is an expression that is evaluated and converted to an integer each time the format item is used. It specifies the number of data-stream character positions that contain (or will contain) the bit string.

2. On input, the character representation of the bit string may occur anywhere within the specified field. Blanks, which may appear before and after the bit string in the field, are ignored. Any necessary conversion occurs when the bit string is assigned to the associated element in the data list. The field width is always required on input, and if it is less than or equal to zero, a null string is assumed. Any character other than 0 or 1 in the string, including embedded blanks, quotation marks, or the letter B, will raise the CONVERSION condition.

3. On output, the character representation of the bit string is left-adjusted in the specified field, and necessary truncation or extension with blanks occurs on the right. Any necessary conversion to bit-string is performed. No quotation marks are inserted, nor is the identifying letter B. The field width need not be specified when the associated element in the data list is a bit string; in this case, the current length of the associated string is used, and the data item completely fills the field. The field width is always required if the data-list item is arithmetic or pictured. If the field width is less than or equal to zero, the format item and its associated element in the data list are skipped, and no characters are placed into the data stream.

The C Format Item

The C format item is:

```
C(real-format-item[,real-format-item])
```

The complex format item describes the external representation of a complex data item.

General rules:

1. Each "real format item" is specified by one of the F, E, or P format items. The P format item must describe fixed-point or floating-point numeric character data; it cannot describe sterling or character-string data.

2. On input, the complex format item describes the real and imaginary parts of the complex data item within adjacent fields in the data stream. If the second real format item is omitted, it is assumed to be the same as the first. The letter I will cause the CONVERSION condition to be raised.

3. On output, the real format items describe the forms of the real and imaginary parts of the complex data item in the data stream. If the second real format item is omitted, it is assumed to be the same as the first. The letter I is never appended to the imaginary part. If the second real format item (or the first, if only one appears) is an F or E item, the internal sign will be printed only if the value of the imaginary part is less than zero. If the real format item is a P item, the sign will be printed only if the S or - or + picture character is specified. If the I is to be appended, it must be specified as a separate data item in the data list, immediately following the variable that specifies the complex item. The I, then, must have a corresponding format item (either A or P).

The COLUMN Format Item

The COLUMN format item is:

```
COLUMN (character-position)
```

The column position format item positions the file to a specified character position within the line. It can be used with either input or output files.

General rules:

1. The "character position" can be specified by an expression, which is evaluated and converted to an integer each time the format item is used.

2. The file is positioned to the specified character positon in the current line. On input, intervening character positions are ignored; on output, they are filled with blanks. If the file is already positioned after the specified character positon, the current line is completed and a new line is started; the format item is then applied to the new line.

3. If the specified character position lies beyond the rightmost character position of the current line, or if

the value of the expression for the character position is less than one, then the character position is assumed to be one.

Note: The rightmost character position is determined as follows:

a. For output files, it is determined by the line size;

b. For input files, the F Compiler uses the length of the current logical record to determine the line size and, hence, the rightmost character position. In the case of V-format records, this line size is equal to the logical record length minus the number of bytes containing control information.

4. The COLUMN format item has no effect unless it is encountered before the data list is exhausted.

The E Format Item

The E format item is:

```
E(field-width,number-of-fractional-digits
        [,number-of-significant-digits])
```

The floating-point format item describes the external representation of decimal arithmetic data in floating-point format.

General rules:

1. The "field width," "number of fractional digits," and "number of significant digits" can be represented by expressions, which are evaluated and converted to integers when the format item is used.

   "Field width" specifies the total number of characters in the field.

   "Number of fractional digits" specifies the number of digits in the mantissa that follow the decimal point.

   "Number of significant digits" specifies the number of digits that must appear in the mantissa.

2. On input, the data item in the data stream is the character representation of an optionally signed decimal floating-point or fixed-point constant located anywhere within the specified field. If the data item is a fixed-point number, an exponent of zero is assumed.

   The external form of a floating-point number is:

$$[+|-]\ \text{mantissa}\left[\left\{\begin{matrix}[E]\\ E\end{matrix}\right\}[+|-]\ \text{exponent}\right]$$

   The mantissa must be a decimal fixed-point constant.

   a. The number can appear anywhere within the specified field; blanks may appear before and after the number in the field and are ignored. If the entire field is blank, the CONVERSION condition is raised. When no decimal point appears, the expression for the number of fractional digits specifies the number of character positions in the mantissa to the right of the assumed decimal point. If a decimal point does appear in the number, it overrides the specification of the number of the fractional digits.

      The value expressed by "field width" includes trailing blanks, the exponent position, the positions for the optional plus or minus signs, the position for the optional letter E, and the position for the optional decimal point in the mantissa.

   b. The exponent is a decimal integer constant. Whenever the exponent and preceding sign or letter E are omitted, a zero exponent is assumed.

3. On output, the internal data is converted to floating-point, and the external data item in the specified field has the following general form:

```
[-] {s-d digits}.{d digits}
     E {+|-} exponent
```

   In this form, s represents the number of significant digits, and d represents the number of fractional digits. The value is rounded if necessary.

   a. The exponent is a two-digit decimal integer constant, which may be two zeros. The exponent is automatically adjusted so that the leading digit of the mantissa is nonzero. When the value is zero, zero suppression is applied to all digit positions (except the first) to the left of the decimal point. All other digit positions contain zero.

b. If the above form of the number does not fill the specified field on output, the number is right-adjusted and extended on the left with blanks. If the number of significant digits is not specified, it is taken to be 1 plus the number of fractional digits. For System/360 implementations, the field width for non-negative values of the data item must be greater than or equal to 5 plus the number of significant digits. For negative values of the data item, the field width must be greater than or equal to 6 plus the number of significant digits. However, if the number of fractional digits is zero, the decimal point is not written, and the above figures for the field width are reduced by 1.

c. The rounding of internal data is as follows: if truncation causes a digit to be lost from the right, and this digit is greater than or equal to 5, then 1 is added to the digit to the left of the truncated digit.

d. If the field width is such that significant digits or the sign is lost, the SIZE condition is raised.

The F Format Item

The F format item is:

```
F(field-width[,number-of-fractional-digits
                [,scaling-factor]])
```

The fixed-point format item describes the external representation of a decimal arithmetic data item in fixed-point format.

General rules:

1. The "field width," "number of fractional digits," and "scaling factor" can be represented by element expressions, which are evaluated and converted to integers when the format item is used.

2. On input, the data item in the data stream is the character representation of an optionally signed decimal fixed-point constant located anywhere within the specified field. Blanks may appear before and after the number in the field and are ignored. If the entire field is blank, it is interpreted as zero.

The number of fractional digits, if not specified, is assumed to be zero.

If no scaling factor is specified and no decimal point appears in the field, the expression for the number of fractional digits specifies the number of digits in the field to the right of the assumed decimal point. If a decimal point actually does appear in the data, it overrides the expression for the number of fractional digits.

If a scaling factor is specified, it effectively multiplies the value of the data item in the data stream by 10 raised to the integral value (p) of the scaling factor. Thus, 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 number 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 appears, or by the expression for the number of fractional digits, in the absence of an actual point.

3. On output, the internal data is converted, if necessary, to fixed-point; the external data is the character representation of a decimal fixed-point number, rounded if necessary, and right-adjusted in the specified field.

If only the field width is specified in the format item, only the integer portion of the number is written; no decimal point appears.

If both the field width and number of fractional digits are specified, but the scale factor is not, both the integer and fractional portions of the number are written. If the value (d) of the number of fractional digits is greater than zero, a decimal point is inserted before the rightmost d digits. Trailing zeros are supplied when the number of fractional digits is less than d (the value d must be less than the field width). Suppression of leading zeros is applied to all digit positions (except the first) to the left of the decimal point.

The rounding of internal data is as follows: if truncation causes a digit to be lost from the right, and this digit is greater than or equal to 5, then 1 is added to the digit to the left of the truncated digit.

The integer value (p) of the scaling factor effectively multiplies the value of the associated element in the data list by 10 raised to the power of p, before it is edited into its external character representation. When the number of fractional digits is zero, only the integer portion of the number is used.

On output, if the value of the fixed-point number is less than zero, a minus sign is prefixed to the external character representation; if it is greater than or equal to zero, no sign appears. Therefore, for negative values of the fixed-point number, the field width specification must include a count of both the sign and the decimal point.

If the field width is such that significant digits or the sign is lost, the SIZE condition is raised.

The LINE Format Item

The LINE format item is:

```
LINE (line-number)
```

The line position format item specifies the particular line on a page of a PRINT file upon which the next data item is to be printed.

General rules:

1. The "line number" can be represented by an expression, which is evaluated and converted to an integer each time the format item is used.
2. The LINE format item specifies that blank lines are to be inserted so that the next line will be the specified line of the current page.
3. If the specified line has already been passed on the current page, or if the specified line is beyond the limits set by the PAGESIZE option of the OPEN statement (or by default), the ENDPAGE condition is raised.
4. If "line number" is less than or equal to zero, it is assumed to be one.
5. The LINE format item has no effect unless it is encountered before the data list is exhausted.

The P Format Item

The P format item is:

```
P 'picture-specification'
```

The picture format item describes the external representation of numeric character data and of character-string data.

The "picture specification" is discussed in detail in Section D, "Picture Specification Characters" and in the discussion of the PICTURE attribute in Section I, "Attributes."

On input, the picture specification describes the form of the data item expected in the data stream and, in the case of a numeric character string, how its arithmetic value is to be interpreted. Note that the picture specification should accurately describe the data in the input stream, including characters represented by editing characters. If the indicated character does not appear in the stream, the CONVERSION condition is raised.

On output, the value of the associated element in the data list is edited to the form specified by the picture specification before it is written into the data stream.

The PAGE Format Item

The PAGE format item is:

```
PAGE
```

The paging format item specifies that a new page is to be established. It can be used only with PRINT files.

General rules:

1. The establishment of a new page implies that the next printing is to be on line one.
2. The PAGE format item has no effect unless it is encountered before the data list is exhausted.

The R Format Item

The R format item is:

```
R (statement-label-designator)
```

The remote format item allows format items in a FORMAT statement to replace the remote format item.

General rules:

1. The "statement label designator" is a label constant or an element 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. (If the procedure is executed recursively, they must be in the same invocation.)

3. There can be no recursion within a FORMAT statement. That is, a remote FORMAT statement cannot contain an R format item that names itself as a statement label designator, nor can it name another remote FORMAT statement that will lead to the naming of the original FORMAT statement. Avoidance of recursion can be 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 also be enabled for the remote FORMAT statement(s) that are referred to.

5. If the GET or PUT statement is the single statement of an on-unit, it cannot contain a remote format item.

The SKIP Format Item

The SKIP format item is:

SKIP[(relative-position-of-next-line)]

The line skipping format item specifies that a new line is to be defined as the current line.

General rules:

1. The "relative position of next line" can be specified by an element expression, which is evaluated and converted to an integer each time the format item is used. It must be greater than zero for non-PRINT files. If it is not, or if it is omitted, 1 is assumed.

2. The new line is the specified number of lines beyond the present line.

3. If the value of the relative position is greater than one, then on input, one or more lines will be ignored; on output, one or more blank lines will be inserted.

4. The value of the relative position may be less than or equal to zero for PRINT files only; the effect is that of a carriage return without line spacing. Characters previously written may be overprinted.

5. If the SKIP format item is not specified at the end of a line, then SKIP (1) is assumed, that is, single spacing.

6. For PRINT files, if the specified relative position is beyond the limit set by the PAGESIZE option of the OPEN statement (or the default), the ENDPAGE condition is raised.

7. The SKIP format item has no effect unless it is encountered before the data list is exhausted.

The X Format Item

The X format item is:

X (field-width)

The spacing format item controls the relative spacing of data items in the data stream. It is not limited to PRINT files.

General rules:

1. The "field width" can be represented by an expression, which is evaluated and converted to an integer each time the format item is used. The integer specifies the number of blanks before the next field of the data stream, relative to the current position in the stream.

2. On input, the specified number of characters is spaced over in the data stream and not transmitted to the program.

3. On output, the specified number of blank characters are inserted into the stream.

4. If the field width is less than zero, it is assumed to be zero.

5. The spacing format item has no effect unless it is encountered before the data list is exhausted.

This section lists the rules for arithmetic conversion and for conversion of problem data types. Each type conversion is listed under a separate heading. In addition to the text, twelve tables appear:

- Tables F-1 through F-4 show the data type of the result of an operation involving two operands of possibly differing types. Note that although the tables are for two operands, these operands could themselves be the result of other operations: any expression involving a number of infix operators will be eventually reduced, during evaluation, to a single infix operation with two operands. Note also that the result is the result of the expression only, and may be converted on subsequent assignment.
- Table F-5 states the rules for computing the precision of the result of an arithmetic conversion.
- Table F-6 states the rules for computing the length of the result of an arithmetic to character-string conversion.
- Table F-7 states the rules for computing the length of the result of an arithmetic to bit-string conversion.
- Table F-8 can be used to find the ceiling (CEIL) of any value between 1 and 15 when that value is multiplied by 3.32 or it can be used to find the ceiling (CEIL) of any value between 1 and 56 when that value is divided by 3.32.
- Tables F-9 through F-12 illustrate conversion in arithmetic expression operations, and they give attributes of the results based upon the operator specified and the attributes of the two operands.

ARITHMETIC CONVERSION

The rules for arithmetic conversion specify the way in which a value is transformed from one arithmetic representation to another. It can be that, as a result of the transformation, the value will change. For example, the number .2, which can be exactly represented as a decimal fixed-point number, cannot be exactly represented in binary. The magnitude of such changes in value depends upon the precisions of the target and source. In expression evaluation, the precision of the target is derived from the precision of the source. In order to estimate and to understand the errors that can occur, the precision rules must be understood; and since the rules also leave some latitude for the implementation, it is helpful to have some knowledge of the way in which conversions are implemented.

Floating-Point Conversion

In System/360 implementations, both decimal and binary floating-point numbers are maintained in the internal hexadecimal form used in System/360. If the specified precision is more than 6 decimal digits, or 21 binary digits, the number is maintained in long floating-point form (14 hexadecimal digits with a hexadecimal exponent). If the precision is 6 decimal digits or less, or 21 binary digits or less, the number is maintained in short floating-point form (6 hexadecimal digits and a hexadecimal exponent).

No actual conversions between binary and decimal are performed on floating-point data. The only precision changes are from long to short, which is done by truncation, and from short to long, which is done by extending with zeros. The declared precision of floating-point data and the base, however, do affect the calculation of target attributes, as well as the attributes of intermediate forms that are determined from the source.

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 value of the real source as the real part and zero as the imaginary part.

Precision Conversion

Precision conversion occurs if the specified target precision is different from the source precision. In particular, there always is a precision change when the source and target are of different bases. It is also possible that there is an actual change in precision when converting from floating-point to fixed-point, because of the way in which floating-point numbers are represented. Precision changes are performed by truncation or by padding with zeros. Floating-point numbers are converted from short precision to long precision by extending with zeros on the right, and from long precision to short precision by truncation on the right.

Fixed-point numbers maintain decimal or binary point alignment and may be truncated on the left or right, or extended with zeros on the left or right.

No indication is given of loss of significant digits on the right. Loss of digits on the left can be checked for if the SIZE condition is enabled. In System/360 implementations, binary fixed-point numbers are stored in words of 31 bits, whatever the declared width. Decimal numbers are always stored as an odd number of digits, since they are maintained in System/360 packed decimal format, with the rightmost four bits of the rightmost byte expressing the sign.

Base Conversion

Changes in base will usually affect only the value of noninteger fixed-point numbers. Some decimal fractions cannot be expressed exactly in binary, and some errors will then occur due to truncation. Some binary fractions will also require more decimal digits for exact representation than are automatically generated by the conversion rules, and this may also cause errors resulting from truncation.

Since the range of binary fixed-point numbers is smaller than the range of decimal fixed-point numbers, it is possible for significant digits to be lost on the left in conversion from decimal to binary. This will raise the SIZE condition, but an interrupt will not occur unless the condition is explicitly enabled by a SIZE prefix.

The natural notation for constants is decimal and, therefore, most constants are written in decimal. The precision of a constant is derived from the way in which it is written. Care should therefore be taken when writing noninteger constants that will be converted to fixed-point binary.

The following examples illustrate how the representation of a decimal constant (.1) is converted when used in an arithmetic expression (such as A+.1). Target attributes are derived from the attributes of A, the operator, and the attributes of the constant, which are, in this case, DECIMAL FIXED (1,1).

```
Attributes of A:   FIXED BIN(10,2)
Value:             .1
Target:            FIXED BIN(5,4)
Final Value:       .0625

Attributes of A:   FLOAT BIN(50)
Value:             .1
Target:            FLOAT BIN(4)
Final Value:       .1>value>.0625
```

Coded Arithmetic to Numeric Character

Coded arithmetic data being converted to numeric character is converted, if necessary, to a decimal value whose scale and precision are determined by the PICTURE attribute of the numeric character item.

Numeric Character to Coded Arithmetic

Numeric character data being converted to coded arithmetic is first interpreted as a decimal item of the scale and precision determined by the corresponding PICTURE attribute. This item is then converted to the base, scale, and precision of the coded arithmetic target.

DATA TYPE CONVERSION

Character-String to Arithmetic

The source string must represent a valid arithmetic constant or complex expression. The constant may optionally be signed, and may be surrounded by blanks, but cannot contain blanks between the sign and the value of the constant, or between the end of the real part and the sign of the imaginary part in a complex expression.

The permitted forms are:

[+|-]arithmetic-constant

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

A null string gives the value zero.

The constant will itself have the attributes of base, scale, mode, and precision. It will be converted to conform with the attributes of the target.

Even when converting from character string to numeric character field, the source must still contain a constant which is valid according to the rules for constants in PL/I source programs. The value of this constant is then converted and edited to the picture representation.

The following example will therefore result in a conversion error:

```
DCL A PICTURE '$$$9V.99';

   A='$17.95';
```

The currency symbol makes the character-string constant invalid for conversion to the arithmetic value of the numeric character variable, even though its character-string value contains a currency symbol.

Correct examples are:

```
A='17.95';

A=17.95;
```

either of which would result in A having the character-string value b$17.95.

For conversion from character string to arithmetic, the attributes assumed for the target are those attributes that would have been assumed if a fixed-point decimal integer of precision (15,0) had appeared in place of the string. (The precision given is that for the F Compiler.)

Arithmetic to Character-String

The arithmetic value is converted to a decimal arithmetic constant. The constant is inserted in an intermediate character string whose length is derived from the attributes of the source (see Table F-6, "Lengths of Converted Character Strings"). Except for the base and precision, the attributes of the constant are the same as the attributes of the source.

In the case of the conversion of expression results, the intermediate string is assigned to the target string, and may be truncated or padded with zeros on the right.

Since the rules of arithmetic to character-string conversion are also used for list-directed and data-directed output, and for evaluating keys (even for REGIONAL files) this type of conversion will be found in most programs, and should be thoroughly understood.

Numeric Character to Character-String

Real numeric character fields are treated as character strings and assigned to the target string from left to right according to the rules for character-string assignment.

The real and imaginary parts of complex numeric character fields are concatenated, and the resulting string is assigned to the target. No character, including I or blank, is inserted between or following the two parts.

Fixed-Point to Character-String

A binary fixed-point source is first converted to decimal, and the decimal precision is derived from the precision of the binary source (see Table F-5, "Precision for Arithmetic Conversions").

A decimal fixed-point source with precision (p,q) is converted to character-string representation as follows:

1. If p>=q>=0 (that is, if the assumed decimal point lies within the field of the internal representation) then:
   - The constant is right adjusted in a field of width p+3.
   - Leading zeros are replaced by blanks, except for a single zero that immediately precedes the decimal point of a fractional number.
   - If the value is negative, a minus sign precedes the first significant digit (or the zero before the point of a fractional number). Positive values are unsigned.
   - Unless the source is an integer, the constant has q fractional digits. If the source is an integer, there is no decimal point.
2. If q is negative or greater than p, a scaling factor is appended to the

Source Attributes	Source Value	Intermediate String	Target Attributes	Result
FIXED DEC(5,0)	2497	'bbbb2497'	CHAR(10)	'bbbb2497bb'
FIXED DEC(5,0)	2497	'bbbb2497'	CHAR(5)	'bbbb2'
FIXED DEC(4,1)	-121.7	'b-121.7'	CHAR(7)	'b-121.7'
FIXED DEC(4,5)	.01217	'b1217F-5'	CHAR(7)	'b1217F-'
FIXED DEC(4,-3)	-3279000	'-3279F+3'	CHAR(8)	'-3279F+3'
FIXED DEC(3,3)	-.567	'-0.567'	CHAR(6)	'-0.567'
FIXED BIN(15,0)	4095	'bbbbb4095'	CHAR(8)	'bbbbb409'
FIXED BIN(3,3)	.375	'bb0.3'	CHAR(4)	'bb0.'
FIXED BIN(15,-15)	-65536	'b-65536F+5'	CHAR(10)	'b-65536F+5'

Figure F-1. Examples of Conversion from Fixed-Point to Character-String

right of the constant. The constant itself is of the same form as an integer. The scaling factor has the form:

```
F{+|-}nnn
```

where {+|-}nnn has the value -q.

The number of digits in the scaling factor is just sufficient to contain the value of q without leading zeros.

The length of the intermediate string is:

```
p+3+k
```

where k is the number of digits necessary to represent the value of q (not including a sign or the letter F). For example, given:

```
DCL A FIXED(4,-3),
    C CHAR(10);
    A=1234.0E3;
    C=A;
```

The intermediate string generated in converting A would be:

```
b1234F+3
```

which, when assigned to C, would give:

```
b1234F+3bb
```

Other examples are shown in Figure F-1.

Floating-Point to Character-String

If the source is binary, its binary precision is converted to the equivalent decimal precision (see Table F-5, "Precision for Arithmetic Conversions").

The decimal source with precision p is converted as if it were transmitted by an E format item of the form E(w,d,s) where:

- w, the length of the intermediate string, is p+6 (for the F Compiler)
- d, the number of fractional digits, is p-1
- s, the number of significant digits, is p

For the F Compiler, an E format item generates a floating-point decimal constant with a signed two-digit exponent. (See Part II, Section E, "Edit-Directed Format Items.")

The following examples illustrate the intermediate string generated for a floating-point to character-string conversion:

```
Source Attributes:   FLOAT DEC(6)
Source Value:        1735x10⁵
Intermediate String: b1.73500E+08

Source Attributes:   FLOAT BIN(20)
Source Value:        -91882x10²
Intermediate String: -9.182200E+06

Source Attributes:   FLOAT DEC(5)
Source Value:        -.0016632
Intermediate String: -1.6632E-03
```

Complex to Character-String

The intermediate string that is generated contains a complex expression. Its length is 1 plus twice the length of the character string generated by a real source with corresponding attributes. The intermediate string consists of two concatenated strings. The left-hand, or real, part consists of a string generated exactly as for a real source. The right-hand, or imaginary, part is always signed, and it has an I appended. The string length of the imaginary part is one character longer than the real part (to allow for the I). The resulting string is a complex expression, with a sign but no blanks between its elements.

The following examples illustrate the intermediate string that results from a complex to character-string conversion:

```
Source:   COMPLEX DEC FLOAT(5)
Value:    17.3+1.5I
Result:   b1.7300E+01+1.5000E+00I

Source:   COMPLEX DEC FIXED(4,3)
Value:    0.133+0.007I
Result:   bbb0.133+0.007I
```

Character-String to Bit-String

The character 1 in the source string becomes the bit 1 in the target string. The character 0 in the source string becomes the bit 0 in the target string. Any character other than 0 and 1 in the source string will raise the CONVERSION condition. A null character string becomes a null bit string.

If the source string is longer than the target, excess characters on the right are ignored (so that excess characters other than 0 or 1 will not raise the CONVERSION condition). If the target is longer than the source, the target is padded on the right with zeros.

Bit-String to Character-String

The bit 0 becomes the character 0, and the bit 1 becomes the character 1. A null bit string becomes a null character string. The generated character string, which has the same length as the source bit string, is assigned to the target.

If the source bit string is shorter than the target character string, the remainder of the target is padded with blanks.

The following are examples of bit-string to character-string conversion:

```
Source Value:          '1011'B
Target Attributes:     CHAR(4)
Result:                '1011'

Source Value:          '10101'B
Target Attributes:     CHAR(10) VAR
Result:                '10101'

Source Value:          '10101'B
Target Attributes:     CHAR(10)
Result:                '10101bbbbb'

Source Value:          '0001'B
Target Attributes:     CHAR(1)
Result:                '0'
```

The CONVERSION condition cannot be raised on conversion from bit to character; however, a character string created by conversion from a bit string can cause a conversion error when reconverted if blanks have been inserted.

Arithmetic to Bit-String

The absolute arithmetic value is first converted to a binary integer, whose precision is the same as the length of the bit-string target as given in Table F-7. This integer, without a sign, is then treated as a bit string. This intermediate string is then assigned to the target.

Examples are shown in Figure F-2.

Bit-String to Arithmetic

For the F Compiler, the effect is as if the bit string were interpreted as an unsigned binary integer of maximum precision (56,0). If the string is longer than 56 bits, bits on the left are ignored: the SIZE condition will be raised if nonzero bits are lost, provided that SIZE is enabled. Note that truncation is on the left, not on the right. The null string gives the value zero; otherwise, the result of a bit-string to arithmetic conversion is always positive.

Source Attributes	Source Value	Intermediate String	Target Attributes	Result
FIXED BIN(10)	15	'0000001111'B	BIT(10)	'0000001111'B
FIXED BIN(1)	1	'1'B	BIT(1)	'1'B
FIXED DEC(1)	1	'0001'B	BIT(1)	'0'B
FIXED BIN(3)	-3	'011'B	BIT(3)	'011'B
FIXED BIN(4,2)	1.25	'01'B	BIT(2)	'01'B
FIXED DEC(2,1)	1.1	'0001'B	BIT(4)	'0001'B
FLOAT BIN(4)	1.25	'0001'B	BIT(5)	'00010'B

Figure F-2. Examples of Conversion From Arithmetic to Bit-String

Table F-1. Data Type of Result of Bit-String Operation

OPERAND TYPES	CODED ARITHMETIC	NUMERIC CHARACTER	CHARACTER STRING	BIT STRING
CODED ARITHMETIC	Bit string	Bit string	Bit string	Bit string
NUMERIC CHARACTER	Bit string	Bit string	Bit string	Bit string
CHARACTER STRING	Bit string	Bit string	Bit string	Bit string
BIT STRING	Bit string	Bit string	Bit string	Bit string

Table F-2. Data Type of Result of Concatenation Operation

OPERAND TYPES	CODED ARITHMETIC	NUMERIC CHARACTER	CHARACTER STRING	BIT STRING
CODED ARITHMETIC	Character string	Character string	Character string	Character string
NUMERIC CHARACTER	Character string	Character string	Character string	Character string
CHARACTER STRING	Character string	Character string	Character string	Character string
BIT STRING	Character string	Character string	Character string	Bit string

Table F-3a. Data Type of Result of Comparison Operation

OPERAND TYPES	CODED ARITHMETIC	NUMERIC CHARACTER	CHARACTER STRING	BIT STRING
CODED ARITHMETIC	Bit string	Bit string	Bit string	Bit string
NUMERIC CHARACTER	Bit string	Bit string	Bit string	Bit string
CHARACTER STRING	Bit string	Bit string	Bit string	Bit string
BIT STRING	Bit string	Bit string	Bit string	Bit string
Note: Although the final result of a comparison operation is always a bit string of length 1, the type of comparison (algebraic, character, or bit) depends on the data type of the intermeidate operands after conversion, which are shown in Table F-3b.				

●Table F-3b. Data Type of Intermediate Operands of Comparison Operation

OPERAND TYPES (before conversion)	CODED ARITHMETIC	NUMERIC CHARACTER	CHARACTER STRING	BIT STRING
CODED ARITHMETIC	Coded arithmetic	Coded arithmetic	Coded arithmetic	Coded arithmetic
NUMERIC CHARACTER	Coded arithmetic	Coded arithmetic	Coded arithmetic	Coded arithmetic
CHARACTER STRING	Coded arithmetic	Coded arithmetic	Character string	Character string
BIT STRING	Coded arithmetic	Coded arithmetic	Character string	Bit string

● Table F-4. Data Type of Result of Arithmetic Operation

OPERAND TYPES	CODED ARITHMETIC	NUMERIC CHARACTER	CHARACTER STRING	BIT STRING
CODED ARITHMETIC	Coded arithmetic	Coded arithmetic	Coded arithmetic	Coded arithmetic
NUMERIC CHARACTER	Coded arithmetic	Coded arithmetic	Coded arithmetic	Coded arithmetic
CHARACTER STRING	Coded arithmetic	Coded arithmetic	Coded arithmetic	Coded arithmetic
BIT STRING	Coded arithmetic	Coded arithmetic	Coded arithmetic	Coded arithmetic

●Table F-5. Precision for Arithmetic Conversions

Source Attributes	Target Attributes	Target Precision
DECIMAL FIXED(p,q)	DECIMAL FLOAT	p
DECIMAL FIXED(p,q)	BINARY FIXED	1+p*3.32,q*3.32 (see note 3)
DECIMAL FIXED(p,q)	BINARY FLOAT	p*3.32
DECIMAL FLOAT(p)	BINARY FLOAT	p*3.32
BINARY FIXED(p,q)	BINARY FLOAT	p
BINARY FIXED(p,q)	DECIMAL FIXED	1+p/3.32,q/3.32 (see note 4)
BINARY FIXED(p,q)	DECIMAL FLOAT	p/3.32
BINARY FLOAT(p)	DECIMAL FLOAT	p/3.32

Notes:

1. In the cases of p*3.32 and p/3.32, the CEIL of the result is taken; the value taken is an integer that is equal to or greater than the result.

2. Target precision never can exceed the implementation-defined maximums, which are 15 for FIXED DECIMAL, 31 for FIXED BINARY, 16 for FLOAT DECIMAL, and 53 for FLOAT BINARY.

3. When q is negative, the following formula applies:

```
(MIN(CEIL(p*3.32)+1,31),CEIL(ABS(q)*3.32)*SIGN(q))
```

4. When q is negative, the following formula applies:

```
(CEIL(p/3.32)+1,CEIL(ABS(q)/3.32*SIGN(q))
```

●Table F-6. Lengths of Converted Character Strings (Arithmetic To Character-String)

Source Attributes	Conditions	Target Length
DECIMAL FIXED(p,q)	If p>=q>=0	p+3
	If q>p or q negative	p+3+k (where k = number of decimal digits to express scale factor)
DECIMAL FLOAT(p)		p+6
Numeric character data		Same as source
Note: Binary data is converted to decimal before conversion to character-string.		

●Table F-7. Lengths of Converted Bit Strings (Arithmetic to Bit-String)

Source Attributes	Target Length
DECIMAL FIXED(p,q)	(p-q)*3.32
DECIMAL FLOAT(p)	p*3.32
BINARY FIXED(p,q)	p-q
BINARY FLOAT(p)	p
Notes: 1. In the cases of p*3.32 and (p-q)*3.32, the CEIL of the result is taken. 2. If q is greater than or equal to p, the result is a null string.	

●Table F-8. Ceiling Values

x	CEIL(x*3.32)	y	CEIL(y/3.32)
1	4	1-3	1
2	7	4-6	2
3	10	7-9	3
4	14	10-13	4
5	17	14-16	5
6	20	17-19	6
7	24	20-23	7
8	27	24-26	8
9	30	27-29	9
10	34	30-33	10
11	37	34-36	11
12	40	37-39	12
13	44	40-43	13
14	47	44-46	14
15	50	47-49	15
		50-53	16
		54-56	17

TABLE OF CEILING VALUES

Table F-8 is intended to aid the programmer in computing the ceiling values used to determine precisions and lengths in conversions. It gives the ceiling for the result of a multiplication by 3.32 of any value (x) between 1 and 15. It also gives the ceiling for the result of a division by 3.32 of any value (y) between 1 and 56.

TABLES FOR RESULTS OF ARITHMETIC OPERATIONS

Tables F-9 through F-12 give the attributes of the results of arithmetic operations, based on the operator specified and the attributes of the two operands. In these tables, the target precisions (i.e., the precisions of the converted operands) can never exceed the implementation-defined maximums, which, for System/360 implementations, are: 15 for FIXED DECIMAL, 31 for FIXED BINARY, 16 for FLOAT DECIMAL, and 53 for FLOAT BINARY.

●Table F-9. Attributes of Result in Addition and Subtraction Operations

Second Operand	First Operand: DECIMAL FIXED(p_1,q_1)	First Operand: DECIMAL FLOAT(p_1)	First Operand: BINARY FIXED(p_1,q_1)	First Operand: BINARY FLOAT(p_1)
DECIMAL FIXED (p_2,q_2)	DECIMAL FIXED(p,q) p=1+MAX(p_1-q_1,p_2-q_2) +MAX(q_1,q_2) q=MAX(q_1,q_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	BINARY FIXED(p,q) p=1+MAX(p_1-q_1,r-s) +MAX(q_1,s) q=MAX(q_1,s) where: r=1+p_2*3.32 s=q_2*3.32	BINARY FLOAT(p) p=MAX(p_1,r) where: r=p_2*3.32
DECIMAL FLOAT (p_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	BINARY FLOAT(p) p=MAX(p_1,r) where: r=p_2*3.32	BINARY FLOAT(p) p=MAX(p_1,r) where: r=p_2*3.32
BINARY FIXED (p_2,q_2)	BINARY FIXED(p,q) p=1+MAX(r-s,p_2-q_2) +MAX(s,q_2) q=MAX(s,q_2) where: r=1+p_1*3.32 s=q_1*3.32	BINARY FLOAT(p) p=MAX(r,p_2) where: r=p_1*3.32	BINARY FIXED(p,q) p=1+MAX(p_1-q_1,p_2-q_2) +MAX(q_1,q_2) q=MAX(q_1,q_2)	BINARY FLOAT(p) p=MAX(p_1,p_2)
BINARY FLOAT (p_2)	BINARY FLOAT(p) p=MAX(r,p_2) where: r=p_1*3.32	BINARY FLOAT(p) p=MAX(r,p_2) where: r=p_1*3.32	BINARY FLOAT(p) p=MAX(p_1,p_2)	BINARY FLOAT(p) p=MAX(p_1,p_2)

●Table F-10. Attributes of Result in Multiplication Operations

Second Operand	First Operand: DECIMAL FIXED(p_1,q_1)	First Operand: DECIMAL FLOAT(p_1)	First Operand: BINARY FIXED(p_1,q_1)	First Operand: BINARY FLOAT(p_1)
DECIMAL FIXED (p_2,q_2)	DECIMAL FIXED(p,q) p=p_1+p_2+1 q=q_1+q_2	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	BINARY FIXED(p,q) p=p_1+r+1 q=q_1+s where: r=1+p_2*3.32 s=q_2*3.32	BINARY FLOAT(p) p=MAX(p_1,r) where: r=p_2*3.32
DECIMAL FLOAT (p_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	BINARY FLOAT(p) p=MAX(p_1,r) where: r=p_2*3.32	BINARY FLOAT(p) p=MAX(p_1,r) where: r=p_2*3.32
BINARY FIXED (p_2,q_2)	BINARY FIXED(p,q) p=r+p_2+1 q=s+q_2 where: r=1+p_1*3.32 s=q_1*3.32	BINARY FLOAT(p) p=MAX(r,p_2) where: r=p_1*3.32	BINARY FIXED(p,q) p=p_1+p_2+1 q=q_1+q_2	BINARY FLOAT(p) p=MAX(p_1,p_2)
BINARY FLOAT (p_2)	BINARY FLOAT(p) p=MAX(r,p_2) where: r=p_1*3.32	BINARY FLOAT(p) p=MAX(r,p_2) where: r=p_1*3.32	BINARY FLOAT(p) p=MAX(p_1,p_2)	BINARY FLOAT(p) p=MAX(p_1,p_2)

●Table F-11. Attributes of Result in Division Operations

		First Operand			
		DECIMAL FIXED(p_1,q_1)	DECIMAL FLOAT(p_1)	BINARY FIXED(p_1,q_1)	BINARY FLOAT(p_1)
Second Operand	DECIMAL FIXED (p_2,q_2)	DECIMAL FIXED(p,q) p=15 q=15-((p_1-q_1)+q_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	BINARY FIXED(p,q) p=31 q=31-((p_1-q_1)+s) where: s=q_2*3.32	BINARY FLOAT(p) p=MAX(p_1,r) where: r=p_2*3.32
	DECIMAL FLOAT (p_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	DECIMAL FLOAT(p) p=MAX(p_1,p_2)	BINARY FLOAT(p) p=MAX(p_1,r) where: r=p_2*3.32	BINARY FLOAT(p) p=MAX(p_1,r) where: r=p_2*3.32
	BINARY FIXED (p_2,q_2)	BINARY FIXED(p) p=31 q=31-((r-s)+q_2) where: r=1+p_1*3.32 s=q_1*3.32	BINARY FLOAT(p) p=MAX(r,p_2) where: r=p_1*3.32	BINARY FIXED(p,q) p=31 q=31-((p_1-q_1)+q_2)	BINARY FLOAT(p) p=MAX(p_1,p_2)
	BINARY FLOAT (p_2)	BINARY FLOAT(p) p=MAX(r,p_2) where: r=p_1*3.32	BINARY FLOAT(p) p=MAX(r,p_2) where: r=p_1*3.32	BINARY FLOAT(p) p=MAX(p_1,p_2)	BINARY FLOAT(p) p=MAX(p_1,p_2)

●Table F-12. Attributes of Result in Exponentiation Operations

	First Operand	Second Operand (Exponent)	Target Attributes of Result
Case (1)	FIXED DECIMAL(p_1,q_1)	Unsigned integer constant with value n	FIXED DECIMAL(p,q) [provided p≤15] p=(p_1+1)*n-1 q=q_1*n
Case (2)	FIXED BINARY(p_1,q_1)	Unsigned integer constant with value n	FIXED BINARY(p,q) [provided p≤31] p=(p_1+1)*n-1 q=q_1*n
Case (3)	FIXED DECIMAL(p_1,q_1) or FLOAT DECIMAL(p_1)	FIXED DECIMAL(p_2,q_2) or FLOAT DECIMAL(p_2)	FLOAT DECIMAL(p) [unless case (1) or (7) is applicable] p=MAX(p_1,p_2)
Case (4)	FIXED BINARY(p_1,q_1) or FLOAT BINARY(p_1)	FIXED DECIMAL(p_2,q_2) or FLOAT DECIMAL(p_2)	FLOAT BINARY(p) [unless case (2) or (7) is applicable] p=MAX(p_1,CEIL(3.32*p_2))
Case (5)	FIXED DECIMAL(p_1,q_1) or FLOAT DECIMAL(p_1)	FIXED BINARY(p_2,q_2) or FLOAT BINARY(p_2)	FLOAT BINARY(p) [unless case (1) or (7) is applicable] p=MAX(CEIL(3.32*p_1),p_2)
Case (6)	FIXED BINARY(p_1,q_1) or FLOAT BINARY(p_1)	FIXED BINARY(p_2,q_2) or FLOAT BINARY(p_2)	FLOAT BINARY(p) [unless case (2) or (7) is applicable] p=MAX(p_1,p_2)
Case (7)	FLOAT DECIMAL(p_1) or FLOAT BINARY(p_1)	FIXED DECIMAL(p_2,0) or FIXED BINARY(p_2,0)	FLOAT(p_1) [with base of first operand]

All of the built-in functions and pseudo-variables that are available to the PL/I programmer are given in this section. The general organization of this section is as follows:

1. Computational Built-in Functions
   a. String-handling built-in functions
   b. Arithmetic built-in functions
   c. Mathematical built-in functions
   d. Array manipulation built-in functions
2. Condition Built-in Functions
3. Based Storage Built-in Functions
4. Multitasking Built-in Functions
5. Miscellaneous Built-in Functions
6. Pseudo-Variables

The computational built-in functions, shown above, provide string handling, arithmetic operations (addition, division, etc.), mathematical operations (trigonometric functions, square root, etc.), and array manipulation. The computational built-in functions are:

String Handling:

BIT	LOW
BOOL	REPEAT
CHAR	STRING
HIGH	SUBSTR
INDEX	UNSPEC
LENGTH	

Arithmetic:

ABS	IMAG
ADD	MAX
BINARY	MIN
CEIL	MOD
COMPLEX	MULTIPLY
CONJG	PRECISION
DECIMAL	REAL
DIVIDE	ROUND
FIXED	SIGN
FLOAT	TRUNC
FLOOR	

Mathematical:

ATAN	LOG10
ATAND	LOG2
ATANH	SIN
COS	SIND
COSD	SINH
COSH	SQRT
ERF	TAN
ERFC	TAND
EXP	TANH
LOG	

Array Manipulation:

ALL	LBOUND
ANY	POLY
DIM	PROD
HBOUND	SUM

The condition built-in functions allow the PL/I programmer to investigate interrupts arising from enabled ON-conditions. The condition built-in functions are:

DATAFIELD	ONFILE
ONCHAR	ONKEY
ONCODE	ONLOC
ONCOUNT	ONSOURCE

The based storage built-in functions are designed to facilitate list processing and the use of based storage. They mainly return special values which can be assigned to locator and area variables. The based storage built-in functions are:

ADDR	NULL
EMPTY	NULLO

The multitasking built-in functions allow the programmer to investigate the current state of a task or asynchronous input/output operation, or the current priority of a task. The multitasking built-in functions are:

COMPLETION
PRIORITY
STATUS

The miscellaneous built-in functions perform various duties; for example, one function provides the current date, another provides a count of data items transmitted during a STREAM input/output operation, while another provides an indication of whether or not a controlled variable is in an allocated state. The miscellaneous built-in functions are:

ALLOCATION	LINENO
COUNT	TIME
DATE	

The section on pseudo-variables gives a short discussion for each of the PL/I pseudo-variables. A more complete description can be found in the discussion of the corresponding built-in function. The pseudo-variables are:

COMPLETION	PRIORITY
COMPLEX	REAL
IMAG	STATUS
ONCHAR	SUBSTR
ONSOURCE	UNSPEC

All of the built-in functions and pseudo-variables are presented in alphabetical order under their proper headings.

COMPUTATIONAL BUILT-IN FUNCTIONS

STRING HANDLING BUILT-IN FUNCTIONS

The functions described in this section may be used for manipulating strings. Unless it is specifically stated otherwise, any argument can be an element expression or an array expression. If an argument is an array, the value returned by the built-in function is an array with bounds identical to that argument (the function having been performed for each element of the array). For those functions where two or more array arguments are allowed, the arguments must have identical bounds. An argument that is specified as "string" can be an expression of any data type, but if it is arithmetic, it is converted to bit-string (if binary base) or character-string (if decimal base) before the function is invoked.

All conversions mentioned in this section are made according to the rules for the conversion of expression operands as specified in Section F, "Problem Data Conversion."

BIT String Built-in Function

Definition: BIT converts a given value to a bit string and returns the result to the point of invocation. This function allows the programmer to control the size of the result of a bit-string conversion.

Reference: BIT (expression [, size])

Arguments: The argument "expression" represents the quantity to be converted to a bit string. The argument "size," when specified, must be a decimal integer constant giving the length of the result. If "size" is not specified, it is determined according to the type conversion rules given in Section F, "Problem Data Conversion." If "expression" is an array expression, "size" applies to each element of the array.

Result: The value returned by this function is "expression" converted to a bit string. The length of this bit string is determined by the integral value of "size," as described above.

BOOL String Built-in Function

Definition: BOOL produces a bit string whose bit representation is a result of a given boolean operation on two given bit strings.

Reference: BOOL (x,y,w)

Arguments: Arguments "x" and "y" are the two bit strings upon which the boolean operation specified by "w" is to be performed. If "x" and "y" are not bit strings, they are converted to bit strings. If "x" and "y" differ in length, the shorter string is extended with zeros on the right to match the length of the longer string.

Argument "w" represents the boolean operation. It is a bit string of length 4 and is defined as $n_1\ n_2\ n_3\ n$, where each n is either 0 or 1. There are 16 possible bit combinations and thus 16 possible boolean operations. As for "x" and "y," "w" is converted to a bit string (of length 4) before the function is invoked, if necessary.

If more than one argument is an array, the arrays must have identical bounds. Note that if only "w" is an array, the returned value is an array with identical bounds, each element of which is the result of the corresponding boolean operation performed on "x" and "y."

Result: The value returned by this function is a bit string, z, whose length is equal to the longer of "x" and "y." Each bit of z is determined by the boolean operation on the corresponding bits of "x" and "y" as follows: the ith bit of z is set to the value of n_1, n_2, n_3, or n depending on the combination of the ith bits of "x" and "y" as shown in the following boolean table:

xi	yi	zi
0	0	n_1
0	1	n_2
1	0	n_3
1	1	n

Example: In the following assignment statement, assume that U and ID have been declared as bit strings, XXX is the string '011'B, YYY is the string '110'B, and the boolean operator is '0110'B:

```
U=ID||BOOL (XXX, YYY, '0110');
```

Further, assume that Z represents the value returned to the point at which BOOL is invoked (that is, Z is a bit string of length 3 that is to be concatenated with ID), then the boolean table for this invocation of BOOL can be defined as:

XXXi	YYYi	Zi
0	0	0
0	1	1
1	0	1
1	1	0

which is interpreted as follows:

> Whenever the ith bits of XXX and YYY are 0 and 0, respectively, the ith bit of Z is 0; whenever the ith bits of XXX and YYY are 0 and 1, respectively, the ith bit of Z is 1, and so on.

Thus, since the first bits of XXX and YYY are 0 and 1, respectively, the first bit of Z is 1; since the second bits of XXX and YYY are 1 and 1, respectively, the second bit of Z is 0; and since the third bits of XXX and YYY are 1 and 0, respectively, the third bit of Z is 1. Therefore, the value returned to the point of invocation is the bit string '101'B.

CHAR String Built-in Function

Definition: CHAR converts a given value to a character string and returns the result to the point of invocation. This function allows the programmer to control the size of the result of a character-string conversion.

Reference: CHAR (expression[, size])

Arguments: The argument "expression" represents the quantity to be converted to a character string. The argument "size," when specified, must be a decimal integer constant giving the length of the result. If "size" is not specified, it is determined according to the type conversion on rules given in Section F, "Problem Data Conversion." If "expression" is an array expression, "size" applies to each element of the array.

Result: The value returned by this function is "expression" converted to a character string. The length of this character string is determined by "size," as described above.

HIGH String Built-in Function

Definition: HIGH forms a character string of a given length from the highest character in the collating sequence; that is, each character in the constructed string is the highest character in the collating sequence.

Reference: HIGH (i)

Argument: The argument, "i," must be a decimal integer constant specifying the length of the string that is to be formed.

Result: The value returned by this function is a character string of length "i;" each character in the string is the highest character in the collating sequence. For System/360 implementations, this character is stored as hexadecimal FF.

INDEX String Built-in Function

Definition: INDEX searches a specified string for a specified bit or character string configuration. If the configuration is found, the starting location of that configuration within the string is returned to the point of invocation.

Reference: INDEX (string, config)

Arguments: Two arguments must be specified. The first argument, "string," represents the string to be searched; the second argument, "config," represents the bit or character string configuration for which "string" is to be searched. If neither argument is a bit string, or if only one argument is a bit string, both arguments are converted to character strings. If both arguments are bit-string, no conversion is performed.

If both arguments are arrays, the arrays must have identical bounds.

Result: The value returned by this function is a binary integer of default precision. This binary integer is either:

1. The location in "string" at which "config" has been found. If more than one "config" exists in "string," the location of the first one found (in a left-to-right sense) will be returned.
2. The value 0, if "config" does not exist within "string" or if either of the arguments has a length of zero.

Example: If ASTRING is a character string containing:

```
'912NAMEA,1,FIRST,2,SECOND'
```

then the statement:

```
I = INDEX(ASTRING,'1,');
```

will return a binary value of ten to the point of invocation. This binary value represents the location of the configuration '1,' within ASTRING. However, if the statement had been:

```
I = INDEX(ASTRING,'1');
```

then a binary value of two would be returned to the point of invocation. This value is the location of the first '1' appearing within ASTRING.

LENGTH String Built-in Function

Definition: LENGTH finds the string length of a given value and returns it to the point of invocation.

Reference: LENGTH (string)

Argument: The argument, "string," represents a character string or a bit string whose length is to be found. The argument need not represent a string; if it does not, it is converted before the function is invoked to a character string (if the argument is DECIMAL) or a bit string (if the argument is BINARY).

Result: The value returned by this function is a fixed binary integer of default precision giving the current length of "string." If "string" is an array expression, an array of identical bounds is returned.

Example: If XYZ is a varying-length character string whose maximum length is 30, but whose current length is 25, then the statement:

```
I = LENGTH(SUBSTR(XYZ,4));
```

will assign a binary value of 22 to I.

LOW String Built-in Function

Definition: LOW forms a character string of specified length from the lowest character in the collating sequence; i.e., each character of the formed string will be the lowest character in the collating sequence.

Reference: LOW (i)

Argument: The argument, "i" must be an unsigned decimal integer constant specifying the length of the string being formed.

Result: The value returned by this function is a character string of length "i"; each character in the string is the lowest character in the collating sequence. For System/360 implementations, this character is stored as hexadecimal 00.

REPEAT String Built-in Function

Definition: REPEAT takes a given string value and forms a new string consisting of the given string value concatenated with itself a specified number of times.

Reference: REPEAT (string,i)

Arguments: The argument "string" represents a character string or bit string from which the new string will be formed. The argument need not represent a string, however; if an argument other than a bit string or character string is specified, it is converted, before the function is invoked, to a bit or character string.

The argument "i" must be an optionally signed decimal integer constant. It represents the number of times that "string" is to be concatenated with itself.

If "string" is an array expression, the value of "i" is applied to each element.

Result: The value returned by this function is "string" concatenated with itself "i" times. In other words, the returned value will be a string containing (i+1) occurrences of the value "string." If "i" is less than or equal to zero, the returned value is identical to the argument (i.e., the converted argument, if the original argument was not a string).

Example: If BSTR is a bit string containing '101'B, the statement

```
A = REPEAT(BSTR,6);
```

will cause the following value to be returned to the point of invocation:

```
'101101101101101101'B
```

STRING String Built-in Function

Definition: STRING concatenates all the elements in an aggregate variable into a single string element.

Reference: STRING(x)

Argument: The argument, "x," is an element, array, or structure variable, composed either entirely of character strings and/or numeric character data, or entirely of bit strings. If "x" is an element variable, the value returned is identical to the value of the variable.

Result: The value returned by this function is an element bit string or character string, the concatenation of all the elements in "x." If "x" contains one or more varying strings, the result is a varying string.

SUBSTR String Built-in Function

Definition: SUBSTR extracts a substring of user-defined length from a given string and returns the substring to the point of invocation. (SUBSTR can also be used as a pseudo-variable.)

Reference: SUBSTR (string,i[,j])

Arguments: The argument "string" represents the string from which a substring will be extracted. If this argument is not a string, it will be converted to a string. Argument "i" represents the starting point of the substring and "j" represents the length of the substring. Arguments "i" and "j" must be integers or expressions that can be converted to integers.

If more than one argument is an array, the arrays must have identical bounds.

Assuming that the length of "string" is k, arguments "i" and "j" must satisfy the following conditions:

1. j must be less than or equal to k and greater than or equal to 0.
2. i must be less than or equal to k and greater than or equal to 1.
3. The value of i+j-1 must be less than or equal to k.

Thus, the substring, as specified by "i" and "j" must lie within "string."

If "j" is not specified, it is assumed to be equal to the value of k-i+1. In other words, it is assumed to be the length of the remainder of "string," beginning at the ith position in "string."

When these conditions are not satisfied, the SUBSTR reference causes the STRINGRANGE condition to be raised, if it is enabled. If STRINGRANGE is not enabled, the result of the erroneous reference is undefined.

Result: The value returned by this function is a varying-length string whose current length is defined as follows:

1. If j=0, the returned value is the null string.
2. If j is greater than 0, the returned value is that substring beginning at the ith character or bit of the first argument and extending j characters or bits.
3. If j is not specified, the returned value is that substring beginning at the ith character or bit and extending to the end of "string."

Example: If AAA is a character string of length 30, the statement:

```
ITEM = SUBSTR(AAA, 7, 14);
```

will cause a 14-character substring to be extracted from AAA, starting at the seventh character of AAA. The extracted string is then returned to the point of invocation, after which it is assigned to ITEM.

UNSPEC String Built-in Function

Definition: UNSPEC returns a bit string that is the internal coded representation of a given value. (UNSPEC can also be used as a pseudo-variable.)

Reference: UNSPEC (x)

Argument: The argument, "x," may be an arithmetic or string constant, variable, or expression, or an area, pointer, or offset variable, whose internal coded representation is to be found.

Result: The value returned by this function is the internal coded representation of "x." This representation is in bit-string form. The length of this string depends upon the attributes of "x," and is defined by System/360 implementations as follows:

1. If "x" is FIXED BINARY of precision (p,q), the length is 32.
2. If "x" is FIXED DECIMAL of precision (p,q), the length is defined as 8*FLOOR ((p+2)/2).
3. If "x" is FLOAT BINARY of precision p, the length is
   a. 32, if p is less than or equal to 21.
   b. 64, if p is greater than 21.
4. If "x" is FLOAT DECIMAL of precision p, the length is
   a. 32, if p is less than or equal to 6.
   b. 64, if p is greater than 6.
5. If "x" is a character-string of length n or a numeric character data item whose character-string value is of length n, the length is 8*n.
6. If "x" is a bit-string of length n, the length is n.
7. If "x" is complex, the length is twice that of the corresponding real value.
8. If "x" is a pointer, the length is 32.
9. If "x" is an offset, the length is 32.
10. If "x" is an area of size n, the length is 8*(n+16).

ARITHMETIC BUILT-IN FUNCTIONS

All values returned by arithmetic built-in functions are in coded arithmetic form. The arguments of these functions should also be in that form. If an argument is not coded arithmetic, then, before the function is invoked, it is converted to coded arithmetic according to the rules stated in Section F, "Problem Data Conversion." Note, therefore, that in the function descriptions below, a reference to an argument always means the converted argument, if conversion was necessary.

In some function descriptions, the phrase "converted to the highest characteristics" is used; this means that the rules for mixed characteristics are followed (these rules are stated in the section "Data Conversion in Arithmetic Operations" in Part I, Chapter 4, "Expressions.")

In general, an argument of an arithmetic built-in function may be an element or array expression. If an argument is an array, the value returned by the built-in function is an array of the same dimension and bounds as the argument (the function having been performed once for each element of the array). Thus, for example, if an array argument is passed to the absolute value function ABS, the returned value is an array, each element of which is the absolute value of the corresponding element in the argument array.

Unless it is specifically stated otherwise:

1. The mode of an argument may be real or complex.
2. The base, scale, mode, and precision of the returned value are determined according to the rules for the conversion of expression operands as given in Section F, "Problem Data Conversion."

In many of these built-in functions, the symbol N is used. This symbol represents the maximum precision that a value may have. It is defined, for System/360 implementations, as follows:

N is 15 for FIXED DECIMAL values

16 for FLOAT DECIMAL values

31 for FIXED BINARY values

53 for FLOAT BINARY values

The precision of decimal and binary floating-point items should be noted when

using the built-in functions ADD, BINARY, DECIMAL, DIVIDE, FLOAT, MULTIPLY, and PRECISION. For decimal floating-point items: if the precision is less than or equal to (6), short floating-point form is used; if the precision is greater than (6), long floating-point form is used. For binary floating-point items: if the precision is less than or equal to (21), short floating-point form is used; if the precision is greater than (21), long floating-point form is used.

ABS Arithmetic Built-in Function

Definition: ABS finds the absolute value of a given quantity and returns it to the point of invocation.

Reference: ABS (x)

Argument: The quantity whose absolute value is to be found is given by "x."

Result: The value returned by this function is the absolute value of "x." If "x" is real, the result is the positive value of "x"; if "x" is complex, the result is the positive square root of the sum of squares of the real and imaginary parts of "x." The mode of the result is real, while the base, scale, and precision are the same as those of "x," with one exception: if "x" is a complex fixed-point value of precision (p,q), the precision of the result is:

```
(MIN(N,p+1),q)
```

ADD Arithmetic Built-in Function

Definition: ADD finds the sum of two given values and returns it to the point of invocation. This function allows the programmer to control the precision of the result of an add operation.

Reference: ADD (x,y,p[,q])

Arguments: Arguments "x" and "y" represent the values to be added. Arguments "p" and "q" must be decimal integer constants specifying the precision of the result; "q" may be signed. If the scale of the result is fixed-point, both "p" and "q" must be specified; if the scale of the result is floating-point, only "p" must be specified. In either case, "p" must not exceed N.

Result: The value returned by this function is the sum of "x" and "y." The precision of the result is determined by "p" and "q"; this precision is maintained throughout the execution of the function.

BINARY Arithmetic Built-in Function

Definition: BINARY converts a given value to binary base and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a binary conversion.

Reference: BINARY (x[,p[,q]])

Arguments: The first argument, "x," represents the value to be converted to binary base. Arguments "p" and "q," when specified, must be decimal integer constants giving the precision of the binary result; "q" may be signed. The precision of a fixed-point result is (p,q); the precision of a floating-point result is (p). If both "p" and "q" are omitted, the precision of the result is determined according to the rules given for base conversion in Section F, "Problem Data Conversion." Note that "q" must be omitted for floating-point arguments.

Result: The value returned by this function is the binary equivalent of "x." The scale and mode of this value are the same as those of "x." The precision is given by "p" and "q."

CEIL Arithmetic Built-in Function

Definition: CEIL determines the smallest integer that is greater than or equal to a given real value and returns that integer to the point of invocation.

Reference: CEIL (x)

Argument: The argument, "x," must not be complex.

Result: The value returned by this function is the smallest integer that is greater than or equal to "x." The base, scale, mode, and precision are the same as those of "x," with one exception: if "x" is a fixed-point value of precision (p,q), the precision of the result is defined as:

(MIN(N,MAX(p-q+1,1)),0)

COMPLEX Arithmetic Built-in Function

Definition: COMPLEX forms a complex number from two given real values and returns it to the point of invocation. (COMPLEX can also be used as a pseudo-variable.)

Reference: COMPLEX (x,y)

Arguments: Arguments "x" and "y" must both be real; "x" represents the real part of the complex number to be formed and "y" represents the imaginary part.

Result: The value returned by this function is the complex number that has been formed from "x" and "y."

CONJG Arithmetic Built-in Function

Definition: CONJG finds the conjugate of a complex value and returns it to the point of invocation. (The conjugate of a complex number is the complex number with the sign of the imaginary part reversed.)

Reference: CONJG (x)

Argument: The argument, "x," is the value whose conjugate is to be found; it must be complex.

Result: The value returned by this function is the conjugate of "x." The base, scale, mode, and precision of the conjugate are the same as those of the argument.

DECIMAL Arithmetic Built-in Function

Definition: DECIMAL converts a given value to decimal base and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a decimal conversion.

Reference: DECIMAL (x[,p[,q]])

Arguments: The first argument, "x," represents the value to be converted to decimal base. Arguments "p" and "q," when specified, must be decimal integer constants giving the precision of the decimal result; "q" may be signed. The precision of a fixed-point result is (p,q); the precision of a floating-point result is (p). If both "p" and "q" are omitted, however, the precision of the result is determined according to the rules given for base conversion in Section F, "Problem Data Conversion." Note that "q" must be omitted for floating-point arguments.

Result: The value returned by this function is the decimal equivalent of the argument "x"; its precision is given by "p" and "q."

DIVIDE Arithmetic Built-in Function

Definition: DIVIDE divides a given value by another given value and returns the quotient to the point of invocation. This function allows the programmer to control the precision of the result of a divide operation.

Reference: DIVIDE (x,y,p[,q])

Arguments: The argument, "x," is the dividend and argument "y" is the divisor. Arguments "p" and "q" ("q" is optional and may be signed) must be decimal integer constants specifying the precision of the result. If the result is a fixed-point value, "p" and "q" must both be specified; if the result is a floating-point value, only "p" must be specified. In either case, "p" must not exceed N.

Result: The value returned by this function is the quotient resulting from the division of "x" by "y." The precision of the result is determined by "p" and "q" as described above; this precision is maintained throughout the execution of the function.

FIXED Arithmetic Built-in Function

Definition: FIXED converts a given value to fixed-point scale and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a fixed-point conversion.

Reference: FIXED (x[,p[,q]])

Argument: The first argument, "x," represents the value to be converted to fixed-point scale. Arguments "p" and "q," when specified, must be decimal integer constants ("q" can be signed) giving the precision of the result, (p,q). For System/360 implementations, if "p" and "q" are omitted, "p" is assumed to be 15 for binary "x" and 5 for decimal "x"; "q" is assumed to be 0.

Result: The value returned by this function is the fixed-point equivalent of the argument "x"; its precision is (p,q).

FLOAT Arithmetic Built-in Function

Definition: FLOAT converts a given value to floating-point scale and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a floating-point conversion.

Reference: FLOAT (x[,p])

Arguments: The first argument, "x," represents the value to be converted to floating-point scale. The second argument, "p," when specified, must be a decimal integer constant giving the precision of the result. For System/360 implementations, if "p" is omitted, it is assumed to be 21 for binary "x" and 6 for decimal "x."

Result: The value returned by this function is the floating-point equivalent of "x"; its precision is "p."

FLOOR Arithmetic Built-in Function

Definition: FLOOR determines the largest integer that does not exceed a given value and returns that integer to the point of invocation.

Reference: FLOOR (x)

Argument: The argument, "x," must not be complex.

Result: The value returned by this function is the largest integer that does not exceed "x." The base, scale, mode, and precision of this value are the same as those of "x," with one exception: if "x" is a fixed-point value of precision (p,q), the precision of the result is:

```
(MIN(N,MAX(p-q+1,1)),0)
```

IMAG Arithmetic Built-in Function

Definition: IMAG extracts the imaginary part of a given complex number and returns it to the point of invocation. (IMAG can also be used as a pseudo-variable.)

Reference: IMAG (x)

Argument: The argument, "x," is the complex value whose imaginary part is to be extracted.

Result: The value returned by this function is the imaginary part of "x." The base, scale, and precision of the imaginary part are unchanged. The mode of the returned value is real.

MAX Arithmetic Built-in Function

Definition: MAX extracts the highest-valued expression from a given set of two or more expressions and returns that value to the point of invocation.

Reference: MAX (x_1,x_2,...,x_n)

Arguments: Two or more arguments must be given. The arguments must not be complex.

Result: The value returned by MAX is the value of the maximum-valued argument. The returned value is converted to conform to the highest characteristics of all the arguments that were specified. If the arguments are fixed-point values and have precisions:

(p_1,q_1), (p_2q_2),..., (p_n,q_n)

then the precision of the result is as follows:

```
(MIN(N,MAX(p1-q1,...,pn-qn)+
    MAX(q1,...,qn)),MAX(q1,...qn))
```

MIN Arithmetic Built-in Function

Definition: MIN extracts the lowest-valued expression from a given set of two or more expressions and returns that value to the point of invocation.

Reference: MIN (x_1,x_2,...,x_n)

Arguments: Two or more arguments must be given. The arguments must not be complex.

Result: The value returned by MIN is the value of the lowest-valued argument. The returned value is converted to conform to the highest characteristics of all the arguments that were specified. If the arguments are fixed-point values and have precisions:

(p_1,q_1), (p_2,q_2),...,(p_n,q_n)

then the precision of the result is as follows:

```
(MIN(N,MAX(p1-q1,...,pn-qn)+
    MAX(q1,...qn)),MAX(q1,...,qn))
```

MOD Arithmetic Built-in Function

Definition: MOD extracts the remainder resulting from the division of one real quantity by another and returns it to the point of invocation.

Reference: MOD (x_1,x_2)

Arguments: Two arguments must be given. They must not be complex. Before the function is invoked, the base and scale of each argument are converted according to the rules for the conversion of expression operands, as given in Section F, "Problem Data Conversion."

Result: The value returned by MOD is the positive remainder resulting from the division of "x_1" by "x_2." If the result is in floating-point scale, its precision is the higher of the precisions of the arguments; if the result is in fixed-point scale, its precision is defined as follows:

```
(MIN(N,p2-q2+MAX(q1,q2)),MAX(q1,q2))
```

where (p_1,q_1) and (p_2,q_2) are the precision of "x_1" and "x_2," respectively.

MULTIPLY Arithmetic Built-in Function

Definition: MULTIPLY finds the product of two given values and returns it to the point of invocation. This function allows the programmer to control the precision of the result of a multiplication operation.

Reference: MULTIPLY (x_1,x_2,p[,q])

Arguments: Arguments "x_1" and "x_2" represent the values to be multiplied. Arguments "p" and "q" ("q" is optional and may be signed) are decimal integer constants specifying the precision of the result. If the result is a fixed-point value, "p" and "q" must both be specified; if the result is a floating-point value, only "p" must be specified. In either case, "p" must not exceed N.

Result: The value returned by this function is the product of "x_1" and "x_2." The precision of the result is as specified; this precision is maintained throughout the execution of the function.

PRECISION Arithmetic Built-in Function

Definition: PRECISION converts a given value to a specified precision and returns the converted value to the point of invocation.

Reference: PRECISION (x,p[,q])

Arguments: The first argument, "x," represents the value to be converted to the specified precision. Arguments "p" and "q" ("q" is optional and may be signed) are decimal integer constants specifying the precision of the result. If "x" is a fixed-point value, "p" and "q" must be specified; if "x" is a floating-point value, only "p" must be specified.

Result: The value returned by this function is the value of "x" converted to the specified precision. The base, scale, and mode of the returned value are the same as those of "x."

REAL Arithmetic Built-in Function

Definition: REAL extracts the real part of a given complex value and returns it to the point of invocation. (REAL can also be used as a pseudo-variable.)

Reference: REAL (x)

Argument: The argument, "x," must be a complex expression.

Result: The value returned by this function is the real part of the complex value represented by "x." The base, scale, and precision of the real part are unchanged.

ROUND Arithmetic Built-in Function

Definition: ROUND rounds a given value at a specified digit and returns the rounded value to the point of invocation.

Reference: ROUND (expression,n)

Arguments: The first argument, "expression," is an element or array representing the value (or values, in the case of an array expression) to be rounded; the second argument, "n," is a signed or unsigned decimal integer constant specifying the digit at which the value of "expression" is to be rounded. If "n" is positive, rounding occurs at the nth digit to the right of the decimal (or binary) point in the value of "expression"; if "n" is zero, rounding occurs at the first digit to the left of the decimal (or binary) point in the value of "expression"; if "n" is negative, rounding occurs at the nth+1 digit to the left of the decimal (or binary) point in the value of "expression." Note that the decimal (or binary) point is assumed to be at the left for floating-point values.

Result: For fixed-point values, ROUND returns the value of "expression" rounded at the nth digit to the right of the decimal (or binary) point for positive "n", or at the first digit to the left of the decimal (or binary) point for zero "n", or at the nth+1 digit to the left of the decimal (or binary) point for negative "n." Thus, when "n" is negative, the returned value is an integer.

If "expression" is a floating-point expression, the second argument is ignored, and the rightmost bit in the internal floating-point representation of the expression's value is set to 1 if it is 0. If the rightmost bit is 1, it is left unchanged.

If "expression" is a string, the returned value is the same string unmodified.

The base, scale, and mode of the returned value are those of the value of "expression".

The precision of the returned value for floating-point expressions is that of "expression"; the precision of the returned value for fixed-point expressions is (MIN(p+1,N),q). The extra digit (p+1) of the returned value for fixed-point expressions is to allow for those cases in which rounding would give a result that could not be expressed in "p" digits, for example, ROUND(9.999,2) would result in 10.000.

Note that the rounding of a negative quantity results in the rounding of the absolute value of that quantity.

SIGN Arithmetic Built-in Function

Definition: SIGN determines whether a value is positive, negative, or zero, and it returns an indication to the point of invocation.

Reference: SIGN (x)

Argument: The argument, "x," must not be complex.

Result: This function returns a real fixed-point binary value of default precision according to the following rules:

1. If the argument is greater than 0, the returned value is 1.
2. If the argument is equal to zero, the returned value is 0.
3. If the argument is less than zero, the returned value is -1.

TRUNC Arithmetic Built-in Function

Definition: TRUNC truncates a given value to an integer as follows: first, it determines whether a given value is positive, negative, or equal to zero. If the value is negative, TRUNC returns the smallest integer that is greater than that value; if the value is positive or equal to zero, TRUNC returns the largest integer that does not exceed that value.

Reference: TRUNC (x)

Argument: The argument, "x," must not be complex.

Result: If "x" is less than zero, the value returned by TRUNC is CEIL(x). If "x" is greater than or equal to zero, the value returned by TRUNC is FLOOR(x). In either case, the base, scale, and mode of the result are the same as those of "x." If "x" is a floating-point value, the precision remains the same. If "x" is a fixed-point value of precision (p,q), the precision of the result is:

(MIN(N,MAX(p-q+1,1)),0)

MATHEMATICAL BUILT-IN FUNCTIONS

All arguments to the mathematical built-in functions should be in coded arithmetic form and in floating-point scale. Any argument that does not conform to this rule is converted to coded arithmetic and floating-point before the function is invoked, according to the rules stated in Section F, "Problem Data Conversion." Note, therefore, that in the function descriptions below, a reference to an argument always means the converted argument, if conversion was necessary.

In general, an argument to a mathematical built-in function may be an element or array expression. If an argument is an array, the value returned by the built-in function is an array of the same dimension and bounds as the argument (the function having been performed once for each element of the array). Thus, for example, an array to the cosine function COS results in an array, each element of which is the cosine of the corresponding element in the argument array.

Unless it is specifically stated otherwise, an argument may be real or complex. Figures G-1 and G-2 at the end of this section provide a quick reference for those mathematical functions that accept either real or complex arguments and those that accept only real arguments.

All of the mathematical built-in functions return coded arithmetic floating-point values. The mode, base, and precision of these values are always the same as those of the arguments.

ATAN Mathematical Built-in Function

Definition: ATAN finds the arctangent of a given value and returns the result expressed in radians, to the point of invocation.

Reference: ATAN (x[,y])

Arguments: The argument, "x," must always be specified; the argument "y" is optional. If "y" is omitted, "x" represents the value whose arctangent is to be found; in such a case, "x" may be real or complex, but if it is complex, it must not be equal to ±1i.

If "y" is specified, then the value whose arctangent is to be found is taken to be the expression x/y. In this case, both "x" and "y" must be real, and both cannot be equal to 0 at the same time.

Result: When "x" alone is specified, the value returned by ATAN depends on the mode of "x." If "x" is real, the returned value is the arctangent of "x," expressed in radians, where:

-pi/2<ATAN(x)<pi/2

If "x" is complex, the arctangent function is multiple-valued, and hence only the principal value can be returned. The principal value of ATAN for a complex argument "x" is defined as follows:

```
-i*ATANH(i*x)
```

If both "x" and "y" are specified, the possible values returned by this function are defined as follows:

1. If y>0, the value is arctangent (x/y) in radians.
2. If x>0 and y=0, the value is (pi/2) radians.
3. If x≥0 and y<0, the value is (pi+ arctangent (x/y)) radians.
4. If x<0 and y=0, the value is (-pi/2) radians.
5. If x<0 and y<0, the value is (-pi+ arctangent (x/y)) radians.

ATAND Mathematical Built-in Function

Definition: ATAND finds the arctangent of a given real value and returns the result, expressed in degrees, to the point of invocation.

Reference: ATAND (x[,y])

Arguments: Arguments "x" and "y" ("y" may be omitted) must be real values. If "y" is omitted, "x" represents the value whose arctangent is to be found. If "y" is specified, the value whose arctangent is to be found is represented by the expression x/y; in this case, both "x" and "y" cannot be equal to 0 at the same time.

Result: If "y" is not specified, the value returned by this function is simply the arctangent of "x," expressed in degrees, where:

```
-90<ATAND(x)<90
```

If "y" is specified, the value returned by this function is ATAN (x,y), except that the value is expressed in degrees and not in radians (see "ATAN Mathematical Built-in Function" in this section); that is, the returned value is defined as:

```
ATAND(x,y) = (180/pi)*ATAN(x,y)
```

AT NH Mathematical Built-in Function

Definition: ATANH finds the inverse hyperbolic tangent of a given value and returns the result to the point of invocation.

Reference: ATANH (x)

rgument: The value whose inverse hyperbolic tangent is to be found is represented by "x." If "x" is real, the absolute value of "x" must not be greater than or equal to 1; that is, for real "x," it is an error if ABS(x)≥1. If "x" is complex, it must not be equal to ±1.

Result: If "x" is real, the value returned by this function is the inverse hyperbolic tangent of "x." For complex "x," the inverse hyperbolic tangent is defined as follows:

```
(LOG((1+x)/(1-x)))/2
```

COS Mathematical Built-in Function

Definition: COS finds the cosine of a given value, which is expressed in radians, and returns the result to the point of invocation.

Reference: COS (x)

Argument: The value whose cosine is to be found is given by "x"; this value can be real or complex and must be expressed in radians.

Result: The value returned by this function is the cosine of "x." For complex argument "x," the cosine of "x" is defined below, where $x = y_1+iy_2$:

$$\cos(x)=\cos(y_1)*\cosh(y_2)-i*\sin(y_1)*\sinh(y_2)$$

COSD Mathematical Built-in Function

Definition: COSD finds the cosine of a given real value, which is expressed in degrees, and returns the result to the point of invocation.

Reference: COSD (x)

Argument: The value whose cosine is to be found is given by "x"; this value must be real and must be expressed in degrees.

Result: The value returned by this function is the cosine of "x."

COSH Mathematical Built-in Function

Definition: COSH finds the hyperbolic cosine of a given value and returns the result to the point of invocation.

Reference: COSH (x)

Argument: The value whose hyperbolic cosine is to be found is given by "x."

Result: The value returned by this function is the hyperbolic cosine of "x." For complex argument "x," the hyperbolic cosine of "x" is defined below, where $x = y_1+iy_2$:

$$\cosh(x)=\cosh(y_1)*\cos(y_2)+i*\sinh(y_1)*\sin(y_2)$$

ERF Mathematical Built-in Function

Definition: ERF finds the error function of a given real value and returns it to the point of invocation.

Reference: ERF (x)

Argument: The value for which the error function is to be found is represented by "x"; this value must be real.

Result: The value returned by this function is defined as follows:

$$ERF(x)= \frac{2}{\sqrt{\Pi}}\int_0^x e^{-t^2}dt$$

ERFC Mathematical Built-in Function

Definition: ERFC finds the complement of the error function (ERF) for a given real value and returns the result to the point of invocation.

Reference: ERFC (x)

Argument: The argument, "x," represents the value whose error function complement is to be found; "x" must be real.

Result: The value returned by this function is defined as follows:

$$ERFC(x) = 1-ERF(x)$$

EXP Mathematical Built-in Function

Definition: EXP raises e (the base of the natural logarithm system) to a given power and returns the result to the point of invocation.

Reference: EXP (x)

Argument: The argument, "x," specifies the power to which e is to be raised.

Result: The value returned by this function is e raised to the power of "x."

LOG Mathematical Built-in Function

Definition: LOG finds the natural logarithm (i.e., base e) of a given value and returns it to the point of invocation.

Reference: LOG (x)

Argument: The argument, "x," is the value whose natural logarithm is to be found. If "x" is real, it must not be less than or equal to 0; if "x" is complex, it must not be equal to 0+0i.

Result: The value returned by this function is the natural logarithm of "x." However, if "x" is complex, the function is multiple-valued; hence, only the principal value can be returned. The principal value has the form w = u±i*v, where v lies in the range:

$$-pi<v\leq pi$$

LOG10 Mathematical Built-in Function

Definition: LOG10 finds the common logarithm (i.e., base 10) of a given real value and returns it to the point of invocation.

Reference: LOG10 (x)

Argument: The argument, "x," represents the value whose common logarithm is to be found; this value must be real and it must not be less than or equal to 0.

Result: The value returned by this function is the common logarithm of "x."

LOG2 Mathematical Built-in Function

Definition: LOG2 finds the binary (i.e., base 2) logarithm of a given real value and returns it to the point of invocation.

Reference: LOG2 (x)

Argument: The argument, "x," is the value whose binary logarithm is to be found; it must be real and it must not be less than or equal to 0.

Result: The value returned to this function is the binary logarithm of "x."

SIN Mathematical Built-in Function

Definition: SIN finds the sine of a given value, which is expressed in radians, and returns it to the point of invocation.

Reference: SIN (x)

Argument: The argument, "x," is the value whose sine is to be found; it must be expressed in radians.

Result: The value returned by this function is the sine of "x." For complex argument "x," the sine of "x" is defined below, where $x = y_1+i*y_2$:

$$\sin(x)=\sin(y_1)*\cosh(y_2)+i*\cos(y_1)*\sinh(y_2)$$

SIND Mathematical Built-in Function

Definition: SIND finds the sine of a given real value, which is expressed in degrees, and returns the result to the point of invocation.

Reference: SIND (x)

Argument: The argument, "x," is the value whose sine is to be found; "x" must be real and it must be expressed in degrees.

Result: The value returned by this function is the sine of "x."

SINH Mathematical Built-in Function

Definition: SINH finds the hyperbolic sine of a given value and returns the result to the point of invocation.

Reference: SINH (x)

Argument: The argument, "x," is the value whose hyperbolic sine is to be found.

Result: The value returned by this function is the hyperbolic sine of "x." For complex argument "x," the hyperbolic sine of "x" is defined below, where $x = y_1+i*y_2$:

$$\sinh(x)=\sinh(y_1)*\cos(y_2)+i*\cosh(y_1)*\sin(y_2)$$

SQRT Mathematical Built-in Function

Definition: SQRT finds the square root of a given value and returns it to the point of invocation.

Reference: SQRT (x)

Argument: The argument, "x," is the value whose square root is to be found. If "x" is real, it must not be less than 0.

Result: If "x" is real, the value returned by this function is the positive square root of "x." If "x" is complex, the square root function is multiple-valued; hence, only the principal value can be returned to the user. The principal value has the form $w = u\pm i*v$, where either $u>0$, or $u=0$ and $v\geq 0$.

TAN Mathematical Built-in Function

Definition: TAN finds the tangent of a given value, which is expressed in radians, and returns it to the point of invocation.

Reference: TAN (x)

Argument: The argument, "x," represents the value whose tangent is to be found; "x" must be expressed in radians.

Result: The value returned by this function is the tangent of "x."

TAND Mathematical Built-in Functions

Definition: TAND finds the tangent of a given real value which is expressed in degrees, and returns the result to the point of invocation.

Reference: TAND (x)

Argument: The argument, "x," represents the value whose tangent is to be found; "x" must be expressed in degrees.

Result: The value returned by this function is the tangent of "x."

TANH Mathematical Built-in Function

Definition: TANH finds the hyperbolic tangent of a given value and returns the result to the point of invocation.

Reference: TANH (x)

Argument: The argument, "x," represents the value whose hyperbolic tangent is to be found.

Result: The value returned by this function is the hyperbolic tangent of "x."

Summary of Mathematical Functions

Figure G-1 summarizes the mathematical built-in functions. In using it, the reader should be aware of the following:

1. A complex argument, "x," is defined as $x = y_1+i*y_2$.
2. The value returned by each function is always in floating-point.
3. The error conditions are those defined by the PL/I Language. Additional error conditions detected by the F-Compiler are described in the publication IBM System/360 Operating System, PL/I Subroutine Library, Computational Subroutines, Form C28-6590.
4. All arguments must be coded arithmetic and floating-point scale, or such that they can be converted to coded arithmetic and floating-point.

ARRAY MANIPULATION BUILT-IN FUNCTIONS

The built-in functions described here may be used for the manipulation of arrays. All of these functions require array arguments (which may be expressions) and return single element values. Note that since these functions return element values, a function reference to any of them is considered an element expression.

Function Reference	Argument Type	Value Returned	Error Conditions
ATAN(x)	real	arctan(x) in radians -(pi/2)<ATAN(x)<pi/2	-
	complex	-i*ATANH(i*x)	x = ±1i
ATAN(x,y)	both real	see function description	error if x=0 and y=0
ATAND(x)	real	arctan(x) in degrees -90<ATAND(x)<90	-
ATAND(x,y)	both real	see function description	error if x=0 and y=0
ATANH(x)	real	arctanh(x)	ABS(x)≥1
	complex	(LOG((1+x)/(1-x)))/2	x = ±1
COS(x) x in radians	real	cosine(x)	-
	complex	$\cos(y_1)*\cosh(y_2)$ $-i*\sin(y_1)*\sinh(y_2)$	-
COSD(x) x in degrees	real	cosine(x)	-
COSH(x)	real	cosh(x)	-
	complex	$\cosh(y_1)*\cos(y_2)$ $+i*\sinh(y_1)*\sin(y_2)$	-
ERF(x)	real	$\frac{2}{\sqrt{\Pi}}\int_0^x e^{-t^2}dt$	-

Figure G-1. Mathematical Built-In Functions

Function Reference	Argument Type	Value Returned	Error Conditions
ERFC(x)	real	1 - ERF(x)	-
EXP(x)	real	e^x	-
	complex	e^x	-
LOG(x)	real	log (x)	x≤0
	complex	log (x) = w where w = u±i*v and -pi<v≤pi	x=0
LOG10(x)	real	$\log_{10}$ (x)	x≤0
LOG2(x)	real	$\log_2(x)$	x≤0
SIN(x) x in radians	real	sin(x)	-
	complex	$\sin(y_1)*\cosh(y_2)$ $+i*\cos(y_1)*\sinh(y_2)$	-
SIND(x) x in degrees	real	sin(x)	-
SINH(x)	real	sinh(x)	-
	complex	$\sinh(y_1)*\cos(y_2)$ $+i*\cosh(y_1)*\sin(y_2)$	-
SQRT(x)	real	$\sqrt{x}$	x<0
	complex	$\sqrt{x}$ = w where w = u±i*v and either u>0, or u=0 and v≥0	-
TAN(x) x in radians	real	tangent(x)	-
	complex	tangent(x)	-
TAND(x) x in degrees	real	tangent(x)	-
TANH(x)	real	tanh(x)	-
	complex	tanh(x)	-

•Figure G-1. Mathematical Built-In Functions (continued)

•

ALL Array Manipulation Function

Definition: ALL tests all bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not the corresponding bits of given array elements are all ones.

Reference: ALL (x)

Argument: The argument, "x," is an array of bit strings. If the elements are not bit strings, they are converted to bit strings.

Result: The value returned by this function is a bit string whose length is equal to the length of the longest element in "x" and whose bit values are determined by the following rule:

If the ith bits of all of the elements in "x" exist and are 1, then the ith bit of the result is 1; otherwise, the ith bit of the result is 0.

ANY Array Manipulation Function

Definition: ANY tests the bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not at least one of the corresponding bits of the given array elements is set to 1.

Reference: ANY (x)

Argument: The argument, "x," is an array of bit strings. If the elements are not bit strings, they are converted to bit strings.

Result: The value returned by this function is a bit string whose length is equal to the length of the longest element in "x" and whose bit values are determined by the following rule:

If the ith bit of any element in "x" exists and is 1, then the ith bit of the result is 1; otherwise, the ith bit of the result is 0.

DIM Array Manipulation Function

Definition: DIM finds the current extent for a specified dimension of a given array and returns it to the point of invocation.

Reference: DIM (x,n)

Arguments: The argument "x" is the array to be investigated; "n" is the dimension of "x," the extent of which is to be found. If "n" is not a binary integer, it is converted to a binary integer of default precision. It is an error if "x" has less than "n" dimensions, if "n" is less than or equal to 0, or if "x" is not currently allocated.

Result: The value returned by this function is a binary integer of default precision, giving the current extent of the nth dimension of "x."

HBOUND Array Manipulation Function

Definition: HBOUND finds the current upper bound for a specified dimension of a given array and returns it to the point of invocation.

Reference: HBOUND (x,n)

Arguments: The argument "x" is the array to be investigated; "n" is the dimension of "x" for which the upper bound is to be found. If "n" is not a binary integer, it is converted to a binary integer of default precision. It is an error if "x" has less than "n" dimensions, if "n" is less than or equal to 0, or if "x" is not currently allocated.

Result: The value returned by this function is a binary integer of default precision giving the current upper bound for the nth dimension of "x."

LBOUND Array Manipulation Function

Definition: LBOUND finds the current lower bound for a specified dimension of a given array and returns it to the point of invocation.

Reference: LBOUND (x,n)

Arguments: The argument "x" is the array to be investigated; "n" is the dimension of "x" for which the lower bound is to be found. If "n" is not a binary integer, it is converted to a binary integer of default precision. It is an error if "x" has less than "n" dimensions, if "n" is less than or equal to 0, or if "x" is not currently allocated.

Result: The value returned by this function is a binary integer of default precision giving the current lower bound of the nth dimension of "x."

POLY Array Manipulation Function

Definition: POLY forms a polynomial from two given arguments and returns the result of the evaluation of that polynomial to the point of invocation.

Reference: POLY (a,x)

Arguments: Arguments "a" and "x" must be one-dimension arrays (vectors). They are defined as follows:

a(m:n)

x(p:q)

where (m:n) and (p:q) represent the bounds of "a" and "x," respectively.

Result: The value returned by this function is defined as:

$$a(m) + \sum_{j=1}^{n-m} \left(a(m+j) * \prod_{i=0}^{j-1} x(p+i) \right)$$

If (q-p)<(n-m-1), then x(p+i)=x(q) for (p+i)>q. If m=n, then the result is a(m).

If "x" is an element variable, it is interpreted as an array of one element, i.e., x(1), and the result is then:

$$\sum_{j=0}^{n-m} a(m+j)*x**j$$

PROD Array Manipulation Function

Definition: PROD finds the product of all of the elements of a given array and returns that product to the point of invocation.

Reference: PROD (x)

Argument: The argument, "x," should be an array of coded arithmetic floating-point elements. If it is not, each element is converted to coded arithmetic and floating-point before being multiplied with the previous product.

Result: The value returned by this function is the product of all of the elements in "x." The scale of the result is floating-point, while the base, mode, and precision are those of the converted elements of "x."

SUM Array Manipulation Function

Definition: SUM finds the sum of all of the elements of a given array and returns that sum to the point of invocation.

Reference: SUM (x)

Argument: The argument, "x," should be an array of coded arithmetic floating-point elements. If it is not, each element is converted to coded arithmetic and floating-point before being summed with the previous total.

Result: The value returned by this function is the sum of all of the elements in "x." The scale of the result is floating-point, while the base, mode, and precision are those of the converted elements of the argument.

CONDITION BUILT-IN FUNCTIONS

The condition built-in functions allow the PL/I programmer to investigate interrupts that arise from enabled ON-conditions. None of these functions requires arguments. Each condition built-in function returns the value described only when executed in an on-unit (or a block activated directly or indirectly by an on-unit) that is entered as a result of an interrupt caused by one of the ON-conditions for which the function can be used. Such an on-unit can be one specific to the condition, or it can be for the ERROR or FINISH condition when these conditions are raised as standard system action. If a condition built-in function is used out of context, the value returned is as described for each function.

The on-units in which each function can be used are given in the function definition.

DATAFIELD Condition Built-in Function

Definition: Whenever the NAME condition is raised, DATAFIELD may be used to extract the contents of the data field that caused the condition to be raised. It can be used only in an on-unit for the NAME condition or in an ERROR or FINISH condition raised as a result of standard system action for the NAME condition.

Reference: DATAFIELD

Result: The value returned by this function is a varying-length character string giving the contents of the data field that caused the NAME condition to be raised. The maximum length of this string is defined by the F Compiler as 255. If DATAFIELD is used out of context, a null string is returned.

ONCHAR Condition Built-in Function

Definition: Whenever the CONVERSION condition is raised, ONCHAR may be used to extract the character the caused that con-

dition to be raised. It can be used only in an on-unit for the CONVERSION condition or in an on-unit for an ERROR or FINISH condition raised as standard system action for the CONVERSION condition. (ONCHAR can also be used as a pseudo-variable.)

Reference: ONCHAR

Result: The value returned by this function is a character string of length 1, containing the character that caused the CONVERSION condition to be raised. This character can be modified in the on-unit by the use of the ONCHAR or ONSOURCE pseudo-variables. If ONCHAR is used out of context, a blank is returned.

ONCODE Condition Built-in Function

Definition: ONCODE can be used in any on-unit to determine the type of interrupt that caused the on-unit to become active.

Reference: ONCODE

Result: ONCODE returns a binary integer of default precision. This "code" defines the type of interrupt that caused the entry into the currently active on-unit. The codes for the F Compiler are given in section H, "ON-Conditions." If ONCODE is used out of context, a value of 4 is returned.

ONCOUNT Condition Built-In Function

Definition: ONCOUNT can be used in any on-unit entered due to the abnormal completion of an input/output event to determine the number of interrupts (including the current one) that remain to be handled when a multiple interrupt has resulted from that abnormal completion. (Multiple interrupts are discussed in Section H, "ON-Conditions.")

Reference: ONCOUNT

Result: ONCOUNT returns a binary value of default precision. If ONCOUNT is used in an on-unit entered as part of a multiple interrupt, this value specifies the corresponding number of equivalent single interrupts (including the current one) that remain to be handled; if ONCOUNT is used in any other case the returned value is zero.

ONFILE Condition Built-in Function

Definition: ONFILE determines the name of the file for which an input/output or CONVERSION condition was raised and returns that name to the point of invocation. This function can be used in the on-unit for any input/output or CONVERSION condition; it also can be used in an on-unit for an ERROR or FINISH condition raised as standard system action for an input/output or CONVERSION condition.

Reference: ONFILE

Result: The value returned by this function is a varying-length character string, of 31-character maximum length, consisting of the name of the file for which an input/output or CONVERSION condition was raised. In the case of a CONVERSION condition, if that condition is not associated with a file, the returned value is the null string.

ONKEY Condition Built-in Function

Definition: ONKEY extracts the value of the key for the record that caused an input/output condition to be raised. This function can be used in the on-unit for an input/output condition or a CONVERSION condition; it can also be used in an on-unit for an ERROR or FINISH condition raised as standard system action for one of the above conditions.

Reference: ONKEY

Result: The value returned by this function is a varying-length character string giving the value of the key for the record that caused an input/output condition to be raised. If the interrupt is not associated with a keyed record, the returned value is the null string.

ONLOC Condition Built-in Function

Definition: Whenever an ON-condition is raised, ONLOC may be used in the on-unit for that condition to determine the entry point to the procedure in which that condition was raised. ONLOC may be used in any on-unit.

Reference: ONLOC

Result: The value returned by this function is a varying-length character string giving the name of the entry point to the

procedure in which the ON-condition was raised. If ONLOC is used out of context, a null string is returned.

ONSOURCE Condition Built-in Function

Definition: Whenever the CONVERSION condition is raised, ONSOURCE may be used to extract the contents of the field that was being processed when the condition was raised. This function can be used in the on-unit for a CONVERSION condition or in an on-unit for an ERROR or FINISH condition raised as standard system action for a CONVERSION condition. (ONSOURCE can also be used as a pseudo-variable.)

Reference: ONSOURCE

Result: The value returned by this function is a varying-length character string (maximum length is 255 for the F Compiler) giving the contents of the field being processed when CONVERSION was raised. This string may be modified in the on-unit by use of the ONCHAR or ONSOURCE pseudo-variable. If ONSOURCE is used out of context, a null string is returned.

BASED STORAGE BUILT-IN FUNCTIONS

The based storage built-in functions generally return special values to program control variables concerned in the use of based storage and list processing. Only ADDR requires an argument.

ADDR Based Storage Built-in Function

Definition: ADDR finds the location at which a given variable has been allocated and returns a pointer value to the point of invocation. The pointer value identifies the location at which the variable has been allocated.

Reference: ADDR (x)

Argument: The argument, "x," is the variable whose location is to be found. It can be any variable that represents an element, an array, a structure, an area, an element of an array, a minor structure, or an element of a structure. It can be of any data type and storage class. For the F Compiler, the variable should not be a bit-string variable forming part of an unaligned array or structure.

Result: ADDR returns a pointer value identifying the location at which "x" has been allocated. If "x" is a parameter, the returned value identifies the corresponding argument (dummy or otherwise). If "x" is a based variable, the returned value is determined from the pointer variable declared with "x"; if this pointer variable has not been set, the value returned by ADDR is undefined. If "x" is an unallocated controlled variable, a null pointer value is returned.

EMPTY Based Storage Built-in Function

Definition: EMPTY clears an area of storage defined by an area variable, by effectively freeing all the allocations contained within the area. The area can then be used for a new set of allocations.

Reference: EMPTY

Arguments: None

Result: EMPTY returns an area of zero size, containing no allocations, to the point of invocation. When this value is assigned to an area variable, all the allocations contained within the area are freed.

Note: The value of the EMPTY built-in function is automatically assigned to all area variables when they are allocated.

NULL Based Storage Built-in Function

Definition: NULL returns a null pointer value (that is, a pointer value that cannot identify any allocation) so as to indicate that a pointer variable does not currently identify an allocation.

Reference: NULL

Arguments: None

Result: The value returned by this function is a null pointer value. This value cannot be converted to offset type.

NULLO Based Storage Built-in Function

Definition: NULLO returns a null offset value (that is, an offset value that cannot

identify any relative location of a based variable allocation) so as to indicate that an offset variable does not currently identify an allocation.

Reference: NULLO

Arguments: None

Result: The value returned by this function is a null offset value. This value cannot be converted to pointer type.

MULTITASKING BUILT-IN FUNCTIONS

The multitasking built-in functions are used during multitasking and during asynchronous input/output operations. They allow the programmer to investigate the relative priority of a task or the current state of execution of a task or asynchronous input/output operation. They all require arguments.

COMPLETION Multitasking Built-in Function

Definition: COMPLETION determines the completion value of a given event variable. (COMPLETION can also be used as a pseudo-variable.)

Reference: COMPLETION (event-name)

Argument: The argument, "event-name", can be an event element or an event array. It represents the event (or events) whose completion value is to be determined. The event can be associated with completion of a task, or with completion of an input/output operation, or it can be user-defined. It can be active or inactive. An array argument causes an array value to be returned.

Result: The value returned by this function is '0'B if the event is incomplete, '1'B if the event is complete.

PRIORITY Multitasking Built-in Function

Definition: PRIORITY determines the relative priority of a given task. (PRIORITY can also be used as a pseudo-variable.)

Reference: PRIORITY (task-name)

Argument: The argument, "task-name," represents the task whose relative priority is to be determined.

Result: The value returned by this task is a fixed binary value of precision (n,0), where n is implementation-defined (15, for the F Compiler). The value is the priority value of the named task, relative to the priority of the task evaluating the function. No interrupt can occur during evaluation of PRIORITY.

STATUS Multitasking Built-in Function

Definition: STATUS determines the status value of a given event variable. (STATUS can also be used as a pseudo-variable.)

Reference: STATUS (event-name)

Argument: The argument, "event-name", can be an event element or an event array. It represents the event (or events) whose status value is to be determined. The event can be associated with completion of a task, or with completion of an input/output operation, or it can be user-defined. It can be active or inactive. An array argument causes an array value to be returned.

Result: The value returned by this function is a fixed binary value of default precision ((15,0) for the F Compiler). It is zero if the event is normal, or nonzero if abnormal.

MISCELLANEOUS BUILT-IN FUNCTIONS

The functions described in this section have little in common with each other and with the other categories of built-in functions. Some require arguments and others do not. Those that do not require arguments will be so identified.

ALLOCATION Built-in Function

Definition: ALLOCATION determines whether or not storage is allocated for a given controlled variable and returns an appropriate indication to the point of invocation.

Reference: ALLOCATION (x)

Argument: The argument, "x," must be an unsubscripted array name, a major structure name, or an element variable name, and it must have the CONTROLLED attribute.

Result: The value returned by this function is defined as follows: if storage has been allocated for "x," the returned value is '1'B (provided that the allocation is known to the task executing the function); if storage has not been allocated for "x," the returned value is '0'B.

COUNT Built-in Function

Definition: COUNT determines the number of data items that were transmitted during the last GET or PUT operation on a given file and returns the result to the point of invocation.

Reference: COUNT (file-name)

Argument: The argument, "file name," represents the file to be investigated. This file must have the STREAM attribute.

Result: The value returned by this function is a binary fixed-point integer of default precision specifying the number of element data items that were transmitted during the last GET or PUT operation on "file name." Note that if an on-unit or procedure is entered during a GET or PUT operation, and within that on-unit or procedure a GET or PUT is executed for the same file, the value of COUNT is reset for the new operation and is not restored when the original GET or PUT is continued.

DATE Built-in Function

Definition: DATE returns the current date to the point of invocation.

Reference: DATE

Arguments: None

Result: The value returned by this function is a character string of length six, in the form yymmdd, where:

yy is the current year

mm is the current month

dd is the current day

Example: If the current date is February 29, 1968, execution of the statement

```
X = DATE;
```

will cause the character string '680229' to be returned to the point of invocation.

LINENO Built-in Function

Definition: LINENO finds the current line number for a file having the PRINT attribute and returns that number to the point of invocation.

Reference: LINENO (file-name)

Argument: The argument, "file name," must be the name of a file having the PRINT attribute.

Result: The value returned by this function is a binary fixed-point integer of default precision specifying the current line number of "file name."

TIME Built-in Function

Definition: TIME returns the current time to the point of invocation.

Reference: TIME

Arguments: None

Result: The value returned by this function is a character string of length nine, in the form hhmmssttt, where:

hh is the current hour of the day

mm is the number of minutes

ss is the number of seconds

ttt is the number of milliseconds in machine-dependent increments

Example: If the current time is 4 P.M., 23 minutes, 19 seconds, and 80 milliseconds, a reference to the TIME function, for some computers, will return the character string '162319080' to the point of invocation.

PSEUDO-VARIABLES

In general, pseudo-variables are certain built-in functions that can appear wherever other variables can appear in order to

receive values. In short, they are built-in functions used as receiving fields. For example, a pseudo-variable may appear on the left of the equal sign in an assignment or DO statement; it may appear in the data list of a GET statement; it may appear as the string name in the STRING option of a PUT statement.

Since all pseudo-variables have built-in function counterparts, only a short description of each pseudo-variable is given here; the discussion of the corresponding built-in function should be consulted for the details. Note that pseudo-variables cannot be nested; for example, the following statement is invalid:

```
UNSPEC(SUBSTR(A,1,2))='00'B;
```

COMPLETION Pseudo-variable

Reference: COMPLETION (event-name)

Description: The named event variable must be inactive and is as described for the COMPLETION built-in function. The value received by this pseudo-variable is a bit-string of length 1. This value sets the completion value of the event variable. A value of '0'B specifies that the event associated with the "event variable" is incomplete; a value of '1'B specifies that the event is complete. No interrupt can take place during assignment to the pseudo-variable.

COMPLEX Pseudo-variable

Reference: COMPLEX (a,b)

Description: Only complex values can be assigned to this pseudo-variable. The real part of the complex value is assigned to the variable "a"; the imaginary part is assigned to the variable "b." If either "a" and "b" is an array, both must be arrays of identical bounds.

IMAG Pseudo-variable

Reference: IMAG (c)

Description: Real or complex values may be assigned to this pseudo-variable. The real value or the real part of the complex value is assigned to the imaginary part of the complex variable "c," which may be an element variable or an array variable.

ONCHAR Pseudo-variable

Reference: ONCHAR

Description: ONCHAR can be used in the on-unit for a CONVERSION condition or in the on-unit for an ERROR or FINISH condition raised as standard system action for a CONVERSION condition; it can also be used in a block directly or indirectly activated by such an on-unit. If ONCHAR is used in some other context, it is an error.

The expression being assigned to ONCHAR is evaluated, converted to a character string of length 1, and assigned to the character that caused the error. The new character will displace the current value of the ONCHAR built-in function, and will be used when the conversion is re-attempted, upon the resumption of execution at the point of interrupt.

ONSOURCE Pseudo-variable

Reference: ONSOURCE

Description: ONSOURCE can be used in the on-unit for a CONVERSION condition or in an on-unit for an ERROR or FINISH condition raised as standard system action for a CONVERSION condition; it can also be used in a block directly or indirectly activated by such an on-unit. If ONSOURCE is used in some other context, it is an error.

The expression being assigned to ONSOURCE is evaluated, converted to a character string, and assigned to the string that caused the CONVERSION condition to be raised. The string will be padded with blanks, if necessary, to match the length of the string that caused the error. This new string displaces the current value of the ONSOURCE built-in function and will be used when the conversion is re-attempted, upon the resumption of execution at the point of interrupt.

PRIORITY Pseudo-variable

Reference: PRIORITY [(task-name)]

Description: The "task-name" is as described for the PRIORITY built-in function, but need not be specified. The value received by this pseudo-variable is a fixed-point binary value of precision (n,0), where n is implementation-defined (15, for the F Compiler). The priority value of the named task variable is adjusted so that it becomes n relative to the

priority that the current task had prior to the assignment. If an active task is associated with the named task variable, its priority is given the same value as the task variable.

If "task-name" is not specified, the task variable associated with the current task (if there is such a variable) is implied, and the priority of this variable is modified; hence, the priority of the current task is modified.

No interrupt can occur during assignment to the PRIORITY pseudo-variable.

The operating system allows a task to change only its own priority or that of any of its immediate subtasks.

REAL Pseudo-variable

Reference: REAL (c)

Description: Real or complex values may be assigned to this pseudo-variable. The real value or the real part of the complex value is assigned to the real part of the complex variable "c," which may be an element variable or an array variable.

STATUS Pseudo-variable

Reference: STATUS (event-name)

Description: The named event variable can be active or inactive, and is as described for the STATUS built-in function. The value received by this pseudo-variable is a fixed point binary value of default precision ((15,0) for the F Compiler). No interrupt can occur during assignment to the pseudo-variable.

SUBSTR Pseudo-variable

Reference: SUBSTR (string,i[,j])

Description: The value being assigned to SUBSTR is assigned to the substring of the character- or bit-string variable "string," as defined for the built-in function SUBSTR. If "string" is an array, i and/or j may be arrays, in which case they must have identical bounds. The remainder of "string" remains unchanged.

UNSPEC Pseudo-variable

Reference: UNSPEC (v)

Description: The letter "v" represents an element or array variable of arithmetic or string type. The value being assigned to UNSPEC is evaluated, converted to a bit string (the length of which is a function of the characteristics of "v" -- see the UNSPEC built-in function), and then assigned to "v," without conversion to the type of "v." If "v" is a string of varying length, its length after the assignment will be the same as that of the bit string assigned to it.

INTRODUCTION

The ON-conditions are those exceptional conditions that can be specified in PL/I by means of an ON statement. If a condition is enabled, the occurrence of the condition will result in an interrupt. The interrupt, in turn, will result in the execution of the current action specification for that condition. If an ON statement for that condition is not in effect, the current action specification is the standard system action for that condition. If an ON statement for that condition is in effect, the current action specification is either SYSTEM, in which case the standard system action for that condition is taken, or an on-unit, in which case the programmer has supplied his own action to be taken for that condition.

If a condition is not enabled (i.e., if it is disabled), and the condition occurs, an interrupt will not take place, and errors may result.

Some conditions are always enabled unless they have been explicitly disabled by condition prefixes; others are always disabled unless they have been explicitly enabled by condition prefixes; and still others are always enabled and cannot be disabled.

Those conditions that are always enabled unless they have been explicitly disabled by condition prefixes are:

CONVERSION

FIXEDOVERFLOW

OVERFLOW

UNDERFLOW

ZERODIVIDE

Each of the above conditions can be disabled by a condition prefix specifying the condition name preceded by NO without intervening blanks. Thus, one of the following names in a condition prefix will disable the respective condition:

NOCONVERSION

NOFIXEDOVERFLOW

NOOVERFLOW

NOUNDERFLOW

NOZERODIVIDE

Such a condition prefix renders the corresponding condition disabled throughout the scope of the prefix; the condition remains enabled outside this scope. (Scope of a condition prefix is discussed in Part I, Chapter 11, "Exceptional Condition Handling and Program Checkout.")

Conversely, those conditions that are always disabled unless they have been enabled by a condition prefix are:

SIZE

SUBSCRIPTRANGE

STRINGRANGE

CHECK

The appearance of one of these four in a condition prefix renders the condition enabled throughout the scope of the prefix; the condition remains disabled outside this scope. Further, a condition prefix specifying NOSIZE, NOSUBSCRIPTRANGE, NOSTRINGRANGE, or NOCHECK will disable the corresponding condition throughout the scope of that prefix.

All other conditions are always enabled and remain so for the duration of the program. These conditions are:

AREA

CONDITION

ENDFILE

ENDPAGE

ERROR

FINISH

KEY

NAME

RECORD

TRANSMIT

UNDEFINEDFILE

Condition Codes (ON-Codes)

The ONCODE built-in function may be used by the programmer in any on-unit to determine the nature of the error or condition that caused entry into that on-unit. The codes corresponding to the conditions and errors checked for by the F Compiler are given below:

Code	Condition/Error
0	FINISH (normal termination, or signaled by STOP or EXIT)
3	Source program error
4	ONCODE function used out of context
9	ERROR (signaled)
10	NAME
20	RECORD (signaled)
21	RECORD (record variable smaller than record size)
22	RECORD (record variable larger than record size)
23	RECORD (attempt to write zero length record)
24	RECORD (zero length record has been read)
40	TRANSMIT (signaled)
41	TRANSMIT (output)
42	TRANSMIT (input)
50	KEY (signaled)
51	KEY (keyed record not found)
52	KEY (attempt to add duplicate key)
53	KEY (key sequence error)
54	KEY (key conversion error)
55	KEY (key specification error)
56	KEY (keyed relative record/track outside data set limit)
57	KEY (no space available to add keyed record)
70	ENDFILE
80	UNDEFINEDFILE (signaled)
81	UNDEFINEDFILE (attribute conflict)
82	UNDEFINEDFILE (access method not supported)
83	UNDEFINEDFILE (blocksize not specified)
84	UNDEFINEDFILE (file cannot be opened, no DD card)
85	UNDEFINEDFILE (error initializing REGIONAL data set)
90	ENDPAGE
300	OVERFLOW
310	FIXEDOVERFLOW
320	ZERODIVIDE
330	UNDERFLOW
340	SIZE (normal)
341	SIZE (I/O)
350	STRINGRANGE
360	AREA (raised by based variable allocation)
361	AREA (raised by area assignment)
362	AREA (signaled)
500	CONDITION
510	CHECK (LABEL)
511	CHECK (variable)
520	SUBSCRIPTRANGE
600	CONVERSION (internal) (signaled)
601	CONVERSION (I/O)
602	CONVERSION (transmit)
603	CONVERSION (error in F-format input)
604	CONVERSION (error in F-format input) (I/O)
605	CONVERSION (error in F-format input) (transmit)
606	CONVERSION (error in E-format input)
607	CONVERSION (error in E-format input) (I/O)
608	CONVERSION (error in E-format input) (transmit)
609	CONVERSION (error in B-format input)
610	CONVERSION (error in B-format input) (I/O)
611	CONVERSION (error in B-format input) (transmit)
612	CONVERSION (character-string to arithmetic)
613	CONVERSION (character-string to arithmetic) (I/O)
614	CONVERSION (character-string to arithmetic) (transmit)
615	CONVERSION (character-string to bit-string)
616	CONVERSION (character-string to bit-string) (I/O)
617	CONVERSION (character-string to bit-string) (transmit)
618	CONVERSION (character to picture)
619	CONVERSION (character to picture) (I/O)
620	CONVERSION (character to picture) (transmit)
621	CONVERSION (P-format input -- decimal)
622	CONVERSION (P-format input -- decimal) (I/O)
623	CONVERSION (P-format input -- decimal) (transmit)
624	CONVERSION (P-format input -- character)
625	CONVERSION (P-format input -- character) (I/O)
626	CONVERSION (P-format input -- character) (transmit)
627	CONVERSION (P-format input -- sterling)
628	CONVERSION (P-format input -- sterling) (I/O)
629	CONVERSION (P-format input -- sterling) (transmit)
1000	Attempt to read output file
1001	Attempt to write input file
1002	GET/PUT string length error
1003	Unacceptable output transmission error

1004	Print option on non-PRINT file
1005	Message length for DISPLAY statements is zero
1006	Illegal array data item for data-directed input
1007	REWRITE not immediately preceded by READ
1008	GET STRING -- unrecognizable data name
1009	Unsupported file operation
1010	File type not supported
1011	Inexplicable I/O error
1012	Outstanding read for update exists
1013	No completed read exists -- incorrect NCP value
1014	Too many incomplete I/O operations
1015	Event variable already in use
1016	Implicit open failures -- cannot proceed
1017	Attempt to rewrite out of sequence
1018	ERROR condition raised when end of file encountered unexpectedly in list-directed or data-directed input, or when field width in format list of edit-directed input would take scan beyond end of file.
1500	Short SQRT error
1501	Long SQRT error
1504	Short LOG error
1505	Long LOG error
1506	Short SIN error
1507	Long SIN error
1508	Short TAN error
1509	Long TAN error
1510	Short ARCTAN error
1511	Long ARCTAN error
1512	Short SINH error
1513	Long SINH error
1514	Short ARCTANH error
1515	Long ARCTANH error
1550	Invalid exponent in short float integer exponentiation
1551	Invalid exponent in long float integer exponentiation
1552	Invalid exponent in short float general exponentiation
1553	Invalid exponent in long float general exponentiation
1554	Invalid exponent in complex short float integer exponentiation
1555	Invalid exponent in complex long float integer exponentiation
1556	Invalid exponent in complex short float general exponentiation
1557	Invalid exponent in complex long float general exponentiation
1558	Invalid argument in short float complex ARCTAN or ARCTANH
1559	Invalid argument in long float complex ARCTAN or ARCTANH
2000	Unacceptable DELAY statement
2001	Unacceptable TIME statement
3000	E-format conversion error
3001	F-format conversion error
3002	A-format conversion error
3003	B-format conversion error
3004	A-format input error
3005	B-format input error
3006	Picture character-string error
3798	ONSOURCE or ONCHAR pseudo-variables used out of context
3799	Improper return from CONVERSION on-unit
3800	Structure length ≥ 16**6 bytes
3801	Virtual origin of array ≥ 16**6 or ≤-16**6
3900	Attempt to wait on inactive and incomplete event
3901	Task variable already active
3902	Event already being waited for
3903	Wait on more than 255 incomplete events
3904	Active event variable as argument to COMPLETION pseudo-variable
3905	Invalid task variable as argument to PRIORITY pseudo-variable
3906	Event variable active in assignment statement
3907	Event variable already active
3908	Attempt to wait for I/O event in wrong task
8091	Invalid operation
8092	Privileged operation
8093	EXECUTE statement executed
8094	Protection violation
8095	Addressing interruption
8096	Specification interruption
8097	Data interruption
9000	Too many active on-units and entry parameter procedures
9001	No invocation count
9002	Invalid free storage (main procedure)
9003	Invalid free VDA

Multiple Interrupts

A multiple interrupt can occur only for an input/output operation that has been associated with an event variable. It occurs during the execution of the WAIT statement naming that event variable, if the event has been completed abnormally (i.e., if one or more conditions occurred during the operation). Since conditions for an input/output event are raised at the execution of the WAIT for that event, the interrupts for these conditions also occur at this time. It is possible for more than one interrupt to occur for an input/output event. The aggregate of interrupts for an input/output event is called a multiple interrupt.

When an input/output event is completed abnormally, the order in which the conditions are raised, and therefore, the order in which the interrupts for these conditions occur, is implementation-defined. If the on-unit for such a condition ends abnormally, then all unprocessed conditions (i.e., remaining interrupts of the multiple interrupt) are ignored; if an

on-unit ends normally, the next condition is processed. If an on-unit has not been established for such a condition or if SYSTEM is in effect, the next condition outstanding will be processed only if the standard system action is to comment and continue; if the standard system action is otherwise, all remaining interrupts in the multiple interrupt will be ignored.

Note: If the UNDEFINEDFILE condition is raised by an attempt at implicit opening, caused by a statement associated with an event variable, the condition is raised immediately, and the interrupt will occur even before the WAIT statement is executed.

SECTION ORGANIZATION

This section presents each condition in its logical grouping, and in alphabetical order within that grouping. In general, the following information is given for each condition:

1. General format -- given only when it consists of more than the condition name.

2. Description -- a discussion of the condition, including the circumstances under which the condition can be raised. Note that an enabled condition can always be raised by a SIGNAL statement; this fact is not included in the descriptions.

3. Result -- the result of the operation that caused the condition to occur. This applies when the condition is disabled as well as when it is enabled. In some cases, the result is not defined; that is, it cannot be predicted. This is stated wherever applicable.

4. Standard system action -- the action taken by the system when an interrupt occurs and the programmer has not specified an on-unit to handle that interrupt.

5. Status -- an indication of the enabled/disabled status of the condition at the start of the program, and how the condition may be disabled (if possible) or enabled.

6. Normal return -- the point to which control is returned as a result of the normal termination of the on-unit. A GO TO statement that transfers control out of an on-unit is an abnormal on-unit termination. Note that if a condition has been raised by the SIGNAL statement, the normal return is always to the statement immediately following SIGNAL.

The conditions are grouped as follows:

1. Computational conditions -- those conditions associated with data handling, expression evaluation, and computation. They are:

   AREA
   CONVERSION
   FIXEDOVERFLOW
   OVERFLOW
   SIZE
   UNDERFLOW
   ZERODIVIDE

2. Input/output conditions -- those conditions associated with data transmission. They are:

   ENDFILE
   ENDPAGE
   KEY
   NAME
   RECORD
   TRANSMIT
   UNDEFINEDFILE

3. Program-checkout conditions -- those conditions that facilitate the debugging of a program. They are:

   CHECK
   SUBSCRIPTRANGE
   STRINGRANGE

4. System action conditions -- those conditions that provide facilities to extend the standard system action that is taken after the occurrence of a condition or at the completion of a program. They are:

   ERROR
   FINISH

5. Programmer-named condition -- the CONDITION condition.

COMPUTATIONAL CONDITIONS

The AREA Condition

Description: The AREA condition is raised in either of the following circumstances:

1. When an attempt is made to allocate a based variable within an area that contains insufficient free storage for the allocation to be made.

2. When an attempt is made to perform an area assignment, and the target area contains insufficient storage to accommodate the allocations in the source area.

Result: If the condition occurs as the result of an attempted allocation, the allocation has no effect; if the condition occurs as a result of an area assignment, the contents of the target area are undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Normal Return: On normal return from the on-unit, the action is as follows:

1. If the condition was raised by an allocation, the allocation is reattempted. If the on-unit has changed the value of a pointer qualifying the reference to the inadequate area so that it points to another area, the allocation is reattempted within the new area.

2. If the condition was raised by an area assignment, or by a SIGNAL statement, execution continues at the point of interrupt.

The CONVERSION Condition

Description: The CONVERSION condition occurs whenever an illegal conversion is attempted on character-string data. This attempt may be made internally or during an input/output operation. For example, the condition occurs when a character other than 0 or 1 exists in a character string being converted to a bit string; other examples are when a character string being converted to a numeric character field contains characters not permitted by the PICTURE specification, or when a string being converted to coded arithmetic data does not contain the character representation of an arithmetic constant.

All conversions of character-string data are carried out character-by-character in a left-to-right sequence and the condition occurs for each invalid character. When an invalid character is encountered, an interrupt occurs (provided, of course, that CONVERSION has not been disabled) and the current action specification for the condition is executed. If the action specification is an on-unit, the invalid character can be corrected within the unit by using the ONSOURCE or ONCHAR pseudo-variables. On return from the on-unit, the conversion of the string is retried from the beginning. For the F Compiler, if the illegal character has not been corrected, a message is printed and the ERROR condition is raised.

Result: When CONVERSION occurs, the contents of the entire result field are undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: CONVERSION is enabled throughout the program, except within the scope of a condition prefix specifying NOCONVERSION.

Normal Return: Upon the normal termination of the on-unit for this condition, control returns to the beginning of the string and the conversion is retried.

The FIXEDOVERFLOW Condition

Description: The FIXEDOVERFLOW condition occurs when the length of the result of a fixed-point arithmetic operation exceeds N. For System/360 implementations, N is 15 for decimal fixed-point values and 31 for binary fixed-point values.

Result: The result of the invalid fixed-point operation is undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: FIXEDOVERFLOW is enabled throughout the program, except within the scope of a condition prefix that specifies NOFIXEDOVERFLOW.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

The OVERFLOW Condition

Description: The OVERFLOW condition occurs when the magnitude of a floating-point number exceeds the permitted maximum. (For System/360 implementations, the magnitude of a floating-point number or intermediate result must not be greater than approximately 10^{75} or 2^{252}.)

Result: The value of such an illegal floating-point number is undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: OVERFLOW is enabled throughout the program, except within the scope of a condition prefix specifying NOOVERFLOW.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

The SIZE Condition

Description: The SIZE condition occurs only when high-order (i.e., leftmost) non-zero binary or decimal digits are lost in an assignment to a variable or a temporary or in an input/output operation. This loss may result from a conversion involving different data types, different bases, different scales, or different precisions.

The SIZE condition differs from the FIXEDOVERFLOW condition in an important sense, i.e., FIXEDOVERFLOW occurs when the size of a calculated fixed-point value exceeds N (the maximum allowed), whereas SIZE is raised when the size of the value being assigned to a data item exceeds the declared (or default) size of the data item. SIZE can be raised on assignment of a value regardless of whether or not FIXEDOVERFLOW was raised in the calculation of that value.

The declared size is not necessarily the actual precision with which the item is held in storage; however, the limit for SIZE is the declared or default size, not the actual size in storage. For example, with the F Compiler, a fixed binary item of precision (20) will occupy a fullword in storage, but SIZE is raised if a value whose size exceeds FIXED BINARY(20) is assigned to it.

Result: The contents of the data item receiving the wrong-sized value are undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: SIZE is disabled within the scope of a NOSIZE condition prefix and elsewhere throughout the program, except within the scope of a condition prefix specifying SIZE.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

The UNDERFLOW Condition

Description: The UNDERFLOW condition occurs when the magnitude of a floating-point number is smaller than the permitted minimum. (For System/360 implementations, the magnitude of a floating-point value may not be less than approximately 10^{-78} or 2^{-260}.)

UNDERFLOW does not occur when equal numbers are subtracted (often called significance error).

Note that, for the F Compiler, the expression X**(-Y) (where Y>0) is evaluated by taking the reciprocal of X**Y; hence, the OVERFLOW condition may be raised instead of the UNDERFLOW condition.

Result: The invalid floating-point value is set to 0.

Standard System Action: In the absence of an on-unit, the system prints a message and continues execution from the point at which the interrupt occurred.

Status: UNDERFLOW is enabled throughout the program, except within the scope of a condition prefix specifying NOUNDERFLOW.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

The ZERODIVIDE Condition

Description: The ZERODIVIDE condition occurs when an attempt is made to divide by zero. This condition is raised for fixed-point and floating-point division.

Result: The result of a division by zero is undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: ZERODIVIDE is enabled throughout the program, except within the scope of a condition prefix specifying NOZERODIVIDE.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

INPUT/OUTPUT CONDITIONS

The input/output conditions are always enabled and cannot appear in condition prefixes; they can be specified only in ON, SIGNAL, and REVERT statements.

The ENDFILE Condition

General Format: ENDFILE (file-name)

Description: The ENDFILE condition can be raised during a GET or READ operation; it is caused by an attempt to read past the file delimiter of the file named in the GET or READ statement. It applies only to SEQUENTIAL files.

If the file is not closed after ENDFILE occurs, then any subsequent GET or READ statement for that file immediately raises the ENDFILE condition again.

If ENDFILE is raised by an input/output statement using the EVENT option, the interrupt does not take place until the execution of a subsequent WAIT statement for that event in the same procedure.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: The ENDFILE condition is always enabled; it cannot be disabled.

Normal Return: Upon the normal termination of the on-unit for the condition, execution continues with the statement immediately following the GET or READ statement that caused the ENDFILE (or, if ENDFILE was raised by a READ with the EVENT option, control passes back to the WAIT statement from which the on-unit was invoked).

The ENDPAGE Condition

General Format: ENDPAGE (file-name)

The "file name" must be the name of a file having the PRINT attribute.

Description: The ENDPAGE condition is raised when a PUT statement results in an attempt to start a new line beyond the limit specified for the current page. This limit can be specified by the PAGESIZE option in an OPEN statement; if PAGESIZE has not been specified, a default limit of 60 applies for the F Compiler. The attempt to exceed the limit may be made during data transmission (including associated format items, if the PUT statement is edit-directed), by the LINE option, or by the SKIP option. ENDPAGE can also be raised by a LINE option or LINE format item that specified a line number less than the current line number.

When ENDPAGE is raised, the current line number is one greater than that specified by the PAGESIZE option (or 61, if the default applies) so that it is possible to continue writing on the same page. The on-unit may start a new page by execution of a PAGE option or a PAGE format item, which sets the current line to 1.

ENDPAGE is raised only once per page. If the on-unit does not start a new page, the current line number may increase indefinitely. If a subsequent LINE option or LINE format item specifies a line number that is less than the current line number, ENDPAGE is not raised, but a new page is started with the current line set to 1.

If ENDPAGE is raised during data transmission, then, on return from the on-unit, the data is written on the current line, which may have been changed by the on-unit. If ENDPAGE results from a LINE or SKIP option, then, on return from the on-unit, the action specified by LINE or SKIP is ignored.

Standard System Action: In the absence of an on-unit, the system starts a new page. If the condition is signalled, execution is unaffected and continues with the statement following the SIGNAL statement.

Status: ENDPAGE is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit for this condition, execution of the PUT statement continues in the manner described above.

The KEY Condition

General Format: KEY (file-name)

Description: The KEY condition can be raised only during operations on keyed records. It is raised in any of the following cases:

1. The keyed record cannot be found.
2. An attempt is made to add a duplicate key.
3. The key is out of sequence.

4. An error occurred in the conversion of the key.

5. The key has not been specified correctly.

6. No space is available to add the keyed record.

If KEY is raised by an input/output statement using the EVENT option, the interrupt does not occur until the execution of a subsequent WAIT statement for that event in the same procedure.

The condition is not raised for a LOCATE statement until actual transmission is attempted (that is, immediately before execution of the next WRITE or LOCATE statement for the file, or immediately before the file is closed); until the error is corrected, the record cannot be transmitted, nor can any further operation take place for the file.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: KEY is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit for this condition, control passes to the statement immediately following the statement that caused KEY to be raised (or, if KEY was raised by an input/output statement with the EVENT option, control passes back to the WAIT statement from which the on-unit was invoked).

The NAME Condition

General Format: NAME (file-name)

Description: The NAME condition can be raised only during a data-directed GET statement. It can be raised either when an identifier in the input stream does not have a counterpart in the data list of the GET statement or when the GET statement has no data list and an identifier that is not known in the block is encountered in the stream.

NAME is raised at the time the unmatched identifier is encountered in the stream.

The programmer may retrieve the data field (i.e., the identifier and its value) containing the unmatched identifier by using the built-in function DATAFIELD in the on-unit.

Standard System Action: In the absence of an on-unit, the system ignores the incorrect data field, prints a message, and continues the execution of the GET statement.

Status: NAME is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit for this condition, the execution of the GET statement continues with the next identifier in the stream.

The RECORD Condition

General Format: RECORD (filename)

Description: The RECORD condition can be raised only during a READ, WRITE, or REWRITE operation. It is raised by any of the following:

1. The size of the record is greater than the size of the variable.

2. The size of the record is less than the size of the variable.

3. A record of zero length has been read.

4. An attempt is made to write a record of zero length.

If the size of the record is greater than the size of the variable, the excess data in the record is lost on input and is unpredictable on output. If the size of the record is less than the size of the variable, the excess data in the variable is not transmitted on output and is unaltered on input. (Thus, if a zero length record is read, the variable contains the same data that it contained before the read operation.) If an attempt is made to write a record of zero length, the attempt is aborted, and, in effect, the statement is ignored.

If RECORD is raised during transmission of an area, the area control field will contain incorrect information

If RECORD is raised by an input/output statement using the EVENT option, the interrupt does not occur until the execution of a subsequent WAIT statement for that event in the same procedure.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: RECORD is always enabled; it cannot be disabled.

Normal Return: Upon normal completion of the on-unit, execution continues with the statement immediately following the one for which RECORD occurred (or if RECORD was raised by an input/output statement with an EVENT option, control returns to the WAIT statement from which the on-unit was invoked).

The TRANSMIT Condition

General Format: TRANSMIT (file-name)

Description: The TRANSMIT condition can be raised during any input/output operation. It is raised by a permanent transmission error and, as a result any data transmit-

ted is potentially incorrect. During input, the condition is raised after assignment of the potentially incorrect data item or record. During output, the condition is raised after the transmission of the potentially incorrect data item or record has been attempted.

If TRANSMIT is raised by an input/output statement using the EVENT option, the interrupt does not take place until the execution of a subsequent WAIT statement for that event in the same procedure.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: TRANSMIT is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit, processing continues with the next data item for STREAM input/output, or the next statement for RECORD input/output (or if TRANSMIT was raised by an input/output statement with an EVENT option, control returns to the WAIT statement from which the on-unit was invoked).

The UNDEFINEDFILE Condition

General Format: UNDEFINEDFILE (file-name)

Description: The UNDEFINEDFILE condition is raised whenever an attempt to open a file is unsuccessful. If the attempt is made by means of an OPEN statement that specifies more than one file name, attempts to open all other files in that statement will be made before the condition is raised. If the condition is raised for more than one file in the same OPEN statement, on-units will be executed according to the order of appearance (taken from left to right) of the file names in that OPEN statement.

If the condition is raised by an implicit file opening in an input/output statement without the EVENT option, then, upon normal return from the on-unit, processing continues with the remainder of the interrupted input/output statement. If the file was not opened in the on-unit, then the statement cannot be continued and the ERROR condition is raised.

If the condition is raised by an implicit file opening in an input/output statement having an EVENT option, then the interrupt occurs before the event variable is initialized. In other words, the event variable retains its previous value and remains inactive. On normal return from the on-unit, the event variable is initialized, that is, it is made active and its completion value is set to '0'B (provided the file has been opened in the on-unit). Processing then continues with the remainder of the interrupted statement. However, if the file has not been opened in the on-unit, the event variable remains uninitialized, the statement cannot be continued, and the ERROR condition is raised.

For the F Compiler, some cases for which the UNDEFINEDFILE condition is raised are as follows:

1. A conflict in attributes exists.
2. The blocksize has not been specified.
3. There is no recognizable DD statement for the file.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: UNDEFINEDFILE is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the final on-unit, control is given to the statement immediately following the statement that caused the condition to be raised (see "Description" for action in the case of an implicit opening).

PROGRAM-CHECKOUT CONDITIONS

The CHECK Condition

General Format: CHECK (name-list)

The "name list" is one or more names separated by commas; a name may be a qualified name. Each name must be one of the following:

1. An entry name
2. A statement label constant
3. An unsubscripted name representing an element, an array, or a structure

The names appearing in a CHECK prefix refer to the names known within the block to which the prefix is attached. A name cannot be a parameter or a variable having the DEFINED or BASED atrributes.

Description: The CHECK condition is raised only within the scope of a CHECK condition prefix. Such a condition prefix may be prefixed only to a PROCEDURE or BEGIN

statement. The CHECK condition is enabled separately for each name in the list of the CHECK prefix. For example, the prefix CHECK (A,B,C) is equivalent to CHECK (A): CHECK (B): CHECK (C). Hence, the action specification can be controlled separately for each name. The REVERT statement can be used to change the action specification for one or more names in the list. Also, a NOCHECK prefix can be used to disable the CHECK condition for a specific name (like CHECK, NOCHECK can appear only as a prefix to a PROCEDURE or BEGIN statement).

If the name of a structure or array of structures appears in the name list following CHECK, such a list is equivalent to one that contains, in the order in which they were declared, the elements of that structure or array of structures. For example, if P is defined:

```
DECLARE 1 P, 2 Q, 2 R, 2 S;
```

then:

```
CHECK (P)
```

is equivalent to:

```
CHECK (Q,R,S)
```

The CHECK condition is raised within the scope of a CHECK prefix in any of the following cases:

1. If a name in the CHECK prefix is a statement label constant, the condition is raised and the interrupt occurs prior to the execution of the statement to which the label is prefixed. If the label is prefixed to a DECLARE or FORMAT statement, the condition is not raised.

2. If a name in the CHECK prefix is a variable (as specified in item 3 of the general format above), the condition is raised whenever the value of the variable, or a generation of any part of the variable, is changed by any statement within the scope of the prefix.

   Specifically, if the identifier ID represents the variable, the condition is raised in the following cases:

   a. ID appears on the left-hand side of an assignment statement. (This applies to BY NAME assignment even if the name mentioned does not appear in the final expansion of the statement.)

   b. ID is set as a result of a pseudovariable appearing on the left-hand side of an assignment statement.

   c. ID appears as the control variable of a DO-group or a repetitive specification in a data list (or it is set as a result of a pseudovariable appearing as the control variable of a DO-group or a repetitive specification in a data list).

   d. ID appears in the data list of an edit-directed or list-directed GET statement.

   e. ID is altered by data-directed input.

   f. ID appears in the REPLY option of a DISPLAY statement.

   g. ID appears in the STRING option of a PUT statement.

   h. ID is passed as an argument to a programmer-defined procedure, no intermediate argument is created, and the procedure terminates with a RETURN or END.

   i. ID appears in the KEYTO or INTO option of a READ statement. Note that if the READ statement has an EVENT option, the CHECK condition will not be raised.

   j. ID is a pointer variable and appears in a SET option.

   Note that in a, b, d, and e above, if ID is a structure, the CHECK condition is raised each time an element of that structure is given a value, but the interrupt for each condition does not occur until after the statement that caused the condition to be raised has been executed completely.

The condition is not raised under any of the following circumstances:

   a. If the value of a variable defined on ID or on part of ID changes in any of the ways described above.

   b. If the parameter that represents the argument ID changes value.

   c. If ID appears in a GO TO or RETURN statement or any statement that involves the execution of a GO TO or RETURN statement.

   d. If ID is set by the INITIAL attribute.

   Note that in all of the above con-

texts, ID can appear in subscripted or qualified form. Note also that ID need not appear in the name list of a CHECK prefix; it only need represent a structure or element contained by, or containing, a name in the list.

The interrupt for a CHECK condition occurs after the statement that caused the condition to be raised has been executed. (Note that an IF statement is considered executed just prior to the execution of the THEN or ELSE clause.) If the statement is a DO statement, the interrupt occurs each time control proceeds sequentially to the statement following the DO statement. If the DO specifies repetitive execution, the interrupt occurs each time the control variable changes value.

Only a data-directed GET statement or a DO statement can cause a condition to be raised more than once for the same appearance of the same name. If a statement causes a CHECK condition to be raised for several names, the conditions will be raised in the left-to-right order of appearance of the names.

3. If a name in the CHECK prefix is an entry name, the condition is raised and the interrupt occurs prior to each invocation of the entry point corresponding to the entry name. The condition is raised only if the entry point is invoked by the entry name given in the prefix.

Result: When CHECK is raised, there is no effect on the statement being executed.

Standard System Action: In the absence of an on-unit, if the name in the name list is a statement-label constant, a statement-label variable, a task name, an event name, an area variable, a locator variable, or an entry name, then for the F Compiler, only the name is printed on SYSPRINT; in all other cases, the name and its new value are printed on SYSPRINT in the format of data-directed output.

Note: Standard system action for the CHECK condition requires access to the variable; consequently, if SIGNAL CHECK is given for an unallocated variable, an error will result, as it would if the variable were accessed by an on-unit.

Status: CHECK is disabled by default and within the scope of a NOCHECK condition prefix. It is enabled only within the scope of a CHECK prefix.

Normal Return: Upon the normal completion of the on-unit for the CHECK condition, execution continues immediately following the point at which the interrupt occurred.

The SUBSCRIPTRANGE Condition

Description: SUBSCRIPTRANGE can be raised whenever a subscript is evaluated and found to lie outside its specified bounds. If more than one subscript is associated with an identifier, e.g., A(I,J,K), SUBSCRIPTRANGE is raised after each erroneous subscript has been checked. Thus, if both I and J in the above example were in error, SUBSCRIPTRANGE would be raised after I was evaluated and again after J was evaluated.

Result: When SUBSCRIPTRANGE has been raised, the value of the illegal subscript is undefined, and, hence, the reference is also undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: SUBSCRIPTRANGE is disabled by default and within the scope of a NOSUBSCRIPTRANGE condition prefix. It is enabled only within the scope of a SUBSCRIPTRANGE condition prefix.

Normal Return: Upon the normal completion of the on-unit for this condition, execution continues immediately following the point at which the condition occurred.

The STRINGRANGE Condition

Definition: The STRINGRANGE condition is raised whenever the lengths of the arguments to a SUBSTR reference fail to comply with the rules described for the SUBSTR built-in function. It is raised for each such reference.

Standard System Action: Execution continues as described for normal return.

Status: STRINGRANGE is disabled by default and within the scope of a NOSTRINGRANGE condition prefix. It is enabled only within the scope of a STRINGRANGE condition prefix.

Normal Return: On normal return from the on-unit, execution continues with a revised SUBSTR reference whose value is defined as follows:

Assuming that the length of the source string (after execution of the on-unit, if specified) is k, the starting point is i, and the length of the substring is j;

1. If i is greater than k the value is the null string.

2. If i is less than or equal to k, the value is that substring beginning at the mth character or bit of the source string 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]

  k-m+1
    [if j is not specified]
```

This means that the new arguments are forced within the limits.

The values of i and j are established before entry to the on-unit; they are not reevaluated on return from the on-unit.

SYSTEM ACTION CONDITIONS

The ERROR Condition

Description: The ERROR condition is raised under the following circumstances:

1. As a result of the standard system action for an ON-condition for which that action is to "print an error message and raise the ERROR condition"

2. As a result of an error (for which there is no ON-condition) occurring during program execution

3. As a result of a SIGNAL ERROR statement

Standard System Action: For the F Compiler, if the condition is raised in the major task, the FINISH condition is raised, and subsequently the major task is terminated. If the condition is raised in any other task, that task is terminated.

Status: ERROR is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit, the standard system action is taken.

The FINISH Condition

Description: The FINISH condition is raised during execution of a statement which would cause the termination of the major task of a PL/I program, that is, by a STOP statement in any task, or an EXIT statement in the major task, or a RETURN or END statement in the initial external procedure of the major task. The condition is also raised by SIGNAL FINISH in any task, and as part of the standard system action for the ERROR condition. The interrupt occurs in the task in which the statement is executed, and any on-unit specified for the condition is executed as part of that task. An abnormal return from the on-unit will avoid any subsequent task termination processes and permit the interrupted task to continue.

Standard System Action: In the absence of an on-unit, no action is taken; that is, execution of the interrupted statement is resumed.

Status: FINISH is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit, execution of the interrupted statement is resumed.

PROGRAMMER-NAMED CONDITION

The CONDITION Condition

General Format: CONDITION (identifier)

The "identifier" must be specified by the programmer. The appearance of an identifier with CONDITION in an ON, SIGNAL, or REVERT statement constitutes a contextual declaration for it; the identifier is given the EXTERNAL attribute.

Description: CONDITION is raised by a SIGNAL statement that specifies the appropriate identifier. The identifier specified in the SIGNAL statement determines which CONDITION condition is to be raised.

Standard System Action: In the absence of an on-unit for this condition, the system prints a message and continues with the statement following SIGNAL.

Status: CONDITION is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit, execution continues with the statement following the SIGNAL statement that caused the interrupt.

A name appearing in a PL/I program may have one of many different meanings. It may, for example, be a variable referring to arithmetic data items; it may be a file name; it may be a variable referring to a character string, or it may be a statement label or a variable referring to a statement label.

Properties, or characteristics, of the values a name represents (for example, arithmetic characteristics of data items represented by an arithmetic variable) and other properties of the name itself (such as scope, storage class, etc.) together make up the set of attributes that can be associated with a name.

The attributes enable the compiler to assign a unique meaning to the identifier specified in a DECLARE statement. For example, if the variable is an arithmetic data variable, the base, scale, mode, and precision attributes must be associated with the name. Associated attributes are those specified in a DECLARE statement or assumed by default.

This section discusses the different attributes. The attributes are grouped by function and then detailed discussions follow, in alphabetic order, showing the rules, defaults, and format for each attribute.

SPECIFICATION OF ATTRIBUTES

Attributes specified in DECLARE statements are separated by blanks. Except for the dimension, length, and precision attribute specifications, they may appear in any order. The dimension attribute specification must immediately follow the array name; the length and precision attribute specifications must follow one of their associated attributes. A comma must follow the last attribute specification for a particular name (or the name itself if no attributes are specified with it), unless it is the last name in the DECLARE statement, in which case the semicolon is used.

FACTORING OF ATTRIBUTES

Attributes common to several names can be factored in a declaration to eliminate repeated specification of the same attribute for many identifiers. Factoring is achieved by enclosing the names in parentheses, and following this by the set of attributes which apply. All factored attributes must apply to all of the names. No factored attribute can be overridden for any of the names, but any name within the list may be given other attributes so long as there is no conflict with the factored attributes. Factoring of attributes is permitted only in the DECLARE statement, but not within an ENTRY attribute declaration. The dimension attribute may be factored. The precision and length attributes can be factored only in conjunction with an associated keyword attribute. Factoring can be nested as shown in the fourth example below.

Names within the parenthesized list are separated by commas.

Note: Structure level numbers can also be factored, but a factored level number must precede the parenthesized list.

```
DECLARE (A,B,C,D) BINARY FIXED (31);

DECLARE (E DECIMAL(6,5)
         F CHARACTER(10)) STATIC;

DECLARE 1 A, 2(B,C,D) (3,2) BINARY
          FIXED (15), ...;

DECLARE ((A,B) FIXED(10), C FLOAT(5))
          EXTERNAL;
```

DATA ATTRIBUTES

PROBLEM DATA

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

BINARY|DECIMAL

FIXED|FLOAT

REAL|COMPLEX

(precision)

PICTURE

The string data attributes are:

BIT|CHARACTER

(length)

VARYING

PICTURE

Other attributes can also be declared for data variables. The INITIAL attribute specifies the initial value to be given to the variable. The DEFINED attribute specifies that the data item is to occupy the same storage area as that assigned to other data. The ALIGNED and UNALIGNED attributes specify the positioning of data elements in storage. The storage class and scope attributes also apply to data.

Other attributes apply only to data aggregates. For array variables, the dimension attribute specifies the number of dimensions and the bounds of an array. The LIKE attribute specifies that the structure variable being declared is to have the same structuring as the structure of the name following the attribute LIKE.

PROGRAM CONTROL DATA

Attributes for program control data specify that the associated name is to be used by the programmer to control the execution of this program. The LABEL, TASK, EVENT, POINTER, OFFSET, and AREA attributes specify program control data.

ENTRY NAME ATTRIBUTES

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

ENTRY

RETURNS

GENERIC

BUILTIN

FILE DESCRIPTION ATTRIBUTES

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

The file description attributes are:

FILE

STREAM|RECORD

INPUT|OUTPUT|UPDATE

PRINT

SEQUENTIAL|DIRECT

BUFFERED|UNBUFFERED

BACKWARDS

ENVIRONMENT(option-list)

KEYED

EXCLUSIVE

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

SCOPE ATTRIBUTES

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

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

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

The INTERNAL attribute specifies that the declared name cannot be known in any other block except those contained in the block in which the declaration is made.

The same identifier may be declared with the INTERNAL attribute in more than one block without regard to whether the attributes given in one block are consistent with the attributes given in another block, since the compiler regards such declarations as referring to different names.

For a discussion of the scope of names, see Part I, Chapter 7, "Recognition of Names."

STORAGE CLASS ATTRIBUTES

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

STATIC

AUTOMATIC

CONTROLLED

BASED

ALPHABETIC LIST OF ATTRIBUTES

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

ABNORMAL and NORMAL

These attributes cause no action in the current version of the F Compiler; they are accepted by the compiler but they have no effect.

ALIGNED and UNALIGNED (Data Attributes)

The ALIGNED and UNALIGNED attributes specify the positioning of data elements in storage, to influence speed of access or storage economy respectively. They may be specified for element, array, or structure variables.

ALIGNED in System/360 implementations specifies that the data element is to be aligned on the storage boundary corresponding to its data type requirement.

UNALIGNED in System/360 implementations specifies that the data element is to be stored contiguously with the data element preceding it, and that a word or doubleword item is to be mapped on the next available byte boundary in a similar manner to character strings of length 4 or 8.

General format:

ALIGNED|UNALIGNED

General rules:

1. Although they are essentially element data attributes, ALIGNED and UNALIGNED can be applied to any array or structure. This is equivalent to applying the attribute to all contained elements that are not explicitly declared with the ALIGNED or UNALIGNED attribute.

2. Application of either attribute to a contained array or structure overrides an ALIGNED or UNALIGNED attribute that otherwise would apply to elements of the contained aggregate by having been specified for the containing structure.

3. The LIKE attribute is expanded before the ALIGNED and UNALIGNED attributes are applied to the contained elements of the LIKE structure variable. The only ALIGNED and UNALIGNED attributes that are carried over from the LIKE structure variable (i.e., A in the example below) are those explicitly specified for substructures and elements of the structure variable.

Example:

```
DECLARE 1 A ALIGNED,
  2 B,                  /* ALIGNED FROM A */
  2 C UNALIGNED,
    3 D;                /* UNALIGNED FROM C */
```

```
DECLARE 1 X UNALIGNED LIKE A;

DECLARE 1 Y LIKE A;
```

The second declare statement is equivalent to:

```
DECLARE 1 X UNALIGNED,
  2 B,              /* UNALIGNED FROM X */
  2 C UNALIGNED,
    3 D;            /* UNALIGNED FROM C */
```

The third declare statement is equivalent to:

```
DECLARE 1 Y,
  2 B,              /* ALIGNED BY DEFAULT */
  2 C UNALIGNED,
    3 D;            /* UNALIGNED FROM C */
```

4. For overlay defining involving bit- and character-class data (see Figure I-1), both the defined item and the overlaid part of the base item must be unaligned. For all other types of defining, equivalent items must be either both ALIGNED or both UNALIGNED.

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

6. If a based variable is used to refer to a generation of another variable, the ALIGNED and UNALIGNED attributes of both variables must agree.

7. Default assumptions for ALIGNED and UNALIGNED are applied on an element basis.

8. POINTER, OFFSET, LABEL, EVENT and AREA cannot be unaligned.

Assumptions:

1. Defaults are applied at element level. The default for bit-string data, character-string data, and numeric character data is UNALIGNED; for all other types of data, the default is ALIGNED.

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

3. Constants take the default for ALIGNED or UNALIGNED.

AREA (Program Control Data Attribute)

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

General format:

```
AREA [(size)]
```

General rules:

1. The area size for areas that are not of static storage class is given by an expression whose integral value specifies the number of units of storage to be reserved. The unit for System/360 implementations is the byte.

2. The size for areas of static storage class must be specified as a constant; for the F Compiler, it must be a decimal integer constant.

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

4. No operators can be applied to area variables.

5. Only the INITIAL CALL form of the INITIAL attribute is allowed with area variables.

6. An area variable cannot be unaligned.

Assumptions:

1. If the size specification is omitted, a default value is assumed. For the F Compiler, this is 1000.

2. An area variable can be contextually declared by its appearance in an OFFSET attribute or an IN option. Note, however, that all contextually declared area variables are given the AUTOMATIC attribute. The F Compiler implementation requires that the variable named in the OFFSET attribute must be based; if a nonbased area variable is named, the offset variable will be changed to a pointer variable. Hence, unless the variable named in the OFFSET attribute is explicitly declared, OFFSET effectively becomes POINTER, and a severe error occurs.

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

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

AUTOMATIC specifies that storage is to be allocated upon each entry to the block to which the storage declaration is internal. The storage is released upon exit from the block. If the block is a procedure that is invoked recursively, the previously allocated storage is "pushed down" upon entry; the latest allocation of storage is "popped up" upon termination of each generation of the recursive procedure (for a discussion of push-down and pop-up stacking, see Part I, Chapter 6, "Blocks, Flow of Control, and Storage Allocation").

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

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

BASED, like CONTROLLED, specifies that full control over storage allocation and freeing will be maintained by the programmer, but by various methods that are described in Chapter 14, "Based Storage and List Processing." Multiple allocations are not stacked but are available at any time; each can be identified by the value of a pointer variable.

General format:

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

General rules:

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

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

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

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

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

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

   a. The pointer variable named in the BASED attribute must be a non-based, unsubscripted, element pointer variable. This applies to explicit pointer qualifiers also.

   b. Whenever a pointer value is needed to complete a based variable reference, and none is explicitly specified, the pointer variable named in the relevant BASED attribute is used.

   c. Based variables cannot have the INITIAL attribute. Based label arrays cannot be initialized by subscripted label prefixes.

   d. When reference is made to a based variable, the data attributes assumed are those of the based variable, while the qualifying pointer variable identifies the location of data.

   e. A based variable can be used to identify and describe existing data; to obtain storage by means of the ALLOCATE statement; or to obtain storage in an output buffer by means of the LOCATE statement.

   f. The relative locations of based variables allocated within an area can be identified by the values of offset variables, but these must be assigned to pointer variables for the purpose of explicit qualification.

   g. The EXTERNAL attribute cannot appear with a based variable declaration, but a based variable reference can be qualified by an external pointer variable.

   h. A based structure can be declared to contain only one adjustable bound or length specification. See "The REFER Option," in Chapter

14, "Based Storage and List Processing."

i. Based variables cannot be transmitted using data-directed input/output.

j. The VARYING attribute cannot be applied to based variables.

Assumptions:

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

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

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

4. A pointer variable can be contextually declared by its appearance in the BASED attribute.

BACKWARDS (File Description Attribute)

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

General format:

BACKWARDS

General rules:

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

2. The BACKWARDS attribute applies to tape files only.

BASED (Storage Class Attibute)

See AUTOMATIC.

BINARY and DECIMAL (Arithmetic Data Attributes)

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

General format:

BINARY|DECIMAL

General rule:

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

Assumptions:

Undeclared identifiers (or identifiers declared only with one or more of the dimensions, UNALIGNED, ALIGNED, scope, and storage class attributes) are assumed to be arithmetic variables with assigned attributes depending upon the initial letter. For identifiers beginning with any letter I through N, the default attributes are REAL FIXED BINARY (15,0). For identifiers beginning with any other alphabetic character, the default attributes are REAL FLOAT DECIMAL (6). If FIXED or FLOAT and/or REAL or COMPLEX are declared, then DECIMAL is assumed. The default precisions are those defined for System/360 implementations.

BIT and CHARACTER (String Attributes)

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

General format:

BIT
(length) [VARYING]
CHARACTER

General rules:

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

2. The VARYING attribute specifies that the variable is to represent varying-length strings, in which case length specifies the maximum length. The current length at any time is the length of the current value. For the

F Compiler, the length of an uninitialized varying-length string is set to zero. VARYING may appear anywhere in the declaration of the string, and it may be factored. VARYING cannot be applied to based variables.

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

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

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

   The asterisk notation can be used for the length attribute specification to indicate that the length is specified elsewhere. For parameters or CONTROLLED variables, the length can be taken from a previous allocation or, for CONTROLLED variables, it can be specified in a subsequent ALLOCATE statement.

   Only one adjustable string length specification can appear in the declaration of a based structure. See "The REFER Option", in Chapter 14.

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

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

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

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

BUFFERED and UNBUFFERED (File Description Attributes)

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

The UNBUFFERED attribute specifies that such records need not pass through buffers. It does not, however, specify that they must not. For the F Compiler, hidden buffers will, in fact, be used if INDEXED or REGIONAL (2) or (3) is specified in the ENVIRONMENT attribute or if the records are variable-length.

General format:

```
BUFFERED|UNBUFFERED
```

General rule:

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

Assumption:

Default is BUFFERED.

BUILTIN (Entry Attribute)

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

General format:

```
BUILTIN
```

General rules:

1. BUILTIN is used to refer to a built-in function or pseudo-variable in a block that is contained in another block in which the same identifier has been declared to have another meaning.
2. If the BUILTIN attribute is declared for an entry name, the entry name can have no other attributes.
3. The BUILTIN attribute cannot be declared for parameters.

CHARACTER (String Attribute)

See BIT.

OMPLEX and REAL (Arithmetic Data Attributes)

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

General format:

```
REAL|COMPLEX
```

General rule:

If a numeric character variable is to represent complex values, the COMPLEX attribute must be specified with the PICTURE attribute. The COMPLEX attribute is the only other arithmetic or string data attribute that can be specified with the PICTURE attribute.

Assumption:

Default is REAL.

CONTROLLED (Storage Class Attribute)

See AUTOMATIC.

DECIMAL (Arithmetic Data Attribute)

See BINARY.

DEFINED (Data Attribute)

The DEFINED attribute specifies that the variable being declared is to represent part or all of the same storage as that assigned to other data. The DEFINED attribute can be declared for element, array, or structure variables.

General format:

```
DEFINED base-identifier
   {[subscript-list]|[POSITION
     (decimal-integer-constant)]}
```

The "base identifier" is an unsubscripted, optionally qualified variable whose storage is also to be represented by the variable being declared. The "subscript list" is a specification used to determine the portion of a base identifier array that the currently declared variable will represent. POSITION is discussed under the rules for overlay defining.

Rules for defining:

1. The INITIAL, storage class, and scope attributes cannot be specified for the defined item. The defined item must be a level 1 variable and it cannot be a parameter. The VARYING attribute must not be specified for either the defined item or the base identifier. It should be noted that although the base can have the EXTERNAL attribute, the defined item always has the INTERNAL attribute and cannot be declared with any scope attribute. If the base is 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 any block.

2. The base identifier must always be known within the block in which the defined item is declared. The base identifier cannot have the DEFINED attribute. It can represent a minor structure. The current F Compiler does not allow the base identifier to be controlled or based.

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 attributes. The lengths need not be the same; however, the length of the defined item must not be greater than the length of the base item. The current F Compiler does not allow correspondence defining for arrays of structures.

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.

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 in the DEFINED attribute specification, each subscript can 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 number of dimensions of the defined item. Thus, 1SUB represents subscripts of the first dimension of the defined array, 2SUB represents the second dimension of the defined array, and so forth. The subscript list following the name of the base array in the DEFINED attribute specification must contain the same number of subscript expressions as there are dimensions of the base array.

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 array can be passed, and an equivalent array can be defined on the corresponding parameter.

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.

The bounds of a defined array must be within the bounds of the base array.

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 items shown in Figure I-1.

Rules for overlay defining:

1. For bit and character class data, 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; that is, the defined item is to begin at the first position of the base.

2. For bit and character class data, 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).

Defined Item	Base Identifier
A coded arithmetic element variable	An unsubscripted coded arithmetic element variable of the same base, scale, mode, and precision
An element label variable	An unsubscripted element label variable
An element event variable	An unsubscripted element event variable
An element task variable	An unsubscripted element task variable
An element pointer variable	An unsubscripted element pointer variable
An element offset variable	An unsubscripted element offset variable
An element area variable	An unsubscripted element area variable
A bit class[1] variable	Bit class[1] data that is neither a cross section of an array nor an array within an array of structures
A character class[2] variable	Character class[2] data that is neither a cross section of an array nor an array within an array of structures
A structure	An identical structure whose makeup is such that matching pairs of items from the structures are valid examples for overlay defining of coded arithmetic, label, task, event, area, offset, and pointer element variables. The elements can also be strings or numeric character data items of matching lengths.

[1]The bit class consists of:
- a. Fixed-length bit strings
- b. Packed structures consisting of items a or c
- c. Packed arrays consisting of items a or b

[2]The character class consists of:
- a. Numeric character data
- b. Fixed-length character strings
- c. Packed structures consisting of items a, b, or d
- d. Packed arrays consisting of items a, b, or c

Figure I-1. Permissible Items for Overlay Defining

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 in the subscript list following the base identifier are evaluated when a reference to the defined item is made.

Examples of Defining

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

In this example of correspondence defining, B is a vector consisting of every even element in the diagonal of the array 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;
```

In this example of overlay defining, PSTRING1 is a character string that represents the concatenation of the two character strings Q and R, which are elements of the structure P. Note that P has the PACKED attribute by default.

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 this example of overlay defining, ALIST refers to the first ten characters of LIST, BLIST refers to the twenty-first through fortieth characters of LIST, and CLIST refers to the eleventh through twentieth characters of LIST.

4.
```
DECLARE 1 A,
          2 B FIXED,
          2 C FLOAT,
        1 X DEFINED A,
          2 Y FIXED,
          2 Z FLOAT;
```

In this example of overlay defining, Y refers to B and Z refers to C.

Note: Although the language rules specify that the attributes (except for length) of the defined item must exactly match the attributes of the base item, the F Compiler allows a programmer to make an exception to this rule, under certain circumstances.

If attributes declared for the defined item differ from those of the base identifier, the compiler notes this with a message at the ERROR level. If, however, the error code of the EXECUTE job control statement of the following step is high enough, linkage editing and execution of the compiled procedure can continue. For example:

```
DECLARE A FIXED BINARY(31),
        B BIT (32) DEFINED A;
```

Compilation of this DECLARE statement would cause an error message to be issued by the compiler. However, execution of the program could be successful, and arithmetic operations performed upon A would result in the change of value of the bit-string variable B.

Dimension (Array Attribute)

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

General format:

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

where "bound" is:

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

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

General rules:

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

2. The bounds specification indicates the bounds as follows:

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

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

   c. If asterisk notation is used, an asterisk must be used for each bounds specification of the array. An asterisk specifies that the actual bounds are to be specified in an ALLOCATE statement, if the variable is CONTROLLED, or in a declaration of an associated argument, if the variable is a simple parameter. Thus, the asterisk notation can be used only for parameters and CONTROLLED variables.

3. Bounds that are expressions are evaluated and converted to integer data -- for System/360 implementations, BINARY(15) -- when storage is allocated for the array. For dummy arguments that are arrays, the bounds are determined at invocation of the block containing the ENTRY attribute. For simple parameters, bounds can be only optionally signed decimal integer constants or asterisks.

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

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

6. If the asterisk notation is used to declare dimensions of an array of structures, all dimension declarations within the major structure must also be asterisks.

7. Only one adjustable array bound specification can appear in the declaration of a based structure. See "The REFER Option" in Chapter 14.

DIRECT and SEQUENTIAL (File Description Attributes)

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

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

General format:

```
SEQUENTIAL|DIRECT
```

General rules:

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

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

Assumptions:

1. Default is SEQUENTIAL for RECORD files.

2. If a file is implicitly opened by an UNLOCK statement, DIRECT is assumed.

ENTRY Attribute

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

General format:

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

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

General rules:

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

2. The ENTRY attribute must be specified for any entry name that is declared elsewhere and not recognized as such within the block if any reference is made to that entry name (such as in an argument list) unless, within the block:

   a. The entry name appears in a CALL statement or a function reference with an argument list, either of which constitutes a contextual declaration of the ENTRY attribute, or

   b. The entry name is declared to have any of the attributes REDUCIBLE, IRREDUCIBLE, SETS, USES, BUILTIN, and RETURNS, all of which (except BUILTIN) imply ENTRY. The ENTRY attribute cannot be specified for a name that is given the BUILTIN or GENERIC attributes.

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

4. Expressions used for length or bounds in an ENTRY attribute specification for non-CONTROLLED parameters are evaluated upon entry to the block to

which the declaration of the ENTRY attribute is internal.

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

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

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

Assumptions:

The ENTRY attribute can be assumed either contextually or by implication, as described in rule 2. The appearance of a name as a label prefix of either a PROCEDURE statement or an ENTRY statement constitutes an explicit declaration of that identifier as an entry name. No defaults are applied for parameters unless attributes and/or level numbers are specified. If only a level number and/or the dimension attribute is specified for a parameter, FLOAT, DECIMAL, and REAL are assumed.

ENVIRONMENT (File Description Attribute)

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

General format:

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

Each option in the "option list" is separated by one or more blanks. The option list is defined individually for each implementation of PL/I. For the F Compiler, it is as follows:

```
[record-format] [BUFFERS(n)]
[data-set-organization]
[volume-disposition] [carriage-control]
[COBOL] [data-management-optimization]
[key-classification]
```

The options may appear in any order.

General rules:

1. The ENVIRONMENT attribute must be included in a DECLARE statement. It cannot be specified as an option of an OPEN statement.

2. The "record format" describes the format of the records to be written or retrieved. The record format specification is as follows:

```
{F(block-size[,record-size])
 V(max-block-size[,max-record-size])
 V{S|BS} (max-block-size
    [,max-record-size])
 U(max-block-size)}
```

F(block-size[,record-size]) specifies fixed-length records with the block size stated in bytes. If the record size is specified (also in number of bytes), it indicates that records are blocked, that is, that each physical record contains more than one logical record. In such cases, the block size must be a simple multiple of the record size. If the record size is not specified, then logical record size is the same as physical record size.

V(max-block-size[,max-record-size]) specifies variable-length records that can be contained within the stated maximum block size. One record is allotted to each block unless the maximum record size is stated, in which case the records are blocked, i.e., more than one record may be put into a block depending on the space available within the block.

V{S|BS} (max-block-size[,max-record-size]) specifies variable-length records that may exceed the maximum block size. If a record exceeds maximum block size, the excess part is placed in the next block or blocks. When VS is specified, there is never more than one record or segment of a record in one block. When VBS is specified, the records are blocked so that a block may contain one or more records or segments of a record. Stating the maximum record size in the VS format or in the VBS format does not affect blocking.

Four bytes of control information per block, plus four bytes per logical record or segment of logical record, are included automatically by the system for variable-length records. These control bytes must be included in the count of maximum block size and maximum record size.

U(max-block-size) specifies records of undefined length up to the maximum stated. No control information is included, since records are not blocked, and logical record size is the same as physical record size.

Neither record size nor block size can exceed 32,760 bytes.

3. The BUFFERS(n) option specifies the number of buffers to be allocated for the data set; this number (n), which is specified by a decimal integer constant, must not exceed 255. For BUFFERED files, one or two buffers are automatically allocated, depending on the access method, unless a greater number is indicated in the BUFFERS(n) option. For UNBUFFERED files that require hidden buffers, one buffer is automatically allocated. If not specified in the ENVIRONMENT option, the buffer count can be specified in the BUFNO subparameter of the associated DD statement.

4. The "data set organization" describes some physical characteristics of the data set and how records are to be written or retrieved. Data set organization is specified by one of the following:

   CONSECUTIVE

   INDEXED

   REGIONAL(1)

   REGIONAL(2)

   REGIONAL(3)

   CONSECUTIVE describes a data set consisting of unkeyed records that are to be written or retrieved in a physically sequential order. This organization is assumed if none is specified. Note the difference between CONSECUTIVE and SEQUENTIAL. CONSECUTIVE specifies physical characteristics of the data set; SEQUENTIAL has no such connotation. A file declared SEQUENTIAL can have any of the five data set organization options.

   INDEXED describes an indexed sequential data set that consists of keyed records, any one of which can be located by means of several levels of indexes.

   REGIONAL (1) describes a data set that consists of records without recorded keys but which can be located by means of a source key that specifies a relative record position within the data set.

   REGIONAL (2) describes a data set that consists of records with recorded keys. A source key specifies the relative record and the recorded key. A search for the record with the specified recorded key starts at the beginning of the track on which the relative record resides.

   REGIONAL (3) describes a data set that consists of records with recorded keys. A source key specifies the relative track and the recorded key. The search is made the same as with REGIONAL (2) except that the search for the record with the specified recorded key is to begin at the relative track indicated.

5. The "volume-disposition" specifies the action to be taken when the end of a magnetic tape volume is reached during access to a data set or when a data set on a magnetic tape volume is closed normally or abnormally. Volume disposition is specified by one of the following ENVIRONMENT attribute options:

   LEAVE

   REWIND

   LEAVE specifies that no repositioning of the volume is to take place if the end of the volume has been reached. The channel can then be freed. If a data set is closed normally or abnormally, LEAVE specifies that the tape is to be positioned at the end of the data set or at the beginning of the data set if a BACKWARDS file is being used. If the data set continues on another volume, the tape is positioned at the end of the current volume or at the beginning if a BACKWARDS file is being used. The channel remains busy during the positioning operation.

   REWIND allows the end-of-volume or data-set-closure tape action to be controlled by the DISP field of the associated DD statement. If DISP=(status,DELETE) is specified in the DD statement, the tape is rewound but not unloaded. If DISP=(status,KEEP|CATLG|UNCATLG) is specified, the tape is rewound and unloaded. If DISP=(status,PASS) is specified, the tape is wound on to the end of the data set, unless a BACKWARDS file is being used, in which case the tape is repositioned at the beginning of the data set. When DISP=(status,PASS) is specified, the channel is kept busy when positioning; in the other two cases the channel is freed when positioning.

6. The "carriage control" specifies that the first character of a record is to be interpreted as a carriage control character. The carriage control options are:

   CTLASA, which specifies that the first character of a record is to be interpreted as an ASA standard carriage control character, and

   CTL360, which specifies that the first character of a record is to be interpreted as an IBM System/360 machine code carriage control character.

7. The COBOL option specifies that the file will contain structures mapped according to the COBOL (F) algorithm. This type of file can be used only with READ INTO and WRITE FROM statements.

8. The "data management optimization" increases program efficiency, in certain circumstances, when DIRECT INDEXED data sets are to be accessed. The data management optimization options are:

   INDEXAREA[(index-area-size)]

   NOWRITE

   INDEXAREA[(index-area-size)] improves the input/output speed of a DIRECT INPUT or DIRECT UPDATE file with INDEXED data set organization, by having the highest level of index placed in main storage. The "index area size," when specified, must be a decimal integer constant whose value lies within the range zero through 32,767. If an index area size is not specified, the highest level index is moved unconditionally into main storage. If an index area size is specified, the highest level index is held in main storage, provided that its size does not exceed that specified. If the specified size is less than zero or greater than 32,767, the compiler issues a warning message and ignores the parameter of the option.

   NOWRITE can be specified only for DIRECT UPDATE files with INDEXED data set organization. It informs the compiler that no records are to be added to the data set and that data management modules concerned solely with adding records are not required; it thus allows the size of the compiled program to be reduced.

9. The "key classification" option GENKEY (generic key), applies only to INDEXED data sets. It enables the programmer to classify keys recorded in a data set and to use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.

   A generic key is a character string that identifies a class of keys: all keys that begin with the string are members of that class. For example, the recorded keys 'ABCD', 'ABCE', and 'ABDF' are all members of the classes identified by the generic keys 'A' and 'AB', and the first two are also members of the class 'ABC'; and the three recorded keys can be considered to be unique members of the classes 'ABCD', 'ABCE', and 'ABDF', respectively.

   The GENKEY option allows the programmer to start sequential reading or updating of an INDEXED data set from the first non-dummy record that has a key in a particular class; the class is identified by the inclusion of its generic key in the KEY option of a READ statement. Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

   If the data set contains no records with keys in the specified class, or if all the records with keys in the specified class are dummy records, the KEY condition is raised and the data set is positioned to read the first record.

   The GENKEY option affects the execution of a READ statement that supplies a source key shorter than the key length specified in the KEYLEN subparameter of the DD statement that defines the data set. GENKEY causes the key to be interpreted as a generic key, and the data set is positioned to the first non-dummy record in the data set whose key begins with the source key. If GENKEY is not specified, a short source key is padded on the right with blanks to the specified key length, and the data set is positioned to the record that has this padded key(if such a record exists).

   The use of the GENKEY option does not affect the result of supplying a source key whose length is greater than or equal to the specified key length. The source key, truncated on the right if necessary, identifies a specific record (whose key can be considered to be the only member of its class).

Assumptions:

CONSECUTIVE data set organization is assumed unless stated otherwise. Tape reels are rewound unless the LEAVE option is specified. If the BUFFERS(n) option is not specified, two buffers are allocated for BUFFERED files, and one is allocated for UNBUFFERED files that require hidden buffers.

EVENT (Program Control Data Attribute)

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

General format:

EVENT

General rules:

1. An identifier may be explicitly declared with the EVENT attribute in a DECLARE statement. It may be contextually declared by its appearance in an EVENT option of a CALL statement, in a WAIT statement, in a DISPLAY statement, or in various input/output statements (see Chapter 8, "Input and Output," and Chapter 15, "Multitasking.")

2. Event names may also have the following attributes:

   Dimension

   Scope (the default is INTERNAL)

   Storage class (the default is AUTOMATIC)

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

3. An event variable has two separate values:

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

   b. A fixed-point value of default precision ((15,0) for the F Compiler) which reflects the status value of the variable. A zero value indicates normal, nonzero indicates abnormal status.

   The values of the event variable can be separately returned by use of the COMPLETION and STATUS built-in functions. The COMPLETION function returns a bit-string value corresponding to the completion value of the variable; STATUS returns a fixed binary value corresponding to the status value.

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

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

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

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

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

   c. Event variable assignment.

   d. By a statement with the EVENT option.

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

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

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

6. On allocation of an event variable, its status and completion values are undefined.

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

with another event, or to modify the completion value of an active event variable by event variable assignment or by use of the COMPLETION pseudo-variable.

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

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

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

11. Event variables cannot be unaligned.

EXCLUSIVE (File Description Attribute)

The EXCLUSIVE attribute specifies that records in a DIRECT UPDATE file may be locked by an accessing task to prevent other tasks from interfering with an operation. The section entitled "The EXCLUSIVE Attribute," in Chapter 15, "Multitasking," contains a table showing the effects of various operations on EXCLUSIVE files and the records contained in them.

General format:

```
EXCLUSIVE
```

General rules:

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

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

3. A DELETE or REWRITE statement referring to a locked record will automatically unlock the record at the end of the DELETE or REWRITE operation; if the record has been locked by another task, the task executing the DELETE or REWRITE statement will wait until the record is unlocked. While a DELETE or REWRITE operation is taking place, the record is always locked.

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

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

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

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

Assumptions:

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

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

EXTERNAL and INTERNAL (Scope Attributes)

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

General format:

```
EXTERNAL|INTERNAL
```

Assumptions:

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

FILE (File Description Attribute)

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

General format:

```
FILE
```

Assumptions:

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

FIXED and FLOAT (Arithmetic Data Attributes)

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

General format:

```
FIXED|FLOAT
```

General rule:

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

Assumptions:

Undeclared identifiers (or identifiers declared only with one or more of the dimension, PACKED, ALIGNED, scope, and storage class attributes) are assumed to be arithmetic variables with assigned attributes depending upon the initial letter. For identifiers beginning with any letter I through N, the default attributes are REAL FIXED BINARY (15,0). For identifiers beginning with any other alphabetic character, the default attributes are REAL FLOAT DECIMAL (6). If BINARY or DECIMAL and/or REAL or COMPLEX are specified, FLOAT is assumed. The default precisions are those defined for System/360 implementations.

FLOAT (Arithmetic Data Attribute)

See FIXED.

GENERIC (Entry Name Attribute)

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

General format:

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

General rules:

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

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

   entry-name attribute-list

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

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

5. When a generic name is referred to, the attributes of the arguments must match exactly the list following the entry name declaration of one 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. Note that no conversion is done for arguments passed to generic functions. Consequently, the precision of a constant or any other expression must match the precision of a parameter.

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

   Base

   Scale

   Mode

   Precision

   PICTURE

   LABEL (but not label list)

   Number of dimensions (but not bounds)

   CHARACTER (but not length)

   BIT (but not length)

   VARYING

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

   FILE (but no other FILE attributes)

   ALIGNED

   PACKED

   AREA (but not size)

   OFFSET (but not specified area variable)

   POINTER

   TASK

   EVENT

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

INITIAL (Data Attribute)

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

General format:

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

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

General rule:

The INITIAL attribute cannot be given for entry names, file names, defined data, structures, parameters, or based variables.

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

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

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

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

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

   (iteration-factor) constant

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

   (iteration-factor) *

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

7. The iteration factor can be an element expression, except for STATIC data, in which case it must be an unsigned decimal integer constant. When storage is allocated for the array, the expression is evaluated to give an integer that specifies the number of iterations.

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

9. For initialization of a string array, if only one parenthesized element expression precedes the string initial value, the expression is interpreted to be a string repetition factor for the string; that is, it is interpreted as a part of the specification of the value for a single element of the array. Consequently, for an expression to cause initialization of more than one element of a string array, both the string repetition factor and the iteration factor must be explicitly stated, even if the string repetition factor is (1). For example, consider the following:

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

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

10. 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. STATIC label variables cannot have the INITIAL attribute.

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. The effect of this appearance is the initialization of that array element to a value that is a constructed label constant for the statement prefixed with the subscripted reference. This statement must be internal to the block containing the declaration of the array. Only one form of initialization can be used for a given label array. If CHECK is specified for a label array and the elements of the label array are initialized by a label prefix, the CHECK condition is not raised at initialization.

13. For the F Compiler, character-string or bit-string data having the STATIC attribute cannot be initialized with complex values.

14 This form of the INITIAL attribute cannot be used in the declaration of locator or area variables.

Rules for form 2:

1. The "entry name" and "argument list" passed must satisfy the condition stated for prologues as discussed in Part I, Chapter 6, "Blocks and Flow of Control."

2. Form 2 cannot be used to initialize STATIC data.

Examples:

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

b.
```
DECLARE MAXVALUE INITIAL (99),
        MINVALUE INITIAL (-99);
```

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

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

e.
```
DECLARE 1 A(8),
          2 B INITIAL (0),
          2 C INITIAL ((8)0);
```

f.
```
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;
```

Example c 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 Example d, 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 Example e, B and C inherit a dimension of (8) but, whereas only the first element of B is initialized, all the elements of C are initialized.

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

INPUT, OUTPUT, and UPDATE (File Description Attributes)

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

General format:

```
INPUT|OUTPUT|UPDATE
```

General rules:

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

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

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

Assumptions:

Default is INPUT. The PRINT attribute implies OUTPUT. The EXCLUSIVE attribute implies UPDATE.

The following assumptions are made when a file is implicitly opened by an input/output statement:

WRITE, PUT	OUTPUT
READ, GET	INPUT
DELETE, REWRITE, UNLOCK	UPDATE

INTERNAL (Scope Attribute)

See EXTERNAL.

IRREDUCIBLE and REDUCIBLE

These attributes cause no action in the current version of the F Compiler other than to imply the ENTRY attribute.

KEYED (File Description Attribute)

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

General format:

```
KEYED
```

General rules:

1. A KEYED file cannot have the attributes STREAM or PRINT.
2. The KEYED attribute can be specified for RECORD files only, and must be associated with direct access devices.
3. The KEYED attribute must be specified for every file with which any of the options KEY, KEYTO, and KEYFROM is used. It need not be specified if none of the options are to be used, even though the corresponding data set may actually contain recorded keys.

Assumption:

The DIRECT and EXCLUSIVE attributes imply KEYED.

LABEL (Program Control Data Attribute)

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

General format:

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

General rules:

1. If a list of statement label constants is given, the variable can have as values only members of the list. The label constants in the list must be known in the block containing the declaration.
2. If the variable is a parameter, its value can be any statement label variable or constant passed as an argument. If the argument is a label variable, the value of the label parameter can be any value permitted for the label variable that is passed.
3. An entry name cannot be a value of a label variable.
4. The parenthesized list of statement label constants can be used in a LABEL attribute specification for a label array. A subscripted label specifying an element of a label array can appear as a statement label prefix, if the label variable is not STATIC, but it cannot appear in an END statement after the keyword END. For further information, see general rule 12 in the discussion of the INITIAL attribute.
5. The INITIAL attribute cannot be specified for STATIC label variables.
6. Labels cannot be unaligned.

Length (String Attribute)

See BIT.

LIKE (Structure Attribute)

The LIKE attribute specifies that the name being declared is a structure variable with the same structuring as that for the name following the attribute keyword LIKE. Substructure names, elementary names, and attributes for substructure names and elementary names are to be identical.

General format:

```
LIKE structure-variable
```

General rules:

1. The "structure variable" can be a major structure name or a minor structure name. It can be a qualified name, but it cannot be subscripted.

2. The "structure variable" must be known in the block containing the LIKE attribute specification. The structure names in all LIKE attributes are associated with declared structures before any LIKE attributes are expanded. For example:

```
DECLARE 1 A, 2 C, 3 E, 3 F,
        1 D, 2 C, 3 G, 3 H;
  .
  .
  .
BEGIN;
  DECLARE 1 A LIKE D, 1 B LIKE A.C;
  .
  .
  .
END;
```

   These declarations result in the following:

   1 A LIKE D is expanded to give:

   1 A, 2 C, 3 G, 3 H

   1 B LIKE A.C is expanded to give:

   1 B, 3 E, 3 F

3. Neither the "structure variable" nor any of its substructures can be declared with the LIKE attribute, nor may the "structure variable" have been completed by the LIKE attribute.

4. Neither additional substructures nor elementary names can be added to the created structure; any level number that immediately follows the "structure variable" in the LIKE attribute specification in a DECLARE statement must be algebraically equal to or less than the level number of the name declared with the LIKE attribute.

5. Attributes of the "structure variable" itself do not carry over to the created structure. For example, storage class attributes do not carry over. If the "structure variable" following the keyword LIKE represents an array of structures, its dimension attribute is not carried over. Attributes of substructure names and elementary names, however, are carried over; contained dimension and length attributes are recomputed. An exception is that this does not apply to the INITIAL attribute for any elements of a label array that has been initialized by prefixing to a statement.

6. If a direct application of the description to the structure declared LIKE would cause an incorrect continuity of level numbers (for example, if a minor structure at level 3 were declared LIKE a major structure at level 1) the level numbers are modified by a constant before application.

7. The LIKE attribute is expanded before the ALIGNED and UNALIGNED attributes are applied to the contained elements of a structure.

NORMAL

See ABNORMAL.

OFFSET and POINTER (Program Control Data Attributes)

The OFFSET and POINTER attributes describe locator variables. A pointer variable can be used in a based variable reference to identify a particular allocation of the based variable. Offset variables identify a location relative to the start of an area; pointer variables identify any location, including those within areas.

General format:

```
POINTER|OFFSET (area-variable)
```

General rules:

1. A pointer variable can be explicitly declared in a DECLARE statement, or it can be contextually declared by its appearance as a pointer qualifier, by its appearance in a BASED attribute, or by its appearance in a SET option.

2. An offset variable must be explicitly declared.

3. The value of a pointer variable can be set in any of the following ways:

   a. With the SET option of a READ statement;

   b. By a LOCATE statement;

   c. By an ALLOCATE statement;

   d. By assignment of the value of another locator variable, or a

locator value returned by a user-defined function;

e. By assignment of an ADDR or NULL built-in function value.

4. The value of an offset variable can be set only by assignment of the value of another locator variable or the value of the NULLO built-in function.

5. Locator variables cannot be operands of any operators other than the comparison operators = and ¬=.

6. Locator data cannot be converted to any other data type, but pointer can be converted to offset, and vice versa.

7. A locator value can be assigned only to a locator variable. When an offset value is assigned to an offset variable, the area variables named in the OFFSET attributes are ignored.

8. Locator data cannot be transmitted using STREAM input/output.

9. Only the INITIAL CALL form of the INITIAL attribute is allowed in locator declarations.

10. Offset variables cannot be used to qualify a based reference.

11. For the F Compiler, the area variable named in an OFFSET attribute must be of based storage class.

12. Pointer variables and offset variables cannot be unaligned.

Assumption:

The variable named in the OFFSET attribute is contextually declared to have the AREA attribute, but its storage class will be automatic; hence, it will not conform to general rule 11, above. For the F Compiler, therefore, an offset declaration without an accompanying explicit area declaration will result in an error. (See also "AREA (Program Control Data Attribute)," in this section.)

OUTPUT (File Description Attribute)

See INPUT.

PICTURE (Data Attribute)

The PICTURE attribute is used to define the internal and external formats of character-string and numeric character data and to specify the editing of data. Numeric character data is data having an arithmetic value but stored internally in character form. Numeric character data must be converted to coded arithmetic before arithmetic operations can be performed.

The picture characters are described in Section D, "Picture Specification Characters."

General format:

```
PICTURE

    'character-picture-specification'

    'numeric-picture-specification'
```

A "picture specification," either character or numeric, is composed of a string of picture characters enclosed in single quotation marks. An individual picture character may be preceded by a repetition factor, which is a decimal integer constant, n, enclosed in parentheses, to indicate repetition of the character n times. If n is zero, the character is ignored. Picture characters are considered to be grouped into fields, some of which contain subfields.

General rules:

1. The "character picture specification" is used to describe a character-string data item. Three characters may be used: A, indicating that the associated position in the data item may contain any alphabetic character or a blank; X, indicating that the associated postion may contain any character; and 9, indicating that the associated position may contain any decimal digit or a blank. A character picture specification must include at least one A or X. Each character picture specification is a single field with no contained subfields.

   Example:

```
DECLARE ORDER# PICTURE
        'AA(3)9X99X(4)9';
```

   This declaration specifies that values of ORDER# are to be character strings of length 13. The string consists of two letters, three digits, any character, two digits, any character, and four digits. For example, the charac-

ter string 'G 42-63-0024' would fit this description.

Editing and suppression characters are not allowed in character picture specifications. Each picture specification character must represent an actual character in the data item.

2. The "numeric picture specification" is used to describe a character item that represents either an arithmetic value or a character-string value, depending upon its use. A numeric picture specification can consist of one or more fields, some of which can be divided into subfields. A single field is used to describe a fixed-point number or the mantissa of a floating-point number. Either may be divided into two subfields, one describing the integer portion, the other describing the fractional portion. For floating-point numbers, a second field is required to describe the exponent; it cannot be divided into subfields. A second field may optionally be used with fixed-point numbers to indicate a scaling factor. Four basic picture characters can be used in a numeric picture specification:

   9 indicating any decimal digit

   V indicating the assumed location of a decimal point. It does not specify an actual character in the character-string value of the data item. The V also indicates the end of a subfield of a picture specification.

   K indicating, for floating-point data items, that the exponent should be assumed to begin at the position associated with the picture character following the K. It does not specify an actual character in the character-string value of the data item, either an E or a sign. The K delimits the two fields of the specification.

   E indicating, for floating-point data items, that the associated position will contain the letter E to indicate the beginning of the exponent. The E also delimits the two fields.

   In addition to these characters, zero suppression characters, editing characters, and sign characters may be included in a numeric picture specification to indicate editing. Editing characters are not a part of the arithmetic value of a numeric character data item, but they are a part of

its character-string value. Repetition factors are allowed in numeric picture specifications.

3. A numeric character data item can have only a decimal base. Its scale and precision are specified by the picture characters. The PICTURE attribute cannot be specified in combination with base, scale, or precision attributes. If the mode of the numeric character data is COMPLEX, however, the COMPLEX attribute must be explicitly stated.

4. The following paragraphs indicate the combinations of picture characters for different arithmetic data formats.

   a. Real decimal fixed-point items are described in the following general form:

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

      The optional field of the picture specification, beginning with the letter F together with a parenthesized, optionally signed decimal integer constant, is a scaling factor that indicates the location of an assumed decimal point if that location is outside the actual data item. The scaling factor has an effect similar to the exponent of a floating-point number; it indicates that the assumed decimal point is "integer" places to the right (or left, if negative) of the position otherwise indicated.

      Sign, editing, and zero suppression picture characters can be included in a fixed-point specification. The V cannot appear more than once in a specification, although it may be used in combination with the decimal point (.) or comma (,) editing characters, which cause insertion of a period or comma. If no V is included, the decimal point is assumed to be to the right of the rightmost digit. Only one sign indication can be included in the first field (the actual sign of the integer in a scaling factor is allowed additionally). The specification must include at least one digit position.

      Example:

      ```
      DECLARE A PICTURE '999V99';
      ```

      This specification describes numeric character items of five digits, two of which are assumed to be fractional digits.

   b. Real decimal floating-point items are described by the following general form:

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

      Both the mantissa field and the exponent field must each contain at least one digit position. The exponent field can contain no more than two digits, since System/360 implementations allow only two digits in the exponent field of a decimal floating-point number. If arithmetic data items are to be assigned to the described variable, the exponent field must contain both of the allowed digit specification characters, or the second digit of the exponent field will be lost and the SIZE condition will be raised.

      Sign, editing, and zero suppression picture characters can be included in a floating-point specification. One sign indication is allowed for each field. Only one V is allowed, and it can appear in the first field only. As with fixed-point specifications, the V may appear in combination with the decimal point editing character (as .V or V.).

   c. Complex numeric character data is described using the general form:

      ```
      PICTURE 'real-picture' COMPLEX
      ```

      The "real picture" is a specification for either a decimal fixed-point or a decimal floating-point data item. The single picture specification describes both parts of a complex number.

5. The precision of a numeric character variable is dependent upon the number of digit positions, actual and conditional. Digit positions can be specified by the following characters:

9 which is an actual digit character

Z, *, Y — which are conditional digit characters specifying zero suppression

T, I, R — which are digit characters specifying an overpunch

$, S, +, - — which are conditional digit drifting characters

Each but the first conditional digit drifting character in a drifting string specifies a digit position. A conditional digit drifting character used alone does not specify a digit position.

Precision of a fixed-point variable is (p,q), where p is the number of digit positions in the picture specification and q is the number of digit positions following V. Precision of a floating-point variable is (p), where p is the number of digit positions preceding the E or K. Indicated static editing characters or insertion characters do not participate in the specification of precision, but they must be counted in the number of characters if the data item is written as output or assigned internally to a character string.

6. A variable representing sterling data items can be specified by using a numeric picture specification that consists of three fields, one each for pounds, shillings, and pence. The pence field may be divided into two subfields. Data so described is stored in character format as three contiguous numbers corresponding to each of the three fields. If any arithmetic operations are specified for the variable, its value is converted to coded fixed-point decimal representing the value in pence. Sterling picture specifications have the following form:

```
PICTURE
   'G [editing-character-1]...
   M pounds-field
   M [separator-1]...
         shillings-field
   M  [separator-2]...
         pence-field
        [editing-character-2]...'
```

Picture specification characters, editing characters, and separators can be used in any of these fields and are discussed in Section D, "Picture Specification Characters."

The precision (p,q) of a sterling numeric character data item is defined as follows:

q = number of fractional digits in the pence field

p = 3+q+(number of digit positions, actual and conditional, in the pounds field)

POINTER (Program Control Data Attribute)

See OFFSET.

POSITION (Data Attribute)

See DEFINED.

Precision (Arithmetic Data Attribute)

The precision attribute is used to specify the minimum number of significant digits to be maintained for the values of the data items, and to specify the scale factor (the assumed position of the binary or decimal point). The precision attribute applies to both binary and decimal data.

General format:

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

The "number of digits" is an unsigned decimal integer constant and "scale factor" is an optionally signed decimal integer constant. The precision attribute specification is often represented, for brevity,

as (p,q), where p represents the "number of digits" and q represents the "scale factor."

General rules:

1. The precision attribute must immediately follow, with or without intervening blanks, the scale (FIXED or FLOAT), base (DECIMAL or BINARY), or mode (REAL or COMPLEX) attribute at the same factoring level.

2. The number of digits specifies the number of digits to be maintained for data items assigned to the variable. The scale factor specifies the number of fractional digits. No point is actually present; its location is assumed.

3. The scale factor can be specified for fixed-point variables only; the number of digits is specified for both fixed-point and floating-point variables.

4. When the scale is FIXED and no scale factor is specified, it is assumed to be zero; that is, the variable is to represent integers.

5. The scale factor can be negative, and it can be larger than the number of digits. A negative scale factor (-q) always specifies integers, with the point assumed to be located q places to the right of the rightmost actual digit. A positive scale factor (q) that is larger than the number of digits always specifies a fraction, with the point assumed to be located q places to the left of the rightmost actual digit. In either case, intervening zeros are assumed, but they are not stored; only the specified number of digits are actually stored.

6. The precision attribute cannot be specified in combination with the PICTURE attribute.

7. The maximum number of digits allowed for System/360 implementations is 15 for decimal fixed-point data, 31 for binary fixed-point data, 16 for decimal floating-point data, and 53 for binary floating-point data.

Assumptions:

The defaults for System/360 implementations are as follows:

(5,0) for DECIMAL FIXED
(15,0) for BINARY FIXED
(6) for DECIMAL FLOAT
(21) for BINARY FLOAT

PRINT (File Description Attribute)

The PRINT attribute specifies that the data of the file is ultimately to be printed. The PAGE and LINE options of the PUT statement and the PAGESIZE option of the OPEN statement can be used only with files having the PRINT attribute. These options are described in Section J, "Statements."

General format:

PRINT

General rules:

1. The PRINT attribute implies the OUTPUT and STREAM attributes.

2. The PRINT attribute conflicts with the RECORD attribute. (However, through the use of the DD statement, RECORD files can be associated with the printer.)

3. The PRINT attribute causes the initial data byte within each record to be reserved for ASA printer control characters. These control characters are set by the PAGE, SKIP, or LINE format items or options.

Assumption:

If no FILE or STRING specification appears in a PUT statement, the standard output file SYSPRINT is assumed.

REAL (Arithmetic Data Attribute)

See COMPLEX.

RECORD and STREAM (File Description Attributes)

The RECORD and STREAM attributes specify the kind of data transmission to be used for the file. STREAM indicates that the data of the file is considered to be a continuous stream of data items, in character form, to be assigned from the stream to variables, or from expressions into the

stream. RECORD indicates that the file consists of a collection of physically separate records, each of which consists of one or more data items in any form. Each record is transmitted as an entity to or from a variable.

General format:

RECORD|STREAM

General rules:

1. A file with the STREAM attribute can be specified only in the OPEN, CLOSE, GET, and PUT statements.
2. A file with the RECORD attribute can be specified only in the OPEN, CLOSE, READ, WRITE, REWRITE, UNLOCK, and DELETE statements.
3. A file with the STREAM attribute cannot have any of the following attributes: UPDATE, DIRECT, SEQUENTIAL, BACKWARDS, BUFFERED, UNBUFFERED, EXCLUSIVE, and KEYED, any of which implies RECORD.
4. A file with the RECORD attribute cannot have the PRINT attribute.

Assumptions:

Default is STREAM. If a file is implicitly opened by a READ, WRITE, REWRITE, UNLOCK, or DELETE statement, RECORD is assumed.

REDUCIBLE

See IRREDUCIBLE.

RETURNS (Entry Name Attribute)

The RETURNS attribute may be specified in a DECLARE statement for an entry name that is used in a function reference within the scope of the declaration. It is used to describe the attributes of the function value returned when that entry name is invoked as a function.

General format:

RETURNS (attribute...)

It is used in the following manner:

```
DECLARE entry-name
    [ENTRY-attribute-specification]
     RETURNS (attribute...);
```

General rules:

1. The "ENTRY attribute specification" consists of the keyword ENTRY with or without associated parameter attribute lists. If parameter attribute lists are not required, the keyword ENTRY is optional, since the RETURNS attribute implies the ENTRY attribute.
2. The attributes in the parenthesized list following the keyword RETURNS are separated by blanks. They must agree with the attributes specified in the PROCEDURE or ENTRY statement to which the entry name is prefixed. If the attributes of the actual value returned do not agree with those declared with the RETURNS attribute, no conversion will be performed.
3. Only arithmetic, string, locator, AREA, and PICTURE attributes can be specified.
4. Length attribute specifications are evaluated on entry to the block containing the RETURNS attribute specification.
5. Unless default attributes for the entry name apply, any invocation of a function must appear within the scope of a RETURNS attribute declaration for the entry name. For an internal function, the RETURNS attribute can be specified only in a DECLARE statement that is internal to the same block as the function procedure.

Assumptions:

If the RETURNS attribute is not specified within the scope of a function reference, the defaults assumed for the returned value are FIXED BINARY (15,0) if the entry name begins with any of the letters I through N; otherwise, the defaults are FLOAT DECIMAL (6). Default precisions are those defined for System/360 implementations.

SEQUENTIAL (File Description Attribute)

See DIRECT.

SETS and USES

These attributes cause no action in the current version of the F Compiler other than to imply the ENTRY attribute.

STATIC (Storage Class Attribute)

See AUTOMATIC.

STREAM (File Description Attribute)

See RECORD.

TASK (Program Control Data Attribute)

The TASK attribute describes a variable that may be used as a task name, to test or control the relative priority of a task.

General format:

TASK

General rules:

1. An identifier can be explicitly declared with the TASK attribute in a DECLARE statement, or it can be contextually declared by its appearance in a TASK option of a CALL statement.
2. Task variables can also have the following attributes:
   a. Dimension
   b. Scope (the default is INTERNAL)
   c. Storage class (the default is AUTOMATIC)
   d. DEFINED (task variables may only be defined on other task names)
3. A task variable can be used in the following contexts only:
   a. In the TASK option of a CALL statement
   b. As an argument of the PRIORITY pseudo-variable or built-in function
   c. As an argument in a CALL statement or function reference
   d. As a parameter in a PROCEDURE or ENTRY statement or in the parameter attribute list of an ENTRY attribute
   e. In an ALLOCATE or FREE statement
4. A task variable may be associated with the priority of a task by including the task name in the TASK option of a CALL statement. A task variable is said to be active if its associated task is active. A task variable must be in an allocated state when it is associated with a task and must not be freed while it is active. An active task variable cannot be associated with another task.
5. A task variable contains a single value, a priority value. This value is a fixed-point binary value of precision (n,0), where n is implementation-defined (15, for the F Compiler). This value can be tested and adjusted by means of the PRIORITY built-in function and pseudo-variable. The built-in function returns the priority of the task argument relative to the priority of the task executing the function. Similarly, the pseudo-variable permits assignment, to the named task variable, of a priority relative to the priority of the task executing the assignment.
6. Structures, arrays, or areas containing task variables cannot take part in assignment or input/output operations.
7. Task data cannot be converted to any other data type.
8. A task variable cannot be passed as an argument if this would require creation of a dummy argument.

UNALIGNED (Data Attribute)

See ALIGNED.

UNBUFFERED (File Description Attribute)

See BUFFERED.

UPDATE (File Description Attribute)

See INPUT.

USES

See SETS.

VARYING (String Attribute)

See BIT.

This section presents the PL/I statements in alphabetical order. (The preprocessor statements are alphabetically arranged at the end of this section.) Most statements are accompanied by the following information:

1. Function -- a short description of the meaning and use of the statement
2. General format -- the syntax of the statement
3. Syntax rules -- rules of syntax that are not reflected in the general format
4. General rules -- rules governing the use of the statement and its meaning in a PL/I program

The ALLOCATE Statement

Function:

The ALLOCATE statement causes storage to be allocated for specified controlled or based 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 controlled variables may both be specified as identifiers in the same ALLOCATE statement.

Syntax rules 2 through 7 apply only to Option 1:

2. "Level" indicates a level number. The first identifier appearing after the keyword ALLOCATE must be a level 1 identifier.
3. Each identifier must represent data of the controlled storage class or be an element of a controlled major structure.
4. "Dimension" indicates a dimension attribute. "Attribute" indicates a BIT, CHARACTER, or INITIAL attribute.
5. A dimension attribute, if present, must specify the same number of dimensions as that declared for the associated identifier.
6. The attribute BIT may appear only with a BIT identifier; CHARACTER may appear only with a CHARACTER identifier.
7. 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 8 and 9 apply only to Option 2:

8. The based variable appearing in the ALLOCATE statement may be an element variable, an array, or a major structure. When it is a major structure, only the major structure name is specified.
9. The SET clause, if present, 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 over-

rides 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 allocation. If the variable has not been previously allocated, the bound or length is undefined.

c. Either the ALLOCATE statement or the DECLARE statement must specify any necessary dimension size, or length attributes for an identifier. Any expression taken from the DECLARE statement is evaluated at the point of allocation using the condition enabling of the ALLOCATE statement, although the names are interpreted in the environment of the DECLARE statement.

d. If, in either an ALLOCATE or a DECLARE statement, bounds, lengths, or area sizes are specified by expressions that contain references to the variable being allocated, the expression are evaluated using the value of the most recent generation of the variable.

3. Upon allocation of an identifier, initial values are assigned to it if the identifier has an INITIAL attribute in either the ALLOCATE statement or DECLARE statement. Expressions or a CALL option in the INITIAL attribute are executed at the point of allocation, using the condition enabling of the ALLOCATE statement, although the names are interpreted in the environment of the declaration. If an INITIAL attribute appears in both DECLARE and ALLOCATE statements, the INITIAL attribute in the ALLOCATE statement is used. If initialization involves reference to the variable being allocated, the reference will be to the new generation of the variable.

4. 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.

6. Any evaluations performed at the time the ALLOCATE statement is executed (e.g., evaluation of expressions in an INITIAL attribute) must not be interdependent; they cannot depend on each other at the same time.

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. The SET clause indicates the pointer variable that is to receive the value identifying the allocation. The SET clause need not name the pointer variable declared with the based variable. If the SET clause is omitted, the pointer that was declared with the based variable is set.

9. If the IN clause appears in the ALLOCATE statement, storage will be allocated in the named area, for the based variable. If sufficient storage does not exist within this area, the AREA condition will be raised.

10. 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. A based structure may contain one adjustable array bound or string length, whose value is taken, on allocation, from the current value of a variable outside the structure (see "The REFER Option", in Chapter 14, "Based Storage and List Processing.") Note that the asterisk notation for bounds and length is not permitted for based variables.

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

12. A based variable transferred as an argument to a procedure cannot appear in an ALLOCATE statement in the called procedure.

Examples:

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

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

N1, N2 = 10;
ALLOCATE A;          The bounds are 10 and
                       10
ALLOCATE A           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 controlled bit string B:

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

ALLOCATE B           Invalid; violates rule
    BIT (*);           2b.
ALLOCATE B;          Invalid; violates rule
                       2b.
ALLOCATE B           The maximum length is
    BIT (N);           N.
ALLOCATE B CHAR-     Invalid; 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 controlled variable in an ALLOCATE statement:

```
DECLARE   A(N,N)   CONTROLLED   INITIAL
((N*N)0);
  .
  .
  .
IF 1 ALLOCATION (A) THEN ALLOCATE A
INITIAL (1,(N-1) ((N)0,1));
  .
  .
  .
ALLOCATE A;
```

4. The following example illustrates three uses of Option 2 of the ALLOCATE statement for based identifiers.

```
DECLARE VALUE BASED (P),
    RATES BASED (Q)
    1 GROUP BASED (R),
      2 DIM FIXED BINARY,
      2 VALUES (N REFER (DIM)),
    TABLE AREA BASED (S),
    N FIXED BINARY,
    T POINTER;
```

a. ALLOCATE VALUE SET (P); Allocates storage for the based variable VALUE and sets the pointer variable P to identify the particular allocation.

b. ALLOCATE GROUP SET (R); Allocates storage for the structure GROUP, and sets the pointer variable R to identify the particular allocation. The current value of N is used to determine the bound of VALUES, and this value is assigned to DIM.

c. ALLOCATE RATES SET (T) IN TABLE; Allocates storage within the area S-> TABLE for the variable RATES. The pointer variable T is set to identify the location within TABLE at which RATES is allocated.

The Assignment Statement

Function:

The assignment statement is used to evaluate an expression and to assign its value to one or more target variables; the target variables may be element, array, or structure variables. The target variables may be indicated by pseudo-variables.

General formats:

The assignment statement has 3 general format options. They are given in Figure J-1.

Syntax rules:

1. In Option 2, each target variable must be an array. If the right-hand side contains arrays of structures, then all target variables must be arrays of structures. The BY NAME option may be given only when the right-hand side contains at least one structure.

2. In Option 3, each target variable must be a structure.

General rules:

1. Aggregate assignments (Options 2 and 3) are expanded into a series of element assignments according to rules 5 through 8.

2. An element assignment is performed as follows:

   a. Subscripts of the target variables, and the second and third arguments of SUBSTR pseudo-variable references, are evaluated from left to right.

   b. The expression on the right-hand side is then evaluated.

   c. For each target variable (in left to right order), the expression is converted to the characteristics of the target variable according

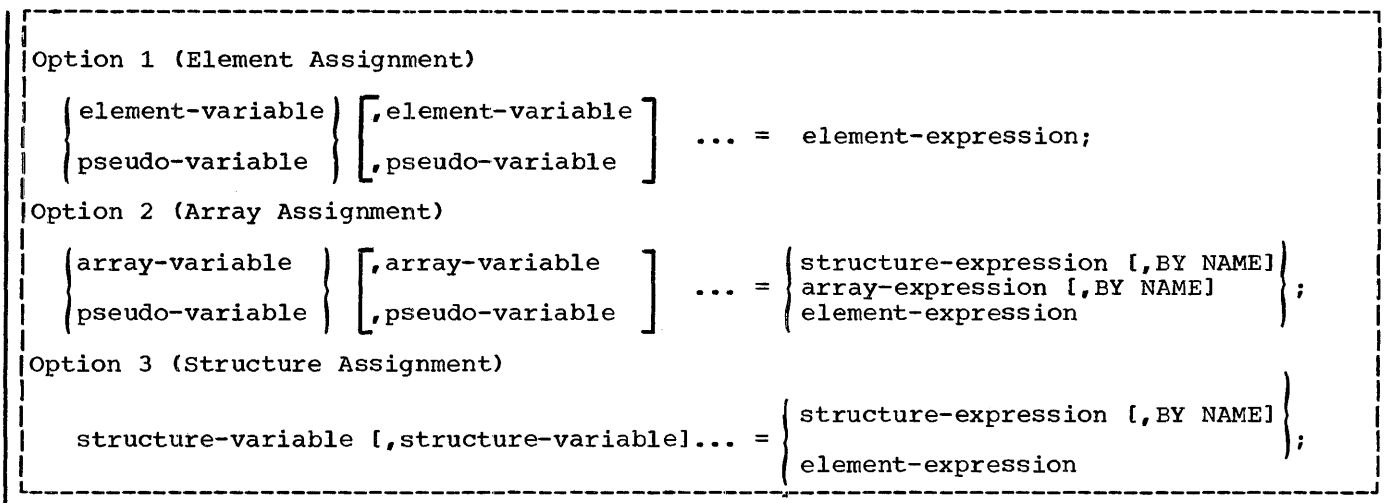


Figure J-1. General Formats of the Assignment Statement

to rules for data conversion (except that whenever a conversion of arithmetic base is involved, the value is converted directly to the precision of the target variable). The converted value is then assigned to the target variable.

3. The following rules apply to string element assignment:

   a. The assignment is performed from left to right, starting with the leftmost position.

   b. If the target variable is a fixed-length string, the expression value is truncated on the right if it is too long or padded on the right (with blanks for character string, zeros for bit strings) if the value is too short. (Note that a string pseudo-variable is considered to be a fixed-length string). The resulting value is assigned to the target.

   c. If the target is a VARYING string and the value of the expression is longer than the maximum length declared for the variable, the value is truncated on the right. The target string obtains a current length equal to its maximum length. If the value of the expression is not longer than the maximum length, the value is assigned; the target string obtains a current length equal to the length of the value.

4. The following rules apply to other element assignments:

   a. If the target is an area variable, the expression must be an area variable or function. The AREA condition will be raised by this assignment if the size of the target area is insufficient for the current extent of the area being assigned.

   b. If the target is a pointer variable, the expression can only be a pointer (or offset) variable or a pointer (or offset) function reference. If the expression is of offset type, its value is converted to pointer.

   c. If the target is an offset variable, the expression can only be an offset (or pointer) variable or an offset (or pointer) function reference. If the expression is of pointer type, its value is converted to offset.

   d. If the target is a label variable, the expression can only be a label variable or label constant. Environmental information (i.e., information that identifies the invocation of the block) is always assigned to the label variable.

   e. If the target is an event variable, the expression can only be an event variable. The assignment is uninterruptable, and it involves both the completion and status values. An event variable does not become active when it has an active event variable assigned to it. It is an error to assign to an active event variable.

f. If the target is a STATUS pseudo-variable, a value can be assigned whether or not the event variable is active. It is an error to assign to a COMPLETION pseudo-variable if the named event variable is active.

5. The first target variable in an aggregate assignment is known as the master variable. If the master variable is an array, then an array expansion (Rule 6) is performed; otherwise, a structure expansion (Rules 7 and 8) is performed. The CHECK condition for assignment to a target variable is not raised during the assignment; it is raised (when suitably enabled) after the assignment is complete. Such CHECK conditions are raised in the written order of the enabled identifiers. In the case of BY NAME assignment, the CHECK condition for the target variable is raised regardless of whether any value is assigned to an item. The label prefix of the original statement is applied to a null statement preceding the other generated statements.

6. In Option 2, all array operands must have the same number of dimensions and identical bounds. The array assignment is expanded into a loop of the form:

```
LABEL: DO j1 = LBOUND(master-variable,1) TO
           HBOUND(master-variable,1);

       DO j2 = LBOUND(master-variable,2) TO
           HBOUND(master-variable,2);
         .
         .
         .

       DO jn = LBOUND(master-variable,n) TO
           HBOUND(master-variable,n);

       generated assignment statement

       END LABEL;
```

In this expansion, n is the number of dimensions of the master variable that are to participate in the assignment. In the generated assignment statement, all array operands are fully subscripted, using (from left to right) the dummy variables j1 to jn. If an array operand appears with no subscripts, it will only have the subscripts j1 to jn; if cross-section notation is used, the asterisks are replaced by j1 to jn. If the original assignment statement (which may have been generated by Rule 7 or Rule 8) has a condition prefix, the generated assignment statement is given this condition prefix. If the original assignment statement (which may have been generated by Rule 8) has a BY NAME option, the generated assignment statement is given a BY NAME option. If the generated assignment statement is a structure assignment, it is expanded as given below.

7. In Option 3, where the BY NAME option is not specified, the following rules apply:

   a. None of the operands can be arrays, although they may be structures that contain arrays.

   b. All of the structure operands must have the same number, k, of immediately contained items.

   c. The assignment statement (which may have been generated by Rule 6) is replaced by k generated assignment statements. The ith generated assignment statement is derived from the original assignment statement by replacing each structure operand by its ith contained item; such generated assignment statements may require further expansion according to Rule 6 or Rule 7. All generated assignment statements are given the condition prefix of the original statement.

8. In Option 3, where the BY NAME option is given, the structure assignment, which may have been generated by Rule 6, is expanded according to steps (a) through (d) below. None of the operands can be arrays.

   a. The first item immediately contained in the master variable is considered.

   b. If each structure operand and target variable has an immediately contained item with the same identifier, an assignment statement is generated as follows: the statement is derived by replacing each structure operand and target variable with its immediately contained item that has this identifier. If any structure contains no such identifier, no statement is generated. If the generated assignment is a structure or array-of-structures assignment, BY NAME is appended. The first generated assignment is given the label prefix of the original assignment statement; all generated assignment statements are given the condition prefix of the original assignment statement.

c. Step b is repeated for each of the items immediately contained in the master variable. The assignments are generated in the order of the items contained in the master variable.

d. Steps a through c may generate further array and structure assignments. These are expanded according to Rules 6 through 8.

Examples:

1. Suppose that the following three structures have been declared.

```
1 ONE                1 TWO
  2 PART1              2 PART1
    3 RED                3 RED
    3 WHITE              3 GREEN
    3 BLUE               3 WHITE
  2 PART2              2 PART2
    3 GREEN              3 BLUE
    3 YELLOW             3 YELLOW
    3 ORANGE(3)          3 ORANGE(3)
  2 PART3
    3 BLACK
    3 WHITE

        1 THREE
          3 PART1
            5 BLACK
            5 WHITE
            5 RED
          3 PART2
            5 YELLOW
            5 WHITE
            5 ORANGE(3)
            5 PURPLE
```

Consider the following assignment:

```
ONE = TWO - 2 * THREE, BY NAME;
```

By Rule 8 this generates:

```
ONE.PART1 = TWO.PART1 - 2 *
  THREE.PART1, BY NAME;

ONE.PART2 = TWO.PART2 - 2 *
  THREE.PART2, BY NAME;
```

Applying Rule 8 again, these statements are replaced by:

```
ONE.PART1.RED = TWO.PART1.RED
  - 2 * THREE.PART1.RED;

ONE.PART1.WHITE = TWO.PART1.WHITE
  - 2 * THREE.PART1.WHITE;

ONE.PART2.YELLOW = TWO.PART2.YELLOW
  - 2 * THREE.PART2.YELLOW;

ONE.PART2.ORANGE = TWO.PART2.ORANGE
  - 2 * THREE.PART2.ORANGE;
```

The final assignment is expanded according to Rule 6.

2. The following example illustrates array assignment (Option 2):

Given the array A

```
2  4
3  6
1  7
4  8
```

and the array B

```
1  5
7  8
3  4
6  3
```

Consider the assignment statement:

```
A = (A+B)**2-A(1,1);
```

After execution, A has the value

```
 7  74
93 189
 9 114
93 114
```

Note that the new value for A(1,1), which is 7, is used in evaluating the expression for all other elements.

3. The following example illustrates string assignment:

Given:

A is a fixed-length string whose value is 'XZ/BQ'.
B is a varying-length string of maximum length 8 whose value is 'MAFY'.
C is a fixed-length string of length 3.
D is a varying-length string of maximum length 5.

Then in the statement:

C=A, the value of C is 'XZ/'.
C='X', the value of C is 'Xbb'.
D=B, the value of D is 'MAFY'.
D=SUBSTR(A,2,3)||SUBSTR(A,2,3), the value of D is 'Z/BZ/'.
SUBSTR(A,2,4)=B, the value of A is 'XMAFY'.
SUBSTR(B,2,2)='R', the value of B is 'MRbY'.
SUBSTR(B,2)='R', the value of B is 'MRbb'.

The BEGIN Statement

Function:

The BEGIN statement heads and identifies a begin block.

General format:

```
BEGIN;
```

Syntax rule:

A label of a BEGIN statement may be subscripted, but such a label cannot appear after an END statement.

General rule:

A BEGIN statement is used in conjunction with an END statement to delimit a begin block. A complete discussion of begin blocks can be found in Part I, Chapter 6, "Blocks, Flow of Control, and Storage Allocation."

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, which can be a generic name, represents the entry point of the procedure invoked.
2. An argument cannot be a condition name.
3. The TASK, EVENT, and PRIORITY options can appear in any order.

General rules:

1. The TASK, EVENT, and PRIORITY options, when used alone or in any combination, specify that the invoked and invoking procedures are to be executed asynchronously. 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 Part I, Chapter 15, "Multitasking.")
2. When the TASK option is used, the task name, if given, is associated with the task created by the CALL. Reference to this name enables the priority of the task to be controlled at some other point by the use of the PRIORITY pseudo-variable and built-in function.
3. When the EVENT option is used, the event name is associated with the completion of the task created by the CALL statement. Another task can then wait for completion of this created task by specifying the event name in a WAIT statement.

   Upon execution of the CALL statement, the event variable is made active, and the completion value is set to '0'B and the status value to 0. Upon termination of the created task, the completion value is set to '1'B and, unless the task has been terminated by a RETURN or END statement, the status is set to 1 if still zero.
4. If the PRIORITY option is used, the expression in the PRIORITY option is evaluated to an integer m, of an implementation-defined precision (15,0). The priority of the named task is then made m relative to the task in which the CALL is executed.

   If a CALL statement with the EVENT or TASK option does not have the PRIORITY option, the priority of the invoked task is made equal to that of the task variable in the TASK option, if there is one, or else made equal to the priority of the invoking task.
5. Expressions in these options, as well as any argument expressions, are evaluated in the task in which the call is executed. This includes execution of any on-units entered as the result of the evaluations.
6. The environment of the invoked procedure is established after evaluation of the expressions named in Rule 5, and before the procedure is invoked.
7. See Part I, Chapter 10, "Subroutines and Functions" for detailed descriptions of the interaction of arguments with the parameters that represent these arguments in the invoked procedure.

Examples:

1.
```
CALL CRITICAL_PATH (A,B*C,D);
                .
                .
                .
CRITICAL_PATH:   PROCEDURE(ALPHA,BETA,
    GAMMA);
                .
                .
                .
               END;
```

2.
```
CALL PAYROLL (NAME, DATE, HRRATE);
```

3.
```
CALL PRINT (A,B) TASK (T2) EVENT (ET2)
PRIORITY (-2);
```

The CLOSE Statement

Function:

The CLOSE statement dissociates the named file from the data set with which it was associated by opening in the current task.

General format:

```
CLOSE     FILE     (file-name)     [,FILE
(file-name)]...;
```

General rules:

1. The FILE(filename) option specifies which file is to be closed. It must appear once. Several files can be closed by one CLOSE statement.

2. A closed file can be reopened.

3. Closing an unopened file, or an already closed file, has no effect.

4. The CLOSE statement cannot be used to close a file in a task different from the one that opened the file.

5. 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.

6. All input/output events that have not been completed before the file is closed are set complete, with a status value of 1.

7. A CLOSE statement unlocks all records in the file previously locked in the task in which the CLOSE appears.

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

Function:

The DECLARE statement is the principal method for explicitly declaring attributes of names.

General format:

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

Syntax rules:

1. Any number of identifiers may be declared in one DECLARE statement.

2. "Level" is a nonzero unsigned decimal integer constant. If a level number is not specified, level 1 is assumed for all element and array variables. Level 1 must be specified for all major structure names. A blank space must separate a level number from the identifier following it.

3. In general, attributes must immediately follow the identifier to which they apply as shown in the general format. However, attributes can be factored (see "Factoring of Attributes" in Section I, "Attributes").

General rules:

1. A particular level 1 identifier can be specified in only one DECLARE statement within a particular block. All attributes given explicitly for that identifier must be declared together in that DECLARE statement. (Note, however, that identifiers having the FILE attribute may be given attributes in an OPEN statement as well. See "The OPEN Statement" in this section and in Part I, Chapter 8, "Input and Output," for further information.)

2. Attributes of external names, in separate blocks and compilations, must be consistent.

3. Labels may be prefixed to DECLARE statements (however, such labels are treated as comments and, hence, have no meaning). Condition prefixes cannot be attached to a DECLARE statement.

The DELAY Statement

Function:

The DELAY statement causes the execution of a task to be suspended for a specified period of time.

General format:

```
DELAY (element-expression);
```

General rule:

Execution of the DELAY statement causes the element expression to be evaluated and converted to an integer n; execution is then suspended for n milliseconds.

Example:

```
DELAY (10);
```

This statement causes execution of the task to be suspended for ten milliseconds.

The DELETE Statement

Function:

The DELETE statement deletes a record from an UPDATE file.

General format:

```
DELETE FILE (file-name)
       [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 be specified.

3. The KEY option must be specified if the file is a DIRECT UPDATE file; it cannot be specified otherwise. The expression is converted to a character string and determines which record is to be deleted.

4. If the file is a SEQUENTIAL UPDATE file, the record to be deleted is the last record that was read; the data set organization must be INDEXED.

5. The EVENT option allows processing to continue while a record is being deleted.

   When control reaches a DELETE statement containing this option, the "event variable" is made active (that is, it cannot be associated with another event) and is given the completion value '0'B, provided that the UNDEFINEDFILE condition is not raised by an implicit file opening (see "Note" below). The event variable remains active and retains its '0'B completion value until control reaches a WAIT statement specifying that event variable. At this time, either of the following can occur:

   a. If the DELETE statement has been executed successfully and neither of the conditions TRANSMIT or KEY has been raised as a result of the DELETE, the event variable is set complete, given the completion value '1'B, and the event variable is made inactive, that is, can be associated with another event.

   b. If the DELETE statement has resulted in the raising of TRANSMIT or KEY, the interrupt for each of these conditions does not occur until the WAIT is encountered. At such time, the corresponding on-units (if any) are entered in the order in which the conditions were raised. After a return from the final on-unit, or if one of the on-units is terminated by a GO TO statement, the event variable is given the completion value '1'B and is made inactive.

Note: If the DELETE statement causes an implicit file opening that results in the raising of UNDEFINEDFILE, the on-unit associated with this condition is entered immediately and the event variable remains unchanged; that is, the event variable remains inactive and retains the same value it had when the DELETE was encountered. If the on-unit does not correct the condition, then, upon normal return from the on-unit, the ERROR condition is raised; if the condition is corrected in the on-unit, that is, if the file is opened successfully, then, upon normal return from the on-unit, the event variable is set to '0'B, it is

made active, and execution of the DELETE statement continues.

6. The DELETE statement unlocks a record only if that record had been locked in the same task in which the DELETE appears.

7. The DELETE statement can cause implicit opening of a file.

Example:

```
DELETE FILE(ALPHA) KEY (DKEY);
```

This statement causes the record identified by DKEY to be deleted from the data set associated with the file ALPHA. If the record was previously locked in the same task, it is unlocked.

The DISPLAY Statement

Function:

The DISPLAY statement causes a message to be displayed to the machine operator. A response may be requested.

General format:

Option 1.

```
DISPLAY (element-expression);
```

Option 2.

```
DISPLAY (element-expression)
REPLY (character-variable)
[EVENT (event-variable)];
```

General rules:

1. Execution of the DISPLAY statement causes the element expression to be evaluated and, where necessary, converted to a varying character string of implementation-defined maximum length (126 characters for the F Compiler). 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. For the F Compiler, the message cannot exceed 126 characters.

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 completion part of the event variable will be given the value '0'B until the reply is completed, when it will be given the value '1'B. The reply is considered complete only after the execution of a WAIT statement naming the event.

Example:

```
DISPLAY ('END OF JOB');
```

This statement causes the message "END OF JOB" to be displayed.

The DO Statement

Function:

The DO statement heads a DO-group and can also be used to specify repetitive execution of the statements within the group.

General formats:

The three format types for the DO statement are shown in Figure J-2.

Syntax rules:

1. In all three types, the DO statement is used in conjunction with the END statement to delimit a DO-group. Only Type 1 does not provide for the repetitive execution of the statements within the group.

2. In Type 3, the variable or pseudovariable must represent a single element; "variable" may be subscripted and/or qualified. Real arithmetic variables are generally used, but label, string, and complex variables are allowed, provided that the expansions given in the general rules below result in valid PL/I programs. Note, however, that if "variable" is a label variable, each "specification" must have the following form:

```
{element-label-variable}
{label-constant        }
        [WHILE (expression)]
```

3. Each expression in a specification must be an element expression.

```
Type 1.  DO;

Type 2.  DO WHILE (element-expression);

Type 3.  DO { pseudo-variable } =specification[,specification]...;
            { variable      }

  where "specification" has the form:

  expression1 [TO expression2 [BY expression3]]  [WHILE(expression4)]
              [BY expression3 [TO expression2]]
```

Figure J-2. General Format of the DO Statement

4. If "BY expression3" is omitted from a "specification," and if "TO expression2" is included, "expression3" is assumed to be 1.

5. If "TO expression2" is omitted from a "specification," repetitive execution continues until it is terminated by the WHILE clause or by some statement within the group.

6. If both "TO expression2" and "BY expression3" are omitted from a specification, it implies a single execution of the group, with the control variable having the value of "expression1". If "WHILE expression4" is included, this single execution will not take place unless "expression4" is true.

General rules:

1. In Type 1, the DO statement only delimits the start of a DO-group; it does not provide for repetitive execution.

2. In Type 2, the DO statement delimits the start of a DO-group and provides for repetitive execution as defined by the following:

```
LABEL:  DO WHILE (expression);
        statement-1
        .
        .
        .
        statement-n
        END;
NEXT:   statement /*STATEMENT
          FOLLOWING THE DO GROUP*/
```

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 /*STATEMENT
          FOLLOWING THE DO GROUP*/
```

3. In Type 3, the DO statement delimits the start of a DO-group and provides for controlled repetitive execution as defined by the following:

```
LABEL:  DO variable (a1,...,an)=
          expression1
          TO expression2
            BY expression3
          WHILE (expression4);
          statement-1
          .
          .
          .
          statement-m
LABEL1: END;
NEXT:   statement
```

This is exactly equivalent to the following expansion:

```
LABEL:  temp1=a1;
        .
        .
        .
        tempn=an;
        e1=expression1;
        e2=expression2;
        e3=expression3;
        v=e1;
```

```
LABEL2: IF (e3>=0)&(v>e2)|
           (e3<0)&(v<e2)
           THEN GO TO NEXT;
        IF (expression4) THEN;
          ELSE GO TO NEXT;
        statement-1
        .
        .
        .
        statement-m
LABEL1: v=v+e3;
        GO TO LABEL2;
NEXT:   statement
```

In the above expansion, $a_1,\ldots,a_n$ are expressions that may appear as subscripts of the control variable; $temp_1 \ldots temp_n$ are compiler-created work areas, with the attributes BINARY FIXED(15), to which the expression values are assigned; v is equivalent to "variable" with the associated "temp" subscripts; "e1," "e2," and "e3" are compiler-created work areas having the attributes of "expression1," "expression2," and "expression3," respectively. In the simplest cases, there are no subscripts (i.e., n=0) and the first statement in the expansion is therefore e1=expression1.

Additional rules for the above expansion follow:

a. The above expansion only shows the result of one "specification." If the DO statement contains more than one "specification," the statement labeled NEXT is the first statement in the expansion for the next "specification." The second expansion is analogous to the first expansion in every respect. Thus, if a second "specification" appeared in the DO statement, the second expansion would look like this:

```
NEXT:   temp1=a1;
          .
          .
          .
        tempn=an;
        e5=expression5;
        .
        .
        .
        v=e5;
LABEL3: IF ... THEN GO TO NEXT1;
        IF (expression8) THEN;
          ELSE GO TO NEXT1;
        statement-1
        .
        .
        .
        statement-m
LABEL4: v=v+e7;
        GO TO LABEL3;
NEXT1:  statement
```

Note that statements 1 through m are not actually duplicated in the program.

b. If the WHILE clause is omitted, the IF statement immediately preceding statement-1 in the expansion is omitted.

c. If "TO expression2" is omitted, the statement "e2=expression2" and the IF statement identified by LABEL2 are omitted.

d. If both "TO expression2" and "BY expression3" are omitted, all statements involving e2 and e3, as well as the statement GO TO LABEL2, are omitted.

4. The WHILE clause in Types 2 and 3 specifies that before each repetition of statement execution, the associated element expression is evaluated, and, if necessary, converted to a bit string. If any bit in the resulting string is 1, the statements of the DO-group are executed. If all bits are 0, then, for Type 2, execution of the DO-group is terminated, while for Type 3, only the execution associated with the "specification" containing the WHILE clause is terminated; repetitive execution for the next "specification," if one exists, then begins.

5. In a "specification," "expression1" represents the initial value of the control variable (i.e., "variable" or "pseudo-variable"); "expression3" represents the increment to be added to the control variable after each execution of the statements in the group; expression2 represents the terminating value of the control variable. Execution of the statements in a DO-group terminates for a "specification" as soon as the value of the control variable is outside the range defined by "expression1" and "expression2." When execution for the last "specification" is terminated, control, in general, 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 Type 1; that is, only if repetitive execution is not specified. Consequently, repetitive DO-groups cannot contain ENTRY statements.

7. The effect of allocating or freeing the control variable within the DO-group is undefined.

The END Statement

Function:

The END statement terminates blocks and groups.

General format:

```
END [label];
```

Syntax rules:

If "label" is specified, it cannot be an element of a label array; that is, it cannot be subscripted.

General rules:

1. If a label follows END, the statement terminates the unterminated group or block headed by the nearest preceding DO, BEGIN, or PROCEDURE statement having that label. It also terminates any unterminated groups or blocks physically within that group or block.

2. If a label does not follow END, the statement terminates that group or block headed by the nearest preceding DO, BEGIN, or PROCEDURE statement for which there is no corresponding END statement.

3. If control reaches an END statement for a procedure, it is treated as a RETURN statement.

The ENTRY Statement

Function:

The ENTRY statement specifies a secondary entry point of a procedure.

General format:

```
entry-name:  [entry-name:]...
   ENTRY [(parameter [,parameter]...)]
         [attribute]...;
```

Syntax rules:

1. The only attributes that may be specified with an ENTRY statement are the arithmetic, string, POINTER, OFFSET, AREA, and PICTURE attributes. The attributes specified determine the characteristics of the value returned by the procedure when it is invoked as a function at this entry point.

2. A condition prefix cannot be specified for an ENTRY statement.

General rules:

1. The relationship established between the parameters of a secondary entry point and the arguments passed to that entry point is exactly the same as that established for primary entry point parameters and arguments. See Part I, Chapter 10, "Subroutines and Functions," for a complete discussion of this subject.

2. As stated in syntax rule 1, the attributes specified with an ENTRY statement determine the characteristics of the value returned by the procedure when it is invoked as a function at this entry point. The value being returned by the procedure (i.e., the value of the expression in a RETURN statement) is converted, if necessary, to correspond to the specified attributes. If the attributes are not specified at the entry point, default attributes are applied, according to the first letter of the entry name used to invoke the entry point.

3. If an ENTRY statement has more than one label, each label is interpreted as though it were a single entry name for a separate ENTRY statement having the same parameter list and explicit attribute specification. For example, consider the statement:

```
A:  I:  ENTRY;
```

This statement is effectively the same as:

```
A:  ENTRY;

I:  ENTRY;
```

Since the attributes of the returned value are not explicitly stated, the characteristics of the value returned by the procedure will depend on whether the entry point has been invoked as A or I.

4. The ENTRY statement must be internal to the procedure for which it defines a secondary entry point. It may not be internal to any block contained in this procedure; nor may it be within a DO-group that specifies repetitive execution.

The EXIT Statement

Function:

The EXIT statement causes immediate termination of the task that contains the statement and all tasks attached by this task. If the EXIT statement is executed in a major task, it is equivalent to a STOP statement.

General format:

```
EXIT;
```

General rule:

If executed in a major task, EXIT causes the FINISH condition to be raised in that task. On normal return from the FINISH on-unit, the task executing the statement, and all of its descendant tasks are terminated. The completion values of the event variables associated with these tasks are set to '1'B, and their status values to 1 (unless they are already non-zero).

The FORMAT Statement

Function:

The FORMAT statement specifies a format list that can be used by edit-directed transmission statements to control the format of the data being transmitted.

General format:

```
label: [label:]... FORMAT (format-list);
```

Syntax rules:

1. The "format list" must be specified according to the rules governing format list specifications with edit-directed transmission as described in Part I, Chapter 8, "Input and Output."

2. At least one "label" must be specified for a FORMAT statement. One of the labels (or a label variable having the value of one of the labels) is the statement label designator appearing in a remote format item. None of the labels can be specified in a GO TO statement.

General rules:

1. A GET or PUT statement may include a remote format item, R, in the format list of an edit-directed data specification. That portion of the format list represented by R must be supplied by a FORMAT statement identified by the statement label specified with R.

2. The remote format item and the FORMAT statement must be internal to the same block.

3. If a condition prefix is associated with a FORMAT statement, it must be identical to the condition prefix associated with the GET or PUT statement referring to that FORMAT statement.

4. When a FORMAT statement is encountered in normal sequential flow, control passes around it, and the CHECK condition will not be raised for a statement label attached to it.

The FREE Statement

Function:

The FREE statement causes the storage allocated for specified based or controlled variables to be freed. For controlled variables, the next most recent allocation in the task is made available, and subsequent references in the task to the identifier refer to that allocation.

General formats:

Option 1

```
FREE controlled-variable
    [,controlled-variable]...;
```

Option 2

```
FREE [pointer-qualifier ->]
    based-variable[IN(area-variable)]
    [,[pointer-qualifier- >]
    based-variable
    [IN(area-variable)]]...;
```

Syntax rules:

1. In Option 1, the "controlled variable" is an element, array, or major structure of the controlled storage class.

2. In Option 2, the "based variable" must be an unsubscripted, level-one based variable.

3. The forms of Option 1 and Option 2 can be combined in the same FREE statement.

General rules:

1. Controlled storage allocated in a task cannot be freed by a descendant task.

2. If a specified nonbased identifier has no allocated storage at the time the FREE statement is executed, it is an error.

Rules 3 through 6 apply only to Option 2.

3. 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.

4. 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 has not been allocated, the results are unpredictable.

5. A based variable can be used to free storage only if that storage has been allocated for a based variable having identical data attributes, including values of bounds, lengths, and area size expressions.

6. The IN option must be specified if the storage to be freed has been allocated using the IN option, and it must have been allocated in the area specified in the FREE statement. The IN option cannot appear in the FREE statement in any other circumstances. Note that area assignment causes allocation of based storage in the target area; such allocations can be freed by the IN option naming the target area.

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 allocation of VALUE identified by pointer P.

```
FREE V->GROUP;
```

Frees that portion of storage which is occupied by the allocation of GROUP identified by pointer V. The value V->DIM is used to determine the bound of VALUES.

The GET Statement

Function:

The GET statement is a STREAM transmission statement that can be used in either of the following ways:

1. It can cause the assignment of data from an external source (that is, from a data set) to one or more internal receiving fields (that is, to one or more variables).

2. It can cause the assignment of data from an internal source (that is, from a character-string variable) to one or more internal receiving fields (that is, to one or more variables).

General format:

```
GET option-list;
```

Following is the format of "option list":

```
[ FILE(filename) | STRING(character-
    string-name) ]
  data-specification [COPY]
    [SKIP[(expression)]]
```

General rules:

1. If neither the FILE(filename) option nor the STRING(character-string-name) option appears, the file option FILE(SYSIN) is assumed.

2. One data specification must appear unless the SKIP option is specified.

3. The options may appear in any order.

4. The filename refers to a file which has been associated, by opening, with the data set which is to provide the values. It must be a STREAM INPUT file.

5. The "character-string name" refers to the character string that is to provide the data to be assigned to the data list. This name may be a reference to a built-in function. Each GET operation using this option always begins at the beginning of the specified string. If the number of characters in this string is less than the total number of characters specified by the data specification, the ERROR condition is raised.

6. When the STRING option is used under data-directed transmission, the ERROR condition is raised if an identifier within the string does not have a match within the data specification.

7. The data specification is as described in Part I, Chapter 8, "Input and Output."

8. If the FILE (filename) option refers to a file that is not open in the current task, the file is implicitly opened in the task for stream output transmission.

9. 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 installation print file.

10. The SKIP option causes a new current line to be defined for the data set. The expression, if present, is converted to an integer w, which must be greater than zero. If not, the F Compiler substitutes a value of 1. The data set is positioned at the start of the wth line relative to the current line. If the expression is omitted, SKIP(1) is assumed. The SKIP option is always executed before any data is transmitted.

Examples:

1. GET LIST (A,B,C);

   Specifies the list-directed transmission of the values to be assigned to A, B and C from the file SYSIN.

2. GET FILE (BETA) EDIT (X,Y,Z) (A(5), F(5,2), A(10));

   Specifies the edit-directed transmission of the values assigned to X, Y and Z from file BETA.

The GO TO Statement

Function:

The GO TO statement causes control to be transferred to the statement identified by the specified label.

General format:

```
{GO TO} {label-constant;                }
{GOTO } {element-label-variable;        }
```

General rules:

1. If an "element label variable" is specified, the value of the label variable determines 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.

2. A GO TO statement cannot pass control to an inactive block.

3. A GO TO statement cannot transfer control from outside a DO-group to a statement inside the DO-group if the DO-group specifies repetitive execution, unless the GO TO terminates a procedure or on-unit invoked from within the DO-group.

4. If a GO TO statement transfers control from within a block to a point not contained within that block, the block is terminated, Also, if the transfer point is contained in a block that did not directly activate the block being

terminated, all intervening blocks in the activation sequence are also terminated (see Part I, Chapter 6, "Blocks, Flow of Control, and Storage Allocation," for examples and details). When one or more blocks are terminated by a GO TO statement, conditions are reinstated and automatic variables are freed just as if the blocks had terminatd in the usual fashion.

5. When a GO TO statement transfers control out of a procedure that has been invoked as a function, the evaluation of the expression that contained the corresponding function reference is discontinued.

The IF Statement

Function:

The IF statement tests the value of a specified expression and controls the flow of execution according to the result of that test.

General format:

```
IF element-expression
        THEN unit-1
        [ELSE unit-2]
```

Syntax rules:

1. Each unit is either a single statement (except DO, END, PROCEDURE, BEGIN, DECLARE, FORMAT, or ENTRY), a DO-group, or a begin block.

2. The IF statement itself is not terminated by a semicolon; however, each "unit" specified must be terminated by a semicolon.

3. Each "unit" may be labeled and may have condition prefixes.

General rules:

1. The element expression is evaluated and, if necessary, converted to a bit string. When the ELSE clause (that is, ELSE and its following "unit") is specified, the following occurs:

   If any bit in the string is 1, "unit-1" is executed, and control then passes to the statement following the IF statement. If all bits in the string have the value 0, "unit-1" is skipped and "unit-2" is executed, after which control passes to the next statement.

   When the ELSE clause is not specified, the following occurs:

   If any bit in the string is 1, "unit-1" is executed, and control then passes to the statement following the IF statement. If all bits are 0, "unit-1" is not executed and control passes to the next statement.

   Each "unit" may contain statements that specify a transfer of control (e.g., GO TO); hence, the normal sequence of the IF statement may be overriden.

2. IF statements may be nested; that is, either "unit", or both, may itself be an IF statement. Since each ELSE clause is always associated with the innermost unmatched IF in the same block or DO-group, an ELSE with a null statement may be required to specify a desired sequence of control.

The LOCATE Statement

Function:

The LOCATE Statement, which applies to BUFFERED OUTPUT files, causes allocation of a based variable in a buffer; it may also cause transmission of a previously allocated based variable.

General format:

```
LOCATE variable;

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

Syntax rules:

1. The options may appear in any order.

2. The "variable" must be an unsubscripted level 1 based variable.

General rules:

1. The FILE(filename) option specifies the file involved. This option must appear.

2. Execution of a LOCATE statement causes the specified based variable to be allocated in the buffer. Components of the based variable that have been specified in REFER options are initialized. A pointer value is assigned to the pointer variable named in the SET option or, if the SET option is

omitted, to the pointer variable specified in the declaration of the based variable. The pointer value identifies the record in the buffer. After execution of the LOCATE statement, values may be assigned to the based variable for subsequent transmission to the file, which will occur immediately before the next LOCATE, WRITE, or CLOSE operation on the file, at which time the record is freed.

3. If the KEYFROM(expression) option appears, the value of the expression is converted to a character string and is used as the key of the record when it is subsequently written.

4. If the FILE(filename) option refers to 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 REC_POINT is set to identify ALPHA in the buffer. Values may subsequently be assigned to ALPHA and the record will be written in the data set associated with file BETA when a subsequent LOCATE or WRITE statement is executed for file BETA or if BETA is closed, either explicitly or implicitly.

The Null Statement

Function:

The null statement causes no action and does not modify sequential statement execution. If the label of a null statement is enabled for the CHECK condition, CHECK is raised whenever control reaches the null statement.

General format:

```
[label:]...;
```

The ON Statement

Function:

The ON statement specifies what action is to be taken (programmer-defined or standard system action) when an interrupt results from the occurrence of the specified exceptional condition.

General format:

```
ON condition[SNAP]{SYSTEM;|on-unit}
```

Syntax rules:

1. The condition may be any of those described in Section H, "ON-Conditions".

2. The "on-unit" represents a programmer-defined action to be taken when an interrupt results from the occurrence of the specified "condition". It can be either a single unlabeled simple statement or an unlabeled begin block. If it is an unlabeled simple statement, it can be any simple statement except BEGIN, DO, END, RETURN, FORMAT, PROCEDURE, or DECLARE. If the on-unit is an unlabeled begin block, any statement can be used freely within that block, with one exception: A RETURN statement can appear only within a procedure nested within the begin block.

3. Since the "on-unit" itself requires a semicolon, no semicolon is shown for the "on-unit" in the general format. However, the word SYSTEM must be followed by a semicolon.

General rules:

1. The ON statement determines how an interrupt occurring for the specified condition is to be handled. Whether the interrupt is handled in a standard system fashion or by a programmer-supplied method is determined by the action specification in the ON statement, as follows:

   a. If the action specification is SYSTEM, the standard system action is taken. The standard system action is not the same for every condition, although for most conditions the system simply prints a message and raises the ERROR condition. The standard system action for each condition is given in Section H, "ON-Conditions." (Note that the standard system action is always taken if an interrupt occurs and no ON statement for the condition is in effect.)

   b. If the action specification is an "on-unit," the programmer has supplied his own interrupt-handling action, namely, the action defined by the statement(s) in the on-unit

itself. The on-unit is not executed when the ON statement is executed; it is executed only when an interrupt results from the occurrence of the specified condition (or if the interrupt results from the condition being signaled by a SIGNAL statement).

2. The action specification (i.e., "on-unit" or SYSTEM) established by executing an ON statement in a given block remains in effect throughout that block and throughout all blocks in any activation sequence initiated by that block, unless it is overridden by the execution of another ON statement or a REVERT statement, as follows:

   a. If a later ON statement specifies the same condition as a prior ON statement and this later ON statement is executed in a block that lies within the activation sequence initiated by the block containing the prior ON statement, the action specification of the prior ON statement is temporarily suspended, or stacked. It can be restored either by the execution of a REVERT statement, or by the termination of the block containing the later ON statement.

   b. If the later ON statement and the prior ON statement are internal to the same invocation of the same block, the effect of the prior ON statement is completely nullified.

3. An on-unit is always treated by the compiler as a procedure internal to the block in which it appears. (Conceptually, it is enclosed in PROCEDURE and END statements.) Any names used in an on-unit do not belong to the invocation of the procedure or block in which the interrupt occurred (and, hence, effectively, the procedure or block in which the on-unit is executed) but, rather, to the environment of the invocation of the procedure or block in which the ON statement for that on-unit was executed. (Remember that an ON statement is executed as it is encountered in statement flow; whereas, the action specification for that ON statement is executed only when the associated interrupt occurs.)

4. A condition raised during execution results in an interrupt if and only if the condition is <u>enabled</u> at the point where it is raised.

   a. The conditions AREA, OVERFLOW, FIXEDOVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, all of the input/output conditions, and the conditions CONDITION, FINISH, and ERROR are enabled by default.

   b. The conditions SIZE, STRINGRANGE, SUBSCRIPTRANGE, and CHECK are disabled by default.

   c. The enabling and disabling of OVERFLOW, FIXEDOVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, SIZE, STRINGRANGE, SUBSCRIPTRANGE, and CHECK can be controlled by condition prefixes.

5. If on-unit is a single statement, it cannot refer to a remote format specification.

6. If SNAP is specified, then when the given condition occurs and the interrupt results, a <u>calling trace</u> is listed; that is, a trace of all of the procedures active at the time the interrupt resulted is printed on SYSPRINT.

<u>The OPEN Statement</u>

Function:

The OPEN statement <u>opens</u> a file by associating a file name with a data set. It also can complete the specification of attributes for the file, if a complete set of attributes has not been declared for the file being opened.

General format:

```
OPEN FILE(file-name)[options-group]
 [,FILE(file-name)[options-group]]...;
```

where "options group" is as follows;

```
[DIRECT|SEQUENTIAL]
[BUFFERED|UNBUFFERED]
[STREAM|RECORD]
[INPUT|OUTPUT|UPDATE]
[KEYED][EXCLUSIVE]
[BACKWARDS]
[TITLE (element-expression)]
[PRINT]
[LINESIZE(element-expression)]
[PAGESIZE(element-expression)]
```

Syntax rules:

1. The INPUT, OUTPUT, UPDATE, STREAM RECORD, DIRECT, SEQUENTIAL, BUFFERED, UNBUFFERED, KEYED, EXCLUSIVE, BACK-

WARDS, and PRINT options specify attributes that augment the attributes specified in the file declaration; for rules governing which of these attributes can be applied together, see Part I, Chapter 8, "Input and Output," and the corresponding attributes in Section I, "Attributes."

2. The options in an "option group" and the FILE option for a file may appear in any order.

3. The "file name" is the name of the file that is to be associated with a data set. Several files can be opened by one OPEN statement.

General rules:

1. The opening of an already open file does not affect the file if the second opening takes place in the same task or an attached task. In such cases, any expressions in the "options_group" are evaluated, but they are not used.

2. If the TITLE option is specified, the "element expression" is converted to a character string, if necessary, the first eight characters of which identify the data set (the ddname) to be associated with the file. If this option does not appear, the first eight characters of the file name (padded or truncated) are taken to be the ddname. Note that this is not the same truncation as that for external names. If the file name is a parameter, the identifier of the original argument passed to the parameter, rather than the identifier of the parameter itself, is used as the identification.

3. The LINESIZE option can be specified only for a STREAM OUTPUT file. The expression is evaluated, converted to an integer, and used as the length of a line during subsequent operations on the file. New lines may be started by use of the printing and control format items or by options in a GET or PUT statement. If an attempt is made to position a file past the end of a line before explicit action to start a new line is taken, a new line is automatically started, and the file is positioned to the start of this new line. If no LINESIZE is given for a PRINT file, an implementation-defined default is supplied. For the F Compiler, this is 120 characters.

   The LINESIZE option cannot be specified for an INPUT file. The line size taken into consideration whenever a SKIP option appears in a GET statement is the line size that was used to create the data set, if any; otherwise, the line size is taken to be the current length of the logical record (minus control bytes, for V-format records).

4. The PAGESIZE option can be specified only for a file having the STREAM and PRINT attributes. The element expression is evaluated and converted to an integer, which represents the maximum number of lines to a page. During subsequent transmission to the PRINT file, a new page may be started by use of the PAGE format item or by the PAGE option in the PUT statement. If a page becomes filled and more data remains to be printed before action to start a new page is taken, the ENDPAGE condition is raised. For the F Compiler, if PAGESIZE is not specified, the default is 60 lines per page.

The PROCEDURE Statement

Function:

The PROCEDURE statement has the following functions:

- It heads a procedure.
- It defines the primary entry point to the procedure.
- It specifies the parameters, if any, for the primary entry point.
- It may specify certain special characteristics that a procedure can have.
- It may specify the attributes of the value that is returned by the procedure if it is invoked as a function at its primary entry point.

General format:

```
entry-name: [entry-name:]...
   PROCEDURE[(parameter[,parameter]...)]
   [OPTIONS (option-list)]
   [RECURSIVE] [data-attributes];
```

Syntax rules:

1. The "data attributes" represent the attributes of the value returned by the procedure when it is invoked as a function at its primary entry point. Only arithmetic, string, pointer, offset, AREA, and PICTURE attributes are allowed.

2. OPTIONS and RECURSIVE are special procedure specifications that the user can specify. They and the "data attributes" may appear in any order and are separated by blanks.

3. The "option list" of OPTIONS specifies one or more additional implementation-defined options. For the F Compiler, the "option list" may contain the MAIN and TASK options, separated by commas. MAIN specifies that the procedure is the initial procedure, to be invoked by the operating system as the first step in the execution of the PL/I program; TASK specifies that the multitasking facilities are to be used.

General rules:

1. When the procedure is invoked, a relationship is established between the arguments passed to the procedure and the parameters that represent those arguments in the invoked procedure. This topic is discussed in Part I, Chapter 10, "Subroutines and Functions."

2. For the F Compiler, OPTIONS may be specified only for an external procedure, and at least one external procedure must have the OPTIONS (MAIN) designation; if more than one is so designated, the operating system will invoke the one that appears first, physically. (If multitasking is to be used, the external procedure must also have the keyword TASK in the OPTIONS attribute.) OPTIONS applies to all of the entry points (both primary and secondary) that the procedure for which it has been declared might have.

3. RECURSIVE must be specified if the procedure might be invoked recursively; that is, if it might be reactivated while it is still active. If specified, it applies to all of the entry points (primary and secondary) that the procedure might have. It applies only to the procedure for which it is declared.

4. The "data attributes" specify the attributes of the value returned by the procedure when it is invoked as a function at its primary entry point. The value specified in the RETURN statement of the invoked procedure is converted to conform with these attributes before it is returned to the invoking procedure.

   If "data attributes" are not specified, default attributes are supplied. In such a case, the name of the entry point (the entry name by which the procedure has been invoked) is used to determine the default base, precision, and scale. (Since the entry point can have several entry names, the default base, precision, and scale can differ according to the entry name.)

5. If a PROCEDURE statement has more than one entry name, the first name can be considered as the only label of the statement; each subsequent entry name can be considered 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 BINARY FIXED:
```

   is effectively the same as:

```
A: PROCEDURE BINARY FIXED;

I: ENTRY BINARY FIXED;
```

The PUT Statement

Function:

The PUT statement is a STREAM transmission statement that can be used in either of the following ways:

1. It can cause the values in one or more internal storage locations to be transmitted to a data set on an external medium.

2. It can cause the values in one or more internal storage locations to be assigned to an internal receiving field (represented by a character-string variable).

General format:

```
PUT[ FILE (file-name)                                  ]
   [STRING (character-string-variable)                 ]
    [data-specification]
   [PAGE [LINE(element-expression)]                    ]
   [SKIP [(element-expression)]                        ];
   [LINE(element-expression)                           ]
```

Syntax rules:

1. If neither the FILE nor STRING option appears, the specification FILE (SYSPRINT) is assumed. If such a PUT statement lies within the scope of a declaration of the identifier SYSPRINT, SYSPRINT must have been

declared as FILE STREAM OUTPUT. If the PUT statement does not lie within the scope of a declaration of SYSPRINT, SYSPRINT is the standard system output file.

2. The FILE option specifies transmission to a data set on an external medium. The file name in this option is the name of the file that has been associated (by implicit or explicit opening) with the data set that is to receive the values. This file must have the OUTPUT and STREAM attributes.

3. The STRING option specifies transmission from internal storage locations (represented by variables or expressions in the "data specification") to a character string (represented by the "character-string variable"). The "character-string variable" can be a string pseudo-variable.

4. The "data specification" option is as described in Part I, Chapter 8, "Input and Output."

5. The PAGE, SKIP, and LINE options cannot appear with the STRING option.

6. The options may appear in any order; at least one must appear.

General rules:

1. If the FILE option is specified, and the "file name" refers to an unopened file, the file is opened implicitly as an OUTPUT file.

2. If the STRING option is specified, the PUT operation begins assigning values to the beginning of the string (that is, at the left-most character position), after appropriate conversions have been performed. Blanks and delimiters are inserted as usual. If the string is not long enough to accomodate the data, the ERROR condition is raised.

3. The PAGE and LINE options can be specified for PRINT files only. All of the options take effect before transmission of any values defined by the data specification, if given. Of the three, only PAGE and LINE may appear in the same PUT statement, in which case, the PAGE option is applied first.

4. The PAGE option causes a new current page to be defined within the data set. If a data specification is present, the transmission of values occurs after the definition of the new page. The page remains current until the execution of a PUT statement with the PAGE option, until a PAGE format item is encountered, or until an ENDPAGE interrupt results in the definition of a new page. A new current page implies line one.

5. The SKIP option causes a new current line to be defined for the data set. The expression, if present, is converted to an integer w, which for non-PRINT files must be greater than zero. The data set is positioned at the start of the wth line relative to the current line. If the expression is omitted, SKIP(1) is assumed.

   For PRINT files w may be less than or equal to zero; in this case, the effect is that of a carriage return with the same current line. If less than w lines remain on the current page when a SKIP(w) is issued, ENDPAGE is raised.

6. 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.

7. If the FILE(filename) option refers to a file that is not open in the current task, the file is opened implicitly in this task for stream output.

Examples:

1. PUT DATA (A,B,C);

   Specifies the data-directed transmission of the values A, B and C to the file SYSPRINT.

2. PUT FILE (LIST) EDIT (X,Y,Z) (A(10)) PAGE;

   Specifies that a new page is to be defined for the print file LIST. The values of X, Y and Z are placed starting in the first printing position of the new page. Each of the values will use the A(10) format item.

The READ Statement

Function:

The READ statement causes a record to be transmitted from a RECORD INPUT or RECORD UPDATE file to a variable or buffer.

General format:

```
READ option-list;
```

Following is the format of "option list":

```
FILE (filename)
      [INTO (variable)                    ]
      [SET(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 is not open in the current task, it is opened.

3. The INTO(variable) option specifies an unsubscripted level 1 variable into which the record is to be read. It cannot be a parameter, nor can it have the DEFINED attribute.

4. The KEY and KEYTO options can be specified for KEYED files only.

5. The KEY option must appear if the file has the DIRECT attribute. The "element expression" is converted to a character string that represents a key. It is this key that determines which record will be read.

   The KEY option may also appear for files having the SEQUENTIAL and KEYED attributes. In such cases, the file is positioned to the record having the specified key. Thereafter, records may be read sequentially from that point on by using READ statements without the KEY option. For System/360 implementations, the data set must have the INDEXED organization.

6. The KEYTO option can be given only if the file has the SEQUENTIAL and KEYED attributes. It specifies that the key of the record being read is to be assigned to the "character-string variable" according to the rules for character-string assignment. If proper assignment cannot be made, the KEY condition is raised. For the F Compiler, the value assigned is as follows:

   a. For REGIONAL (1), the eight character regional number, padded or truncated on the left to the declared length of the character-string variable

   b. For REGIONAL (2) and REGIONAL (3), the recorded key without the regional number, padded or truncated on the right to the declared length of the character-string variable

   c. For INDEXED, the recorded key, padded or truncated on the right to the declared length of the character-string variable

   The KEY condition will not be raised for such padding or truncation.

   Note: The KEYTO option cannot specify a variable declared with a numeric picture specification.

7. The EVENT option allows processing to continue while a record is being read or ignored. This option cannot be specified for a SEQUENTIAL BUFFERED file.

   When control reaches a READ statement containing this option, the "event variable" is made active (that is, it cannot be associated with another event) and is given the completion value '0'B, provided that the UNDEFINEDFILE condition is not raised by an implicit file opening (see "Note" below). The event variable remains active and retains its '0'B completion value until control reaches a WAIT statement specifying that event variable. At this time, either of the following can occur:

   a. If the READ statement has been executed successfully and none of the conditions ENDFILE, TRANSMIT, KEY or RECORD has been raised as a result of the READ, the event variable is set complete (given the completion value '1'B), and the event variable is made inactive, that is, it can be associated with another event.

b. If the READ statement has resulted in the raising of ENDFILE, TRANSMIT, KEY, or RECORD, the interrupt for each of these conditions does not occur until the WAIT is encountered. At such time, the corresponding on-units (if any) are entered in the order in which the conditions were raised. After a return from the final on-unit, or if one of the on-units is terminated by a GO TO statement, the event variable is given the completion value '1'B and is made inactive.

Note: If the READ statement causes an implicit file opening that results in the raising of UNDEFINEDFILE, the on-unit associated with this condition is entered immediately and the event variable remains unchanged; that is, the event variable remains inactive and retains the same value it had when the READ was encountered. If the on-unit does not correct the condition, then, upon normal return from the on-unit, the ERROR condition is raised; if the condition is corrected in the on-unit, that is, if the file is opened successfully, then, upon normal return from the on-unit, the event variable is set to '0'B, it is made active, and execution of the READ statement continues.

8. 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, REWRITE, or DELETE statement that specifies the same key, by a CLOSE statement, or by completion of task in which the record was locked.

Note that a record is considered locked only for tasks other than the task that actually locks it; in other words, a locked record can always be read by the task that locked it and still remain locked as far as other tasks are concerned (unless, of course, the record has been explicitly unlocked by one of the above methods).

9. The SET option specifies that the record is to be read into a buffer and that a pointer value is to be assigned to the named locator variable. The pointer value identifies the record in the buffer. If the locator variable is an offset variable, the pointer value is implicitly converted.

10. The IGNORE option may be specified for SEQUENTIAL INPUT and SEQUENTIAL UPDATE files. The expression in the IGNORE option is evaluated and converted to an integer. If the value, n, is greater than zero, n records are ignored; a subsequent READ statement for the file will access the (n+1)th record. A READ statement without an INTO, SET, or IGNORE option is equivalent to a READ with an IGNORE(1).

11. A keyed file being accessed sequentially may be positioned by issuing a READ statement with the KEY option. The specified key will be used to identify the record required. Thereafter, records may be read sequentially from that point by use of READ statements without the KEY option. This applies to INPUT and UPDATE files.

For BUFFERED SEQUENTIAL files, two positioning statements can be used, with the following formats:

```
READ  FILE (filename)
      INTO (variable)
      KEY (expression);

READ  FILE (filename)
      SET (pointer-variable)
      KEY (expression);
```

For UNBUFFERED SEQUENTIAL files, only the first form shown immediately above can be used, and it may be specified with the EVENT option.

Examples:

1. READ FILE (ALPHA) SET (REC_IDENT);

   The next record from the data set associated with ALPHA is made available and the pointer variable REC_IDENT is set to identify the record in the buffer.

2. READ FILE (BETA) KEY (VALUE) INTO (WORK);

   The record identified by the key VALUE is transmitted from the data set associated with BETA into the variable WORK.

The RETURN Statement

Function:

The RETURN statement terminates execution of the procedure that contains the RETURN statement. If the procedure has not been invoked as a task, the RETURN statement returns control to the invoking procedure. The RETURN statement may also return a value.

General format:

Option 1.

```
RETURN;
```

Option 2.

```
RETURN (expression);
```

General rules:

1. Only the RETURN statement in Option 1 can be used to terminate procedures not invoked as function procedures; control is returned to the point logically following the invocation.

   Option 1 represents the only form of the RETURN statement that can be used to terminate a procedure initiated as a task. If the RETURN statement terminates the major task, the FINISH condition is raised prior to the execution of any termination processes. If the RETURN statement terminates any other task, the completion value of the associated event variable (if any) is set to '1'B, and the status value is left unchanged.

2. The RETURN statement in Option 2 is used to terminate a procedure invoked as a function procedure only. Control is returned to the point of invocation, and the value returned to the function reference is the value of the expression specified converted to conform to the attributes declared for the invoked entry point. These attributes may be explicitly specified at the entry point; they are otherwise implied by the initial letter of the entry name through which the procedure is invoked.

3. If control reaches an END statement corresponding to the end of a procedure, this END statement is treated as a RETURN statement (of the Option 1 form) for the procedure.

Example:

```
A:  PROCEDURE (X,Y) FIXED;
    DECLARE (X,Y) FLOAT;
    .
    .
    .
    RETURN (X**2+Y**2);
    END;
B:  PROCEDURE;
    DECLARE A ENTRY RETURNS (FIXED);
    .
    .
    .
    R = A(P,Q);
    .
    .
    .
    END;
```

In the assignment statement (R=A(P,Q);), procedure B invokes procedure A as a function. Procedure B specifies that the scalar expression in the RETURN statement is to be evaluated; since X and Y are floating-point variables and the PROCEDURE statement specifies that the value returned is to be fixed point, the value of the expression is converted to fixed point, and this value is returned to B.

The REVERT Statement

Function:

The REVERT statement nullifies the effect of the current action specification for the specified condition only if the current action specification is the result of an ON statement executed within the same invocation of the block in which the REVERT statement is executed. When this is true, the action specification that was in effect for the specified condition when the block containing the REVERT statement was invoked is re-established and once again takes effect.

General format:

```
REVERT condition;
```

Syntax rule:

The "condition" is any of those described in Section H, "ON-Conditions."

General rule:

The execution of a REVERT statement has the effect described above only if (1) an ON statement, specifying the same condition and internal to the same block, was executed after the block was activated and (2) the execution of no other similar REVERT statement has intervened. If either of these two conditions is not met, the REVERT statement is treated as a null statement.

The REWRITE Statement

Function:

The REWRITE statement can be used only for update files. It replaces an existing record in a data set.

General format:

```
REWRITE FILE (file-name)
   [FROM(variable)]
   [KEY (element-expression)]
   [EVENT (event-variable)]
```

Syntax rules:

1. The FILE specification, which includes the file name, and the options may be specified in any order.

2. The file name is the name of the file containing the record to be rewritten. The file must have the UPDATE attribute.

3. The "variable" in the FROM option must be an unsubscripted level 1 variable (that is, a variable not contained in an array or structure). It cannot have the DEFINED attribute and it cannot be a parameter. It represents the record that will replace the existing record in the specified file.

General rules:

1. If the file whose name appears in the FILE specification has not been opened, it is opened implicitly with the attributes RECORD and UPDATE.

2. The KEY option must appear if the file has the DIRECT attribute; it cannot appear otherwise. The element-expression is converted to a character string. This character string is the source key that determines which record is to be rewritten. For INDEXED SEQUENTIAL files in System/360 implementations, the key must be the same as the one it replaces.

3. The FROM option must be specified for UPDATE files having either the DIRECT attribute or both the SEQUENTIAL and UNBUFFERED attributes. A REWRITE statement in which the FROM option has not been specified has the following effect: if the last record was read by a READ statement with the INTO option, REWRITE without FROM has no effect on the record in the data set; if the last record was read by a READ statement with the SET option, the record will be updated by whatever assignments were made in the buffer identified by the pointer variable in the SET option.

4. The EVENT option allows processing to continue while a record is being rewritten. This option cannot be specified for a SEQUENTIAL BUFFERED file.

   When control reaches a REWRITE statement containing this option, the event variable is made active (that is, it cannot be associated with another event) and is given the completion value '0'B, provided that the UNDEFINEDFILE condition is not raised by an implicit file opening (see "Note" below). The event variable remains active and retains its '0'B completion value until control reaches a WAIT statement specifying that event variable. At this time, either of the following can occur:

   a. If the REWRITE statement has been executed successfully and none of the conditions TRANSMIT, KEY, or RECORD has been raised as a result of the REWRITE, the event variable is set complete (given the completion value '1'B), and the event variable is made inactive (that is, it can be associated with another event).

   b. If the REWRITE statement has resulted in the raising of TRANSMIT, KEY, or RECORD, the interrupt for each of these conditions does not occur until the WAIT is encountered. At such time, the corresponding on-units (if any) are entered in the order in which the conditions were raised. After a return from the final on-unit, or if one of the on-units is terminated by a GO TO statement, the event variable is given the completion value '1'B and is made inactive.

   Note: If the REWRITE statement causes an implicit file opening that results in the raising of UNDEFINEDFILE, the on-unit associated with this condition

is entered immediately and the event variable remains unchanged, that is, the event variable remains inactive and retains the same value it had when the REWRITE was encountered. If the on-unit does not correct the condition, then, upon normal return from the on-unit, the ERROR condition is raised; if the condition is corrected in the on-unit, that is, if the file is opened successfully, then, upon normal return from the on-unit, the event variable is set to '0'B, it is made active, and execution of the REWRITE statement continues.

5. If the record rewritten is one that was locked in the same task, it becomes unlocked.

The SIGNAL Statement

Function:

The SIGNAL statement simulates the occurence of an interrupt. It may be used to test the current action specification for the associated condition.

General format:

```
SIGNAL condition;
```

Syntax rule:

The "condition" is any one of those described in Section H, "ON-Conditions."

General rules:

1. When a SIGNAL statement is executed, it is as if the specified condition has actually occurred. Sequential execution is interrupted and control is transferred to the current on-unit for the specified condition. After the on-unit has been executed, control normally returns to the statement immediately following the SIGNAL statement.

2. The on-condition CONDITION can cause an interrupt only as a result of its specification in a SIGNAL statement.

3. If the specified condition is disabled, no interrupt occurs, and the SIGNAL statement becomes equivalent to a null statement.

4. If there is no current on-unit for the specified condition, then the standard system action for the condition is performed.

The STOP Statement

Function:

The STOP statement causes immediate termination of the major task and all subtasks

General format:

```
STOP;
```

General rule:

Prior to any termination activity the FINISH condition is raised in the task in which the STOP is executed. On normal return from the FINISH on-unit, all tasks in the program are terminated.

The UNLOCK Statement

Function:

The UNLOCK statement makes the specified locked record available to other tasks for operations on the record.

General format:

```
UNLOCK option-list;
```

Following is the format of "option list":

```
FILE(filename) KEY(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.

3. In the KEY(expression) option, the "expression" is converted to a character string and determines which record is unlocked.

4. A record can be unlocked only by the task which locked it.

The WAIT statement

Function:

The execution of a WAIT statement within an activation of a block retains control for that activation of that block within the WAIT statement until certain specified events have completed.

General format:

```
WAIT (event-name [,event-name]...)
  [(element-expression)];
```

General rules:

1. Control for a given block activation remains within this statement until, at possibly separate times during the execution of the statement, the condition

   ```
   COMPLETION(event-name) = '1'B
   ```

   has been satisfied, for some or all of the event names in the list.

2. If the optional expression does not appear, all the event names in the list must satisfy the above condition before control returns to the next statement in this task following the WAIT.

3. If the optional expression appears, the expression is evaluated when the WAIT statement is executed and converted to an integer. This integer specifies the number of events in the list that must satisfy the above condition before control for the block passes to the statement following the WAIT. Of course, if an on-unit entered due to the WAIT is terminated abnormally, control might not pass to the statement following the WAIT.

   If the value of the expression is zero or negative, the WAIT statement is treated as a null statement. If the value of the expression is greater than the number, n, of 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.

4. If the event variable named in the list has been associated with a task in its attaching CALL statement, then the condition in Rule 1, will be satisfied on termination of that task.

5. If the event variable named in the list is associated with an input/output operation initiated in the same task as the WAIT, the condition in Rule 1 will be satisfied when the input/output operation is completed. The execution of the WAIT is a necessary part of the completion of an input/output operation. If prior to, or during, the WAIT all transmission associated with the input/output operation is terminated, then the WAIT performs the following action: If the transmission has finished without requiring any input/output conditions to be raised, the event variable is set complete (i.e., COMPLETION(event name) = '1'B). If the transmission has been terminated but has required conditions to be raised, the event variable is set abnormal (i.e., STATUS(event name) = 1) and all the required ON conditions are raised. On return from the last on-unit, the event variable is set complete.

6. The order in which ON conditions for different input/output events are raised is not dependent on the order of appearance of the event names in the list. If an ON condition for one event is raised, then all other conditions for that event are raised before the WAIT is terminated or before any other input/output conditions are raised unless an abnormal return is made from one of the on-units thus entered. The raising of ON conditions for one event implies nothing about the completion or termination of transmission of other events in the list.

7. If an abnormal return is made from any on-unit entered from a WAIT, the associated event variable is set complete, the execution of the WAIT is terminated, and control passes to the point specified by the abnormal return.

8. If some of the event names in the WAIT list are associated with input/output operations and have not been set complete before the WAIT is terminated (either because enough events have been completed or due to an abnormal return), then these incomplete events will not be set complete until the execution of another WAIT referring to these events in this same task.

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 is a RECORD transmission statement that transfers a record from a variable in internal storage to an OUTPUT or UPDATE file.

General format:

```
WRITE FILE (file-name) FROM (variable)
   [KEYFROM(element-expression) ]
   [EVENT(event-variable)];
```

Syntax rules:

1. The FILE and FROM specifications and the KEYFROM and EVENT options may appear in any order.

2. The "file name" specifies the file in which the record is to be written. This file must be a RECORD file that has either the OUTPUT attribute or the DIRECT and UPDATE attributes.

3. The "variable" in the FROM specification must be an unsubscripted level 1 variable (i.e., a variable not contained in an array or structure). It cannot have the DEFINED attribute and it cannot be a parameter. It contains the record to be written.

General rules:

1. If the file is not open, it is opened implicitly with the attributes RECORD and OUTPUT (unless UPDATE has been declared).

2. If the KEYFROM option is specified, the "element expression" is converted to a character string. This character string is the source key that specifies the relative location in the data set where the record is written. For REGIONAL (2), REGIONAL (3), and INDEXED data sets, with the F Compiler KEYFROM also specifies a recorded key whose length is determined by the KEYLEN subparameter.

3. The EVENT option allows processing to continue while a record is being written. This option cannot be specified for a SEQUENTIAL BUFFERED file.

   When control reaches a WRITE statement containing this option, the "event variable" is made active (that is, it cannot be associated with another event) and is given the completion value '0'B, provided that the UNDEFINEDFILE condition is not raised by an implicit file opening (see "Note" below). The event variable remains active and retains its '0'B completion value until control reaches a WAIT statement specifying that event variable. At this time, either of the following can occur:

   a. If the WRITE statement has been executed successfully and none of the conditions TRANSMIT, KEY, or RECORD has been raised as a result of the WRITE, the event variable is set complete (given the completion value '1'B), and the event variable is made inactive, that is, it can be associated with another event.

   b. If the WRITE statement has resulted in the raising of TRANSMIT, KEY, or RECORD, the interrupt for each of these conditions does not occur until the WAIT is encountered. At such time, the corresponding on-units (if any) are entered in the order in which the conditions were raised. After a return from the final on-unit, or if one of the on-units is terminated by a GO TO statement, the event variable is given the completion value ('1'B) and is made inactive.

   Note: If the WRITE statement causes an implicit file opening that results in the raising of UNDEFINEDFILE, the on-unit associated with this condition is entered immediately and the event variable remains unchanged; that is,

the event variable remains inactive and retains the same value it had when the WRITE was encountered. If the on-unit does not correct the condition, then, upon normal return from the on-unit, the ERROR condition is raised; if the condition is corrected in the on-unit, that is, if the file is opened successfully, then upon normal return from the on-unit, the event variable is set to '0'B, it is made active, and execution of the WRITE statement continues.

PREPROCESSOR STATEMENTS

All of the statements that can be executed at the preprocessor stage are presented alphabetically in this section.

The %ACTIVATE Statement

Function:

The appearance of an identifier in a %ACTIVATE statement makes it active and eligible for replacement; that is, any subsequent encounter of that identifier in a nonpreprocessor statement, while the identifier is active, will initiate replacement activity.

General format:

```
%[label:]... ACTIVATE identifier
        [,identifier]...;
```

Syntax rules:

1. Each identifier must be either a preprocessor variable or a preprocessor procedure name.
2. A %ACTIVATE statement cannot appear within a preprocessor procedure.

General rules:

1. An identifier cannot be activated initially by a %ACTIVATE statement; the appearance of that identifier in a %DECLARE statement serves that purpose. An identifier can be activated by a %ACTIVATE statement only after it has been deactivated by a %DEACTIVATE statement.
2. When an identifier is active (and has been given a value -- if it is a preprocessor variable) any encounter of that identifier within a nonpreprocessor statement will initiate replacement activity in all cases except when the identifier appears within a comment or within single quotes.

Example:

If the source program contains the following sequence of 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 preprocessed text generated by the above would be as follows (replacement blanks are not shown):

```
S = I*A(I)*3;

R = 20*T*2;
```

The % Assignment Statement

Function:

The % assignment statement is used to evaluate preprocessor expressions and to assign the result to a preprocessor variable.

General format:

```
%[label:]... preprocessor-variable =
             preprocessor-expression;
```

General rule:

When the value assigned to a preprocessor variable is a character string, this character string should not contain a preprocessor statement, nor should it contain unmatched comment or string delimiters.

The %DEACTIVATE Statement

Function:

The appearance of an identifier in a %DEACTIVATE statement makes it inactive and ineligible for replacement; that is, any subsequent encounter of that identifier in a nonpreprocessor statement will not initiate any replacement activity (unless, of course, the identifier has been reactivated in the interim).

General format:

```
%[label:]... DEACTIVATE identifier
        [,identifier]...;
```

Syntax rules:

1. Each "identifier" must be either a preprocessor variable, the SUBSTR built-in function, or a preprocessor procedure name.
2. A %DEACTIVATE statement cannot appear within a preprocessor procedure.

General rule:

The deactivation of an identifier does not strip it of its value, nor does it prevent it from receiving new values in subsequent preprocessor statements. Deactivation simply prevents any replacement for a particular identifier from taking place.

The %DECLARE Statement

Function:

The %DECLARE statement establishes an identifier as a preprocessor variable or a preprocessor procedure name and also serves to activate that identifier. An identifier must appear in a %DECLARE statement before it can be used as a variable or a procedure name in any other preprocessor statement.

General format:

The general format is shown in Figure J-3.

Syntax rules:

1. CHARACTER or FIXED must be specified if the "identifier" is a preprocessor variable; an entry declaration must be specified if the "identifier" is a preprocessor procedure name.
2. Only the attributes shown in the above format can be specified in a %DECLARE statement.
3. Factoring of attributes is allowed as for nonpreprocessor DECLARE statements.
4. Any label attached to a %DECLARE statement is ignored by the scan.

General rules:

1. No length can be specified with the CHARACTER attribute. If CHARACTER is specified, it is assumed that the associated identifier represents a varying-length character string that has no maximum length.
2. A preprocessor declaration is not known until it has been encountered by the scan. If a reference to a preprocessor variable or procedure is encountered in a preprocessor statement before the declaration for that variable or procedure has been scanned, then the reference is in error.
3. The scope of all preprocessor variables, procedure names, and labels is the entire source program scanned by the preprocessor, not including any preprocessor procedures that redeclare such identifiers. The scope of a declaration in a preprocessor procedure is limited to that procedure.

```
%[label:]...DECLARE identifier {FIXED|CHARACTER|entry-declaration}

                  [,identifier {FIXED|CHARACTER|entry-declaration}]...;

where the format of "entry declaration" is:

                        ENTRY[([CHARACTER|FIXED]
                             [,[CHARACTER|FIXED]]...)]
                        RETURNS(CHARACTER|FIXED)
```

Figure J-3. General Format of the %DECLARE Statement

4. An entry declaration must be specified for each preprocessor procedure in the source program. The ENTRY attribute specifies the number (and attributes, if desired) of the parameters of the procedure; the RETURNS attribute specifies the attribute of the value returned by that procedure.

   The ENTRY attribute in the entry declaration must account for every parameter specified in the %PROCEDURE statement of the preprocessor procedure. If the procedure has no parameters, ENTRY must be specified without the parenthesized list following; if the procedure has one parameter, ENTRY followed by empty closed parentheses -- ENTRY () -- will suffice; if the procedure has more than one parameter, the place of each additional parameter must be kept by a comma. Thus, ENTRY (,,FIXED) specifies three parameters, the third of which has the attribute FIXED; the preprocessor makes no assumptions about the attributes of the first two.

   The RETURNS attribute specifies the attribute of the value to be returned by the preprocessor procedure to the point of invocation. If, in fact, the attribute of the value does not agree with the attribute specified by RETURNS, no conversion is performed and errors may result.

   See "Preprocessor Procedures" in Part I, Chapter 12, "Compile-Time Facilities," for a discussion of the above attributes, as well as a discussion of the association of arguments and parameters at the time of invocation.

5. After a %DECLARE statement has been executed, it is replaced by a null statement so that any subsequent scanning through the statement has no effect.

The %DO Statement

Function:

The %DO statement is used in conjunction with a %END statement to delimit a preprocessor DO-group. It cannot be used in any other way.

General format:

```
%[label:]...DO [ i=m1 [ TO m2 [BY m3] ] ];
               [      [ BY m3 [TO m2] ] ]
```

Syntax rule:

The "i" represents a preprocessor variable, and "m1," "m2," and "m3" are preprocessor expressions.

General rule:

The expansion of a preprocessor DO-group is the same as the expansion for a corresponding nonpreprocessor DO-group and "i," "m1," "m2," and "m3" have the same meaning that the corresponding expressions in a nonpreprocessor DO-group have.

See "Preprocessor DO-Groups" in Part I, Chapter 12, "Compile-Time Facilities," for a discussion and an example of its use.

The %END Statement

Function:

The %END statement is used in conjunction with %DO or %PROCEDURE statements to delimit preprocessor DO-groups or preprocessor procedures.

General format:

```
% [label:]...  END [label];
```

Syntax rule:

The label following END must be a label of a %PROCEDURE or %DO statement. Multiple closure is permitted.

The %GO TO Statement

Function:

The %GO TO statement causes the preprocessor to continue its scan at the specified label.

General format:

```
% [label:]...  {GO TO|GOTO} label;
```

General rules:

1. The label following the keyword GO TO determines the point to which the scan will be transferred. It must be a label of a preprocessor statement, although it cannot be the label of a preprocessor procedure.

2. A preprocessor GO TO statement appearing within a preprocessor procedure cannot transfer control to a point outside of that procedure. In other words, the label following GO TO must be contained within the procedure.

3. See "The %INCLUDE Statement" for a restriction regarding the use of %GO TO with included text.

The %IF Statement

Function:

The %IF statement can control the flow of the scan according to the value of a preprocessor expression.

General format:

```
%[label:]...IF preprocessor-expression
        %THEN preprocessor-clause-1
        [%ELSE preprocessor-clause-2]
```

Syntax rule:

A preprocessor clause is any single preprocessor statement other than %DECLARE, %PROCEDURE, %END, or %DO (percent symbol included) or a preprocessor DO-group (percent symbols included). Otherwise, the syntax is the same as that for non-preprocessor IF statements.

General rules:

1. The preprocessor 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, clause-1 is executed and clause-2, if present, is ignored; if all bits are 0, clause-1 is ignored and clause-2, if present, is executed. In either case, the scan resumes immediately following the IF statement, unless, of course, a %GO TO in one of the clauses causes the scan to resume elsewhere.

2. %IF statements can be nested according to the rules for nesting nonpreprocessor IF statements.

The %INCLUDE Statement

Function:

The %INCLUDE statement is used to include (incorporate) strings of external text into the source program being scanned. This included text can contribute to the preprocessed text being formed.

General format:

The %INCLUDE statement is defined as follows for the F Compiler:

```
%[label:]...  INCLUDE
        { ddname-1 (member-name-1) }
        { member-name-1            }
        [ ,ddname-2 (member-name-2) ]
        [ ,member-name-2            ]   ...;
```

Syntax rules:

1. Each "ddname" and "member name" pair identifies the external text to be incorporated into the source program. This external text must be a member of a partitioned data set.

2. A "ddname" specifies the ddname occurring in the name field of the appropriate DD statement. Its associated "member name" specifies the name of the data set member to be incorporated. If "ddname" is omitted, SYSLIB is assumed, and the SYSLIB DD statement is required.

3. A %INCLUDE statement cannot be used in a preprocessor procedure.

General rules:

1. Included text can contain nonpreprocessor and/or preprocessor statements.

2. The included text is scanned, in sequence, in the same manner as the source program; that is, preprocessor statements are executed and replacements are made where required.

3. %INCLUDE statements can be nested. In other words, included text also can contain %INCLUDE statements. A %GO TO statement in included text can transfer control to a point in the source program or in any included text at an outer level of nesting, but the reverse is not permitted. An analogous situation exists for nested DO-groups that specify iterative execution: control can be transferred from an inner

group to an outer, containing group, but not from an outer group into an inner, contained group.

4. Preprocessor statements in included text must be complete. It is not permissible, for example, to have half of a %IF statement in included text and half in the other part of the source program.

Example:

If the source program contained the following sequence of statements:

```
%DECLARE (FILENAME1, FILENAME2)
        CHARACTER;

% FILENAME1 = 'MASTER';

% FILENAME2 = 'NEWFILE';

% INCLUDE DCLS;
```

and if the SYSLIB member name DCLS contained:

```
DECLARE (FILENAME1, FILENAME2)
        FILE RECORD INPUT
        DIRECT KEYED;
```

then the following would be inserted into the preprocessed text:

```
DECLARE (MASTER, NEWFILE)
        FILE RECORD INPUT
        DIRECT KEYED;
```

Note that this is a way in which a central library of file declarations can be used, with each user supplying his own names for the files being declared.

The % Null Statement

Function:

The % null statement can be used to provide transfer targets for %GO TO statements. It is also useful for balancing ELSE clauses in nested %IF statements.

General format:

```
% [label:]...;
```

The %PROCEDURE Statement

Function:

The %PROCEDURE statement is used in conjunction with a %END statement to delimit a preprocessor procedure. Such a preprocessor procedure is an internal function procedure that can be executed only at the preprocessor stage.

General format:

```
% label: [label:]...  PROCEDURE
     [(identifier [, identifier]...)]
     {CHARACTER|FIXED};
```

Syntax rules:

1. Each "identifier" is a parameter of the function procedure.

2. One of the attributes CHARACTER or FIXED must be specified to indicate the type of value returned by the function procedure. There can be no default.

General rules:

1. The only statements and groups that can be used within a preprocessor procedure are:

   a. the preprocessor assignment statement

   b. the preprocessor DECLARE statement

   c. the preprocessor DO-group

   d. the preprocessor GO TO statement

   e. the preprocessor IF statement

   f. the preprocessor null statement

   g. the preprocessor RETURN statement

   All of these statements and the DO-group must adhere to the syntax and general rules given for them in this section, with one exception; all percent symbols must be omitted.

2. A GO TO statement appearing in a preprocessor procedure cannot transfer control to a point outside of that procedure.

3. As implied by general rule 1, preprocessor procedures cannot be nested.

4. A preprocessor procedure can be invoked by a function reference in a preprocessor statement, or, if the

function procedure name is active, by the encounter of that name in a non-preprocessor statement.

The Preprocessor RETURN Statement

Function:

The preprocessor RETURN statement can be used only in a preprocessor procedure and, therefore, can have no leading %. It returns a value as well as control back to the point from which the preprocessor procedure was invoked.

General format:

```
[label:]... RETURN
    (preprocessor-expression);
```

General rule:

The value of the preprocessor expression is converted to the attribute specified in the %PROCEDURE statement before it is passed back to the point of invocation. If the point of invocation is in a nonpreprocessor statement, replacement activity can be performed on the returned value after that value has replaced the procedure reference.

This section provides definitions for most of the terms used in this publication.

access: the act that encompasses the reference to and retrieval of data.

action specification: in an ON statement, the on-unit or the single keyword SYSTEM, either of which specifies the action to be taken whenever an interrupt results from raising of the named condition.

activation: institution of execution of a block. A procedure block is activated when it is invoked at any of its entry points; a begin block is activated when it is encountered in normal sequential flow.

active:

1. the state in which a block is said to be after activation and before termination.

2. the state in which a preprocessor variable or preprocessor procedure is said to be when its value can replace the corresponding identifier in source program text.

3. the state in which an event variable is said to be as a result of its appearance in the EVENT option of an executed RECORD input/output statement. An event variable remains active, and, hence, cannot be associated with any other input/output operation, until a WAIT statement naming that event variable has been executed.

additive attributes: file attributes for which there are no defaults and which, if required, must always be stated explicitly.

address: a specific storage location at which a data item can be stored.

adjustable (bounds and lengths): bounds or lengths that may be different for different allocations of the associated variable. Adjustable bounds and lengths are specified as variables, expressions, or asterisks, which are evaluated separately at each allocation. They cannot be used for STATIC data.

allocated variable: a variable with which storage has been associated.

allocation: the association of storage with a variable.

alphabetic character: any of the characters A through Z and the alphabetic extenders #, $, and @.

alphameric character: an alphabetic character or a digit.

alternative attributes: attributes that may be chosen from groups of two or more alternatives. If none is specified, a default is assumed.

area: a block of storage defined by an area variable and reserved, on allocation, for the allocation of based variables.

arithmetic conversion: the transformation of a value from one arithmetic representation to another arithmetic representation.

argument: an expression, file name, statement label constant or variable, mathematical built-in function name, or entry name passed to an invoked procedure as part of the procedure reference.

arithmetic data: data that has the characteristics of base, scale, mode, and precision. It includes coded arithmetic data and numeric character data.

arithmetic operators: any of the prefix operators, + and -, or the infix operators +, -, *, /, and **.

array: a named, ordered collection of data elements, all of which have identical attributes. An array has dimensions, and elements that are identified by subscripts. An array can also be an ordered collection of identical structures.

array of structures: an ordered collection of structures formed by giving the dimension attribute to the name of a structure.

assignment: giving a value to a variable.

asynchronous: describes either the overlap of an input/output operation with the execution of statements, or the concurrent execution of procedures, using multiple flows of control.

attachment of a task: the invocation of a procedure that is to be executed asynchronously with the invoking procedure.

attribute: a descriptive property associated with a name or expression to describe a characteristic of data items, of a

file, or of an entry point the name may represent.

automatic storage: storage that is allocated at the activation of a block and released at the termination of that block.

base: the number system in terms of which an arithmetic value is represented. In PL/I, the base is binary or decimal.

based storage: storage whose allocation and release is controlled by the programmer, with immediate access to all unfreed allocations.

begin block: a collection of statements headed by a BEGIN statement and ended by an END statement that delimits the scope of names and, in general, is activated by normal sequential statement flow. It controls the allocation and freeing of automatic storage declared in that block.

binary: the number system based on the value 2.

bit: a binary digit, either 0 or 1.

bit string: a string composed of zero or more bits.

bit-string operators: the operators ¬(not), &(and), and |(or).

block: a begin block or a procedure block.

bounds: the upper and lower limits of an array dimension.

buffer: an intermediate area, used in input/output operations, into which a record is read during input and from which a record is written during output.

built-in function: one of the PL/I-defined functions.

call: the invocation of a subroutine by means of the CALL statement or the CALL option of the INITIAL attribute.

character string: A string composed of zero or more characters from the data character set.

coded arithmetic data: arithmetic data whose characteristics are given by the base, scale, mode, and precision attributes. The types for System/360 are packed decimal, binary full words, and hexadecimal floating-point.

comment: a string of characters, used for documentation, which is preceded by /* and terminated by */ and which is treated as a blank.

comparison operators: the operators ¬< < <= ¬= = >= > ¬>

compile time: in general, the time during which a source program is translated into an object module. In PL/I, it is the time during which a source program can be altered (preprocessed), if desired, and then translated into an object program.

compiler: a translator that converts a source program into an object module. It consists of two stages, a preprocessor and a processor.

complex data: arithmetic data consisting of a real part and an imaginary part.

compound statement: a statement that contains other statements. IF and ON are the only compound statements.

concatenation: the operation that connects two strings in the order indicated, thus forming one string whose length is equal to the sum of the lengths of the two strings. It is specified by the operator ||.

condition name: a language keyword that represents an exceptional condition that might arise during execution of a program.

condition prefix: a parenthesized list of one or more condition names prefixed to a statement by a colon. It determines whether or not the program is to be interrupted if one of the specified conditions occurs within the scope of the prefix. Condition names within the list are separated by commas.

constant: an arithmetic or string data item that does not have a name; a statement label.

contained in: all of the text of a block except any entry names of that block. (A label of a BEGIN statement is not contained in the begin block defined by that statement.)

contextual declaration: the association of attributes with an identifier according to the context in which the identifier appears.

controlled storage: storage whose allocation and release is controlled by the programmer, with immediate access to the latest allocation only.

conversion: the transformation of a value from one representation to another.

cross section of an array: every element represented by the extent of at least one dimension of an array. An asterisk in the place of a subscript in an array reference

indicates the entire extent of that dimension.

data: representation of information or of value.

data character set: all of those characters whose bit configuration is recognized by the computer in use.

data-directed transmission: the type of STREAM input and output in which self-identifying data of the type, variable-name = value, is transmitted.

data item: a single unit of data; it is synonymous with "element."

data list: a list of expressions used in a STREAM input/output specification that represent storage areas to which data items are to be assigned during input or from which data items are to be obtained on output. (On input, the list may contain only variables.)

data set: a collection of data external to the program.

data specification: the portion of a stream-oriented data transmission statement that specifies the mode of transmission (DATA, LIST, or EDIT) and includes the data list and, for edit-directed transmission, the format list.

deactivated: the state in which a preprocessor variable is said to be when its value cannot replace the corresponding identifier in source program text.

decimal: the number system based on the value 10.

declaration: the association of attributes with an identifier explicitly, contextually, or implicitly.

default: the alternative assumed when an identifier has not been declared to have one of two or more alternative attributes.

delimiter: any valid special character or combination of special characters used to separate identifiers and constants, or statements from one another.

dimensionality: the number of bounds specifications associated with an array.

disabled: the state in which the occurrence of a particular condition will not result in a program interrupt.

DO-group: a sequence of statements headed by a DO statement and closed by its corresponding END statement.

dummy argument: a compiler-assigned variable for an argument that has no programmer-assigned name or whose attributes do not agree with those declared with the ENTRY attribute for the corresponding parameter.

edit-directed transmission: that type of STREAM transmission for which both a data list and a format list are specified.

element: a single data item as opposed to a collection of data items, such as a structure or an array. (Sometimes called a "scalar item.")

element variable: a variable that can represent only a single value at any one point in time.

enabled: that state in which the occurrence of a particular condition will result in a program interrupt.

entry name: a label of a PROCEDURE or ENTRY statement.

entry point: a point in a procedure at which it may be invoked by reference to the entry name. (See primary entry point and secondary entry point.)

epilogue: those processes which occur at the termination of a block.

event: an identifiable point in the execution of a program.

event name: the identifier used to refer to an event variable.

event variable: a variable associated with an event; its value shows whether an event is complete and the status of the completion.

exceptional condition: an occurrence, which can cause a program interrupt, of an unexpected situation, such as an overflow error, or an occurrence of an expected situation, such as an end of file, that occurs at an unpredictable time.

explicit declaration: the assignment of attributes to an identifier by means of the DECLARE statement, the appearance of the identifier as a label, or the appearance of the identifier in a parameter list.

exponent (of floating-point constant): a decimal integer constant specifying the power to which the base of the floating-point number is to be raised.

expression: the representation of a value; examples are variables and constants appearing alone or in combination with operators, and function references. The term "expression" refers to an element

expression, an array expression, or a structure expression.

external declaration: an explicit or contextual declaration of the EXTERNAL attribute for an identifier. Such an identifier is known in all other blocks for which such a declaration exists.

external name: an identifier which has the EXTERNAL attribute.

external procedure: a procedure that is not contained in any other procedure.

field (in the data stream): that portion of the data stream whose width, in number of characters, is defined by a single data or spacing format item.

field (of a picture specification): a character-string picture specification or that portion (or all) of a numeric character picture specification that describes a fixed-point number. If more than one field appears in a single specification, they are divided by the F scaling-factor character for fixed-point data, the K or E exponent character for floating-point data, or the M field-separator for sterling data.

file: a symbolic representation, within a program, of a data set.

file name: a symbolic name used within a program to refer to a data set.

format item: a specification used in edit-directed transmission to describe the representation of a data item in the stream or to control the format of a printed page.

format list: a list of format items required for an edit-directed data specification.

function: a procedure that is invoked by the appearance of one of its entry names in a function reference.

function reference: the appearance of an entry name in an expression, usually in conjunction with an argument list.

generation (of a block): a particular activation of a block.

generation (of data): a particular allocation of controlled or automatic storage.

generic key: a character string that identifies a class of keys: all keys that begin with the string are members of that class. For example, the recorded keys 'ABCD', 'ABCE', and 'ABDF' are all members of the classes identified by the generic keys 'A' and 'AB', and the first two are also members of the class 'ABC'; and the three recorded keys can be considered to be unique members of the classes 'ABCD', 'ABCE', and 'ABDF', respectively.

generic name: the name of a family of entry names. A reference to the name is replaced by the entry name whose entry attribute matches the attributes of the argument list.

group: a DO-group.

identifier: a string of alphameric and break characters, not contained in a comment or constant, preceded and followed by a delimiter and whose initial character is alphabetic.

imaginary number: a number whose factors include the square root of -1.

implicit declaration: association of attributes with an identifier used as a variable without having been explicitly or contextually declared; default attributes apply, depending upon the initial letter of the identifier.

inactive block: a procedure or begin block that has not been activated or that has been terminated.

inactive event variable: an event variable that is not currently associated with an event.

infix operator: an operator that defines an operation between two operands.

initial procedure: an external procedure whose PROCEDURE statement has the OPTIONS (MAIN) attribute. Every PL/I program must have an initial procedure. It is invoked automatically as the first step in the execution of a program.

input/output: the transfer of data between an external medium and internal storage.

interleaving of subscripts: a subscript notation used with subscripted qualified names that allows one or more of the required subscripts to immediately follow any of the component names.

internal block: a block that is contained within another block.

internal name: an identifier that has the INTERNAL attribute.

internal procedure: a procedure that is contained in another block.

internal to: all of the text contained in a block except that text contained in another block. Thus the text of an internal block (except for its entry names) is

not internal to the containing block. Note: An entry name of a block is not contained in that block.

interrupt: the suspension of normal program activities as the result of the occurrence of an enabled condition.

invoke: to activate a procedure at one of its entry points; to enter an on-unit.

invoked procedure: a procedure that has been activated at one of its entry points.

invoking block: a block containing a statement that activates another block.

iteration factor: an expression that specifies:

1. the number of consecutive elements of an array that are to be initialized with a given constant.
2. the number of times a given format item or list of format items is to be used in succession in a format list.

key: see source key and recorded key.

key class: a set of keys that begin with a common character string; this character string is the generic key for the class.

keyword: an identifier that is part of the language and which, when used in the proper context, has a specific meaning to the compiler.

known: a term that is used to indicate the scope of an identifier. For example, an identifier is always known in the block in which it has been declared.

label constant: synonymous with statement label.

label prefix: an unparenthesized identifier prefixed to a statement by a colon.

leading zeros: zeros that have no significance in the value of an arithmetic number; all zeros to the left of the first significant digit (1 through 9) of a number.

level number: an unsigned decimal integer constant specifying the hierachy of a name in a structure. It appears to the left of the name and is separated from it by a blank.

level-one variable: a major structure name; any unsubscripted data variable not contained within a structure.

list-directed transmission: the type of STREAM transmission in which data in the stream appears as constants separated by blanks or commas.

list processing: the use of based variables and locator variables to build chains or lists of data.

locator variable: a variable whose value identifies an allocation of a based variable in storage. Pointer variables and offset variables are the two types of locator variables.

major structure: a structure whose name is declared with level number 1.

major task: the task that has control at the outset of execution of a program.

minor structure: a structure whose name is declared with a level number greater than 1.

mode: real or complex designation for an arithmetic value.

multiple declaration: two or more declarations of the same identifier internal to the same block without different qualifications, or two or more EXTERNAL declarations of the same identifier as different names within a single program.

multiprogramming: the use of a computing system to execute a number of instructions concurrently.

multitasking: the PL/I facility that allows the programmer to make use of the multiprogramming capability of a system.

name: an identifier that has been declared.

nesting:

1. the occurrence of a block within another block.
2. the occurrence of a group within another group.
3. the occurrence of an IF statement in a THEN clause or an ELSE clause.
4. the occurrence of a function reference as an argument of another function reference.

null locator value: a special locator value that cannot identify any location in storage; it gives a positive indication that a locator variable does not currently identify any allocation of a based variable.

null string: a string data item of zero length.

numeric character data: arithmetic data described by a picture that is stored in character form. It has both an arithmetic value and a character-string value. The picture must not contain either an A or an X picture specification character.

offset variable: a locator variable whose value identifies a location in storage, relative to the start of an area.

on-unit: the action to be executed upon the occurrence of the ON-condition named in the containing ON statement.

operator: a symbol specifying an operation to be performed. See arithmetic operators, bit-string operators, comparison operators, and concatenation.

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

packed decimal: the System/360 internal representation of a fixed-point decimal data item.

parameter: a name in an invoked procedure that is used to represent an argument passed to that procedure.

parameter-attribute list: a description in an ENTRY attribute specification that lists attributes of parameters of the named entry point. This enables dummy arguments to be created correctly.

picture: a character-by-character specification describing the composition and attributes of numeric character and character-string data. It allows editing.

point of invocation: the point in the invoking block at which the procedure reference to the invoked procedure appears.

pointer variable: a locator variable whose value identifies an absolute location in storage.

precision: the value range of an arithmetic variable expressed as the total number of digits allowed and, for fixed-point variables, the assumed location of the decimal (or binary) point.

prefix: a label or a parenthesized list of condition names connected by a colon to the beginning of a statement.

prefix operator: an operator that precedes, and is associated with, a single operand. The prefix operators are ¬ + -

preprocessed text: the output from the first stage of compile-time activity. This output is a sequence of characters that is altered source program text and which serves as input to the processor stage in which the actual compilation is performed.

preprocessor: the first of the two compiler stages. At this stage the source program is examined for preprocessor statements which are then executed, resulting in the alteration of the source program text.

preprocessor statements: special statements appearing in the source program that specify how the source program text is to be altered; they are identified by a leading percent sign and are executed as they are encountered by the preprocessor (they appear without the percent sign in preprocessor procedures).

primary entry point: the entry point named in the PROCEDURE statement.

priority: the value that determines whether a task will take precedence over another task.

problem data: string or arithmetic data that is processed by a PL/I program.

procedure: a block of statements, headed by a PROCEDURE statement and ended by an END statement, that defines a program region and delimits the scope of names and that is activated by a reference to its name. It controls allocation and freeing of automatic storage declared in that block.

procedure reference: a function or subroutine reference.

processor: the second of the two compiler stages. The stage at which the preprocessed text is compiled into an object module.

program: a set of one or more external procedures, one of which must have the OPTIONS(MAIN) attribute in its PROCEDURE statement.

program control data: data used in a PL/I program to affect the execution of the program. Program control data consists of the following types: label, event, task, pointer, offset, and area.

prologue: those processes that occur at the activation of a block.

pseudo-variable: one of the built-in function names that can be used as a receiving field.

pushed-down stack: a stack of allocations to which new allocations are added and removed from the top on a last-in, first-out basis.

qualified name: a sequence of names of structure members connected by periods, to uniquely identify a component of a structure. Any of the names may be subscripted.

receiving field: any field to which a value may be assigned.

record: the unit of transmission in a RECORD input or output operation; in the internal form of a level-one variable.

recorded key: a character string recorded in a direct-access volume to identify the data record that immediately follows.

recursion: the reactivation of a procedure while it is already active.

repetition factor: a parenthesized unsigned decimal integer constant preceding a string configuration as a shorthand representation of a string constant. The repetition factor specifies the number of occurrences that make up the actual constant. In picture specifications, the repetition factor specifies repetition of a single picture character.

repetitive specification: an element of a data list that specifies controlled iteration to transmit a list of data items, generally used in conjunction with arrays.

returned value: the value returned by a function procedure to the point of invocation.

scale: fixed- or floating-point representation of an arithmetic value.

scope (of a condition prefix): the range of a program throughout which a condition prefix applies.

scope (of a name): the range of a program throughout which a name has a particular interpretation.

secondary entry point: an entry point defined by a label of an ENTRY statement within a procedure.

source key: a character string referred to in a RECORD transmission statement that identifies a particular record within a direct-access data set. The source key may or may not also contain, as its first part, a substring to be compared with, or written as, a recorded key to positively identify the record. Note: The source key can be identical to the recorded key.

source program: the program that serves as input to the compiler. The source program may contain preprocessor statements.

standard file: a file assumed by the compiler in the absence of a FILE or STRING option in a GET or PUT statement (the standard files are: SYSIN for input, SYSPRINT for output).

statement: a basic element of a PL/I program that is used to delimit a portion of the program, to describe data used in the program, or to specify action to be taken.

statement label: an identifying name prefixed to any statement other than a PROCEDURE or ENTRY statement.

statement label variable: a variable declared with the LABEL attribute and thus able to assume as its value a statement label.

static storage: storage that is allocated before execution of the program begins and that remains allocated for the duration of the program.

stream: data being transferred from or to an external medium represented as a continuous string of data items in character form.

string: a connected sequence of characters or bits that is treated as a single data item.

structure: a hierarchical set of names that refers to an aggregate of data items that may or may not have different attributes.

subfield: the integer description portion or the fraction description portion of a picture specification field that describes a noninteger fixed-point data item. The subfields are divided by the picture character V.

subroutine: a procedure that is invoked by a CALL statement or a CALL option. A subroutine cannot return a value to the invoking block, but it can alter the value of variables that are known within the invoking block.

subscript: an element expression specifying a location within a dimension of an array. A subscript can also be an asterisk, in which case it specifies the entire extent of the dimension.

subtask: a task that is attached by another task; any task attached by this subtask is a subtask of both tasks.

synchronous: describes serial execution of a program, using a single flow of control.

task: the execution of one or more procedures.

task name: the identifier used to refer to a task variable.

task variable: a variable whose value gives the relative priority of a task.

termination of block: cessation of execution of a block, and the return of control to the activating block by means of a RETURN or END statement, or the transfer of control to the activating block or some other active block by means of a GO TO statement. A return of control to the operating system via a RETURN or END statement in the initial procedure or a STOP or EXIT statement in any block results in the termination of the program. See epiloque.

termination of task: conclusion of the flow of control for a task.

variable: a name that represents data. Its attributes remain constant, but it can represent different values at different times. Variables fall into three categories: element, array, and structure variables. Variables may be subscripted and/or qualified.

IBM Technical Newsletter

File Number	S360-29
Re: Form No.	C28-8201-1
This Newsletter No.	N33-6008
Date	May 1, 1968
Previous Newsletter Nos.	None

IBM SYSTEM/360
PL/I REFERENCE MANUAL

This Technical Newsletter provides replacement pages for IBM System/360, PL/I Reference Manual, Form C28-8201-1. Pages to be inserted and removed are listed below.

Pages to be Inserted	Pages to be Removed
41,42,42.1	41,42
97-100	97-100
103-105,105.1,106	103-106
131-134	131-134
163,164	163,164
169,170	169,170
201-204,204.1	201-204
233,234	233,234
237,237.1,238	237,238
251,252	251,252
269-274,274.1	269-274
277-284,284.1	277-284
289,290	289,290
295,296	295,296

A change to the text is indicated by a vertical line to the left of the change.

The specifications contained in this Technical Newsletter correspond to Release 16 of IBM System/360 Operating System. Significant changes or additions will be reported in subsequent revisions or technical newsletters.

Summary of Amendments

Information has been added about the STRING built-in function; the UNALIGNED attribute; and the INDEXAREA, NOWRITE, and REWIND options of the ENVIRONMENT attribute.

Note: Please file this cover letter at the back of the manual to provide a record of changes.

20 Aug 1969

IBM United Kingdom Laboratories Ltd., Programming Publications, Hursley Park, Winchester, Hampshire, England.

PRINTED IN U.S.A.

IBM Technical Newsletter

File Number S360-29

Re: Form No. C28-8201-1

This Newsletter No. N33-6009

Date November 21, 1968

Previous Newsletter Nos. N33-6008

IBM SYSTEM/360
PL/I REFERENCE MANUAL

This Technical Newsletter, a part of Release 17, of IBM System/360 Operating System, provides replacement pages for the PL/I Reference Manual, Form C28-8201-1. These replacement pages remain in effect for subsequent releases unless specifically altered. Pages to be inserted and/or removed are listed below.

Cover,preface
15,16
29,30
33,34
34.1(added)
43,44
51,52
52.1(added)
65,66
69,70
77,78
81,82
85,86
97-102
102.1(added)
105
105.1(removed and not replaced)
106
106.1,106.2(added)
115,116
139,140
143-148
148.1(added)
155,156
156.1(added)
165-168
168.1(added)
171,172
175,176
181-184
193,194
201-206
206.1(added)
217-222
225-228
235-238
238.1(added)
241,242

IBM United Kingdom Laboratories Ltd., Programming Publications, Hursley Park, Winchester, Hampshire, England.

PRINTED IN U.S.A.

242.1(added)
247,248
253,254
257-264
264.1(added)
267,268
271-274.1
277-290
290.1,290.2(added)
305,306
309,310
315,316
323,324
333-338

A change to the text or a small change to an illustration is indicated by a vertical line to the left of the change; a changed or added illustration is denoted by the symbol • to the left of the caption.

Summary of Amendments

Information on the following items has been added:

Initialization of arrays of structures

Distinction between recursive and reentrant procedures

Variable-length records and their formats (V, VS, and VBS)

Key classification (GENKEY)

Standard system action for the ERROR condition

Numeric character data and the P format item

Precision of floating-point items

Condition code 1018

Maximum length of DISPLAY character strings

Termination of task

The descriptions of the following items have been clarified or expanded:

Concatenation of VARYING strings

Rounding of E and F format items

Parameter attribute lists without the ENTRY attribute

Varying-length string arguments and parameters

REFER option

ROUND built-in function

SIZE condition

ENDPAGE standard system action

There are, in addition to these amendments, some editorial changes to make this publication consistent with the reclassification to Restricted Distribution of the IBM System/360 PL/I Language Specifications, formerly Form C28-6571-4.

Note: Please file this cover letter at the back of the manual to provide a record of changes.

✓ 2 1 Sept 69

IBM Technical Newsletter

File Number	S360-29
Re: Form No.	C28-8201-1
This Newsletter No.	N33-6011
Date	January 31, 1969
Previous Newsletter Nos.	N33-6008 N33-6009

IBM SYSTEM/360
PL/I REFERENCE MANUAL

This Technical Newsletter, an amendment to the publication issued with Release 17 of IBM System/360 Operating System, provides replacement pages for the PL/I Reference Manual, C28-8201-1. These replacement pages remain in effect for subsequent releases unless specifically altered. Pages to be inserted and removed are listed below.

279,280
305,306
315,316

Changes in the text are indicated by a vertical line to the left of the change.

Summary of Amendments

Brackets and braces, omitted on the original pages, have been inserted.

✓ 24 Sept 1969

IBM United Kingdom Laboratories Ltd., Programming Publications, Hursley Park, Winchester, Hampshire, England.

PRINTED IN U.S.A.

READER'S COMMENT FORM

IBM System/360
PL/I Reference Manual

Form No. C28-8201-1

How did you use this publication?

As a reference source ☐

As a class-room text ☐

As a self-study text ☐

Based on your own experience, rate this publication:

As a reference source—Very Good☐ Good☐ Fair☐ Poor☐ Very Poor☐

As a text—Very Good☐ Good☐ Fair☐ Poor☐ Very Poor☐

What is your occupation?

We would appreciate your specific comments; please give page and line references where appropriate. If you wish a reply, be sure to include your name and address.

● Thank you for your cooperation. No postage necessary if mailed in U.S.A.

YOUR COMMENTS PLEASE . . . .

This SRL bulletin is one of a series which serves as reference sources for systems analysts, programmers and operators of IBM systems. Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

fold fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY...

IBM CORPORATION

112 EAST POST ROAD,
WHITE PLAINS, N.Y. 10601.

Attention: Department 813

fold fold

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

IBM System/360 Printed in U.S.A. C28-8201-1

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]