File Number S360(Mod.20)-29
Form C33-6007-1

DPS

Systems Reference Library

# IBM System/360 Model 20 Disk Programming System PL/I

This publication provides the information required for writing and running Model 20 PL/I programs that are to be compiled and link-edited using the Model 20 PL/I compiler under control of the IBM System/360 Model 20 Disk Programming System.

Part I, "Model 20 PL/I Language Features", and Part II, "Model 20 PL/I Syntax Rules", are composed of discussions and examples that explain the different features of the language and their interrelationships, their syntax notation and rules. Part III, "Model 20 PL/I as Part of the Disk Programming System", introduces the main components of the Disk Programming System, explains job control, and discusses compilation, link-editing, and execution of a Model 20 PL/I program.

More detailed information about the Disk Programming System can be found in the publications IBM System/360 Model 20, Disk Programming System, Control and Service Programs, Form C24-9006, and IBM System/360 Model 20, Guide to the Disk Programming System, Form C33-6000.

First Edition (December, 1968)

Changes are made from time to time to the specifications herein; before using this publication in connection with the operation of IBM equipment, refer to the latest IBM System/360 Model 20 SRL Newsletter, Form N20-0361 for the editions that are applicable and current.

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.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Laboratories, Programming Publications, 703 Boeblingen/Germany, P.O. Box 210.

# Contents

**Part I. Model 20 PL/I Language Features**

## Part II. Model 20 PL/I Syntax Rules

## Part III. Model 20 PL/I as Part of the Disk Programming System

# Introduction

## Reasons for Conceiving PL/I

Throughout the relatively brief history of electronic data processing, computers have been used mainly in two fields of activity - the commercial and the scientific.

Consequently, programmers generally have specialized in one field or the other. High-level languages like COBOL for commercial programming and FORTRAN for scientific programming have emphasized this divergence.

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

Now, however, the situation is changing. Business and industry have discovered new uses for the computer. The commercial programmer finds himself concerned with more complex computations in statistical forecasting and in programming for operations research. In science and engineering, the programmer needs a language to simplify the preparation of reports, to sort and edit technical data.

Today's computing systems have been designed to cope with all of these computing problems. They handle commercial and scientific/engineering programs with equal ease, with new power and new speed.

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

This is why PL/I was conceived. PL/I is a programming language designed to cover as wide a range of programming applications as possible. It can be used to solve both commercial and scientific/engineering problems. PL/I has been designed so that any programmer, no matter how brief or extensive his knowledge, can use it easily at his own level. It is simple for the beginning programmer, it is powerful for the experienced one.

A programmer need not know everything about PL/I to be able to use it. An experienced programmer can use PL/I to specify almost every detail of every step of a highly complicated program. A beginner can take advantage of the many automatic features of the language to do much of his work for him.

PL/I has also been designed to reduce the cost of programming, including the cost of training programmers who need to be trained in one programming language only. Another factor that contributes to programming cost is the machine dependency of the traditional programming languages, which means that frequently a program must be rewritten, sometimes because the system under which it is used has changed, sometimes because it is to be run on a new machine. Often, rewriting a program costs as much as writing it in the first place.

## Basic Characteristics of PL/I

A PL/I _program_ is written in form of _statements_ like for example C = A + B. The statements, whose sequence follows the logical flow of the program, are grouped together into blocks called _procedures_. A procedure defines a section of the program or a complete program. The task of a procedure is the execution of a particular _job_ or part of a job. The same procedure can be used in a number of different programs. Consequently, a change made in one procedure effectively makes a change in all programs that use it.

PL/I provides many options in statements, in descriptions of data or files, giving a lot of flexibility in writing programs. Wherever there are alternatives, the compiler makes an assumption if no choice is stated by the programmer. In each case, the assumption, called default, is the alternative that would be required in the majority of situations. The default concept is an important part of the simplicity of PL/I. In many cases, the beginning programmer need not even know that alternatives exist. PL/I is much less machine dependent than most commonly used programming languages, for example the Assembler Language.

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.

Model 20 PL/I is a subset of the full language. It is upward compatible with System/360 DOS/TOS PL/I provided the same input/output devices are available. The user can write both scientific/engineering and commercial programs in Model 20 PL/I.

## How to Use the Publication

This publication is designed as a reference book for the Model 20 PL/I programmer. Its three-part format allows a presentation of the material in such a way that references can be found quickly.

Part I, which may be read sequentially, describes the different features of the language and their interrelationship. Part II, which is organized purely from the reference point of view, brings rules and syntactic descriptions. Part III discusses the basic features of the Model 20 PL/I compiler, describes program compilation and execution, and brings all information needed to execute a Model 20 PL/I program.

This publication reflects features of the Model 20 PL/I compiler. Consequently, a number of features of the full PL/I language are not described in this publication, because they are not part of Model 20 PL/I.

Language features that are limited against the full PL/I language are described in the light of the limitations. Wherever a description here differs from the full language, it is not to be regarded as a respecification of the language, but merely a description of Model 20 PL/I. The publication is designed to provide all the implementation information needed to write programs in Model 20 PL/I and to run them under the Model 20 PL/I compiler.

Implementation features identified by the phrase "for IBM System/360 implementations ..." apply to all implementations for IBM System/360 computers. Features identified by the phrase "for the Model 20 PL/I Compiler ..." apply specifically to the IBM Model 20 PL/I Compiler under the System/360 Model 20 Disk Programming System.

# Syntax Notation

Throughout this publication, wherever a PL/I statement -- or some other combination of elements -- appears in the text, this statement or phrase is written using a uniform system of notation.

This notation is not part of Model 20 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 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 Model 20 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 all in lower-case letters and one portion all in upper-case letters, and the two portions must be separated by a hyphen.

   All such variables used are defined in the manual either syntactically, that is, this notation, or by giving a verbal definition.

   Examples:

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

   b) filename: This denotes the occurrence of the notation variable named filename.

   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, followed by the literal occurrence of the the word FIXED and 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, or

   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 $\left\{\begin{array}{l}\text{FIXED}\\ \text{FLOAT}\end{array}\right\}$

   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 publication to display alternatives.

6. Square brackets [ ] denote options. Anything enclosed in brackets may appear or may not appear at all in the syntactic unit. Brackets can serve the additional purpose of delimiting a syntactic unit.

   Example:

   ```
   FILE(filename)[KEY(expression)]
   ```

   This denotes the literal occurrence of the word FILE followed by the notation variable filename enclosed in parentheses and optionally followed by the literal occurrence of the word KEY with its notation variable expression enclosed in parentheses. If, in rule 4, the two alternatives also were optional, the vertical stacking would be within square brackets instead of 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.

# Part I

# Model 20 PL/I Language Features

# Basic Characteristics of Model 20 PL/I

The modularity of PL/I, that is, the ease with which combinations of language features can be used to meet different needs, is one of the most important characteristics of PL/I; in fact, it is the base on which PL/I has been built.

This chapter briefly discusses most of the basic features to give you an overall description of the language. Each feature is treated in more detail in subsequent sections.

## Program Structure

A Model 20 PL/I program is constructed of statements that are logically grouped together into one or more blocks called procedures. A program always comprises a main procedure and, usually, a number of other procedures that perform specific functions.

The rules defining the use of procedures, communication between procedures, the meaning of names, and the allocation of storage are fundamental for the understanding of PL/I.

## 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. In our context, data is generally defined as a representation of information in the form of digits and characters that have certain characteristics called attributes. PL/I deals with arithmetic data, character-string data, and program-control data, such as labels and pointers (addresses). It provides you with features to perform arithmetic operations, logical operations (e.g., comparison), and operations and functions for manipulating character strings. In order to be able to perform these operations, data items are usually given names.

The compiler must be able to determine, for every name used in a program, the complete set of attributes associated with that name. You may specify these attributes explicitly by means of a DECLARE statement, or the compiler may determine all or some of the attributes by context or by default if you do not specify them.

## Default Assumptions

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

## Storage Allocation

PL/I provides you with more flexibility in the allocation of main storage than most other programming languages. The storage areas for data in a PL/I program may be assigned statically, that is, when the program is loaded, or dynamically, that is, when the individual procedures are executed.

There are three different storage classes in Model 20 PL/I: STATIC, AUTOMATIC, and BASED. In general, the default storage class in Model 20 PL/I is AUTOMATIC. Storage for data with the storage class attribute STATIC is statically allocated, while for data with the attribute AUTOMATIC or BASED it is dynamically allocated.

## Expressions

Calculations in PL/I are specified by expressions. The meaning of an expression in PL/I is similar to that of an expression in elementary algebra. For example the expression.

```
A + B * C
```

specifies multiplication of the value of B by the value of C and addition of the value of A to the result. The data used in an expression must be of the same type; that is, there can be no mixing of data types in an expression. For example, a character-string cannot be added to an arithmetic value.

The results of the evaluation of expressions may be assigned to variables. Variables are names representing data. An example of an assignment statement is:

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

This means: Evaluate the expression on the right and assign it to X. The type of the result of the expression must be consistent with the type of X.

## Data Collections

Data variables can be grouped into either arrays or structures. An array is composed of elements of the same characteristics. A structure is a collection of variables, not necessarily alike in characteristics. Individual items of an array are referred to by the subscripted name of the array; individual items of a structure are referred to by names given to them.

Expressions cannot be specified for arrays or structures, but for elementary components of arrays or structures. Consider the following the assignment statement:

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

The names used in this assignment statement could be elements of structures or arrays, but not arrays or structures themselves.

## Input/Output

Input/output (I/O) is the transmission of data from an external storage medium to internal (main) storage and vice-versa. There are two classes of I/O in PL/I: stream-oriented and record-oriented I/O.

Stream-oriented I/O 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 and are converted automatically to conform, in main storage, to the attributes of the variables to which they are assigned. Similarly, on output, data items are converted one by one to external character form and are added to a conceptually countinuous stream of characters.

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 certain line size and page size. PL/I provides you with the facilities to detect the end of a page and to specify the beginning of a line or a page.

Record I/O 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 record is treated as a whole, and because no conversion is performed, this form of I/O is potentially more efficient than stream-oriented I/O, although the actual efficiency of each class will, of course, depend on the type of problem to be solved.

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.

## 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 has facilities for detecting a variety of exceptional conditions. It allows you to specify the conditions for which, should they arise, you want an interrupt to occur, and also the action to be taken when such interrupt does occur.

# Program Elements

In most programming languages, the length of an individual instruction or statement is limited by the size of a single punch card. If a statement exceeds the size of one card, a notation must be made, usually with a punch in some particular card column, to indicate that the statement is continued on the following card.

PL/I has no such artificial limitation. There is no fixed-length format for input although the Model 20 PL/I compiler reserves some card columns: the first column of every card in a program must be blank, and columns 73 through 80 of these cards are ignored and can contain any information, for example, card sequence numbers.

Within the available card area, you can write your program without considering special coding forms and without having to ensure 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 end of the previous statement, or any number of blanks may intervene.

## Character Sets

You may write your programs in one of two character sets; either a 60-character set or a 48-character set. 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, alphabetic characters, and special characters. There are ten digits: the decimal digits 0 through 9. 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 three alphabetic characters $, #, and @ can be used to cause the printing of letters that are not included in the standard English alphabet.

There are 21 special characters. They are as follows:

| Name | Character |
|---|---|
| Blank | |
| Equal 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 | \| (used only in Model 20 PL/I in combination with a second "or" symbol (\|\|) as concatenation operator) |
| "Greater than" symbol | > |
| "Less than" symbol | < |
| Break Character[1] | _ |
| Question mark* | ? |

*not used in Model 20 PL/I

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.

The rules for PL/I sometimes specify that an "alphameric" character must be used in certain coding. The term alphameric refers to any of the 29 alphabetic characters and the 10 digits, but not to the 21 special characters.

Note: The question mark, at present, has no specific use in the PL/I language, even though it is included in the 60-character set. The percent symbol and the "And" symbol have no meaning in Model 20 PL/I, although they do have a meaning in the full PL/I language used with higher System/360 models.

[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.

The restrictions for this character set are described in *Appendix C. Character Sets With EBCDIC and Card-Punch Codes*.

### 48-Character Set

The 48-character set is composed of 48 characters of the 60-character set. In all but five cases, the characters of the reduced set can be combined to represent the missing characters of the larger set. For example, the semicolon (;) is not included in the 48-character set, but a comma followed by a point (,.), without intervening blanks, can be used to represent it. The five characters that cannot be represented are the commercial "at" sign, the number sign, the break character, the question mark, and the percent symbol.

The restrictions and changes for this character set are described in *Appendix C. Character Sets with EBCDIC and Card-Punch Codes*.

## Using the Character Sets

All elements that make up a PL/I program are constructed exclusively from the PL/I character sets, with two exceptions: character-string constants and comments may contain any character in the EBCDIC character set.

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

There are three types of operators: arithmetic, comparison, and string operators.

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 *string* operator is:

|| denoting concatenation

Figure 1 shows some of the functions of other special characters.

### Identifiers

In a PL/I program, you give names or labels to data, statements, files, and procedures. In creating a name or label, you must observe the syntactic rules for creating the identifier.

An identifier is a single alphabetic character or a string of up to 31 alphameric and break characters, not contained in a comment or constant, and preceded and followed by a blank or some other delimiter (which may be an operator or a special character (except a decimal point)); the

| Name | Character | Use |
|---|---|---|
| comma | , | separates elements of a list |
| period | . | indicates decimal point |
| semicolon | ; | terminates statements |
| assignment symbol | = | indicates assignment of values[1] |
| colon | : | connects prefixes to statements |
| blank | | separates elements of a statement |
| apostrophe | ' | encloses string constants |
| parentheses | () | enclose lists; specify information associated with various keywords; in conjunction with operators and operands, delimit portions of an operational expression |

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

Figure 1. Some Functions of Special Characters

initial character of the string must be alphabetic.

Language keywords are also 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. A complete list of keywords and their use is contained in Appendix D. Model 20 PL/I Keywords.

Note: Only when using the 48-character set, some PL/I keywords are reserved words. Keywords 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. (Those keywords that are reserved are given in the section Recognition of Names.

An identifier must not exceed 31 characters in length. For the Model 20 PL/I compiler, some identifiers, as discussed in later sections, must not exceed six characters in length; this limitation applies to certain names, called external names, that may be referred to by other procedures.

The following are examples of identifiers you could use for names or labels:

```
A
FILE2
LOOP_3
RATE_OF_PAY
#32
```

The third and the fourth example illustrate the use of the break character to improve readability of an identifier, since blanks are not permitted in identifiers.

Examples of illegal identifiers are:

```
/*ABC
23AC
A*B
BEG IN
```

An identifier must not contain special characters; it must not start with a digit; and it must not contain embedded blanks.

## The Use of Blanks

You may use blanks freely throughout a PL/I program. You may or may not use them before and after 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, ¬=) must not contain blanks.

Other cases that require or permit blanks are noted in the text where the feature of the language is discussed. See Figure 2 for examples.

## Comments

Frequently you may want to insert comments into your programs to clarify the action that is taken at a given point. These comments enable someone unfamiliar with the program to follow your line of thought, and they are helpful to you when looking back over program sections that were written earlier.

Comments are permitted wherever blanks are allowed in a program. You may insert them between statements or in the middle of a statement without affecting the compilation of your program.

The character pair, /*, indicates the beginning of a comment. The same characters reversed, */, indicate its end. No blanks or other characters must separate these two characters; 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.

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

/*SO COULD +#$c%&-THIS!*/
```

| | | |
|---|---|---|
| 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 |

Figure 2. Examples of the Use of Blanks

In comments you may use any characters recognized by the system hardware. This includes characters that are not in the PL/I character set, such as the cent sign in the second example above.

Note: The length of a single comment must not exceed 5 cards. However, any number of comments may occur consecutively.

## Basic Program Structure

A PL/I program is made up of basic program elements called statements. There are two types of statements: simple and compound. These statements make up larger program elements called DO-groups and procedures.

### SIMPLE AND COMPOUND STATEMENTS

There are three types of simple statements in PL/I: keyword, assignment, and null statements, each of which is terminated by a semicolon.

A keyword statement has a keyword to indicate the function of the statement. For example:

GOTO LOOP; (GOTO is a keyword; a blank between GO and TO is optional.

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

A = B + C; (This is an assignment statement; it does not contain a keyword).

The null statement consists of a semicolon only and indicates that no operation is to be performed. The null statement may be used in connection with interrupts.

; (Null statement)

A compound statement is a statement that contains more than one simple statement. It is terminated by the semicolon of the last simple statement.

There are two compound statements: the IF statement and the ON statement. Examples of compound statements are:

1. IF A<B THEN A = B + C;

   This compound statement is terminated by the semicolon of the simple statement A = B + C;. The IF statement may be nested which means that it may contain other compound statements.

2. ON UNDERFLOW GOTO UNFIX;

   The ON compound statement is always composed of only two simple statements terminated by a semicolon.

### Statement Prefixes

Both simple and compound statements may have one or more prefixes. We have two types of prefixes in PL/I: the label prefix and the condition prefix.

A label prefix is used to identify a statement so that it can be referred to at some other point in a program. A label prefix is an identifier that precedes the statement and is connected to the statement by a colon. Most statements may have one or more labels. If you specify more than one label, you may use them interchangeably to refer to the statement. For example:

```
LABEL1:LABEL2:A=B;
```

PROCEDURE statements, however, must have one and only one label. The label prefix of a PROCEDURE statement is known as an entry name. The label prefix of any other statement is known as a statement label. DECLARE statements must not be preceded by any prefix.

A condition prefix is used to specify whether or not program 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 the execution of a program. In Model 20 PL/I only PROCEDURE statements may have condition prefixes. An example is OVERFLOW. The OVERFLOW condition arises when the exponent of a floating-point value exceeds the maximum allowed (representing a maximum value of about $10^{49}$).

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. There must not be any blanks 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. Only one condition prefix must be attached to the PROCEDURE statement, and the parenthesized list must be followed by a colon. A condition prefix precedes the entire

PROCEDURE statement, including the entry name for the PROCEDURE statement.

Example:

```
(CONVERSION, NOOVERFLOW): PROGM1:
     PROCEDURE;
```

The condition prefix indicates that an interrupt is to occur if the CONVERSION condition arises, but that no interrupt is to occur if the OVERFLOW condition arises. Note that the condition prefix precedes the entryname PROGM1.

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, you can easily remove the prefix.

Example:

```
(CONVERSION, NOOVERFLOW):
PROGM1:  PROCEDURE;
```

If no condition prefix precedes a procedure statement, or if not all possible conditions are explicitly stated, the compiler assumes default values.

Condition prefixes are discussed in the section entitled Exceptional Condition Handling.

## GROUPS AND PROCEDURES

A group, also called a DO-group, is a sequence of statements headed by a DO statement and terminated by a corresponding END statement, as follows:

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

DO-groups are used for control purposes and, in general, they can appear wherever single statements can appear. One DO-group may contain another DO-group; that is, a DO-group may be nested. DO-groups normally are used to specify an iterative process and/or in the IF compound statement.

A procedure is a sequence of statements headed by a PROCEDURE statement and terminated by an END statement, as follows:

```
label:  PROCEDURE;
          .
          .
          .
        END;
```

With Model 20 PL/I, a procedure may contain any statement except another PROCEDURE statement. Thus, unlike DO-groups, procedures cannot be nested.

The label preceding a procedure statement is called the entry name of the procedure, or the procedure name. A procedure is invoked (activated) by a procedure reference, that is, a special reference to the procedure name. The point at which the procedure reference appears in a program (for example: CALL PROGM1;), is called the point of invocation; the procedure containing the procedure reference is called the invoking procedure.

# Data Elements

Data is generally defined as a representation of information.

In PL/I, you refer to a data item, arithmetic or character-string, 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. The characteristics of a variable are not immediately apparent from 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 you may use to describe data are discussed briefly in this section. A complete discussion of each attribute appears in Part II of this manual, under Attributes.

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

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 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 a 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 the second statement, only the digit 2 is a constant.

In preparing a PL/I program, you 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 data you may use 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 the arithmetic and character-string data types. Program-control data is used to control the execution of the program. Statement labels and pointers are types of program-control data.

### PROBLEM DATA

The types of problem data available in Model 20 PL/I are arithmetic and character-string.

#### ARITHMETIC DATA

An arithmetic data item is one with a numeric value, that is, a number. It may be a decimal constant, like for example 215.8, or it may be the value of a variable, for example, 2.158 assigned to a variable. In Model 20 PL/I, all arithmetic data items must be written as decimal, either fixed-decimal or float-decimal data items.

Arithmetic data items have the characteristics of base, scale, and precision. For data items represented by an arithmetic variable, the characteristics have to be specified by attributes declared for the variable name, or they are assumed by default.

Base. The base of an arithmetic data item in Model 20 PL/I is decimal, that is ten.

Scale. The scale of an arithmetic data item is either fixed-point or floating-point. A decimal fixed-point data item is a decimal number in which the position of the decimal point is fixed. It 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 decimal number followed by an integer exponent that may or may not be signed. The exponent specifies the assumed position of the decimal point, relative to the position in which it actually appears.

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

Base and scale of arithmetic variables are specified by keywords; DECIMAL for base and FIXED and FLOAT for scale. Precision is specified by decimal integer constants enclosed in parentheses.

Whenever you assign a data item to a fixed-point variable, the precision you have declared for that variable is maintained. The assigned item is aligned on the assumed decimal point of the variable. Leading zeros are inserted if the assigned decimal item contains fewer integer digits than declared; trailing zeros are inserted if an assigned decimal item contains fewer fractional digits than declared. Truncation on the left or right may occur if the so aligned value has too many digits to the left or right of the assumed decimal point.

### Decimal Fixed-Point Data

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

Examples of fixed-point decimal constants as you may write them in a program are:

```
3.141593
-5280
455.3
.00003
234.
234
```

The keywords for decimal fixed-point variables are DECIMAL and FIXED. Precision is stated by two unsigned decimal integer constants, separated by a comma and enclosed in parentheses. The first specifies the total number of digits; the second, the scale factor, 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.

To define PI (which should assume the value 3.141593) in this way, we could use the following statement:

```
DCL PI FIXED DECIMAL (7,6);
```

This defines the identifier or variable PI as a fixed-point decimal item of not more than seven digits, six of which are to the right of the decimal point. In this declaration, of course, no value has yet been assigned to PI. This could be done later in the program with the following assignment statement:

```
PI = 3.141593;
```

The value could also be assigned in the DCL statement, specifying the INITIAL attribute, as follows:

```
DECLARE PI FIXED DECIMAL (7,6)
     INITIAL (3.141593);
```

This not only defines the identifier PI but gives it an initial value of 3.141593. The value may be retained throughout the program, as is probable in this case, or it may be changed during execution by an assignment statement.

The maximum number of decimal digits allowed in Model 20 PL/I is 15. Default precision, assumed when no specification is made, is 5,0. The internal form of decimal fixed-point data is packed decimal. Packed decimal is stored in 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 as an even number. Any such extra digit is in the high-order position (that is, to the left of the leftmost decimal digit), and it participates in any operations performed upon the data item, such as in a comparison operation. Note, however, that this extra digit is disregarded when evaluating the precision of arithmetic expressions.

The fixed-decimal values that can be represented in Model 20 PL/I are in the range of $10^{-50}$ to $10^{49}$, even though the declared scale factor must lie in the range of 0 to 15. Note, however, that not only the individual values, but also the values resulting from the evaluation of expressions must be within that range.

### Decimal Floating-Point Data

A decimal floating-point constant is written as one or more digits, with an optional sign and decimal point, referred to as the mantissa, followed by the letter E, followed by a decimal integer exponent that specifies a power of ten. The mantissa has the same format as a decimal fixed-point constant. Both, mantissa and exponent, may be preceded by a plus or minus sign.

Consider the following example in which the decimal number 312.5 10-17 would be written as:

```
+312.5E-17
```

The digits preceding the letter E are the mantissa preceded by an optional plus sign. The digits following the letter E are the exponent preceded by a minus sign.

Other examples of decimal floating-point constants as you may write them in a program are:

```
15E-23
15E23
4E-3
48333E44
438E0
5.E-12
314159E-6
.00314159E2
```

The last two examples represent the same value, namely 0.314159.

The keyword attributes that describe decimal floating-point variables are DECIMAL and FLOAT. Precision is stated by a decimal integer constant enclosed in parentheses. It specifies the number of digits to be maintained preceding the E. 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 DECIMAL or FLOAT.

Consider the following 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 five significant digits.

The maximum precision allowed for decimal floating-point data items in Model 20 PL/I is 15; the exponent must not exceed two digits. The value V that can be expressed is in the range of $10^{-51} \leq V < 10^{49}$, and V=0. The default precision is 6. The internal representation of decimal floating-point data may be in either short or long floating-point form. If the declared precision is less than or equal to 6, short floating-point form, otherwise long floating-point form is used.

The internal representation of decimal floating-point data is based on the representation of decimal numbers by mantissa and exponent. The first byte contains the exponent incremented by 50 as a two-digit decimal number without sign (but assumed to be positive). The following four, respectively 8 bytes (depending on whether short or long floating-point form is used) contain the mantissa without any leading zeros as a seven-digit or 15-digit decimal number, respectively, with the decimal point assumed to the left of the leftmost digit. Floating-point numbers in internal representation are normalized, i.e., the leftmost digit of the mantissa is not zero. The (normalized) value zero is represented in a special notation having zeros in all digit positions of characteristic and mantissa.

Note that all variables that have not been explicitely declared and whose names do not start with any of the letters I to N are assumed to be arithmetic decimal floating-point variables of six digits. (An identifier starting with any of the letters I to N must be explicitly declared).

### 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 attrtlyibute and a numeric picture specification. The format is:

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

The picture specification is a string of picture characters (e.g., 9 and V) used to represent a decimal fixed-point or floating-point value. The basic form of a numeric-picture specification is the picture character 9 specified one or more times and the optional picture character V, which is used to indicate the assumed location of a decimal point. The picture specification must be enclosed in apostrophes. An example of declaring a picture variable is:

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

This example specifies that any value assigned to PRICE is to be maintained as a string of five decimal digits in character form, with a decimal point assumed to precede the rightmost two digits.

In some cases it might be convenient to use repetition factors in numeric-picture specifications. A repetition factor is a decimal integer constant, enclosed in parentheses, that indicates how often the immediately following picture character is to be repeated. For example, the following picture specification would result in the same field as the example shown above:

```
   DECLARE PRICE PICTURE '(3)9V(2)9'
```

In Model 20 PL/I, numeric-character data is stored in zoned decimal format. If it is to be used in arithmetic computations it is automatically converted to coded arithmetic.

Although numeric-character data is in character form like a character string, and although it is aligned on the decimal point like packed decimal data, it is processed differently from the way either packed decimal items or character strings are processed. Editing characters (like the point and the dollar sign) can be specified for insertion into a numeric-character data item, and such characters are actually stored within the data item. Consequently, when the data item is assigned to 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. (Note that character-string data cannot be assigned to numeric-character variables). 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 assigned to character strings, which were then printed, they would look exactly the same. They do not, however, always function in the same may. For example, look at the following assignment statements:

```
   VALUE = PRICE;
   COST = PRICE;
   VALUE = COST;
   PRICE = COST;
```

After the first two assignment statements have been executed, the value of VALUE would be 001228 (with an assumed decimal point before the last two digits) 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; after the assignments, the arithmetic value of PRICE is contained in VALUE in packed decimal 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. The third and fourth assignment statements are invalid. The value of COST cannot be assigned to VALUE because a character string cannot be converted to packed decimal form. The value of COST cannot be assigned to PRICE because, in Model 20 PL/I, a character string cannot be converted to numeric-character format.

Other editing characters (including zero suppression characters) and insertion characters (like, for example, an asterisk), can be used in numeric-character specifications.

Note that the number of possible digit positions in the fixed part of a picture declaration must range between 1 and 15, inclusively. The total length of a picture, including editing characters, must not exceed 30 characters. The V character, however, does not count since it represents only an assumed, not an actual point. A picture or editing character preceded by an repetition factor (n) counts n times.

For complete discussions of picture characters, see Part II, the section Picture Specification Characters and the discussion of the PICTURE attribute in the section Attributes.

### CHARACTER-STRING DATA

You may think of a character string as a connected sequence of characters that is treated as a single data item. The length of the string is the number of characters it contains.

A character string can include any digit, letter, or special character that is contained in the EBCDIC-character set. Any blank included in a character string is considered an integral character of the data item and is included in the count of the length. Comments cannot be inserted in a character string. The comment, as well as the comment delimiters (/* and */), will be considered to be part of the character-string data.

When writing a program, you have to enclose character-string constants in apostrophes. If an apostrophe is a character in a string, it has to be written as two apostrophes with no intervening blank. The length of a character string is the number of characters between the enclosing apostrophes. If two apostrophes are used within the string to represent apostrophes, they are counted as a single character.

Consider the following examples of character-string constants:

```
'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 actual 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 repetition factor may range between 1 and 255.

The keyword attribute for declaring a character-string variable is CHARACTER which may be abbreviated as CHAR. The length of the character-string variable is declared by a decimal integer constant, enclosed in parentheses. The length specification must follow the keyword CHARACTER or CHAR. For example:

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

This DECLARE statement specifies that the identifier NAME is to represent a character-string data item that is 15 characters long. The values of this variable, that is, different character strings, are to be assigned during the execution of the program. Most data items, however, can also be given an initial value by declaring the name with the INITIAL attribute and listing the initial value. For example:

```
DECLARE NAME CHARACTER(15)
      INITIAL ('JOHN DOE');
```

Although the declared length is 15, the length of the string assigned by the INITIAL attribute contains only 8 characters. Blanks are added automatically to the right to fill out the length. The first character assigned is always left-adjusted, and, if necessary, blanks are added on the right. In this case, the string would be stored as the characters JOHN DOE, followed by 7 blanks.

A character string is assigned from left to right. If the actual string is longer than the declared length, the string is truncated on the right, that is, the rightmost characters are lost.

Note: If truncation occurs, there will be no interrupt. There is no ON-condition in Model 20 PL/I to deal with character-string truncation.

Character-string data in System/360 implementations is maintained internally in character format, that is, each character occupies one byte of main storage. The maximum length allowed by the Model 20 PL/I Compiler for variables declared with the CHARACTER attribute is 255. The maximum length allowed for a character-string constant after application of repetition factors is also 255. The minimum length in either case is one.

## PROGRAM-CONTROL DATA

The types of program-control data in Model 20 PL/I are label and pointer data.

### LABEL DATA

A statement label 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 separates the label from the statement, as follows:

```
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 shown in the following example:

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

ABCDE, as it is used above, can be classified as a statement-label constant. A statement-label variable is a variable to which statement-label constants can be assigned in the program. 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;
```

### POINTER DATA

A pointer variable is the name of a pointer which is used to point to a location in storage. A pointer is, in effect, the address of data in storage.

The keyword attribute for declaring pointer variables is POINTER. For information on the use of pointer variables, refer to the sections Data_Transmission, and Based_Variables_and_Pointer_Variables.

## Data Organization

In PL/I, we have single data elements or collections of data elements, called arrays or structures, depending on their composition. A variable that represents a single element is called an element_variable. A variable that represents a collection of data elements is either an array_variable or structure_variable.

### ARRAYS

An array is a named table of data elements all of which have identical attributes. Only the array itself is given a name. An individual element of an array is referred to by its relative position within the array. The relative position is specified by a subscript (enclosed in parentheses) following the array name, with or without intervening blanks.

Assume TABLE has been declared to be an array of 12 elements. TABLE(1) refers to the first data element in the array, TABLE(2) to the second, TABLE(3) to the third, etc. Each of the numbers, (1), (2), or (3), is a subscript that gives the relative position, within TABLE, of a particular data element. TABLE(1), TABLE(2), and TABLE(3) all refer to single elements and are element variables. The entire array is referred to by the unsubscripted name TABLE. TABLE is an array variable.

An array variable is declared in a DECLARE statement by giving its name, the number of elements in the array, and the attributes of the items. Consider the following example:

```
DECLARE TABLE (12) DECIMAL FIXED (2);
```

This specifies that TABLE refers to an array of 12 data elements, each of which will have a value that can be represented by two decimal digits. TABLE, as declared above, might look as follows:

| Element_Value | Reference |
|---|---|
| 31 | TABLE(1) |
| 43 | (2) |
| 42 | (3) |
| 57 | (4) |
| 64 | (5) |
| 73 | (6) |
| 79 | (7) |
| 79 | (8) |
| 69 | (9) |
| 58 | (10) |
| 49 | (11) |
| 40 | (12) |

Thus, TABLE(1) would refer to the data item 31, TABLE(6) to 73, TABLE(12) to 40. The expression TABLE(7) + TABLE(1) would yield a value of 110.

Assume that the values assigned to TABLE represent the average temperature of the months of a particular year. TABLE(1) is the January average, etc. As TABLE was declared in the previous DECLARE statement, the data items could be referred to singly or as a whole.For various reasons, you may want to consider the year as divided into quarters; it might be convenient to be able to use one reference to the average temperatures of a quarter of a year and another to specify months in a quarter. For this purpose, TABLE may be declared as follows:

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

In this statement, TABLE is declared to be a two-dimensional array of 12 data items; that is, TABLE is considered to consist of four lists of three items each. It has two dimensions, one with a bound of four, one with a bound of three. The data might be recorded in storage in exactly the same way as with the first declaration, but conceptually it is ordered differently.

The upper bound of the dimension is the end of it, 4 and 3 in this case; the lower bound or the beginning of a dimension is always assumed to be 1. The extent of the dimension is the number of integers between and including the specified end. Thus, the terms bound and extent, while conceptually different, have the same value in Model 20 PL/I.

Note the difference between a subscript and the dimension-attribute specification. The latter, which appears in the declaration of an array, specifies the dimensioning and the number of elements in an array. Subscripts are used in other references to identify specific elements within the array.

Following are two different ways in which the arrangement might be conceptually illustrated. The first shows TABLE as consisting of four consecutive lists of three items each; the second shows it as a matrix of four rows and three columns.

| Element Value | Reference |
|---|---|
| 31 | TABLE(1,1) |
| 43 | (1,2) |
| 42 | (1,3) |
| 57 | (2,1) |
| 64 | (2,2) |
| 73 | (2,3) |
| 79 | (3,1) |
| 79 | (3,2) |
| 69 | (3,3) |
| 58 | (4,1) |
| 49 | (4,2) |
| 40 | (4,3) |

| TABLE | (n,1) | (n,2) | (n,3) |
|---|---|---|---|
| (1,m) | 31 | 43 | 42 |
| (2,m) | 57 | 64 | 73 |
| (3,m) | 79 | 79 | 69 |
| (4,m) | 58 | 49 | 40 |

You may refer to an element of the above described TABLE by a subscripted name with two parenthesized subscripts, separated by a comma. For example, TABLE(2,1) would specify the first element in the second list or row, in this case, the data item 57.

The Model 20 PL/I Compiler allows a maximum of three dimensions to be declared for an array. The above described TABLE could, in fact, also be declared as a three-dimensional array with the following DECLARE statement:

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

Note that the number of specifications, separated by commas, is the same as the number of dimensions, and that the product of the numbers is equal to the number of items in the array: (12), (4,3), (2,3,2).

Using the same data, TABLE (2,3,2) might be illustrated as follows:

| Element Value | Reference |
|---|---|
| 31 | TABLE(1,1,1) |
| 43 | (1,1,2) |
| 42 | (1,2,1) |
| 57 | (1,2,2) |
| 64 | (1,3,1) |
| 73 | (1,3,2) |
| 79 | (2,1,1) |
| 79 | (2,1,2) |
| 69 | (2,2,1) |
| 58 | (2,2,2) |
| 49 | (2,3,1) |
| 40 | (2,3,2) |

| TABLE (1,n,m) | (1,n,1) | (1,n,2) |
|---|---|---|
| (1,1,m) | 31 | 43 |
| (1,2,m) | 42 | 57 |
| (1,3,m | 64 | 73 |

| TABLE (2,n,m) | (2,n,1) | (2,n,2) |
|---|---|---|
| (2,1,m) | 79 | 79 |
| (2,2,m) | 69 | 58 |
| (2,3,m) | 49 | 40 |

The dimension attribute (2,3,2) specifies that TABLE represents a list of 12 data items and that the list will be referred to as if it consists of two sub-lists, each of which is further divided into three sub-lists of two items each.

The examples of arrays shown in this section are arrays of arithmetic data. Character strings and statement labels may also be collected into arrays. Note, however, that pointers may not be collected into arrays.

### Expressions as Subscripts

The subscripts of a subscripted name need not be constants as shown in the above examples. Subscripts are frequently expressed as variables or expressions. We could, for example, use TABLE(I,J*K) to refer to the different elements of TABLE by varying the values of I, J, and K.

Note that, although a subscript can be an expression, each bound of a dimension-attribute declaration must be an unsigned decimal integer constant and that the value of a subscript must lie within the extent of the corresponding dimension. If the result of a subscript expression (such as J*K above) is not a fixed-decimal integer, it is converted to FIXED DECIMAL (5,0) in Model 20 PL/I. Note also that the number of subscripts in a reference must agree with the number of dimensions in the declaration.

## STRUCTURES

A structure is a collection of data elements that need not have identical characteristics, but that have a logical relationship to one another.

Like an array, the entire structure is given a name that can be used to refer to the entire collection of data. Unlike in an array, however, each element and groups of elements of a structure also have names. A structure is a hierarchical collection of names referring to elements. These elements, each of which may be a single data item or an array are at the bottom of the hierarchy. At the top of the hierarchy is the structure name, which represents the entire collection of elements.

Consider a program to calculate a weekly payroll. One employee, John J. Doe, whose man-number is 68584, works 40 hours of regular time and five hours of overtime. He is paid $4.00 per hour for regular time and $6.00 for overtime.

His weekly pay record, with all the above information, is read and assigned to a structure named PAYROLL. The information could be ordered:

```
DOE JOHN J 68584 40 05 400 600
```

If this data were referred to merely by the name PAYROLL, it might be treated as a character string; but, if the data were declared as a character string, it would be difficult to get to individual items within the string, and arithmetic operations would involve conversion. However, a name can also be given to each element. The names for John Doe's pay record and the data each name represents might, conceptually, look like this:

```
         / LASTNAME      DOE
        (  FIRSTNAME     JOHN
         \ MIDDLENAME    J
PAYROLL  ) MAN_NO        68584
         \ REGLHOURS     40
         ( OVTMHRS       05
         \ STRATE        400
          \OVRTMRATE     600
```

Thus, we could refer to the entire collection of data items by the name PAYROLL, or we can refer to an individual item by an individual name.

It is often convenient to subdivide the entire collection into smaller logical collections, to be able to refer collectively to more than one, but not all, of the variables in a structure. In a structure, such subdivisions are also given names. The above example might be subdivided as follows:

```
                      (LAST     DOE
         / NAME       <FIRST    JOHN
        (             (MIDDLE   J
        ) MAN_NO                68584
PAYROLL <
        ) HRS         (REGLR    40
        (             (OVTM    .05
         \ RATE       (STRATE   400
                      (OVRTM    600
```

The major structure, PAYROLL, contains the substructures, NAME, HRS, and RATE. MAN_NO is not a substructure but an elementary name because it represents only a single data item.

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

When a structure is declared, the level of each name is indicated by a level number. The major-structure name, at the first level, is always given the level number 1. Each name at a deeper level is given a greater number to indicate the level depth. The above structure could, for example, be declared as follows:

```
DECLARE 1 PAYROLL,
    2 NAME,
       3 LAST,
       3 FIRST,
       3 MIDDLE,
    2 MAN_NO,
    2 HRS,
       3 REGLR,
       3 OVTM,
    2 RATE,
       3 STRATE,
       3 OVRTM;
```

Note that the pattern of indention is used only for readability. The statement could be written in a continuous string as DECLARE 1 PAYROLL, 2 NAME, 3 LAST, etc.

The order of appearance of names in a DECLARE statement, along with their level numbers, determines the structuring. Except for the major-structure name, which must be declared with the level number 1, any number up to 255 may be used in Model 20 PL/I. Note, however, that only a maximum of 8 physical structure levels may be specified in structure declarations.

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

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

The level numbers of the above example might have been declared as follows:

```
DECLARE 1 PAYROLL,
    8 NAME,
       20 LAST,
       20 FIRST,
        9 MIDDLE,
    6 MAN_NO,
    2 HRS,
       3 REGLR,
       3 OVTM,
    2 RATE,
       255 STRATE,
       255 OVRTM;
```

Exactly the same structuring would result.

When a structure is declared, attributes may be specified for each of the elementary names. For example:

```
DECLARE 1 PAYROLL,
    2 NAME,
       3 LAST CHARACTER (12),
       3 FIRST CHARACTER (8),
       3 MIDDLE CHARACTER (1),
    2 MAN_NO CHARACTER (5),
    2 HRS,
       3 REGLR FIXED DECIMAL (2)
       3 OVTM FIXED DECIMAL (2),
    2 RATE,
       3 STRATE FIXED DECIMAL (3,2),
       3 OVRTM FIXED DECIMAL (3,2);
```

Note: Level numbers are specified with structure names only in the DECLARE statement. In references to the structure or its elements, no level numbers are used. Only structures can be declared with level numbers; a level number cannot be declared with any other identifier.

### Qualified Names

All names within a single procedure must be unique. But within structures, it is often convenient to be able to use the same identifier for related names. In the above structure, for example, it would be convenient to refer to the items in HRS and RATE as "regular hours" and "regular rate" and "overtime hours" and "overtime rate". In fact, the elements can be given the same names. The last portion of the structure might be declared:

```
2 HRS,
   3 REGLR,
   3 OVRTIM,
2 RATE,
   3 REGLR,
   3 OVRTIM;
```

The use of a qualified name in referring to the individual item avoids ambiguity. A qualified name is a substructure or element name that is made unique by qualifying it with one or more names of a higher level. The individual names within a qualified name are separated by a period. A qualified name must not contain embedded blanks or comments. The above items could be referred to by the following qualified names:

```
HRS.REGLR
RATE.REGLR
HRS.OVRTIM
RATE.OVRTIM
```

None of the names in PAYROLL, except PAYROLL itself, need be unique within the procedure in which it is declared. Each of them could be qualified. For example:

```
PAYROLL.NAME
PAYROLL.NAME.LAST
  or    NAME.LAST
  or    PAYROLL.LAST
```

Qualification need go only so far as necessary to make the name unique. Tntermediate qualifying names can be omitted. The name PAYROLL.LAST is a valid reference to the name PAYROLL.NAME.LAST.

Note: The length of qualified names must not exceed 2 cards.

## RESPECIFICATION OF DATA

The DEFINED attribute specifies that the name of a data element, a structure, or an array is to refer to the same storage area as the name given to other data. For example, in the declaration

```
DECLARE LIST (20,20),
    LIST_A (20,20) DEFINED LIST;
```

LIST is a 20 by 20 two-dimensional array. LIST_A is an identical array referring to the same storage area as LIST. The reference to the same storage area is achieved by using the DEFINED attribute. The effect is that a reference to an element in LIST_A is the same as a reference to the corresponding element in LIST, and vice versa. Thus, a change to an element in LIST_A will at the same time, be an identical change to the corresponding element of LIST. This use of the DEFINED attribute is called *simple defining*.

The DEFINED attribute can also be used for so-called *string-overlay defining*. This type of defining specifies that the *defined item* (the item having the DEFINED attribute; e.g., LIST_A above) is to refer to all or part of the storage area occupied by the *base identifier* (the identifier following the keyword DEFINED; e.g., LIST above). For example:

```
DECLARE 1 P, 2 Q CHARACTER (25),
             2 R CHARACTER (50),
        PSTRING1 CHAR (60) DEFINED P;
```

In this example, PSTRING1 is a character string of length 60 defined on the structure P. The first character of Q through the last character in R can be considered as one string of 75 characters in length. PSTRING1 refers to the first 60 characters of that string, that is, the 25 characters of Q effectively concatenated with (that is, connected to) the first 35 characters of R.

### Initialization of Data

The INITIAL attribute, which may be abbreviated as INIT is used to specify an initial value for a variable. The initial value is assigned to the variable at the time storage is allocated to it. For example:

```
DCL NAME CHARACTER (10) INITIAL
       ('JOHN DOE');
DCL PI FIXED DECIMAL (5,4) INIT
       (3.1416);
```

When storage is allocated to NAME, the character string 'JOHN DOE' (padded with blanks on the right up to ten characters) will be assigned to it. When storage is allocated to PI, it will be initialized to the value of 3.1416. The initial value of a variable may be retained throughout the program, or it may be changed during execution.

For a STATIC variable, that is, a variable for which storage remains allocated throughout the entire execution of a program, any value specified in an INITIAL attribute is assigned only once. For AUTOMATIC variables, that is, variables for which storage is allocated whenever the declaring procedure is activated, any INITIAL values will be assigned at each activation. INITIAL values cannot be declared for BASED variables, DEFINED variables, STATIC label variables, and POINTER variables. In a structure declaration, the INITIAL attribute can only be used in the declaration of elementary names.

The INITIAL attribute may be specified for element variables and arrays. Note, however, that is cannot be specified for arrays of the storage class AUTOMATIC.

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

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

In the first example, only the first three elements of A are initialized; the remainder of the array is not. 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 with the same value.

Note that the depth of nested iteration factors in an INITIAL attribute is restricted to 3.

The iteration factor should not be confused with the character-string *repetition factor*. Consider the following example:

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

This INITIAL attribute 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 character-string array, a single parenthesized factor preceding a character-string constant is assumed to be a string-repetition factor (as in (10)'A'). If more than one appears, the one immediately preceding the character string is the string-repetition factor, while all factors preceding this repetition factor are iteration factors.

# Expressions

An expression is a representation of a value. An expression may be a single constant or an element variable, a function reference, or a combination of them, including operators and other delimiters. An expression that contains operators is an operational expression. The constants and element variables of an operational expression are called operands.

In the examples below, assume that the variables have been declared as follows:

```
DECLARE A(10,10) DECIMAL FIXED (15),
   B(10,10) DECIMAL FIXED (15),
   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 DECIMAL FIXED (8),
   D DECIMAL FIXED (8);
```

Examples of 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
C
A(10,10)
```

All except the last two examples are operational expressions. The last two are element variables. Note that the expression A(10,10) is not the same as A(10,10) in the DECLARE statement. In the DECLARE statement, (10,10) is a dimension attribute specifying a two-dimensional array of 100 element variables. In the expression A(10,10), (10,10) is a subscript referring to the last element of the array A.

A single operational expression may contain a number of arithmetic operations, as shown in the following example:

```
A(4,4)+B(3,3)-C*(D/(B(2,2)-A(1,1)))**C
```

Parentheses within an expression indicate that the parenthesized portion is considered as a single value in relation to its surrounding operators. The parenthesized portion of an operational expression is evaluated first, with the innermost parenthesized portion taking precedence. In the above example, the expression (B(2,2)-A(1,1)) is evaluated first, before the value of D is divided by the result of the subtraction.

Although an operational expression may contain more than one data item, it represents a single value that may appear in a number of different PL/I statements. The most common occurrence of operational expressions is in the form of assignment statements. Such as:

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

In this example, all of the three operands are element variables. The assignment symbol (=) indicates that the value of the expression on the right (A + B) is to be assigned to the variable C on the left.

## Expression Operations

An operational expression can specify one or more single operations. The class of operation depends on the class of operator specified for the operation. There are three classes of operations: arithmetic, comparison, and concatenation.

### ARITHMETIC OPERATIONS

An arithmetic operation is represented by one or two operands in combination with one of the following operators:

```
+ - * / **
```

The plus and the minus sign can appear either as prefix operators (for example +A or -A) or as infix operators (that is, between operators, such as A + B or A - B). The other arithmetic operators can appear only as infix operators. All operands of an arithmetic operation must be arithmetic (for example, a character-string variable in combination with an arithmetic operator will lead to an error).

An expression may contain a number of 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.

A single variable may have more than one prefix operator. 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, thus restoring it to the original arithmetic value of A.

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

### CONVERSION OF OPERANDS IN ARITHMETIC OPERATIONS

The two operands of an arithmetic operation may differ in type, precision, and scale. The necessary conversions are performed automatically according to the rules listed below.

#### Type

Numeric-character operands (digits recorded in character form as in the PICTURE specification) are converted to coded arithmetic form. 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.

#### Precision

If only precisions differ, no conversion takes place.

#### 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 if 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 with the precision (p,0). In such a case, no conversion is necessary, but the result will be floating-point.

Consider the following example:

```
DECLARE A FLOAT,
        B FIXED DECIMAL (5,0),
        C FLOAT;
          .
          .
          .
        C = A ** B;
```

The result of A ** B will be in floating-point form and will be assigned to the variable C. The exponentiation is, however, executed with the FIXED value of B.

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). For example:

   ```
   DECLARE A FIXED DECIMAL,
           B FIXED DECIMAL (5,2),
           C FLOAT;
             .
             .
             .
           C = A ** B;
   ```

   The precision of the value of B has a scale factor of two. Since it is not an integer, both operands are converted to floating-point form.

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

3. The first operand is also converted to floating-point if precisions indicate that the result of the fixed-point exponentiation would exceed the maximum number of digits allowed (i.e., 15 decimal digits).

### FORMATS OF RESULTS OF ARITHMETIC OPERATIONS

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

After the required conversions have taken place, the arithmetic operation is performed. If the maximum precision has been exceeded, the result is truncated, that is, digits are lost regardless of the scale of the operands. In some cases involving fixed-point data, high-order digits may be lost when scale factors are such that decimal-point alignment does not allow for the declared number of digits. In floating-point operations low-order digits may be lost even when the maximum precision has not been reached. The scale and precision of the result depend upon the operands and the operator involved.

For prefix operations, the result has the same scale and precision as the converted operand (for example. (+5.0 has the same scale and precision as (-5.0)).

For infix operations, the result depends on the scale of the operands, as described in the following sections.

### Floating-Point Operands

If the converted operands of an infix operation are of floating-point scale, the result is of floating-point scale. The precision of the result is the greater of the precisions of the two operands. For example:

```
DECLARE A FLOAT (5),
        B FLOAT (7);
```

The precision of the value of (A+B) will be 7. Note that even though the maximum precision of 15 has not been reached, low-order digits may be lost due to different exponents in A, B, and A+B.

### Fixed-Point Operands

If the converted operands of an infix operation are of fixed-point scale, the result is of fixed-point scale. The precision of a fixed-point result varies according to the type of operation performed.

The symbols used in the formulas for computing the precision of the fixed-point results are as follows:

p represents the total number of digits of the result

q represents the scale factor of the result

p1 represents the total number of digits of the first operand

q1 represents the scale factor of the first operand

p2 represents the total number of digits of the second operand

q2 represents the scale factor of the second operand

Addition and Subtraction. The total number of digits in the result is equal to one plus the number of integer digits of the operand with the greater number of integer digits, plus the number of fractional digits of the operand with the greater number of fractional digits. The total number of positions cannot exceed the maximum number of digits allowed (15 decimal digits). The scale factor of the result is equal to the larger scale factor of the two operands.

Formulas:

p = 1 + maximum (p1 - q1, p2 - q2) + maximum (q1, q2)

q = maximum (q1, q2)

Consider the following example in which, for explanation purposes, the variables have been broken into parts:

$$\overbrace{\underset{A}{12354}.\underset{B}{\overset{<q1>}{2385}}}^{p1} + \overbrace{\underset{C}{222}.\underset{D}{\overset{<q2>}{11111}}}^{p2}$$

The total number of digits (p) in the result would be equal to 1 plus the number of digits in A (p1-q1) plus the number of digits in D (maximum (q1,q2)). The scale factor of the result would be equal to the number of digits in D. The 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 the first operand, plus the number of digits in the second operand. The total number of digits cannot exceed the maximum number of digits allowed for the implementation (that is, 15). The scale factor of the result is the sum of the scale factors of the two operands.

Formulas:

p = 1 + p1 + p2

q = q1 + q2

Consider the following example:

$$\underset{A}{345}.\underset{B}{432} * \underset{C}{22}.\underset{D}{45}$$

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

*Division*. The total number of digits in the quotient is always 15 (maximum allowed). The scale factor of the quotient depends on 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 (always 15) minus the sum of A and D.

Formulas:

$$p = 15$$
$$q = 15 - ((p1 - q1) + q2)$$

Consider the following example:

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

The total number of digits in the quotient would be 15 (the maximum 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). The 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*. Also note from the above formulas that the result of a fixed-point division can have a scale factor greater than zero even though the operands might have a scale factor of zero, and that the result of fixed-point division can have a negative scale factor even though negative scale factors cannot be explicitly declared in Model 20 PL/I.

Examples:

```
00432.432 / 2
432.432 / 2.0000
```

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

*Caution*: In the use of fixed-point division operations, take care 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 this or a subsequent operation to to be left-truncated by exceeding the maximum number (15) of digits allowed.

*Exponentiation*. If the second operand (the exponent) is a positive decimal integer constant, the total number of digits in the result is one less than the number of digits in the first operand plus 1 multiplied by the *value* of the second operand (the exponent). The scale factor of the result is equal to the scale factor of the first operand multiplied by the value of the second operand (the exponent).

*Note*: In the exponentiation operation x**y, some special cases are defined as follows:

1. If $x = 0$ and $y>0$, the result is 0.
2. If $x = 0$ and $y \leq 0$, the ERROR condition is raised.
3. If $x \neq 0$ and $y = 0$, the result is 1.
4. If $x < 0$ and y is not fixed-point with precision (p,0), the ERROR condition is raised.

Formulas:

$$p = (p1 + 1) * (\text{value-of-exponent}) - 1$$
$$q = q1 * (\text{value-of-exponent})$$

Consider the following example:

```
32 ** 5
```

The total number of digits in the result would be 14. We arrive at this number by multiplying one plus the number of digits in the first operand (1 + 2) by the value of the exponent (5) 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).

Figures 3 through 6 show the results of arithmetic operations.

## COMPARISON OPERATIONS

In PL/I, we specify a comparison operation by combining operands with one of the following operators:

```
 <  (less than)
¬<  (not les than)
<=  (less than or equal to)
 =  (equal to)
¬=  (not equal to)
>=  (greater than or equal to)
>   (greater than)
¬>  (not greater than)
```

There are three types of comparisons:

1. *Algebraic* comparison, that is, the comparison of signed arithmetic values in coded arithmetic form. If operands differ in scale and precision, they are converted according to the rules for arithmetic operations. Numeric-character data is converted to coded arithmetic format before comparison.

2. *Character comparison*, which is a left-to-right, character-by-character comparison of character data according to the System/360 collating sequence.

3. *Pointer comparison*, for which only the operators = and ¬= are allowed. Both operands must be valid pointer expressions, since there is no conversion of program-control data.

The operands of a comparison operation must be of the same type; that is, both must be arithmetic or both must be character strings. If operands of a character-string comparison are of different lengths, the shorter operand is extended on the right with blanks.

The result of the comparison operation always is a "truth" value in Model 20 PL/I; the value is 'true' if the relationship is true, and 'false' if the relationship is not true.

The only occurrence of comparison operations is in the IF statement, as shown in the following example:

```
IF A + C = B
    THEN action-if-true
    ELSE action-if-false
```

The evaluation of the expression A + C = B yields either 'true' or 'false'. Depending on the result, either the THEN portion or the ELSE portion of the IF statement is executed. Note that the comparison operations in IF statements can involve only element values; arrays or structures are not permitted.

Only comparison operations of "equal" and "not equal" are valid for comparisons of pointer-variable operands. Labels must not be compared.

| | | First Operand | |
|---|---|---|---|
| | | DECIMAL FIXED (p1,q1) | DECIMAL FLOAT (p1) |
| Second | DECIMAL FIXED (p2,q2) | DECIMAL FIXED (p,q)<br>p=1+MAX(p1-q1,p2-q2)<br>+MAX(q1,q2);<br>q=MAX(q1,q2) | DECIMAL FLOAT(p)<br>p=MAX(p1,p2) |
| Operand | DECIMAL FLOAT (p2) | DECIMAL FLOAT (p)<br>p=MAX(p1,p2) | DECIMAL FLOAT (p)<br>p=MAX(p1,p2) |

Figure 3. Attributes of Results of Addition and Subtraction Operations

| | | First Operand | |
|---|---|---|---|
| | | DECIMAL FIXED (p1,q1) | DECIMAL FLOAT(p1) |
| Second Operand | DECIMAL FIXED (p2,q2) | DECIMAL FIXED(p,q)<br>p=p1+p2+1<br>q=q1+q2 | DECIMAL FLOAT(p)<br>p=MAX(p1,p2) |
| | DECIMAL FLOAT (p2) | DECIMAL FLOAT (p)<br>p=MAX(p1,p2) | DECIMAL FLOAT (p)<br>p=MAX(p1,p2) |

Figure 4. Attributes of Result of Multiplication Operations

| | | First Operand | |
|---|---|---|---|
| | | DECIMAL FIXED(p1,q1) | DECIMAL FLOAT(p1) |
| Second Operand | DECIMAL FIXED (p2,q2) | DECIMAL FIXED(p,q)<br>p=15<br>q=15-((p1-q1)+q2) | DECIMAL FLOAT(p)<br>p=MAX(p1,p2) |
| | DECIMAL FLOAT (p2) | DECIMAL FLOAT(p)<br>p=MAX(p1,p2) | DECIMAL FLOAT(p)<br>p=MAX(p1,p2) |

Figure 5. Attributes of Results of Division Operations

| First Operand | Second Operand (Exponent) | Target Attributes of Result |
|---|---|---|
| FIXED DECIMAL (p1,q1) | Unsigned integer constant with value n | FIXED DECIMAL (p,q)<br>[provided p ≥ 15]<br>p = (p1+1)* n - 1<br>q = q1*n |
| FIXED DECIMAL (p1,q1)<br>or<br>FLOAT DECIMAL (p1) | FIXED DECIMAL (p2,q2)<br>FLOAT DECIMAL (p2) | FLOAT DECIMAL (p)<br>(unless the case above is applicable)<br>p = MAX (p1,p2)<br>p = MAX (p1,p2) |

Figure 6. Attributes of Results of Exponentiation Operations

## CONCATENATION OPERATIONS

A concatenation operation is specified by combining operands with the concatenation symbol ||.

The symbol signifies that the operands, which must be character strings or numeric-character data, are to be joined in such a way that the last character of the operand to the left will immediately precede the first character of the operand to the right, with no intervening characters. Numeric-character data items are converted to character strings before concatenation takes place. The result of a concatenation operation is a character string whose length is equal to the sum of the lengths of the two character-string operands.

For example, if A represents the character string '010234', B the character string '101', C the character string 'XY,Z', and D the character string 'AA/BB', then

```
A||B     yields '010234101'
A||A||B  yields '010234010234101'
C||D     yields 'XY,ZAA/BB'
D||D     yields 'AA/BBXY,Z'
B||D     yields '101AA/BB'
```

## PRIORITY OF OPERATORS

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

```
** prefix+ prefix-      (highest)
*    /                      |
infix+   infix-             |
||                          ↓
< ¬< <= = ¬= >= > ¬>     (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 lower priority.

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

Note that in Model 20 PL/I only one comparison operation can appear in one expression. In case there is a comparison operation, it can appear only in the expression immediately following the IF in the IF statement.

The order of evaluation of the expression in the IF statement:

```
IF A * B + D ¬= C THEN . . .
```

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

```
(A) * (B)
(A * B) + (D)
(A * B + D) ¬= (C)
```

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

```
IF A * (B + D) ¬= C THEN . . .
```

In such an expression, the expression enclosed in parentheses is evaluated first and reduced to a single value, before it is considered in relation to the surrounding operators. In case two parenthesized expressions are surrounding an operator there is, however, no rule within the language that specifies which of the parenthesized expression would be evaluated first.

In other words, the priority of the operators is defined only within a string consisting of operands and operators only. It does not necessarily hold true for an entire expression. Consider the following example:

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

The priority of the operators specifies, in this case, only that the exponentiation will occur before the addition. It does not specify the order of operation in relation to the evaluation of the other operand (A - B).

Any operational expression (except a prefix expression) must eventually be reduced to a final single infix operation. The operands and operator of that operation determine the attributes of the result of the entire expression.

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 * C > D ** E - F
```

In this case, the operators indicate that the final operation will be:

```
(A + B * C) > (D ** E - F)
```

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

```
(A + (B * C)) > ((D ** E) - F)
```

It is undefined in the language which of the two outer parentheses is evaluated first.

## Expressions Containing Function References

In Model 20 PL/I, an operand of an expression is usually a constant or an element variable. An operand can, however also be an expression representing a value which is the result of a computation.

Consider the following example:

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

In this example, the expression SQRT(C) represents a value which is equal to the square root of C. Such an expression is called a _function reference_.

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 procedure written to perform specific computations on the data contained in the list and to substitute the computed value for the function reference.

Assume, in the above example, that C has the value 16. The function reference SQRT( C) causes the execution of the procedure SQRT, which 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 procedure 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 all built-in functions of Model 20 PL/I appears in Part II, under _Built-in Functions_.) You may also write your own functions for specific purposes (as described under _Arguments and Parameters_) and use the names of those functions in function references.

_Note_: Besides returning a value, a function may change the value of any variable in the expression containing the function reference. In this case, the result of the expression is undefined and compatibility with other System/360 compilers is not guaranteed.

The use of function references 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 must, however, not be used in place of a variable representing a receiving field, such as to the left of the assignment statement.

There is, however, one built-in function that can be used as a _pseudo-variable_, i.e., in a receiving field: the SUBSTR function. The SUBSTR pseudo-variable is the SUBSTR built-in function name 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 built-in function name SUBSTR is used both in a normal function reference and as a pseudo-variable.

The SUBSTR built-in function (on the right side of the assignment symbol) extracts a substring of specified length from the named string. The pseudo-variable SUBSTR (to the left of the assignment symbol) indicates the location, within a named string, that is the receiving field, i.e., it replaces a substring of specified length in the named string.

In the above example, a substring five characters in length, beginning with the 20th character 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 the 20th through the 24th characters of B. The first five characters of A remain unchanged, as do all characters of the string B.

_Note_: None of the functions you may write for specific purposes can be used as a pseudo-variable.

The built-in function SUBSTR is discussed in Part II, under _Built-In Functions_.

# Statement Classification

This section describes the statements available in Model 20 PL/I, their functions and purpose. Examples of their use are also shown.

A more detailed description of each statement may be found in Part II, under Statement, and in the section Data Transmission which shows examples.

The statements used in Model 20 PL/I can be grouped into the following six classes:

- descriptive statements,
- input/output statements,
- data-movement and computational statements,
- program-flow control statements,
- exception-control statements, and
- program-structure statements.

The names of the classes are merely descriptive, 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 types of data. Each data item, except a constant, is referred to in the program by a name. Model 20 PL/I requires that the names and characteristics or attributes of data items referred to must be known at the time the program is compiled, that is, translated into machine language.

### The DECLARE Statement

With the DECLARE statement you specify the attributes of the data assigned to each variable. But although it is the characteristics of the data you describe with the attributes, it is the data name with which the declared attributes are associated. Consequently, when a value is assigned to a name (or variable) whose attributes describe characteristics that are different from the attributes of the data, the value will be converted where possible so that it will have the characteristics of the variable to which it is assigned. For example, when fixed-point data is assigned to a variable that has the FLOAT attribute the data item is converted to floating-point representation. If conversion is not possible, a diagnostic message is given by the compiler.

A variable for which a complete set of attributes has not been specified, is given default attributes by the compiler.

You always need DECLARE statements for fixed-point decimal variables, character-string variables, filenames, entry names pointer variables, label variables, arrays and structures, data with the EXTERNAL STATIC, BASED, DEFINED or INITIAL attribute, all data with the PICTURE attribute and the built-in function DATE. A RETURNS attribute declaration must be made for the name of any function that returns a value with attributes different from the default attribute FLOAT DECIMAL(6) which is applicable if the name of the function starts with a letter other than I through N. You find a complete discussion of the RETURNS attribute in the section Arguments and Parameters.

DECLARE statements may be an important part of the documentation of a program; consequently, 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, separate DECLARE statements can be used for different groups of names. This can make modification easier and the interpretation of diagnostics clearer. Note, however, that a structure must be completely declared in one DECLARE statement.

### The FORMAT Statement

The FORMAT statement may be thought of as describing the layout of data on an external medium, for example on a page or on an input card. You will find a complete discussion of the FORMAT statement in Part II, under Statements, and in the section Data Transmission.

## Input/Output Statements

Input/output statements cause a transfer of data between main storage and an external storage medium, such as disk, tape, or card.

In the following list, the statements that cause a transfer of data are grouped into two classes, Record I/O and Stream I/O:

Record I/O Transfer Statements:

READ
WRITE
REWRITE
LOCATE

Stream I/O Transfer Statements:

GET
PUT

There are two important differences between RECORD transmission and STREAM transmission. In STREAM transmission, the file on the external medium is considered a continuous stream of data items, in character form. On input, the data items are assigned from the stream to variables; on output they are transferred from variables into the stream. In RECORD transmission, the file is considered to be a collection of physically separate records that are transmitted as an entity to or from variables. STREAM transmission implies conversion. All of the items in the stream are in character form. On input, they are converted automatically to conform to the attributes of the variables to which they are assigned; on output, data items are converted, if necessary, to characters. In RECORD transmission, there is no conversion; data is transmitted, exactly as it is recorded either internally or on the external medium.

Record transmission is mainly used for processing large files that are written in an internal representation, such as packed-decimal. Stream transmission may be used for processing keypunched data and for producing readable output, such as lists or tables, where editing is required. Record transmission is faster because no conversion is involved.

### RECORD I/O Transfer Statements

The READ statement causes a transfer of data (records) from the external medium to main storage. The WRITE statement causes a transfer of records from main storage to the external medium. The LOCATE statement causes the creation of new records by making space available in a buffer in which the record may be built. The REWRITE statement replaces a record in an UPDATE file that has been read from disk.

### STREAM I/O Transfer Statements

STREAM transmission is always sequential and can be performed only by means of GET and PUT statements. The GET statement causes a transfer of data from the external medium to main storage, and the PUT statement from main storage to the external medium. In STREAM transmission, data is considered to be a stream of individual data items. Record boundaries are generally ignored, but synchronization with record boundaries is possible.

Note: GET and PUT statements can also be used for internal data movement in connection with the STRING option. The GET and PUT statements with the STRING option are discussed under Data Movement and Computational Statements in this section.

### Input/Output Control Statements

Other I/O statements that affect the transfer of data are input/output control statements. They are:

OPEN
CLOSE

The OPEN statement associates a filename (as declared in your program) with a file (the actual data recorded on an external medium) and prepares the file for processing.

An OPEN statement need not always be written for STREAM transmission. Execution of a GET or PUT statement with the name of an unopened file specified will result in the automatic (implicit) opening of the file before data transmission takes place. However, the OPEN statement can be used to force opening of a file at a specific time during program execution, e.g., after issuing a message to mount a specific tape. For RECORD transmission, the OPEN statement must always be present.

The CLOSE statement dissociates a file from the file declaration used in the program and terminates processing of the file. All files are automatically closed at the termination of a program, so that a CLOSE statement is not always required. It may be used, for example, when the same device on which a file resides is to be used for another file in the course of the program.

### The DISPLAY Statement

The DISPLAY statement is used to display a one-byte message on the CPU console, usually to the operator. It is used, with the REPLY option, to allow the operator to communicate with the program by returning a one-byte message. Execution of the program is suspended until the operator acknowledges the message. An example of the use of the DISPLAY statement is shown in Part III, under Linking PL/I Programs With Assembler Procedures.

## 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 operational expression.

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 can also be used for internal data movement. The PUT statement can, in addition, specify computations to be performed.

### The Assignment Statement

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

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

The first form is used purely for internal data movement. The value of the variable (or constant) to the right of the assignment symbol is assigned to the variable on the left. The second form includes an operational expression (B + C), whose value is to be assigned to the variable to the left of the assignment symbol (A). The second statement specifies both computations and 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, because the result of the computation to the right of the assignment symbol is converted to conform to the attributes of the variable to the left of the assignment symbol.

In the assignment statement A = B + C, the variable on the left must be an element variable or an array; it must not be a structure. The expression on the right must only contain single constants and/or element variables as operands. Thus, this form of the assignment statement can be used only to assign single items to element variables or arrays. In the assignment statement, A = B, the variables to the left and right may either be both array names or both structure names or an array name on the left side and an element variable or constant on the right side.

If they are array names, both arrays must have identical dimensions and bounds. If they are structure names, both structures must have identical structuring and the attributes (including arithmetic attributes) of all corresponding variables in the two structures must be identical.

Example:

```
DECLARE

   A1(10,10) FLOAT, A2(10,10) FIXED,
   A3(100) FIXED,
   A4(5,5) FLOAT, V1 FIXED,
   1 S1, 2((E1,E2) CHAR(10), F FIXED(7,2)),
   1 S2, 2(E CHAR(20), F FIXED(7,2)),
   1 S3, 2((E1,E2) CHAR(10), F FIXED(7,2));
```

Valid assignment statements are:

```
   A1 = A1(5,1);
   A1 = A2;
   A1 = V1;
   S1 = S3;
```

Invalid assignment statements are:

```
  A1 = A3;       Dimensions not identical.
  A1 = A4;       Bounds not identical.
  A1 = A1 + A2;  Right side is an opera-
                 tional expression, not a
                 name.
  S1 = S2;       Structuring not identical.
```

### The GET STRING and PUT STRING Statements

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 character-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 the variable FIRST, the next character to the variable MIDDLE, and the remaining 17 characters to the variable LAST.

The PUT statement with the STRING option specifies the reverse operation, that is, the values of the specified variables are to be concatenated into a character string and assigned to 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 first 12 characters of FIRST, the first character of MIDDLE, and the first seven characters of LAST are to be concatenated in that order, and assigned to the string variable NAME. If FIRST or LAST are shorter than 12 or 17 characters, respectively, then blanks are added at the end of FIRST or LAST until the specified length of 12 or 17, respectively, has been reached.

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) EDIT
        (A * 3, B + C)
        (F(15),F(15));
```

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. Note that while arithmetic to character-string and character-string to arithmetic conversions are not allowed in Model 20 PL/I, they can be effectively achieved by the GET STRING and PUT STRING operations, respectively; however, note also that this can be inefficient because of the amount of execution time and main storage that is required.

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

## Program-Flow Control Statements

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

```
GO TO
IF
DO
CALL
RETURN
END
```

### The GO TO Statement

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

If you use subscripted label variables, that is, labels grouped into an array, you can control the switch by varying the subscript. Usually, however, simple program-flow control statements are the most efficient.

Note: The keyword GO TO may be written either as two separate words or as a single word, GOTO.

### The IF Statement

The IF statement provides the most common conditional branch and is 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-if-true") is executed. After execution of the THEN clause, control branches around the ELSE clause (the "action-if-false"), and execution continues with the next statement. Note that the THEN clause can contain a GO TO statement or some other program-flow 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 program-flow control statement that causes a branch, either conditional or unconditional. If the THEN clause contains only a GO TO statement, for example, there is normally 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 keyword IF cannot contain more than one comparison operation. However, nested IF statements can be used to test for more than one condition. Consider the following example:

```
IF A = B
      THEN IF A = C
          THEN D = E;
          ELSE F = G;
      ELSE F = A;
      GO TO LABEL_1;
```

In the example, E is assigned to D only if A is equal to both B and C. If A is equal to B, but not to C, then G is assigned to F. If A is not equal to B, then A is assigned to F. If either the innermost THEN clause (D = E) or the innermost ELSE clause (F = G) is executed, control skips to the GO TO statement following the final ELSE clause.

In a series of nested IF statements, each ELSE clause is paired with an IF, starting at the innermost level. IF-condition tests in nested IF statements are made in the order from autermost to innermost IF. As soon as a test is reached that is not true, the checking stops, and the matching ELSE clause is executed. Control is then transferred to the statement following the entire series of IF statements, unless it is directed otherwise by a GOTO statement in the ELSE clause.

In the nesting of IF statements, an associated ELSE clause may or may not appear for the outermost IF. But every nested IF must have an associated ELSE clause when any IF statement at a higher level requires an associated ELSE clause.

Assume that a programmer writing the above nested IF statements does not want to provide a second alternative for the innermost IF statement. If A is equal to B but not equal to C, he wants to go to the statement labelled LABEL_1. To achieve this, he would have to insert a null statement, as follows:

```
IF A = B
      THEN IF A = C
          THEN D = E;
          ELSE;
      ELSE F = A;
      GOTO LABEL_1;
```

An ELSE with a null statement as its clause is called a null ELSE.

In this example, if A is equal to B but is not equal to C, the second alternative of the innermost IF is chosen. Since it is a null ELSE, control is transferred out of the entire nest to the next statement, which is GOTO LABEL_1.

The examples have illustrated the nesting of IF statements only to the second level. Deeper nesting is, however, allowed. In Model 20 PL/I, the number of IF and DO statements in one nest must not exceed 20. Any IF statement, at any level, may have a DO group as either or both of its alternative actions.

The DO Statement

The most common use of the DO statement is to specify that a group of statements is to be executed and re-executed one or more times while a control variable is incremented each time control passes through the loop. Such a group might have the form:

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

A DO group is delimited by the DO and END statements. In this example, the DO statement specifies that the statements between the DO and END statements are to be executed, as a group, ten times before control passes to the next statement. The variable COUNTER is used to control the number of times the group is executed. When the DO group is executed for the first time, COUNTER is assigned the value 1. Then the group of statements is executed. When the END statement is reached, COUNTER is incremented by one, and control is transferred back to the beginning of the group where COUNTER is tested to see whether it does not exceed ten. This looping continues until the value of COUNTER exceeds 10, then control passes to the statement following the loop. The above example is exactly equivalent to the following:

```
       COUNTER = 1;
LOOP:  IF COUNTER > 10
          THEN GO TO NEXT;
            .
            .
            .
       COUNTER = COUNTER + 1;
       GO TO LOOP;
NEXT:  next statement
```

Note that since the test is made after COUNTER is incremented, its value at the end of the loop will be one incrementation larger than the number of times the loop is executed. In this case, the value of

COUNTER will be 11 when execution of the loop is terminated.

The variable COUNTER, either as used in the DO statement or as used above, would have to be declared to represent values as great as 11; e.g., DECIMAL FIXED (2).

In the preceding example, the value of COUNTER is increased by 1 each time the DO statement is executed. An incrementation of one is assumed unless some other specification is made. Consider the following example:

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

This DO statement causes the initial value of COUNTER to be set to two. Each time the DO statement is executed thereafter, the value is increased by two. Thus, the statements of the DO group would be executed five times, and the final value of COUNTER would be 12.

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

```
   DO COUNTER = 1 TO 10;
         ARRAY(COUNTER) = COUNTER;
         END;
```

In this example, each element of ARRAY is set to 1, 2, 3,...10, respectively.

Note: The control variable of a DO group, in our example COUNTER, must not be changed during execution of the DO group by its appearance on the left side of an assignment statement or in a data list of a GET statement or indirectly through a procedure that has been called in the DO group.

DO groups, like IF statements, may be nested. Consider the following example:

```
   DO I = 1 TO 10;
   statement 1
   statement 2
   statement 3
         DO J = 1 TO 10;
         statement 1a
         statement 2a
         statement 3a
         END;
   statement 4
   statement 5
   statement 6
   END;
```

The statements of the outer DO group - the outer DO-END and statements 1 through 6 - would be executed ten times. The statements of the inner DO group - the inner DO-END and statements 1a to 3a - would be executed 100 times, ten times for each execution of the outer DO group. When the first DO statement is executed the first time, the counter variable I is assigned the value 1. Then statements 1 through 3 are executed. When control reaches the second DO statement, the variable J is assigned the value 1, and the inner loop is executed ten times. Control then passes on to statements 4, 5, and 6. When the final END statement is reached, control returns to the first DO statement. The counter I is incremented to 2, and execution proceeds through statements 1 through 3. When the second DO statement is reached for the second time, the counter J is reset to 1, and the inner DO group again is executed ten times before control passes to statement 4 for its second execution. The process is repeated until the outer DO group has been executed ten times. The inner DO group goes through its entire looping process immediately following each execution of statement 3.

The example shows nesting only to the second level. In Model 20 PL/I, the maximum number of DO statements allowed in one nest is 12.

The Non-Iterative DO Statement. The DO statement need not specify repeated execution of a DO group. You can use a simple DO statement in conjunction with a DO group, 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 or ELSE clause of an IF statement. For example:

```
   IF A = B
         THEN DO;
           .
           .
           .
         END;
         ELSE DO;
           .
           .
           .
         END;
```

### The CALL, RETURN, and END Statements

A procedure (except the main procedure of a program) or a function is invoked (or activated) by a CALL statement that names the entry point of the procedure. Control is returned to the invoking (or activating) procedure when a RETURN statement is executed in the called procedure or when the execution of the END statement terminates the called procedure.

The RETURN statement with a parenthesized expression is used to return a value to a function reference. This form can be used only to return from a procedure that has been invoked by a function reference. The RETURN statement without a parenthesized expression cannot be used in this case.

A program normally is terminated by the execution of the END or RETURN statement of the main procedure, either of which returns control to the Monitor program.

Consider the following example:

```
 FIRST:  PROCEDURE OPTIONS (MAIN);
           .
           .
           .
         CALL SECOND;
           .
           .
           .
         END;
SECOND:  PROCEDURE;
           .
           .
           .
         IF A ¬=B
            THEN RETURN;
              C = A ** 3 + PRODCT(X,Y,Z);
           .
           .
           .
         END;
PRODCT:  PROCEDURE (A,B,C);
           .
           .
           .
         IF A>B + C
            THEN RETURN (0);
            ELSE RETURN (A-B*C);
         END;
```

In the above example, FIRST is the name of the main procedure in the program which consists of the three procedures FIRST, SECOND, and PRODCT. During program execution, the procedure SECOND is invoked by the CALL statement CALL SECOND in the main procedure FIRST. When SECOND is executed, the IF statement is encountered. Depending on the result of the comparison operation, control is either returned to the statement immediately following the CALL statement in the invoking procedure FIRST, or the statement containing the function reference PRODCT(X,Y,Z) is executed invoking PRODCT. When PRODCT is executed, another IF statement is encountered and a test is made. Depending upon the result of the comparison operation, one of the two RETURN statements is executed returning control together with the evaluated result of an expression to SECOND. At the termination of SECOND, control is passed back to the main procedure.

## Exception-Control Statements

The 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.

In general, an exceptional condition is the occurrence of an unexpected action, such as an overflow, 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 this part of the manual, under Exceptional_Condition_Handling.

### The_ON_Statement

The ON statement is used to specify the action to be taken when a program interrupt occurs due to a specific condition. ON statements can be used with a number of different conditions. For each of these conditions, a standard system action is specified as part of the Model 20 PL/I, and if no ON statement for a condition has been executed and is in force at the time an interrupt occurs, the standard system action will be taken. For most conditions, the standard system action is to print a message and terminate execution.

The ON statement is a compound statement that contains a GOTO or null statement. For example:

```
ON UNDERFLOW GO TO ERROR;
```

In the above statement, GO TO ERROR is the contained statement, or the ON-unit. ERROR is the label of a statement or of the first of several statements that specify what action is to be taken, whether to try to recover from the error or to note the error and continue with other computations. If the standard system action is to be taken if an interrupt occurs, the keyword SYSTEM is used in place of the ON-unit, as follows:

```
ON UNDERFLOW SYSTEM;
```

The ON-unit can also be the null statement. The example

```
ON UNDERFLOW;
```

specifies that when an interrupt occurs as the result of an underflow condition, the interrupt is to be ignored and execution is to continue from the point at which the interrupt occurred. If an ON statement for UNDERFLOW was not in force at the time of an interrupt, the standard system action would be taken.

Note: If the condition of the ON statement is CONVERSION, ENDFILE, or KEY, the action must not be the null statement.

The effect of the ON statement, either standard system action or any other action specified by the programmer, can be changed within a procedure by the execution of another ON statement naming the same condition with either another ON-unit or with the word SYSTEM, which re-establishes standard system action. The action in effect at the time another procedure is activated is passed to the activated procedure and remains in effect in that procedure and in other procedures activated by it, unless another ON statement for the same condition is executed. When control returns to an activating procedure, actions are re-established as they existed in that procedure.

```
   FIRST:  PROCEDURE OPTIONS (MAIN);
             .
             .
             .
     ON1:  ON FIXEDOVERFLOW GOTO L1;
             .
             .
             .
           CALL SECOND;
             .
             .
             .
           END;
  SECOND:  PROCEDURE;
             .
             .
             .
     ON2:  ON FIXEDOVERFLOW GO TO L2;
             .
             .
             .
     ON3:  ON FIXEDOVERFLOW SYSTEM;
             .
             .
             .
           END;
```

Until statement ON1 is executed the standard system action is taken in the case of a FIXEDOVERFLOW condition. After execution of the statement ON1, control goes to a statement with the Label L1 when a FIXEDOVERFLOW condition occurs. After calling the procedure SECOND, the program still branches to the same statement L1 when the FIXEDOVERFLOW condition is raised, even though the Label L1 may not be known in the procedure SECOND, until the statement ON2 is executed. Then control branches to statement L2 declared in SECOND instead of L1 declared in FIRST in the case of FIXEDOVERFLOW. After executing the statement ON3, the standard system action is taken in the case of FIXEDOVERFLOW. After returning from procedure SECOND, the action of statement ON1 is reestablished, i.e., branching to L1.

Note: In this example it is assumed that the statement of each of the two procedures are executed sequentially. If this is not the case due to a GO TO statement or DO-group with iteration, the same ON-statement may be executed more than once during the time of activity of a procedure.

## Program-Structure Statements

Program-structure statements are those statements used to divide a program into procedures and DO-groups. These statements are the PROCEDURE statement, the END statement, and the DO statement.

### The PROCEDURE Statement

A program may consist of a single procedure or of several separate procedures. Each procedure in a program is headed by a PROCEDURE statement and ended by an END statement, as follows:

```
   FIRST:  PROCEDURE;
             .
             .
             .
           END;
```

Each procedure must have a name, that is, each PROCEDURE statement must be labelled. In the example above, FIRST is the label of the PROCEDURE statement. The procedure name specifies the entry point through which control can be passed to the procedure.

Control does not pass automatically from one procedure to another. Each procedure to be executed, except the first, must be invoked, or called, from some other procedure. This may be done with either the CALL statement or by the appearance of a procedure name in an expression. If it appears in an expression, the procedure is called a function. A function reference causes a single value to be computed and returned to the function reference for use in the evaluation of the expression.

Communication between two procedures is achieved by

1. passing arguments (that is, expressions) from an invoking procedure to the invoked procedure,

2. returning values from the invoked procedure, and

3. referring to names that are known within both procedures (that is, names declared as EXTERNAL in both procedures).

### The DO Statement

Another kind of program structure is provided by the DO-group, which is delimited by a DO statement and the associated END statement. See the DO statement as discussed above under the heading Program-Flow Control Statements.

This section discusses how control passes within a program from one procedure to the next, how procedures are activated and terminated, and how storage may be allocated for data within procedures.

## The Procedure

As we have already stated, a procedure is headed by a PROCEDURE statement and ended by an END statement, as follows:

```
label:  PROCEDURE;
          .
          .
          .
        END;
```

Each procedure must have a name, that is a label. The label denotes the entry point through which control can be passed to the procedure.

The division of a program into several procedures is a feature of PL/I that provides a special convenience to programmers. The procedures can be written separately, compiled separately or together, and executed as a single program. A long program can be divided into logical blocks (or procedures); special procedures can be written for special purposes. The division of a program into procedures also provides great economy in the use of main storage space.

### ACTIVATION OF A PROCEDURE

Control does not pass automatically from one procedure to the next. Each procedure, except the first, must be invoked, or called, from some other procedure, where the entry name of the procedure to be invoked must appear

- after the keyword CALL in the CALL statement or
- as a function reference (see the section Arguments and Parameters for details).

When a CALL statement or a function reference is executed, the procedure with the specified entry name is activated or invoked. Control is transferred to the specified entry point. The point at which the procedure reference occurs, is called the point of invocation. The procedure in which the reference is made is called the invoking procedure. The invoking procedure remains active even though control is transferred from it to the procedure it invokes. Whenever a procedure is invoked, execution begins with the first executable statement in the invoked procedure.

The first procedure in a program, called the initial or main procedure, can only be activated by the Monitor program of the DPS. The main procedure must always have the OPTIONS (MAIN) attribute specified in its PROCEDURE statement, as follows:

```
FIRST:  PROCEDURE OPTIONS (MAIN);
          .
          .
          .
        CALL A;
        CALL B;
          .
          .
          .
        END;
```

In this example, FIRST is the initial procedure that invokes other procedures in the program.

Following is a summary of rules that apply to the activation of procedures.

- A program becomes active when the initial procedure is activated by the Monitor program.
- Except for the initial procedure, all procedures contained in a program are activated by a reference to them, either in a CALL statement or in a function reference.
- A procedure cannot be invoked while it is active.
- The initial procedure remains active for the duration of the program.
- All activated procedures remain active until they are terminated (see below).

### TERMINATION OF A PROCEDURE

In general, a procedure is terminated when one of the following conditions occurs:

1. Control reaches a RETURN statement within a 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 is resumed at the statement immediately following the CALL statement. If the point of invocation is 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. A GO TO statement causes the termination of one or more procedures by branching back to one of the series of invoking procedures. In this case, the destination of the GO TO must be specified by a label variable. If one of the terminated procedures has been invoked as a function reference in an expression, the evaluation of the expression is discontinued.

### PROGRAM TERMINATION

A program is terminated when one of the following conditions occurs:

1. Control reaches a RETURN statement or the final END statement in the main procedure. This is the normal termination of a program.

2. An error condition arises from which the system cannot recover. In this case, the standard system action results in a return of control to the Monitor program.

## Storage Allocation

Any name (or variable) that represents a data item, actually represents the location in main storage where that data item is recorded. The compiler analyzes the attributes of a variable to determine the length of the storage area that is needed and _when_ it will have to be available.

When a location in main storage has been associated with a variable, the storage is said to be _allocated_ to the variable. _Allocation_ for a given variable may take place _dynamically_, that is, during execution of a procedure, or _statically_, that is, before execution of a program.

### DYNAMIC ALLOCATION

The fact that certain variables are used in one procedure of a program and not in others, makes it possible to allocate storage dynamically, that is, to allocate the same storage space, at different times, to different variables.

We have, in Model 20 PL/I, two types of dynamic storage, _automatic_ and _based_.

### Automatic Storage

If during execution of a single procedure, but nowhere else in the program, a 100-character field is required for the variable TABLE, the space need not actually be allocated until execution of that procedure begins. If the value of TABLE is not needed when the procedure is invoked again, there is no need to keep the space reserved after execution of the procedure is completed. The storage area can be used for other purposes.

Such dynamic use of main storage is called automatic allocation. Variables of the automatic storage class are declared with the AUTOMATIC attribute. They are allocated storage space when the procedure is invoked and this storage space is freed when the procedure is completed. Once a storage area is freed, the value of the corresponding variable is lost.

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.

### Based Storage

A variable of the based storage class is declared with the BASED attribute; you associate a based variable with storage by means of a READ or LOCATE statement with the SET option or by means of the ADDR built-in function. This causes the attributes of the _based variable_ to be applied to the main storage area "pointed to" by a _pointer variable_ associated with the based variable.

The pointer variable can be manipulated so that the attributes of the based variable apply to different storage areas. That is, the value of the pointer variable (which represents an address) can be changed so that the storage area pointed to by the old pointer value is no longer associated with the based variable. The attributes of the based variable now apply to the storage area pointed to by the new value of the pointer variable.

You will find a complete discussion of this topic in the section _Based Variables and Pointer Variables_.

### STATIC ALLOCATION

Whenever the value of a variable must be saved between different invocations of the same procedures, storage for that variable has to be allocated statically, that is, storage is allocated before execution of the program and remains allocated throughout the entire duration of the program. For a detailed description of the EXTERNAL attribute, refer to the following section *Recognition of Names*).

### Static Storage

All variables of the static storage class have the STATIC attribute. Variables with the EXTERNAL attribute must always be of the static storage class. For such variables, STATIC is the default storage-class attribute and need not be explicitly declared.

Consider the following example:

```
CNTRL: PROCEDURE OPTIONS (MAIN);
       DECLARE X FIXED (5,0) INIT (1)
                   STATIC,
               Y FIXED (5,0) INIT (1)
                   STATIC EXTERNAL,
               Z FIXED (5,0) INIT (1)
                   STATIC;
           .
           .
           .
       CALL OUTP;
           .
           .
           .
       END;
OUTP:  PROCEDURE;
       DECLARE X FIXED (5,0) INIT (1)
                   STATIC EXTERNAL,
               Y FIXED (5,0) INIT(1) STATIC
                   EXTERNAL;
           .
           .
           .
       X = X+1;
       END;
```

Before execution of the program begins, all static variables are allocated main storage space. Thus, in the above example, X, Y, and Z are allocated storage and initialized to 1 when the program is loaded into main storage. Note that X in CNTRL and X in OUTP are different variables, because they have not been declared EXTERNAL in both, while Y in CNTRL and Y in OUTP refer to the same variable. Before execution of OUTP is terminated, X is increased by 1. If OUTP is invoked a second time, X has the value 2. Note that even though OUTP could be activated and terminated several times, X, being STATIC, retains its value between each termination and re-activation of OUTP. The storage space associated with X remains allocated to X throughout the duration of the program.

### INITIAL DATA

Variables, whether main storage space is allocated to them dynamically or statically, may be given *initial values* at the time of storage allocation.

Static variables are *initialized* only once, that is, before execution of a program begins. Automatic variables are re-initialized at each activation of the declaring procedure.

*Note*: The INITIAL attribute may be specified only for element variables, elementary names in structures, and arrays. If specified for arrays, however, these must not belong to the AUTOMATIC storage class. It cannot be specified for STATIC label variables.

## Prologues and Epilogues

Each time a procedure is activated, certain activities must be performed before control reaches the first executable statement in the procedure. These activities are called a *prologue*. Similarly, when a procedure is terminated, certain activities must be performed before control can be transferred out of the procedure. These activities are referred to as an *epilogue*.

Prologues and epilogues are set up by the compiler and not by the programmer. They are discussed here because knowledge of them may assist you in improving the performance of your programs.

### PROLOGUES

A prologue is a routine set up by the compiler and logically attached to the beginning of a procedure. It is executed as the first step after activation of the procedure. The main activities performed by a prologue are:

- Allocation of main storage for automatic variables.
- Establishment of the inherited ON-units.
- Allocation of storage for dummy arguments (which are discussed in the section *Arguments and Parameters*) that may be passed at the time another procedure is invoked.

## EPILOGUES

An epilogue is a routine set up by the compiler and logically appended to the end of a procedure. It is executed as the final step before termination of the procedure. The main activities performed by an epilogue are:

- Re-establishment of the ON-units as they existed before the procedure was activated.

- Release of all main storage that has been allocated to automatic variables in the procedure.

A PL/I program consists of a collection of identifiers, constants, and special characters used as operators or delimiters. Identifiers may be either keywords or names with a meaning specified by the programmer. The PL/I language is constructed in such a way that the compiler can usually determine from context whether or not an identifier is a keyword, so that there are very few reserved words i.e., words that you may not use as identifiers in your programs (see note below). Any identifier that is not a reserved word may be used as a name; the only restriction is that at any point in a procedure a name can have one and only one meaning. For example, the same name cannot be used in the same procedure for both a file and a floating-point variable.

Note: With the 60-character set there are no reserved words. When using the 48-character set, you may not use the following operators as identifiers in your program: GT, GE, NE, LE, LT, NG, NL, CAT, NOT, OR, AND, and PT (even though the last four identifiers are not used as keywords in Model 20 PL/I). These are fully reserved. No other keywords are reserved.

It is not necessary for a name to retain the same meaning throughout a program. A name declared within a procedure has a meaning only within that procedure. Outside the procedure it is unknown unless it has been declared with the EXTERNAL attribute and has also been declared in another procedure with the same attributes including the EXTERNAL attribute.

That part of a program in which the meaning of a declared name is known, is called the scope of a name. A name in a Model 20 PL/I program can be declared by

- explicit declaration
- contextual declaration, or
- implicit declaration.

## Explicit Declaration

A name is explicitly declared if you specify it:

- in a DECLARE statement,
- in a parameter list (that is, the parenthesized list that follows the keyword PROCEDURE in a PROCEDURE statement,
- as a statement label, or
- as the label of a PROCEDURE statement (that is, an entry name).

The appearance of a name in a parameter list has the same effect as a DECLARE statement for that name following the PROCEDURE statement in which the parameter list occurs (though the same name may also appear in a DECLARE statement in the same procedure). The default assumptions for parameters are the same as for other variables.

The appearance of a statement label constitutes an explicit declaration equivalent to the declaration of a variable in a DECLARE statement.

The scope of an explicitly declared name is the procedure in which the name has been declared.

## Contextual Declaration

When an identifier that has not been explicitly declared appears

- to the left of the assignment symbol in an assignment statement, or
- to the left of the assignment symbol in a DO statement (or in a repetitive specification in a data list), or
- in the data list of a GET statement,
- as a built-in function name,

the identifier is said to be contextually declared. However, a name can only be contextually declared if its name does not start with one of the letters I to N. In Model 20 PL/I, a contextually declared identifier is always a scalar arithmetic variable with the default attributes FLOAT DECIMAL(6) AUTOMATIC INTERNAL. This rule is illustrated by the example below:

```
P:   PROCEDURE (PAR);
     DECLARE N FIXED DECIMAL (5,2);
      .
      .
      .
     A = N + 1;
LBL:  GET FILE (FL1) EDIT (N,B) (F(7,2),
          E(12,4));
      .
      .
      .
     END;
```

The names PAR, N, LBL, and P are declared explicitly by their appearance in a parameter list, a DECLARE statement, and as a statement or procedure label, respectively. A and B are declared contextually by their appearance on the left side of the assignment symbol and in the data list of a GET statement, respectively. FL1 must be declared in another DECLARE statement (not shown in this example) in the same procedure, since file names cannot be declared contextually in Model 20 PL/I.

Built-In Functions

The name of a built-in function, that is, a procedure that is part of the compiler and that is invoked by means of a function reference, is contextually declared by its appearance in the function reference. It must not be explicitly declared by means of a DECLARE statement.

Exception: The DATE built-in function must be explicitly declared in the invoking procedure by means of a DECLARE statement and with the attribute BUILTIN.

Note 1: The name of a built-in function may be used in a procedure to describe an item other than that function. In this case, it must not appear in a function reference in that procedure. Consider the following statements:

```
   DCL SIN(10);
        .
        .
        .
   X=SIN(5);
```

In this example, SIN is explicitly declared to be an array of 10 data elements. When the statement X=SIN(5); is evaluated the array element and not the built-in function SIN is taken. The explicit declaration takes recedence over the contextual declaration.

## 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 INTERNAL, and it is by default an arithmetic variable with the attributes FLOAT DECIMAL(6) AUTOMATIC. Note, however, that the name of an implicitly declared variable must not start with any of the letters I to N.

An implicitly declared variable can only obtain a value if it appears as an argument in a procedure invocation. There is no other way for an implicitly declared variable to obtain a value. Consider the following example:

```
ALPHA:  PROCEDURE;
           .
           .
           .
        D = E;
        CALL BETA(C);
        A = C + 1;
           .
           .
           .
        END;
 BETA:  PROCEDURE(X);
        DCL X FLOAT DECIMAL(6);
        X = 52;
           .
           .
           .
        END;
```

In the above example, E in procedure ALPHA is an implicitly declared variable that cannot assume any value. The variable C which appears as an argument in the CALL invoking BETA, is also an implicitly declared variable. In BETA, however, the argument C, represented by the parameter X, is explicitly declared and given a value. Thus, when BETA returns control to ALPHA and the instruction A = C + 1 is executed, C has been given a value, since both C and X refer to the same data item in main storage.

Note: The attributes associated with a name comprise those explicitly, contextually, or implicitly declared, as well as those assumed by default. The default for each attribute is given in Part II under Attributes.

## INTERNAL and EXTERNAL Names

The scope of a name declared with the INTERNAL attribute is the procedure in which it is declared. A declaration of that name in another procedure refers to a different item with a different scope.

A name with the EXTERNAL attribute may be declared more than once in the same program, in different procedures. All declarations of the name that specify the EXTERNAL attribute refer to the same item. The scope of the name is the sum of the scopes of its individual declarations within the program.

Since these declarations all refer to the same thing, they must have the same set of attributes. It may be impossible for the compiler to check this, since the declarations appear in different procedures; therefore, take care to ensure that different declarations of the same name with the EXTERNAL attribute have matching attributes (including the INITIAL attribute, if present).

In Model 20 PL/I, the length of a name with the EXTERNAL attribute is restricted to six characters. This restriction also applies to names that are EXTERNAL by default, such as file names and entry names of procedures.

## Multiple Declarations and Ambiguous References

Two or more declarations of the same identifier within the same procedure constitute a *multiple declaration*, unless all but one of the identifiers is declared within structures 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 with different attributes and with the EXTERNAL attribute constitute a multiple declaration, that is, a name declared with the EXTERNAL attribute in different procedures must have identical attributes in all procedures in which it is declared EXTERNAL.

Multiple declarations are errors.

A name need have only enough qualification to make the name unique. 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;
DECLARE 1 B, 2 A, 3 C, 3 E;
 A.C = D.E;
```

In this example, A.C refers to C in the first declaration because C in the second declaration has another complete name qualification B.A.C; A.C is a unique qualified reference; D.E refers to E in the first declaration.

```
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 variable declared in 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.

# Data Transmission

PL/I provides input and output statements, which cause data to be transmitted between main storage and an external storage medium. Transmission of data from an external storage medium to main storage is called input, and transmission of data from main storage to an external storage medium is called output.

In order to understand the basic concepts of input and output, it is necessary to define the following terms: file, volume, block, record, and filename.

A file is a collection of data on an external storage medium.

Files can be stored on a variety of external storage media, such as punched cards, reels of magnetic tape, and packs of magnetic disks. Despite their variety, external storage media have many common characteristics that permit standard methods of collecting, storing, and transmitting data. For convenience, thus, we use the general term volume 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 file are arranged in distinct physical groupings called blocks. These blocks allow the file to be transmitted and processed in portions rather than as a unit. For processing purposes, each block consists of one or more logical parts called records, each of which can contain 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 parts, the logical subdivisions are called logical records.

When a physical record contains two or more logical records, the records are said to be blocked. Blocking of records often permits more compact and efficient use of external 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 interrecord gaps is reduced, and much more data can be stored on a full length of tape. For example, on a tape with a density of 800 characters/ inch and interrecord gaps 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, which would be a total of 1.6 inches. Thus, 100 records would now occupy 16 inches of tape only; 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 with its physical organization in external storage.

To be able, now, to deal with the data items of a file, that is, to read them into main storage or to write them onto the external storage media, a filename has to be associated with a file. You do this in your program.

A filename is the symbolic reference, within a program, to a file stored on an external medium.

You associate a filename with a file by declaring the filename for the file in your program and by specifying attributes that describe the file and the manner in which it will be handled. For example, the INPUT attribute specifies that a file is to be read; OUTPUT specifies that a file is to be written. Unlike a file, however, a filename has significance only in a program; it does not exist external to the program. For example, if you use the same file again, you may specify a different filename and partly different attributes for it.

## Types of Data Transmission

There are two types of data transmission you can use in a PL/I program, stream-oriented transmission and record-oriented transmission.

In stream-oriented transmission, the data in the file is considered to be a continuous stream of data items in character form. Consequently, characters in the input stream are interpreted and converted where necessary to the specified internal format. Whether conversion is to take place is determined by the attributes of the variable to which a data item is assigned. On output, data items in internal format are converted where necessary to character form and added to the output stream. The statements GET and PUT 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_ contained in each GET and PUT statement.

Although data in the file is in record format, 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 file 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 file; on output, it is transmitted exactly as it is recorded internally.

The input and output statements used for record-oriented transmission are READ, WRITE, REWRITE and LOCATE. The READ, WRITE, and REWRITE statements cause a single logical record to be transmitted to or from a data variable or, in the case of READ with the SET option, to an intermediate work area in main storage, called a buffer, which can be accessed by the program. The LOCATE statement allocates an area in a buffer to which data for an output record can be assigned.

Note that although records may be blocked, in which case actually the physical record is transmitted to or from the file as an entity, each data transmission statement in record I/O is concerned with a logical record. Blocked records are deblocked automatically.

The following discussion of filenames and file attributes will be of particular interest to a programmer using record-oriented transmission. File handling is simpler when using stream-oriented transmission, and fewer attributes are applicable to files read or written by stream-oriented transmission.

## File Declarations

To allow a source program to deal primarily with the logical aspects of data rather than with its physical organization in a file, PL/I employs a symbolic representation of a file, the filename.

PL/I requires a filename to be declared and allows the characteristics associated with that filename to be described by keywords called _file attributes_. The DECLARE statement specifying the filename and the associated attributes is referred to as _file declaration_.

### _FILE ATTRIBUTES_

The following lists show file attributes that are applicable to each type of data transmission:

| _Record Transmission_ | _Stream Transmission_ |
|---|---|
| FILE | FILE |
| RECORD | STREAM |
| INPUT | INPUT |
| OUTPUT | OUTPUT |
| UPDATE | PRINT |
| SEQUENTIAL | ENVIRONMENT |
| DIRECT | |
| KEYED | |
| BACKWARDS | |
| ENVIRONMENT | |

A detailed description of each of these attributes appears in Part II, _Attributes_. Following is a brief description of each attribute and its use in a file declaration.

#### _The FILE Attribute_

The FILE attribute states that the identifier (or name) associated with FILE is a filename. For example, MASTER is declared to be a filename in the following statement:

```
DECLARE MASTER FILE
```

The FILE attribute must be explicitly declared for every filename, and it must always be the first attribute declared in a file declaration.

### 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 of a group and if one of the alternatives is required, a default attribute is assumed in most cases.

| Group | Alternatives | Default |
|---|---|---|
| Transmission | STREAM \| RECORD | STREAM |
| Function | INPUT \| OUTPUT \| UPDATE | no default |
| Access | SEQUENTIAL \| DIRECT | SEQUENTIAL |

Figure 7. Groups and Default Attributes for Alternative File Attributes

PL/I provides three groups of alternative file attributes. Each group is discussed individually. The groups and the default for each alternative file attribute is shown in Figure 7.

Note: No default is applied for the function attributes; one of them must be specified in each file declaration. The scope of a filename must always be EXTERNAL. The EXTERNAL attribute can be explicitly declared in a file declaration. If it is not declared, it is assumed by default.

An additive attribute is one that must be stated explicitly or is implied by another explicitly stated attribute. The ENVIRONMENT attribute must be declared explicitly with every filename. An additive attribute can never be applied by default. The additive attribute KEYED is implied by the DIRECT attribute.

The additive attributes are:

```
   PRINT
   BACKWARDS
   KEYED
   ENVIRONMENT
```

### 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 the file associated with the filename to be treated as a continuous stream of data items recorded in character format.

The RECORD attribute causes the file associated with the filename to be treated as a sequence of logical records, each record consisting of one or more data items recorded in any format.

```
   DECLARE MASTER FILE RECORD
   DECLARE DETAIL FILE STREAM
```

### The INPUT, OUTPUT, and UPDATE Attributes

The function attributes determine the direction of data transmission. The INPUT attribute applies to files that are to be read only. The OUTPUT attribute applies to files that are to be created or extended, and hence are to be written only. The UPDATE attribute describes a file that can 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.

```
DECLARE
   DETAIL FILE RECORD INPUT
   REPORT FILE STREAM OUTPUT
   MASTER FILE RECORD UPDATE
```

### The SEQUENTIAL and DIRECT Attributes

The access attributes are used only in conjunction with the RECORD attribute and describe how the records in the file are to be accessed, sequentially or directly. STREAM transmission is always sequential.

The SEQUENTIAL attribute specifies that physically or logically successive records in the file are to be accessed sequentially.

```
DCL PAYROLL FILE RECORD INPUT SEQUENTIAL
```

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 physical or logical 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 file must be read from or written on a direct-access device, for example, a disk drive.

```
DCL MASTER FILE RECORD UPDATE DIRECT
   [KEYED]
```

### 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 is to appear on printed pages, although it may first be written on some other medium. The PRINT attribute specifies that the print lines are to be created with the first character position reserved for a printer-control character, which is inserted automatically.

```
DCL OUT_F FILE STREAM OUTPUT PRINT
```

### The BACKWARDS Attribute

The BACKWARDS attribute indicates that a file is to be accessed in reverse order, beginning with the last logical record and ending with the first logical record. The BACKWARDS attribute can be used only in connection with the RECORD SEQUENTIAL, and INPUT attributes and only with files on magnetic tape.

```
DCL IN_FLE FILE RECORD INPUT SEQUENTIAL
   BACKWARDS
```

### The KEYED Attribute

The KEYED attribute is used only in connection with INDEXED files. It indicates that each record in the file has a key and can be accessed using one of the key options (KEY or KEYFROM) of data transmission statements.

```
DCL REPORT FILE INPUT DIRECT KEYED
```

The use of keys is discussed in detail in Part III, under Input/Output.

### The ENVIRONMENT Attribute

The ENVIRONMENT attribute specifies information about the physical organization of the file associated with a filename. These characteristics are indicated in a parenthesized option list in the ENVIRONMENT attribute specification. They are not part of the PL/I language, but are defined to be recognized by a specific compiler. The option list for the Model 20 PL/I Compiler is discussed in Part III, under Input/Output.

```
DCL OUTPUT FILE STREAM OUTPUT PRINT
   ENVIRONMENT (option-list);
```

Note: As stated earlier in this section, each file must be explicitly declared; the FILE attribute and the ENVIRONMENT attribute must appear in every file declaration.

## Opening and Closing Files

Before the data of a file can be transmitted by input or output statements, certain preparative actions must be taken, such as checking for the availability of the external storage medium, and positioning the medium. 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 used during the opening of the file.

Model 20 PL/I provides two statements, OPEN and CLOSE, to perform these functions. All files for which RECORD transmission has been specified must be explicitly opened before any data can be transmitted. However, files for which STREAM transmission has been specified, need not be opened explicitly. If you do not specify an OPEN statement for such a file, the file is opened automatically when the first GET or PUT is executed; this form of opening is referred to as implicit opening. Automatic preparation is exactly the same as if an explicit OPEN had been executed before the GET or PUT. With both STREAM and RECORD transmission, all files that have not been closed before completion of a program will be closed automatically upon completion of the program. With the exception of INDEXED files, all files that have been explicitly closed may be reopened.

The following discussions show the effect of OPEN and CLOSE 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 (filename) [option-list]
     [, FILE(filename) [option-list]]...;
```

The OPEN statement is executed by routines that are loaded dynamically by the system at the time the OPEN statement is executed.

For a file that has to be opened explicitly, the OPEN statement must be executed before any I/O statements are executed for the same file.

### The CLOSE Statement

The basic format of the CLOSE statement is:

```
CLOSE FILE (filename)
     [,FILE(filename)]...;
```

Executing a CLOSE statement dissociates the specified filename from the file with which it became associated when the file was opened for, say, input. When using the same file again for, say output, another file declaration has to be made for it before it can be accessed.

Note: Closing an already closed file or opening an already opened file has no effect.

## Environmental Considerations for Data Files

The PL/I object program produced by the Model 20 PL/I compiler is designed to be executed under control on the Model 20 Disk Programming System (DPS). The DPS provides data management facilities that control the organization, location, storage, and retrieval of files. 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 files organizations supported by the data-management facilities of the DPS.

### DEVICE INDEPENDENCE OF INPUT AND OUTPUT STATEMENTS

The input and output statements of a Model 20 PL/I program are concerned with the logical organization of a file and not with its physical characteristics.

Some of the detailed information ultimately required by the PL/I program to process a file -- information such as I/O unit numbers and recording density -- does not appear in the PL/I program at all. It is supplied by means of DPS job-control statements at the time the PL/I object program is executed. (Job-control statements are described in Part III of this publication under *Job Control*). This means that the PL/I program need not be recompiled when changes to this information are made.

Other information, such as the I/O device type to be used and the organization of a file to be read or written, is noted in the PL/I program in the ENVIRONMENT attribute of the file declaration. Hence, changes to this type of information only affect this attribute. They require no changes to the actual I/O statements in the PL/I program. However, if changes to the ENVIRONMENT attribute have been made, the PL/I program must be recompiled before it can be executed in accordance with the new information.

I/O statements are *device-independent* to a large extent. This characteristic of PL/I allows you to write a program without any specific knowledge of the I/O devices that will be used for its execution.

### The ENVIRONMENT Attribute

The ENVIRONMENT attribute provides information about the physical organization of the associated file. This information allows the compiler to determine the method of accessing the file.

For the Model 20 PL/I Compiler, the ENVIRONMENT attribute has the following general form:

```
                ┌CONSECUTIVE┐
ENVIRONMENT (   └ INDEXED   ┘
┌F (blocksize [,recordsize])┐
┤V (maxblocksize)           ├
└U (maxblocksize)           ┘
 [BUFFERS ({1|2})]
  MEDIUM (symbolic-device-address,
              device-type)
[CTLASA]
[LEAVE]
[NOTAPEMK]
[NOLABEL]
[VERIFY]
[KEYLENGTH (decimal-integer-constant)]
[EXTENTNUMBER (decimal-integer-constant)]
[OFLTRACKS (decimal-integer-constant)]
[KEYLOC (decimal-integer-constant)]
[ALTTAPE]
[NOWRITE]);
```

The individual options of the ENVIRONMENT attribute must be separated by at least one blank. They are discussed in detail in Part III of this publication, in the section *Input/Output*. Examples of complete file declarations can be found in Part III, under *Two Complete Programming Examples*.

## Stream-Oriented and Record-Oriented Data Transmission

As we have discussed earlier in this section, PL/I provides two types of data transmission, stream-oriented and record-oriented.

With *stream-oriented transmission*, a file is considered to be a continuous stream of data items in character format. Data items are transferred from the stream to program variables or from program variables (or expressions) into the stream, with appropriate conversion from or to character format. Stream-oriented transmission statements ignore the boundaries between records.

With *record-oriented transmission*, a file 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 either a program variable or a buffer.

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 file or write a specified logical record into the file. Other record-oriented transmission statements are REWRITE and LOCATE.

The same file can be processed at one time by STREAM transmission and at another time by RECORD transmission; however, the file would have to be in character format to be acceptable for stream transmission.

You specify the method of transmission for a file by declaring the associated filename with either the STREAM or the RECORD attribute.

## *STREAM-ORIENTED TRANSMISSION*

The Model 20 PL/I language provides only one mode of stream transmission: the *edit-directed mode*.

Edit-directed transmission uses the

```
   GET and
   PUT
```

statements for input and output. These statements require the following information:

1. The filename associated with the file from which data is to be obtained or in which data is to be written.

2. A list of program variables which are to receive data items 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 can include constants and expressions.

3. A list containing the format of each data item in the stream. This list is called a *format list*.

*Note*: For PRINT OUTPUT files, you may specify the option PAGE or SKIP instead of or in addition to the data list and format list.

For edit-directed transmission, the general format of the GET and PUT statements is as follows:

```
GET FILE (filename) EDIT (data-list)
         (format-list);

PUT FILE (filename)
  ⎧EDIT (data-list)(format-list)        ⎫
  ⎨option                               ⎬ ;
  ⎩option EDIT(data-list)(format-list)  ⎭
```

whereby *option* may either be PAGE or SKIP.

The *data specification* consists of two parts: the *data list* and the *format list*. Each must be enclosed in parentheses.

On input, the data list contains one or more variables. Values whose format is described in the format list are assigned to these variables. On *output*, the data list may, in addition to variables, also contain constants and other expressions. For each item in the data list, the external format that it is to assume is described in the format list of the PUT statement.

The format list contains one or more *format items*. There are three types of format items:

- *data format items*, which describe whether data items in the stream are characters or arithmetic values in character form:

- *control format items*, which describe page-control, line-control, and spacing operations;

- *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).

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 to be assigned to a variable is specified by the corresponding format item in the format list. The format item also specifies how the associated data item is to be stored internally in character format or as an arithmetic value.

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.

For either input or output, the first data-format item in the format list 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 the format list contains fewer format items than there are items in the associated data list, the format list is reused; 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.

An array variable in a data list is equivalent to *n* data items in the data list, where *n* is the number of data items in the array. Each of the data items in the array will be associated with a separate use of the data format item (consider the third example below).

Names of major and minor structures must not appear im a data list in Model 20 PL/I.

If a data list is associated with a format list that contains *control format items*, the control format action is executed before the next data item is paired with the next *data format item*. For example, on output, lines or pages are skipped before the next item is printed.

The specified data transmission is complete when the last item in the data list has been processed with its corresponding format item. Subsequent format items, including control format items, are ignored.

On output, each data item occupies precisely the field length specified by its corresponding format item in the format list.

*Note*: Data in STREAM files is actually transferred in blocks. Thus, when a STREAM file is closed, the last block is transferred regardless of whether or not it is completely filled with data. This may lead to unpredictable results since, in this case, the end of the data may not coincide with the end of the file. You must therefore ensure that the end of the data is clearly identified.

Consider the following examples:

```
1.  GET FILE (MASTER) EDIT
         (NAME, DATA, SALARY)
         (A(20), X(2), A(6), F(6,2));

2.  PUT FILE (OUTPUT) EDIT
         ('INVENTORY='||INUM,INVCODE)
         (A(20), F(5));

3.  GET FILE (MASTER) EDIT
         (ARRAY, DATA, SALARY)
         (20(A(8)),A(6), F(6,2));
```

The first example specifies that the first 20 characters in the stream are to be treated as a character string (the format item A identifies characters) and assigned to the variable NAME. The next two characters are to be skipped (this is specified by the skipping format item X). The next six characters 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. F is the data format item for fixed-point arithmetic variables.

The second example specifies that the character string 'INVENTORY=' is to be concatenated with the value of character string 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 treated as an optionally signed decimal fixed-point integer constant and placed in the stream right-adjusted in a field with a width of five characters (leading characters are blank). Note that operational expressions can appear in output data lists only.

Assume that, for the third example, ARRAY has been declared as follows:

```
   DECLARE ARRAY (4,5) CHARACTER(8);
```

The example specifies that the first 160 characters in the data stream are to be assigned to the variable ARRAY in character format.

## DATA LISTS

Edit-directed GET and PUT statements require a data list to specify the data items to be transmitted. The general format of the data list is as follows:

```
   (element [,element]...)
```

The nature of the elements depends upon whether the data list is used for input or for output. The following rules apply:

1. On input, a data-list element can be one of the following: an *element variable* or an *array variable*, or the *SUBSTR pseudo-variable*.

2. On *output*, a data list element can be one of the following: an *expression*, an *array variable*.

3. The elements of a data list must be of arithmetic or character-string data type.

4. As shown in the general format, a data list must always be enclosed in parentheses.

### *Repetitive Specifications in a Data List*

Data lists may contain repetitive specifications. Repetitive specifications in a data list are used for the transmission of arrays. The general format of a data list containing repetitive specifications is as follows:

```
(element [,element]...DO
   control-variable = specification)
```

where specification has the following format:

```
   expression_1 [TO expression_2] [BY
                expression_3]
```

Note_1: If both options, TO and BY, are present in a data list, TO must occur first. Note also that the control variable must be an arithmetic element variable.

Note_2: Repetitive specifications in data lists may be nested.

The expressions in the specification, which are the same as in a DO statement, are described as follows:

a) Each expression in the specification is an expression.

b) In the specification, expression_1 represents the starting value of the control variable. Expression_3 represents the increment to be added to the control variable after each repetition of the DO-group. Expression_2 represents the terminating value of the control variable.

Consider the following example:

```
   DO I = 1 TO 10 BY 2
```

In this expression, I is the control variable, 1 is the starting value of the control variable (expression_1), 10 is is the terminating value of the control variable (expression_2), and 2 is the increment to be added to the control variable after each repetition (expression_3).

Repetitive specifications in data lists may be nested. Each DO portion must be delimited on the right with a right parenthesis (with its matching left parenthesis added to the beginning of the list element to be repeated).

When repetitive specifications are nested, the rightmost_DO is at the outer level_of_nesting (DO I = 1 TO 2 in the example below).

Consider the following example:

```
DCL A (2,5,10);

GET FILE(INPUT) EDIT
    ((((A(I,J,K) DO K = 1 TO 10 BY 5)
    DO J = 1 TO 5) DO I = 1 TO 2))
       (format-list);
```

In this example, every fifth element of a three-dimensional array of 100 elements in main storage is filled with a data item from the input stream. Note the four sets of parentheses: The outermost set is required by the data list; the second set is required by the outer repetition, the third by the second repetition etc. The sequence of elements transmitted is equivalent to the sequence of elements that would be transmitted if following nested DO group were executed:

```
DO I = 1 TO 2;
   DO J = 1 TO 5;
      DO K = 1 TO 10 BY 5;
      GET FILE (INPUT) EDIT
          (A(I,J,K)) (format-list);
      END;
   END;
END;
```

This nested DO-group gives values to the elements of the array A in the following order:

```
A(1,1,1), A(1,1,6), A(1,2,1), A(1,2,6), ...
```

Note: Although the DO keyword is used in the data list, a corresponding END statement is not allowed. Note also that a nest of repetitions in a GET or PUT statement must not contain more than three repetitions.

If a data list element is an array name (that is, a name without subscripts), the elements of that array are transmitted beginning with the first element in the array and proceeding until the last element has been transmitted.

Consider the following example:

```
   GET FILE(INPUT) EDIT (A) (format-list);
```

If we consider A to be declared as above, data items from the stream are transmitted to the elements of A in the following order:

```
   A(1,1,1),A(1,1,2),A(1,1,3),...
   A(1,1,10),A(1,2,1)...A(1,5,10),
   A(2,1,1),A(2,1,2),...A(2,5,10)
```

If, in a data list used in an input statement for 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 FILE(INPUT) EDIT
     (N,X,J,SUBSTR(NAME,J,5))(format-list);
```

When this statement is executed, data is transmitted and assigned in the following order (assuming X is a two-by-two array):

1. A new value is assigned to N.

2. Elements are assigned to the array X in the order X(1,1), X(1,2), X(2,1), and X(2,2).

3. A new value is assigned to J.

4. A substring of length 5 is assigned to the string variable NAME beginning at the Jth character.

## FORMAT LISTS

Each edit-directed data list requires its own format list. The format list immediately follows its associated data list in the GET or PUT statement and has the following general format:

```
GET FILE (filename) EDIT
    (data-list) (format-list);
```

where *format list* is defined as:

```
⎧item           ⎫         ⎡,item            ⎤
⎨n item         ⎬         ⎢,n item          ⎥
⎩n(format-list) ⎭         ⎣,n(format-list)  ⎦
```

In the general format, *item* represents a format item of any of the types described below. The letter *n* represents an iteration factor, which must be an unsigned decimal integer constant. A blank or left parenthesis must separate the constant and the following item or format list, respectively. The iteration factor specifies that the associated, that is, the immediately following format item or format list is to be used *n* successive times.

There are three types of format items: data format items, control format items, and the remote format item.

*Data format items* specify whether data in the stream are characters or arithmetic values in character format.

*Control format items* specify page-skipping, line-skipping, and spacing operations. The page-skipping format item (PAGE) can only be used for files having the PRINT attribute. The line-skipping and spacing format items (SKIP and X, respectively) can be used for both PRINT and non-PRINT files, including input files.

*Note*: For files having the attribute PRINT, the PAGE and SKIP format items can also be used outside the format list as an option of the PUT statement. (See the description of the PUT statement in Part II, *Statements*).

The *remote format item* allows reference to format items specified in a separate FORMAT statement elsewhere in the procedure.

Detailed discussions of the various types of format items appear in Part II of this publication, in the section *Edit-Directed Format Items*. The following discussions show how you may use format items 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 whether to interpret the external data as *arithmetic* or *character-string* 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 (arithmetic data in the stream is in character representation and is converted to fixed-point or floating-point representation where applicable). On output, the value of the associated element in the data list is converted, where necessary, to the *character representation* specified by the format item and is inserted into the data stream.

There are three *data-format items*:

- the F-item for fixed-point data,

- the E-item for floating-point data, and

- The A-item for character-string data.

The specifications used with the format items are discussed in detail in Part II, in the section *Edit-Directed Format Items*.

The following examples discuss the use of format items:

1. GET FILE (INPUT) EDIT (ITEM) (A(20));

   This statement causes the next 20 characters in the file called INPUT to be assigned to ITEM, which must be a character-string variable. If it is not a character-string variable, an error results.

2. PUT FILE (MASKFL) 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 output file MASKFL. A decimal point is to be inserted before the last two numeric characters. The number will be right-adjusted in a field of six characters. Leading zeros more than one digit to the left of the decimal point will be changed to blanks, and, if necessary, a minus sign will be placed to the left of the first numeric character. If a minus sign appears, it will replace one leading blank. Consequently, the F(6,2) specification will always allow all digits, the point, and a possible sign to appear.

3. 
```
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 will override this assumption. The value of the number is converted to the attributes of ESTIMATE and assigned to this variable.

4. 
```
GET FILE(A) EDIT (NAME, TOTAL) (A(5),
   F(4,0));
```

   When this statement is executed, the first five characters read are assigned to the variable NAME. The next four characters containing possible leading and/or trailing blanks, are then assigned to the variable TOTAL.

### Control Format Items

Control-format items comprise the following types:

- the spacing format item X,
- the PAGE format item, and
- the SKIP format item.

The spacing format item X specifies relative spacing in the data stream. It can be used with PRINT and non-PRINT files, in GET as well as PUT statements.

The printing format item PAGE can be used only for PRINT files and, consequently, appear only in PUT statements. It specifies that printing is to continue on a new page.

The format item SKIP can be used with PRINT and non-PRINT files, in GET as well as PUT statements. For output, it specifies that printing is to continue on a new line (or, with non-PRINT files, output has to start with a new logical record). For input, it specifies that the remainder of a logical record (the size of which is specified in the file declaration) is to be skipped and reading is to continue at the beginning of the next one.

The format items X and SKIP generally include decimal integer constants, which specify the width of the field to be spaced over, or the number of lines or records to be skipped.

The following examples illustrate the use of the control format items:

1. 
```
GET FILE(IN) EDIT (NUMBER, REBATE)
    (A(5), X(5), A(5));
```

   This statement treats the next 15 characters from the input file IN as follows: the first five characters are assigned to NUMBER, the next five characters are spaced over and ignored, and the following five characters are assigned to REBATE.

2. 
```
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 integer value of COUNT.

3. The following example shows the combined use of control format items in an output file.

```
PUT FILE (OUT)EDIT
     ('QUARTERLY STATEMENT')
     (PAGE, SKIP(2), A(19));
PUT FILE (OUT)EDIT
     (ACCT#, BOUGHT, SOLD, PAYMENT,
     BALANCE)
     (SKIP(3),A(6),X(7),F(7,2),X(8),
     F(7,2),X(7),
     F(7,2),X(7),F(7,2));
```

   The first PUT statement specifies that the heading QUARTERLY STATEMENT is to be written starting with the first character position on line three of a new page in the file OUT. 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 position of the sixth line; the value of BOUGHT is to be written in the same line after skipping 7 character positions; the value of SOLD after skipping 8 character positions, beginning at character position 29, etc.

   Note: The number of lines specified in the SKIP format item must not exceed 3.

4. The following statements show the use of the SKIP and X format items in a GET statement.

```
    GET FILE(IN) EDIT (PART, SEQUENCE)
       (SKIP,X(1),A(71),A(8));

    GET FILE(IN) EDIT (DETAIL)
       (X(19),F(15,2));
```

The SKIP format item in the first GET statement specifies that reading is to continue with the next record. The X(1) format item specifies that the first character of this record is to be skipped and that characters 2 to 72 are to be assigned to the variable PART, and characters 73 to 80 to the variable SEQUENCE.

The second GET statement specifies that the first 19 characters of an input record are to be skipped, and that the next 15 characters are to be assigned to the variable DETAIL.

Note: Control format items are executed at the time they are encountered in the format list. Any control-format item that appears in the format list but has not yet been processed when 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 appear in the same procedure. 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) EDIT (CODE) (F(1));
   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 first GET statement reads a code. This code is tested and, depending on the result, the label variable SWITCH is assigned the value L1 or L2. Thus, the second GET statement can use either of the two FORMAT statements, depending on the current value of SWITCH and, hence, depending on the code that has been read by the first GET.

Another advantage of the remote format item is that it allows many GET/PUT statements to share the same format list.

Note: If the format list contains a remote format item that is contained in a replication nest, the remote format item must not be at a depth greater than 2.

## 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 option of the OPEN statement.

For example:

```
DECLARE REPORT FILE OUTPUT PRINT
        ENVIRONMENT (option-list);

OPEN FILE (REPORT) PAGESIZE(55);
```

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. This can be useful, for example, if a footing is to be written at the bottom of a page. Consider the following example:

```
      ON ENDPAGE (REPORT) GO TO FOOT;
            .
            .
            .
FOOT: PUT FILE(REPORT) SKIP EDIT
            (FOOTING) (A);
      PUT FILE (REPORT) PAGE;
      N = N + 1;
      PUT FILE(REPORT) EDIT ('PAGE ',N)
            (A,F(3));
      PUT FILE(REPORT) SKIP (3);
      GO TO NEXT;
```

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 GO TO FOOT statement will be executed. The first PUT statement specifies that a line is to be skipped, and the value of FOOTING, 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 (N) is incremented, and the character string 'PAGE ' and the new page number N are printed. Note that a blank is included as part of the character string to separate the word from the page number. The F(3) is a format item and allows the page number to go as high as 999. (Format items are discussed in Part II, under Edit-Directed Format Items). The final PUT statement causes two lines to be skipped, so that the next printing will be on line 4. The GO TO NEXT; statement transfers control to the statement labeled NEXT.

The maximum number of characters to be printed on each line (i.e., the line size) is equal to the fixed-length record size specified in the ENVIRONMENT attribute for the file minus one. If you try to write more than the maximum number of characters specified in one line, i.e., without skipping to a new line or page, the excess characters will automatically be placed on the next line.

The PAGESIZE option can be specified only for a file with the PRINT attribute, and it can be specified only in the OPEN statement. The default value for PAGESIZE is 60 lines.

## SUMMARY OF STREAM I/O STATEMENTS

The following is a summary of the I/O statements used with STREAM transmission, along with their options, according to file attributes (the statements are discussed individually in detail in Part II, under Statements).

STREAM INPUT:

```
GET FILE(filename) EDIT
    (data-list) (format-list);
```

STREAM OUTPUT:

```
PUT FILE(filename) EDIT
    (data-list) (format-list);
```

STREAM OUTPUT PRINT:

```
PUT FILE (filename)
/EDIT(data-list) (format-list)       \
\PAGE                                 |
<SKIP(n)                              >
/PAGE EDIT(data-list) (format-list)   |   ;
\SKIP(n) EDIT(data-list) (format-list)/
```

Format lists may contain the following format items:

| | |
|---|---|
| A, E, F | which may be used for any STREAM file |
| PAGE | which may be used only with STREAM OUTPUT PRINT files |
| SKIP,X | which may be used with any STREAM file |
| R | which may be used with any STREAM file |

## RECORD-ORIENTED TRANSMISSION

Files that contain discrete records or which are to be created as a collection of discrete records, may be manipulated with record-oriented input/output statements. These statements are READ, WRITE, REWRITE, and LOCATE.

A general description of these statements is contained in this section. They are described completely in Part II, in the section Statements.

Each record obtained from a file or transferred to a file is defined in terms of data attributes of a variable (usually a structure). For input, the record is obtained from the input file and assigned, without conversion, to the variable. For output, the data is transmitted without conversion to the output file.

The variable whose value is transmitted from or to a file can be

(1) an element variable that is

- not part of an array or structure,
- not a label or pointer variable;

(2) a structure of level 1, i.e., a major structure.

## RECORD I/O STATEMENTS

There are four RECORD I/O statements you can use for data transmission. They are:

READ
WRITE
REWRITE
LOCATE

These I/O statements can be used only in combination with options. The options are: for the READ statement FILE, INTO, SET, and KEY; for the WRITE statement FILE, FROM and KEYFROM; for the REWRITE statement FILE, FROM and KEY; for the LOCATE statement FILE and SET. Each option, its use and its purpose in the pertinent I/O statement, is individually discussed below.

The type of I/O statement and option(s) that you select to transmit a record depends on

(1) the form in which the file is organized on the external medium (CONSECUTIVE or INDEXED);

(2) the method you want to use to access the file (SEQUENTIAL or DIRECT);

(3) the type of activity for which you want to use the file (INPUT, OUTPUT, or UPDATE);

(4) the area in main storage which is to be allocated to the variable containing the record (an I/O buffer set aside by the compiler as a result of the BUFFERS option in the ENVIRONMENT attribute, or a separate area elsewhere in main storage).

The READ statement with the INTO option causes a record to be transmitted from the file to a variable allocated in STATIC or AUTOMATIC storage. You can use it with any INPUT or UPDATE file. In case of blocked records, the READ statement provides for automatic deblocking, so that in your program, you are always concerned with logical records only.

The READ statement with the SET option is used to read a record from a CONSECUTIVE INPUT or UPDATE file if it is desired to operate upon input data in a buffer. This saves main-storage space. The record is made accessible within the buffer by use of a pointer which is set automatically by the SET option to point to the desired logical record in the buffer. For further details, refer to the _SET Option_ below.

The WRITE statement causes a record to be transmitted from main storage to the file. It can be used with any OUTPUT file, and with DIRECT UPDATE, but _not_ with SEQUENTIAL UPDATE. 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 on a direct-access storage device. For SEQUENTIAL UPDATE files, the REWRITE statement specifies that the last record read from the file is to be rewritten in place. 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, which must always have the SET option specified, is used to create an output record in a buffer. The logical record to be created in the output buffer is accessed by means of a pointer that is automatically set by the SET option. For further details refer to the _SET Option_ in this section. The LOCATE statement with the SET option can be used only with CONSECUTIVE OUTPUT files.

## OPTIONS OF RECORD I/O STATEMENTS

The options you may use in RECORD I/O statements differ according to the attributes of the associated file and the purpose of the statement. A list of all the allowed combinations for each type of file is given later in this section.

Each option consists of a _keyword_ followed by a parenthesized specification.

### _The FILE Option_

The FILE option (also called the FILE specification) specifies the name of the file upon which the operation is to take place. It must appear as the first option in every RECORD I/O statement, and consists of the keyword FILE, followed by the filename 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 specifies a variable to which the logical record is to be assigned. It can be used in the READ statement for any type of INPUT or UPDATE file. The variable can be a based variable, if the associated pointer variable has a value. Consider the following statement:

```
READ FILE (DETAIL) INTO (RECORD_1);
```

This specifies that the next sequential record is to be assigned to the variable RECORD_1.

### _The SET Option_

The SET option sets a pointer variable so that it points to a logical record in a buffer. It can be used with the READ statement for CONSECUTIVE INPUT or UPDATE files. It is also used with the LOCATE statement for CONSECUTIVE OUTPUT files. Note that it cannot be used with KEYED files. Consider the following examples:

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

READ FILE (MASTER) SET (REC_ID);
```

This example specifies that the next record from the file MASTER is to be read and that the pointer variable REC_ID is to be set (automatically) to point to that record in the I/O buffer. If the logical record is part of a record block and is not the first record in the block, the actual result of the statement will be merely to set the value of the pointer to point to the next logical record in the block. The value of REC_ID must be associated with a BASED variable, so that the fields of the record can be accessed.

The name MASTER_RECORD is the based variable that is used to describe a record located in a buffer. Fields of the record must conform to the attributes declared for MASTER_RECORD. REC_ID is the pointer variable that identifies the position of MASTER RECORD within the buffer. The pointer variable is declared explicitly.

After reading a record from the file MASTER, the structure declaration MASTER_RECORD is "overlaid" on the buffer so that you can access the data in it and process them by using the names declared in the based structure. The statements

```
LOCATE MASTER_RECORD FILE
      (PAYROLL)SET(REC_ID);
MASTER_RECORD=PAYRECORD;
```

specify that the based variable MASTER_RECORD is to be allocated storage in a buffer and that its location is to be assigned to the pointer variable REC_ID, which must have been declared as the pointer to the based variable as shown above. If the record MASTER_RECORD is part of a record block, the next LOCATE statement may only allocate storage in the buffer to the next logical record in the same block. The record is actually written when the block is completed by a LOCATE or WRITE statement, or by a following CLOSE statement regardless of whether or not the block is complete.

After setting the pointer in the LOCATE statement, you have to assign to the associated based variable the data items in the record which is to be stored in the output buffer, as shown in the second statement above.

The pointer REC_ID is set to point to the location of the based variable MASTER_RECORD in the output buffer. Then the data items that are to be transmitted are assigned to the based variable in the assignment statement.

Both MASTER_RECORD and PAYRECORD must have identical structuring and attributes. Based variables may be element variables, structure variables, or array variables.

### The FROM Option

The FROM option must be used in the WRITE statement for any OUTPUT file and in the WRITE or REWRITE statement for a DIRECT UPDATE file. You may also use it in the REWRITE statement for a SEQUENTIAL UPDATE file. The FROM option specifies the variable from which the record is to be written. Consider the following statements.

```
   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 KEY Option

The KEY option applies only to files of INDEXED organization. The "key" is the control field in the record i.e., the field according to which the file is organized. The KEY option must be used in the READ and REWRITE (or WRITE) statements for DIRECT files with the INPUT or UPDATE attribute. You may use the KEY option in the READ statement of SEQUENTIAL files with the INPUT or UPDATE attribute and INDEXED organization. Any file for which the KEY option is used must have the KEYED attribute.

If a REWRITE is executed for an INDEXED file you must ensure that the key portion of the record is not changed.

If an INDEXED file is being read sequentially, the KEY option can be used to position the file at a specific record. Subsequent READ statements without the KEY option will cause sequential reading to continue from that point in the file.

Keys for INDEXED SEQUENTIAL OUTPUT files must be in ascending sequence.

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 must be a _character string_ whose length is adjusted to the length specified in the KEYLENGTH option of the ENVIRONMENT attribute. The source key must exactly match the _recorded key_ in the file. The recorded key is embedded in the data part of each logical record. The key location within the record must be specified in the KEYLOC(n) option of the ENVIRONMENT attribute.

The expression in the KEY option must result in a valid key. Consider the following statements:

```
READ FILE(MASTER)INTO(PAY_REC)KEY(NAME);
REWRITE FILE(MASTER)FROM(PAY_REC)KEY(NAME);
```

The first statement specifies that the record of the file MASTER with a key identical to the value of the variable NAME is to be read into the variable PAY_REC.

The second statement specifies that the record of the file MASTER with a key identical to the value of the variable NAME is to be updated.

### The KEYFROM Option

The KEYFROM option is used only for files of INDEXED organization. You may use it in a WRITE statement to create or extend an INDEXED OUTPUT file or to add new records to an INDEXED UPDATE file. Consequently, it can appear in a WRITE statement for an INDEXED SEQUENTIAL OUTPUT file or for an INDEXED DIRECT UPDATE file. It cannot be used for files with the CONSECUTIVE attribute. Any file for which the KEYFROM option is specified must have the KEYED attribute.

If a WRITE is executed for a file with INDEXED organization, the key value specified in the KEYFROM option is moved automatically to the position in the record specified by KEYLOC (in the ENVIRONMENT attribute).

The KEYFROM option specifies the logical location, within the file, where the record is to be written. It specifies the recorded key, whose value is used to determine the location. It is written with the keyword KEYFROM followed by a parenthesized expression. The expression always has to result in a character string. The length of the recorded key has to be specified in the KEYLENGTH option of the ENVIRONMENT attribute. Consider the following example:

```
WRITE FILE(PAYROLL)FROM(PAY_REC)
     KEYFROM(NAME||ADDRESS);
```

This statement specifies that the value of PAY_REC is to be written into the location specified by the value of the character-string variable NAME concatenated with the character-string variable ADDRESS. The _source key_ is to be a concatenation of the value of NAME and the value of ADDRESS, and is to be written as the _recorded key_.

## SUMMARY OF RECORD I/O STATEMENTS AND ASSOCIATED OPTIONS

This section provides a summary of the allowed RECORD I/O statements, along with their options, according to file attributes.

_CONSECUTIVE INPUT_:

```
READ FILE (filename)INTO(variable);
READ FILE (filename)SET(pointer-variable);
```

_CONSECUTIVE OUTPUT_:

```
WRITE FILE (filename)FROM(variable);
LOCATE variable FILE (filename)
   SET(pointer-variable);
```

_CONSECUTIVE UPDATE_:

```
READ FILE(filename)INTO(variable);
READ FILE(filename)SET(pointer-variable);
REWRITE FILE(filename);
REWRITE FILE(filename)FROM(variable);
```

_INDEXED SEQUENTIAL INPUT_:

```
READ FILE(filename)INTO(variable)
              [KEY(expression)];
```

_INDEXED SEQUENTIAL OUTPUT_:

```
WRITE FILE(file-name)FROM(variable)
              KEYFROM(expression);
```

_INDEXED SEQUENTIAL UPDATE_:

```
READ FILE(filename)INTO(variable)
              [KEY(expression)];

REWRITE FILE(filename)FROM(variable);
```

_INDEXED DIRECT INPUT_:

```
READ FILE(filename)
     INTO(variable)KEY(expression);
```

_INDEXED DIRECT UPDATE_:

```
READ FILE(filename)
     INTO(variable)KEY(expression);

REWRITE FILE(filename)
     FROM(variable)KEY(expression);

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

## NOTES ON FILE ORGANIZATION AND ACCESS METHODS USED WITH RECORD-ORIENTED TRANSMISSION

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

1. SEQUENTIAL specifies that the accessing, creation, or modification of the records in a file is performed in a particular order, that is, from the first record of the file to the last (or from the last to the first if the BACKWARDS attribute has been specified).

2. DIRECT specifies that the accessing or modification of the records in a file is performed in random order. The particular record to be operated upon is identified by a specified key.

3. A file of INDEXED organization that is accessed, created, or modified by the SEQUENTIAL access method has recorded keys. The keys may be ignored while accessing sequentially. The way to create a file containing recorded keys is as an INDEXED SEQUENTIAL OUTPUT file. It is then written in INDEXED organization and can later be accessed by either the SEQUENTIAL or the DIRECT method.

4. INDEXED SEQUENTIAL INPUT and INDEXED SEQUENTIAL UPDATE files may be positioned to a particular record within the file by a READ operation that specifies the key of the desired record. Thereafter, successive READ statements without the KEY option will cause sequential reading to continue from that point in the file. This kind of accessing may be used only if the KEYED attribute is specified in the file declaration.

5. Existing records of a SEQUENTIAL UPDATE file can be rewritten, modified, or ignored, but the number of records cannot be increased or decreased. An existing record in an UPDATE file is replaced through the use of a REWRITE statement.

6. The FROM option in a REWRITE statement for a SEQUENTIAL UPDATE file must name the variable from which the data is to be rewritten.

7. If the READ INTO option is used with a CONSECUTIVE UPDATE file and the next REWRITE statement does not make use of the FROM option, the record in the file is not updated.

8. A WRITE statement adds a new record to a file, while a REWRITE statement replaces an existing record. Thus, a WRITE statement may be used with OUTPUT files and INDEXED DIRECT UPDATE files, but a REWRITE statement may be used with UPDATE files only. Moreover, for INDEXED DIRECT UPDATE 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 file, but also specifies an identifying key to be recorded in the file.

# Editing and Character-String Handling

The data manipulations that can be performed by arithmetic, comparison and the concatenation operations are extended in PL/I by a variety of character-string handling and editing features. These features are specified by data attributes, statement options, built-in functions, and the pseudo-variable SUBSTR.

Following is a general description of each feature, along with illustrative examples.

## Editing by Assignment

The most fundamental form of editing performed by assignment is the conversion of the value assigned to a field to a form that agrees with the attributes of the receiving field. By making the assigned value conform to the attributes of the receiving field, the type, precision, or length of the assigned value may be changed. Such alteration can involve the addition of digits or characters to, and the deletion of digits or characters from the converted item.

### THE ASSIGNMENT STATEMENT

A simple assignment statement can be used for the type of "editing" described above.

#### Type Conversion

Consider the following example:

```
   DCL A DECIMAL FIXED (5,2),
       PROD DECIMAL FLOAT (8);
            .
            .
            .
       A = PROD;
```

Assume that PROD has obtained a value in a statement preceding the assignment statement. This value would then be stored in PROD according to the attributes declared for PROD. The assignment statement causes the value of PROD to be converted to a fixed-decimal value and to be stored in A.

#### Altering the Length of Character-String Data

When a character-string value is assigned to a character-string variable, it is, if necessary, truncated or extended on the right to conform to the declared length of the receiving variable. For example, assume SUBJECT has the attribute CHARACTER(10), indicating a character string of ten characters. Consider the following statement:

```
   SUBJECT = 'TRANSFORMATIONS';
```

The length of the string on the right is fifteen characters; therefore, the string will be truncated on the right so that the last five characters are lost 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 character-string variable, the assigned string is extended on the right with blank characters. Assume that SUBJECT still has the attribute CHARACTER(10). Then the following two statements assign equivalent values to SUBJECT:

```
   SUBJECT = 'PHYSICS';
   SUBJECT = 'PHYSICSbbb';
```

The letter b indicates a blank character.

### OTHER FORMS OF ASSIGNMENT

In addition to the assignment statement, PL/I provides two other ways of assignment that involve editing. Both of them use GET and PUT statements. In one of them actual input and output operations are performed, while in the other data movement is entirely internal.

#### Assignment by STREAM I/O

STREAM I/O operations can be treated as a form of assignment, although transmission occurs between the internal and external storage facilities of the computer.

Stream-oriented I/O operations provide a variety of editing functions that are applied when data items are read or written. These editing functions are similar to those of 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.

Note: Record-oriented I/O operations 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.

### Assignment by Using the STRING Option in the GET and PUT Statements

With the STRING option in the GET and PUT statements you can cause transmission of data between main storage locations rather than between the main storage and external storage facilities. In GET and PUT statements, the FILE option, specified by FILE (filename), is replaced by the STRING option, as shown in the following general format:

```
GET STRING (character-string-variable)
            EDIT (data-list) (format-list);

PUT STRING (character-string-variable)
            EDIT (data-list) (format-list);
```

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 variable. The PUT statement specifies that data items of the data list are to be assigned to the specified character-string variable.

The STRING option is used with edit-directed transmission, which considers the character-string variable to be a continuous string of characters. This 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),F(8));
```

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 multiplied by the value of RATE, and the product is to be handled like F-format output and assigned to the next eight character positions of RECORD.

Frequently, it is necessary to read records of different formats, each of which carries with it an indication of its format in the form of a code. The STRING option provides an easy way to handle such records; for example:

```
READ FILE (INPTR) 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 INPTR. 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 using an F-format item. 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 main 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),F(8));
WRITE FILE (OUTPRT) FROM (RECORD);
```

The X(1) in the format list of the PUT statement specifies that the first character assigned to the character-string variable RECORD is to be a blank, which means "skip two lines", when the file eventually is printed. 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 is 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 numeric character values. (A detailed discussion of each picture character, together with examples of its use, appears in Part II of this publication, in the section Picture-Specification Characters. The following discussions are concerned with general principles that govern the use of the picture specification).

A picture specification is used to describe numeric-character data, which is data that represents a numeric value. It is therefore also referred to as a numeric-character variable.

A picture specification is always enclosed in apostrophes and is used with a PICTURE attribute in a DECLARE statement:

```
DECLARE PAYMT PICTURE '$999V.99';
```

In addition to the picture character 9 (which is used to represent a digit), picture specifications can contain other picture characters that are used to edit numeric-character data. The general functions performed by these additional picture characters are described in Editing Numeric-Character Data below.

As opposed to character-string variables, for which assignment is always from left to right and padding and truncation are on the right, assignment to a numeric-character variable depends upon the location of the assumed decimal point (specified by the picture character V). Values assigned to numeric-character variables 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 the picture character 9 (one or more times), 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 are usually different. Editing characters are actually stored internally in the specified positions of the variable. The editing characters then are considered to be part of the character-string value of the variable; they are not, however, a part of its arithmetic value, which involves only decimal digits, the assumed location of a decimal point, and the sign (if 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 then 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.

No arithmetic variable (except another identical numeric-character variable) can be defined on a numeric-character variable without causing an error.

## EDITING NUMERIC-CHARACTER DATA

The basic picture character of a numeric-character field is 9. Consider the following example:

```
DECLARE COUNT PICTURE '99999';
```

Although COUNT is a string of five characters, it can only contain numeric digits; it is a numeric-character variable whose value can be interpreted as a five-digit unsigned 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 variable. Assume, for example, that COUNT as declared above appears in the following assignment statement:

```
COUNT = 123.45;
```

When the assignment is performed, the decimal point of the constant is aligned on the assumed point declared for the numeric-character variable, and the two rightmost digits are lost. Two zero digits are then inserted on the left side. The effect of the above assignment therefore, is equivalent to that of the following statement:

```
COUNT = 00123;
```

With the picture character V, you can specify an assumed decimal point to be anywhere in a numeric-character variable:

```
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 digits. The decimal point of a value assigned to TOTAL will be aligned between the third and fourth digit positions 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 converted to a character string and then printed, no decimal point would appear in the printed field; its character-string value does not include a decimal point.

A picture specification can contain a decimal-point insertion character (.). It merely indicates that a point is to be included in the character representation of the value. Therefore, the decimal point is part of the character-string value. It does, however, not cause decimal-point alignment during assignment, since it is not part of the arithmetic value. Only the picture character V causes alignment of the decimal point. 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 part of the arithmetic value; it is, however, part of the character-string value. The decimal-point insertion character can appear on either side of the character V. (See Part II, Picture-Specification Characters).

The following two statements assign the same value 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 insertion character that appears in the picture specification for RATE. If RATE were interpreted as a character string and then 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 insertion 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.

In addition to the decimal-point insertion character, PL/I provides two other insertion characters: comma (,) and blank (B), which are used in the same way as the decimal-point insertion character. 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 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 '*bb*12' (where the letter *b* indicates a blank character).

Other picture characters control the appearance of signs and the currency symbol ($) in specified positions of the numeric-character data item. For example, a dollar sign can be inserted 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, would be 1245 with a precision of (4,2), or 12.45.

The picture specification can also specify floating-point formats. These formats are discussed in Part II, *Picture-Specification Characters*.

### *Purpose of Numeric-Character Variables*

The principal purpose of a 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 $**172.50; the asterisks might be used to discourage an attempt to alter the amount. This could be done, for example, with the specification '$****9.99'.

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 require the compiler to insert extra coding. Note also that any editing characters in the picture specification will be disregarded in the conversion. Consider the following example:

```
DECLARE RESULT FIXED DECIMAL (3,2),
        COST PICTURE '$9V.99';
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 2.20 (i.e., 220 with precision (3,2)). If RESULT is printed, neither the $ symbol nor the decimal point will be printed.

## Built-In Functions for Character-String Handling

PL/I provides a number of built-in functions for character-string handling that add power to the string-handling facilities of the language. One of these functions, SUBSTR, can also be used as a pseudo-variable. Following are brief descriptions of the functions (more detailed descriptions *appear in Part II, Built-In Functions and the Pseudo-Variable SUBSTR*).

The *CHAR built-in function* converts a specified data item to a character string. The built-in function allows you to specify the length of the converted string and thus override the length that would result from the standard rules of data conversion.

The *SUBSTR built-in function*, which can also serve as a *pseudo-variable* representing 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 *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 the IBM System/360, 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 the IBM System/360, the character is hexadecimal 00.

# Arguments and Parameters

Data can be referred to in a procedure only if the names identifying that data are known within that procedure, that is, if the procedure lies within the scope of the names. The scope of a name is usually the procedure in which it is declared. The scope can, however, be extended in one of two ways:

- by specifying the *EXTERNAL attribute* for the name, or

- by *passing* the name as an *argument* to a procedure that is to be activated (or invoked).

The type of argument you can pass to an invoked procedure may either be a *variable name* or an *expression* in Model 20 PL/I. File names, labels, and entry names cannot be passed as arguments.

Arguments are passed in a parenthesized list, called an *argument list*, contained in the invoking statement, which may be a CALL statement or a function reference. Different names or expressions in an argument list are separated by commas.

Arguments passed to an invoked procedure must be accepted by that procedure. This is done by the explicit declaration of one or more *parameters* in a parenthesized list in the PROCEDURE statement of the invoked procedure. 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 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. Note that, although an argument and the corresponding parameter refer to the same storage area, they may have different names.

The example below illustrates how parameters and arguments may be used:

```
ALPHA: PROCEDURE;
        DCL BETA ENTRY;
        DCL NAME CHAR (20),
            ITEM CHAR (5);
       .
       .
       CALL BETA (NAME, ITEM);
       .
       .
       .
       END;

BETA:  PROCEDURE (FIELD, OBJECT);
        DCL FIELD CHAR (20),
            OBJECT CHAR (5);
        .
        .
        .
        PUT FILE (OUT) EDIT (FIELD, OBJECT)
        (A(20), X(10), A(5));
        END;
```

In the procedure ALPHA, NAME is declared as a string of 20 characters, ITEM as a string of five characters. The CALL statement in ALPHA invokes the procedure BETA, and the parenthesized list included in this procedure reference contains the two arguments being passed to BETA. The PROCEDURE statement defining BETA declares two parameters, FIELD and OBJECT. When BETA is invoked, NAME is associated with FIELD and ITEM with OBJECT. Each reference to FIELD in BETA is treated as a reference to NAME, and each reference to OBJECT is treated as a reference to ITEM. Therefore, the PUT statement causes the values of NAME and ITEM to be written in the file named OUT.

*Note*: The entry name of the invoked procedure (in the example, BETA) must appear in a DECLARE statement with the ENTRY attribute in the invoking procedure. Excepted from this rule are built-in functions, which are discussed later in this section.

The passing of arguments usually involves the passing of names and not merely of the values represented by these names. Storage allocated for a *variable* before it is passed as an argument is not duplicated in the invoked procedure. Any change of value specified for a parameter in the invoked procedure actually is a change in the value of the argument in the invoking procedure. Such changes remain in effect when control is returned to the invoking procedure.

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 be the same. For example, the program is in error, if a parameter has the attribute FIXED and the corresponding argument has the attribute FLOAT.

A name is explicitly declared to be a parameter by its appearance in the parameter list of a PROCEDURE statement. However, its attributes, unless the default attributes apply, must be explicitly stated within that procedure in a DECLARE statement.

Through the specification of arguments and parameters, procedures and functions can be used throughout a program to perform the same operations upon many different data items whose names may be known only within the invoking procedure.

The difference between a normal procedure and a procedure referred to as a function is that a function usually returns a value to the invoking procedure, whereas a normal procedure does not return any value to the invoking procedure. Functions and procedures are invoked by function and procedure references, respectively, that may or may not contain arguments.

Note: An exception is the main procedure of a program which initially is invoked by the system and cannot be called by any other procedure in a program.

## Passing Arguments to Procedures

Arguments are passed to a procedure in the invoking CALL statement, which is known as procedure reference. The general format of the procedure reference is as follows:

```
CALL entry-name(argument
      [,argument...]);
```

Whenever a procedure is invoked, the arguments in the invoking statement are associated with the parameters of the entry point, and control is then passed to the invoked procedure. The invoked procedure is thus activated, and execution begins.

Upon termination of an invoked procedure, control normally is returned to the invoking procedure. An invoked procedure can be terminated in any of the following ways:

1. Control reaches a GOTO statement that transfers control to an external label.

2. Control reaches the final END statement of the procedure. Execution of this statement causes control to be returned to the first executable statement following the statement that invoked the procedure. This is considered to be the normal return.

3. Control reaches a RETURN statement in the invoked procedure. This causes the same normal return as is caused by the END statement.

4. An error condition encountered in an invoked procedure abnormally terminates execution of that procedure and of the entire program if the error cannot be recovered.

The following example illustrates how an invoked procedure interacts with the procedure that invokes it:

```
A: PROCEDURE;
     DCL READCM ENTRY;
     DCL RATE FIXED (10,3),
         TIME FIXED (5,2),
         DISTANCE FIXED (15,5);
      .
      .
      .
     CALL READCM (RATE, TIME, DISTANCE);
      .
      .
      .
     END;

READCM: PROCEDURE (W,X,Y);
           DCL W FIXED (10,3),
               X FIXED (5,2),
               Y FIXED (15,5);
        GET FILE(INPUT) EDIT (W,X,Y)
                 (F(10,3),F(5,2),F(15,5));
         .
         .
         .
        Y = W * X;
        IF Y > 0 THEN RETURN;
        ELSE PUT FILE(OUTPUT)EDIT
             ('ERROR READCM')(A(12));
        END;
```

The arguments RATE, TIME, and DISTANCE are passed to the parameters W, X, and Y. Consequently, in the invoked procedure, a reference to W is the same as a reference to RATE, X the same as TIME, and Y the same as DISTANCE. This means that any change to the values of W, X, or Y in procedure READCM is a change to the values of RATE, TIME, or DISTANCE, respectively, in procedure A.

## Passing Arguments to Functions

A function is a procedure that usually requires arguments to be passed to it when it is invoked. Unlike a procedure, which is invoked by a CALL statement, 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 procedure, a function can operate upon the arguments passed to it and upon other known data. But unlike a procedure, a function is written to compute a *single value* which is returned, together with control, to the point of invocation, the function reference. This single value can be an *arithmetic*, *character-string*, *picture*, or *pointer value*. An example of a function reference is contained in the following procedure:

```
MAINP: PROCEDURE OPTIONS (MAIN);
       DECLARE SPROD ENTRY;
          .
          .
          .
       X = Y ** 3 + SPROD (A,B,C);
          .
          .
          .
       END;
```

In this 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;
```

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, and neither their names nor the name of the invoked function (SPROD) start with any of the letters I through N, the default attributes of FLOAT DECIMAL (6) are applied to each argument and parameter. 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* usually terminates a function and returns control to the invoking procedure. Its use in a function differs somewhat from its use in a procedure; *in a function*, not only does it return *control*, but it also returns a *value* to the point of invocation. The general format of the RETURN statement, when it is used in a function, is as follows:

```
   RETURN (expression);
```

The expression must be present and must represent a *single value*.

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.

### *ATTRIBUTES OF VALUE RETURNED BY FUNCTION*

You may declare the attributes of the value to be returned by a function in two ways:

1. You can declare them *by default*.

2. You can declare them *explicitly* following the parameter list in the function PROCEDURE statement.

Note that the value of the expression in the RETURN statement is converted within the function, wherever 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. The default attributes FLOAT DECIMAL(6) are therefore applied, since the name of the name of the invoked procedure (SPROD) does not start with any of the letters I through N. Since FLOAT DECIMAL(6) are the attributes that the returned value is expected to have, no conflict exists.

The following example gives you an illustration of how you can declare attributes for the returned value in the PROCEDURE statement. Assume that the PROCEDURE statement for SPROD has been specified as follows:

```
   SPROD: PROCEDURE (U,V,W) RETURNS (FIXED
       DECIMAL);
```

With this declaration, the value returned by SPROD will have the attributes FIXED and DECIMAL. These attributes differ from the ones that would be assigned by default. To avoid possible error conditions, you would have to specify this difference in the invoking as well as the invoked procedure. You can do this with the RETURNS attribute.

### The RETURNS Attribute

The RETURNS attribute has to be specified when a function returns a value that has attributes other than the default attributes FLOAT DECIMAL (6). It appears in the invoking as well as the invoked procedure.

For the invoking procedure, you specify it in a DECLARE statement which must contain the entry name of the function to be invoked and an attributes list. The attributes list specifies the attributes of the value returned by that function. In the invoking procedure the RETURNS attribute appears in the following general form:

```
DECLARE entry-name RETURNS
    (attributes-list);
```

The RETURNS attribute specifies that within the invoking procedure the value returned from the named function is to be treated as though it had the attributes given in the attributes list. The word treated is used because no conversion is performed in an invoking procedure upon any value returned to it. Therefore, if the attributes of the returned value do not agree with those in the attributes list of the RETURNS attribute, an error will probably result.

Thus, in order to specify to the compiler that in MAINP the value returned by SPROD is to be handled as a FIXED DECIMAL value, the following declaration must be given within MAINP:

```
DECLARE SPROD [ENTRY] RETURNS (FIXED
    DECIMAL);
```

The ENTRY attribute may or may not be specified together with the RETURNS attribute.

For the invoked procedure, you have to specify the RETURNS attribute in the PROCEDURE statement, following the parameter list. In the invoked procedure, the RETURNS attribute appears in the following general form:

```
entry-name:  PROCEDURE (parameter-list)
             RETURNS (attributes-list);
```

The RETURNS attribute in the PROCEDURE statement of the invoked procedure specifies that the value returned has to have the attributes specified in the attributes list. Thus, the PROCEDURE statement of SPROD would have to look as already shown above:

```
SPROD:  PROCEDURE (U,V,W) RETURNS (FIXED
    DECIMAL);
```

Conclusively, let us state again when and where the RETURNS attribute has to be specified. It has to be specified

a) in the invoking procedure in the DECLARE statement together with the entry name of the invoked procedure and the attributes list for the value to be returned;

b) in the invoked procedure (which can only be a function) in the PROCEDURE statement.

It is important to note some of the things that are implied in the above discussion. Principally, you should remember that during compilation of the invoking procedure, 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, the compiler can only assume that the values returned will have the default attributes DECIMAL FLOAT (6) unless the initial letter of the entry name is I through N, in which case the RETURNS attribute has to be specified. No conversion is performed for values returned by a function. Therefore, the attributes of the value to be returned must be the same in the invoking as well as in the invoked procedure. The RETURNS attribute must be declared for a function that returns any value with attributes not consistent with default attributes.

## BUILT-IN FUNCTIONS

Similar to function procedures which you can write yourself, 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 character strings. Built-in functions are invoked the same way programmer-defined functions are invoked. Like programmer-defined functions they can only return element values in Model 20 PL/I.

Note: Some built-in functions actually are compiled as in-line code (that is, as though the code of the built-in function actually appeared within the source program) rather than as procedure invocations. Built-in functions can only be referred to in a source program by function references.

Neither the ENTRY attribute nor the RETURNS attribute can be specified for any built-in function name. The name appearing in a function reference is recognized without the need for any further identification; attributes of values returned by built-in functions are known by the compiler.

Built-in function names are PL/I keywords. They are not reserved. You can use any built-in-function name in your program as a name to refer to any data you have defined.

### The ENTRY Attribute

The ENTRY attribute specifies that the associated identifier is an entry name.

The general format of the ENTRY attribute is as follows:

```
DECLARE entry-name ENTRY;
```

You must specify the ENTRY attribute for each entry name appearing in

a) a CALL statement,

b) a function reference referring to a function which returns a value with the default attributes FLOAT DECIMAL(6), that is, a function for which the RETURNS attribute has not been specified.

You must not specify the ENTRY attribute for any built-in function.

Consider the following example, which illustrates the use of the ENTRY and RETURNS attributes:

```
A: PROCEDURE;
      DCL B ENTRY,
          FUNCTN ENTRY,
          C RETURNS (FIXED DECIMAL);
      DCL V DECIMAL FIXED (4),
          W DECIMAL FIXED (3);
        .
        .
        .
    CALL B;
    X = FUNCTN (Y,Z);
        .
        .
        .
    U = C(V,W);
    D = SQRT(U);
        .
        .
        .
    END;
B: PROCEDURE;
        .
        .
        .
     END;
FUNCTN: PROCEDURE(A,B);
        .
        .
        .
    RETURN(A ** B);
     END;
C: PROCEDURE(E,D) RETURNS(FIXED DECIMAL);
     DCL E FIXED DECIMAL (4),
           D FIXED DECIMAL (3);
        .
        .
        .
     RETURN(E * D);
     END;
```

In this example, the procedure A invokes three other procedures (B, FUNCTN, and C) and the built-in function SQRT. B is a normal procedure which returns control to A when its END statement is executed. No arguments are passed to it, and no values are returned. Only the ENTRY attribute has to be specified for it. FUNCTN is a function which is referred to in a function reference (X = FUNCTN(Y,Z)). Since no attributes are declared for the arguments and their corresponding parameters, and their names do not start with any of the letters I to N, they are assumed to have the default attributes. Only the ENTRY attribute has to be specified. The function C (invoked in the function reference U = C(V,W)) returns values not having the default attributes. This means, that the RETURNS attribute has to be specified in both the invoking procedure (A) and in the invoked function (C). The ENTRY attribute may or may not be specified in the invoking procedure. The built-in function SQRT must not be declared with any attributes; it is recognized by the compiler as a built-in function; the attributes of the value returned by SQRT are known to the compiler.

## Relationship of Arguments and Parameters

When a function or procedure is invoked, a relationship is established between the arguments of the invoking statement or expression and the parameters of the invoked procedure. This relationship is dependent upon whether or not dummy arguments are created.

### DUMMY ARGUMENTS

In the introductory discussion of arguments and parameters it was pointed out that the name of an argument and not its value is passed to a procedure 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 allocate storage for such values and assign an internal name for each. These internal names are called dummy_arguments. You cannot access these dummy arguments in the compiler, but you 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 the following cases:

1. If an argument is a constant. For example:

   ```
   CALL X(7.5);
   ```

2. If an argument is an expression involving operators. For example:

   ```
   CALL X(A+B);
   CALL X(+A);
   ```

3. If an argument is itself a function reference containing arguments. For example:

   ```
   CALL X(SIN(Y));
   ```

4. If an argument is an expression in parentheses. For example:

   ```
   CALL X((A));
   ```

   You may enclose an argument in parentheses, as shown in this example, if you want to pass an argument to a procedure, but do not want to change the value of the argument in the invoked procedure.

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 created.

## ARGUMENT_AND_PARAMETER_TYPES

In general, an argument and its corresponding parameter may be of any type, with the following exceptions: file names, entry names and labels. An argument may be a pointer provided that the corresponding parameter is also a pointer; it may be a character string provided that the corresponding parameter is also a character string, etc. However, not all argument/ parameter relationships are so clear-cut. Some need further definition. Such cases are given below.

If a parameter is an array_name, the argument must be an array_name. The data attributes of the argument must agree with those of the parameter. The bounds of the array argument must agree with the bounds of the array parameter.

If a parameter is a structure_name, the argument must be a structure_name. The relative structuring of the argument and the parameter must be the same; the level numbers need not be identical. The data attributes of the elements of the structure argument must match those of the corresponding elements of the parameter.

If a parameter is an element_variable, i.e., a variable that is neither a structure name nor an array name, the argument must be an 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.

A parameter has no storage class and therefore cannot be declared with any storage-class attribute. All arguments must be either STATIC or AUTOMATIC; they cannot be BASED.

Note that the scale and precision of an arithmetic constant passed as an argument must be the same as that of its corresponding parameter. Similarly, the length of a character-string constant passed as an argument must be the same as that of its corresponding parameter.

When a PL/I program is executed, a large number of exceptional conditions are monitored by the system. These conditions are automatically detected whenever they arise. Exceptional conditions may be errors, such as underflow or an input/output transmission error, or they may be conditions that are expected but infrequent, such as the end of file or the end of a page when output is being printed.

Each of the conditions for which a test may be made has been given a name in PL/I. You can use these names to control the handling of exceptional conditions. The condition names are keywords of the PL/I language. For keywords and descriptions of all exceptional conditions, see Part II ON Conditions.

## Enabled Conditions and Established Action

A condition that can occur and cause an interrupt and that is being monitored by the system, is said to be enabled. Any action specified to take place when such an enabled condition arises and causes an interrupt, is said to be established.

The conditions are checked automatically, and when they occur, the system will take control and perform some standard action specified for the condition. All conditions are enabled by default, and the standard system action is established for them.

The most common condition is the ERROR condition. A large number of different errors may cause this condition to arise. Standard system action for the ERROR condition is to terminate the program.

You may specify in your programs whether or not you want some conditions to be enabled, that is, whether or not you want them to cause an interrupt when they arise. If a condition is disabled, occurrence of the condition will not cause an interrupt.

All input/output conditions and the ERROR conditions are always enabled and cannot be disabled. All of the computational conditions may be enabled or disabled. You have to explicitly disable them if you do not want them 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 prefixed to a procedure-statement label by a colon. The condition prefix always precedes the procedure-statement label. A condition name in a prefix list indicates that the corresponding condition is enabled within the scope of the procedure to whose label it is prefixed. The condition names can be preceded by the word NO, without a separating blank or other delimiter, to indicate that the corresponding condition is disabled.

Condition prefixes are effective during the execution of all statements within one procedure including the END statement. However, they are not in effect during the execution of any other procedures which may be invoked by that procedure.

Consider the following example:

```
(NOCONVERSION,NOFIXEDOVERFLOW):
A: PROCEDURE;
     .
     .
     .
   CALL B;
     .
     .
     .
   END;
(NOCONVERSION):
B: PROCEDURE;
     .
     .
     .
   END;
```

In this example, the condition prefix NOCONVERSION disables that condition for procedure A; it is repeated for procedure B. Although B is invoked by A, the condition prefixes have to be repeated if they are to apply to B as well. The condition prefix NOFIXEDOVERFLOW specifies that the condition FIXEDOVERFLOW is to be disabled in A, that is, no interrupt is to occur when that condition arises; however, during execution of B, FIXEDOVERFLOW (as well as all other conditions except CONVERSION) is enabled by default, since NOFIXEDOVERFLOW is not specified in the condition-prefix list to B.

## The ON-Statement

A system action exists for every condition, and if a condition arises, the system action will be performed unless an ON statement has been executed specifying an alternative action for that statement. With the ON statement you can 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 if the condition has been disabled in a condition prefix.

The form of the ON statement is:

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

(For a full description, see Part II, Statements).

The keyword SYSTEM specifies standard system action whenever an interrupt occurs:

```
ON FIXEDOVERFLOW SYSTEM;
```

It reestablishes standard system action for a condition for which some other action has previously been established. In the statement

```
ON FIXEDOVERFLOW GO TO ERROR;
```

GOTO ERROR is the ON-unit. You can use the ON-unit to specify an alternative action to be taken whenever an interrupt occurs. In the above example, ERROR is the label of a statement or the first of several statements that specify the action to be taken, for example to try to recover from the error or to register the error and continue processing.

An ON-unit must either be the null statement or a GO TO statement. A null statement effectively causes the interrupt to be ignored and, in general, causes control to be returned to the point logically following the point at which the interrupt occurred. Thus, the effect of a null ON-unit is to say: "When an interrupt occurs as a result of this condition, do nothing except continue". The above example with a null statement would look as follows:

```
ON FIXEDOVERFLOW;
```

The semicolon (;) is the null statement.

The use of the null statement is not the same as disabling a condition, for two reasons: a) a null statement can be specified for any condition (except ENDFILE, KEY, and CONVERSION), but not all conditions can be disabled; b) 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 the condition, transfer control to the ON-unit whenever an interrupt occurs, although, in the ON-unit, no action other than returning control is taken.

Note: The specific point to which control returns from a null ON-unit varies for different conditions. In most cases, control returns to the point that immediately follows the operation in which the condition arose. The section ON-Conditions in Part II gives the point of return for all conditions for which a null ON-unit can be specified. Return from a null ON-unit is a normal return.

If an ON-unit is a GO TO statement, control is, when an interrupt occurs for the specified ON-condition, transferred to the label specified in the GO TO statement, as described above. Linkage to the point at which the interrupt occurred is thus lost and a normal return cannot occur.

### Scope of the ON-Statement

The ON-statement specifies that a specific action is to be taken for a named condition, that is, the ON-statement associates a condition with a specific action. Once this association is established, it remains in effect until it is overridden by another ON-statement specifying the same condition, or until the procedure in which it appears is terminated.

An established interrupt action (established by an ON-statement, not a condition prefix) passes from a procedure to any procedure it invokes, and the action remains in force for all subsequently activated procedures unless it is overridden by the execution of another ON-statement for the same condition. If it is overridden, the new action, extablished in an invoked procedure, remains in force only until that procedure is terminated. When control returns to the activating procedure, all interrupt actions that were established at the point of invocation, are reestablished. This makes it impossible for an invoked procedure to alter the interrupt action established for the invoking procedure.

If more than one ON-statement for the same condition appears in the same procedure, each subsequently executed ON-statement overrides the previously established action. Re-establishment is only possible through the execution of another ON-statement (for example, by transferring control to an overridden ON-statement).

Consider the following example:

```
A: PROCEDURE;
   ON FIXEDOVERFLOW GOTO A_ERR;
      .
      .
      .
   CALL B;
      .
      .
      .
    END;
B: PROCEDURE;
   ON FIXEDOVERFLOW GOTO B_ERR;
      .
      .
      .
   CALL C;
      .
      .
      .
   END;
C: PROCEDURE;
   ON FIXEDOVERFLOW GOTO C_ERR;
      .
      .
      .
   ON FIXEDOVERFLOW GOTO D_ERR;
      .
      .
      .
   END;
```

The ON-statement in procedure A establishes the action to be taken for the FIXEDOVERFLOW error occurring within A. (Note that FIXEDOVERFLOW is enabled by default and therefore does not require a condition prefix to enable it).

The action specification made in A is carried over into procedure B, because B is invoked by A. Within B, however, the action established in A is overridden by another action specification. When procedure C is called, the action established in B for FIXEDOVERFLOW remains active until it is overridden by the first action specification which, in turn, is overridden by the second action specification in procedure B.

When C returns control to B, the action specified for the FIXEDOVERFLOW error in B is re-established (ON FIXEDOVERFLOW GOTO B_ERR;). When B returns control to A, the action specified in A is re-established (ON FIXEDOVERFLOW GOTO A_ERR;). Standard system action is taken for all other conditions enabled by default.

# Based Variables and Pointer Variables

For each identifier you use in your PL/I programs, the compiler must be able to determine the associated attributes in order to generate correct code.

In addition to determining the type of operation, the compiler must be able to determine the address of each operand. In some cases, the compiler must generate code that will determine the address when the program is executed. The storage class of a variable determines the way in which the address is obtained. There are three distinct classes:

1. Static Storage: The address of an identifier is determined when the program is loaded.

2. Automatic Storage: The address is determined upon entry to the procedure.

3. Based Storage: The address is contained in a pointer variable. The contents of this pointer variable may change during program execution, so that the same identifier can have different addresses at different times.

It is the third class, based storage, with which this section is concerned.

## Pointer Variables

A special type of variable, the pointer variable, is used to locate data in storage; that is, the data in storage is "pointed to" by the pointer variable. Consequently, a pointer variable may be thought of as an address.

## Based Variables

A based variable is a description of data that can be applied to different locations in storage, depending upon the value of the associated pointer variable.

Based variables and pointer variables are used with record-oriented input/output. They allow you to operate upon records in a buffer without allocating storage in addition to the buffers.

With the pointer variable, you can

1. explicitly specify the address of a record to be operated upon in the buffer, and

2. locate, in the buffer, the record that is to be transmitted by record-oriented input/output.

With the based variable, you describe the record pointed to by the pointer variable; that is, the record in the buffer pointed to by the pointer variable is treated as if it had the attributes of the associated based variable.

When a based variable is declared, it must be associated with a pointer that has been explicitly declared. The form of the declaration is:

```
DECLARE identifier BASED
        (pointer-variable);
DECLARE pointer-variable POINTER;
```

For example:

```
DECLARE P POINTER;
DECLARE A BASED (P);
```

Whenever a reference is made to A, the address of A will be the value of the associated pointer variable, P in this case. For example:

```
A = A + 1;
```

In this statement, the pointer used to determine the address of A will, in both cases, be P.

So long as an associated pointer variable has a valid value, any reference to the based variable will always refer to the location in storage identified by the pointer variable.

A restriction imposed by the Model 20 PL/I compiler is that the pointer name used in the declaration of a based variable must be an unsubscripted, unqualified element variable. Pointer variables must not be elements of structures nor of arrays.

## Values of Pointer Variables

Before a reference can be made to a based variable, a value must be given to the pointer associated with it. This can be done in any of four different ways:

1. with the SET option of a READ statement,

2. with the SET option of a LOCATE statement,

3. by assignment of the value of another pointer,

4. by assignment of the value returned by the ADDR built-in function.

READ and SET

```
READ FILE (file-name) SET (pointer);
```

The READ statement with a SET option which can be used only for CONSECUTIVE files, causes a record to be read into a buffer and the specified pointer variable to be set to point to the record in the buffer. A based variable, declared with the same pointer, can then be used to refer to different fields of the record.

A based variable is not a variable for which main storage is reserved, but a pattern which will be overlaid on data in main storage pointed to by the associated pointer variable; that is, if the based variable is a structure variable, the data pointed to by the associated pointer is treated as if it had the same structuring as the based variable. A reference to an element of the based variable has the same effect as if the record had been read directly into the structure described by the based variable.

When records are blocked, the first execution of a READ statement with the SET option causes the transmission of a block of physical records to a buffer and the pointer to be set to point to the beginning of the first logical record. The second execution of the READ statement causes the pointer variable to be set to point to the location of the second logical record in the block already in the buffer.

When records are unblocked, each execution of a READ statement with the SET option causes actual data transmission from the file to the buffer. In this case, the pointer always has the same value.

LOCATE and SET

```
LOCATE variable FILE(filename)SET(pointer);
```

The LOCATE statement, which must always have the SET option and can only be used for CONSECUTIVE files, allocates storage for a based variable in an output buffer. The action is similar to that of a READ and SET, in that the based variable is, in effect, overlaid on the buffer. The LOCATE statement sets the pointer variable to point to the location that a logical record will have in an output buffer after it has been assigned to the associated based variable.

When records are blocked, physical transmission from an output buffer to an output file occurs only after the entire block is in the buffer. Therefore, for blocked records, a LOCATE statement is executed repeatedly before actual data transmission occurs; and for each execution of a LOCATE statement, a pointer variable is set to point to the location of the next logical record to be constructed in the buffer.

Assignment of Pointer Value

```
pointer-variable = pointer-variable;
```

The value of a pointer variable can be assigned to another pointer variable by a simple assignment statement. Assume that Q and P are pointer variables and that P has a valid pointer value.

```
   Q = P;
```

In this statement, Q would point to an input buffer if P had been set by a READ statement, or to an output buffer if P had been set by a LOCATE statement.

Assignment of the ADDR Function Value

The general form in which an ADDR built-in function appears in a statements is:

```
   pointer-variable = ADDR(variable);
```

The value returned to an ADDR function reference 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(A);
```

Execution of this assignment statement will give the pointer variable P a value so that it points to the location of the data variable A. The value of an ADDR function reference can be assigned to a pointer variable only.

The argument of the ADDR function reference can be a variable that represents an element, an array, an element of an array, a major structure, a minor structure, or an element of a structure.

Since the ADDR function can be used to set a pointer to point to a nonbased variable, this facility allows the use of a based variable to refer to the value of a nonbased variable.

The data thus pointed to may then be referred to by means of the pointer value and based variable, provided the attributes of the based variable are compatible with that of the variable identified by the pointer. The rules for the relation between an argument and a parameter also apply to the relation between a variable identified by a pointer and the based variable used to refer to it.

Example:

```
     DECLARE ARRAY (10,10) STATIC EXTERNAL
     FIXED, (P,Q,R) POINTER,
     VALUE BASED (P) FIXED,
     1 GROUP AUTOMATIC,
       2 GROUP1,
          3 A FIXED,
          3 B CHARACTER (2),
       2 GROUP2,
          3 C CHAR (1),
          3 D FLOAT,
     1 DESCRIPTION BASED (Q),
       2 A FIXED,
       2 B CHARACTER (2),
     SWITCH CHAR (1) BASED (R);
```

```
P = ADDR (ARRAY (I,J));
```

This statement assigns a value to the pointer P so that it will point to the location of the (I,J)th element of ARRAY. When using the based variable, VALUE it will be overlaid on the (I,J)th element of ARRAY.

```
P = ADDR (GROUP.A);
```

Provides for the use of the based variable VALUE in referring to GROUP.A.

```
Q = ADDR (GROUP.GROUP1);
```

Provides for the use of the based variable DESCRIPTION in referring to the minor structure GROUP.GROUP1.

```
R = ADDR (GROUP.C);
```

Provides for the use of the based variable SWITCH in referring to GROUP.C.

## Restrictions on Pointer Variables

Because a pointer is very closely related to an address, its value is strongly dependent upon the implementation in which it is used. In order to reduce implementation dependence, some restrictions are made on the use of pointer variables.

1. Pointer variables must not be elements of structures or arrays.

2. Pointer variables must not be operands of any operations except the comparison operations specified by the operators = and ¬=.

3. The value of a pointer variable can be assigned only to another pointer variable.

4. Pointer variables cannot be used for STREAM input and output. When used in RECORD input and output, a pointer value written as output cannot be assumed to point to the same data if it is read back in.

## Use of Based Storage and Pointers

The based storage and pointer handling facilities provided by the Model 20 PL/I compiler are primarily intended to permit processing of records in input and output buffers. This can result in a significant saving of storage, particularly when many different record types exist in the same file.

Many different declarations of based variables can be associated with the same pointer. The effect of this is that once the pointer has been given a value, say by a READ statement with a SET option, then any of the record descriptions associated with the pointer may be used to refer to the record in the buffer. For example:

```
  DECLARE P POINTER;
  DECLARE 1 ISSUE BASED(P),
          2 CODE CHAR(1),
          2 PART_NO PIC '(7)9',
          2 QTY PIC '9999',
          2 DEPT PIC '99',
          2 JOB_NO PIC '(4)9',
        1 RECEIPT BASED (P),
          2 CODE CHAR(1),
          2 PART_NO PIC '(7)9',
          2 QTY PIC '9999',
          2 SUPPLIER PIC '(5)9';

  READ FILE (TRANS) SET (P);
  IF ISSUE.CODE = 'R' THEN GOTO RL1;
  IF SUPPLIER>1000 THEN GOTO INHS1;
```

In this example, the two record descriptions ISSUE and RECEIPT are associated with the same pointer. Once a record has been read and P has been set, the record code (CODE) is tested to determine whether a record with the structure of ISSUE or that of RECEIPT has been read. Depending on the result, either record type is processed. The records do not require working storage, since the pointer refers to a position within the buffer.

The records can also contain variables other than character strings and numeric character fields. Any number of records can be associated with the same pointer. When the pointer is given a value, all of the records will refer to the same storage area and will effectively be overlaid.

Such overlaying of record descriptions can be machine de‸endent and should be used with care.

## Pointer Manipulation

Important for the manipulation of pointer variables is the ADDR built-in function, which has already been briefly discussed. It requires one argument, the name of a variable, and it returns a value that points to the variable. It can be used to find the address of an element variable, an array variable, an element of an array, a major structure, a minor structure, or an element of a structure.

The argument in the ADDR function reference may be the name of a nonbased or based variable.

When using the ADDR function with arrays and structures, it is important to note that the ADDR of the first element of an array or structure is the same as the ADDR of the array or structure itself.

For example, given the following declarations:

```
DECLARE P POINTER;
DECLARE B(10,10) BASED (P),
        A(10,10);
```

ADDR(A(1,1)) is the same as ADDR(A), and with the following assignment:

```
P = ADDR(A);
```

B(1,1) will refer to the first element of A.

When writing your programs, it is entirely your responsibility to ensure that such references do access meaningful storage locations, which must have been allocated in some other way and whose attributes are correct.

# Part II

# Model 20 PL/I Syntax Rules

## Picture Specification Characters

Picture specification characters appear in a PICTURE attribute. You can use them to specify the editing operations to be performed on the associated data item. A discussion of the concepts of picture specifications appears in Part I, in the section Editing and Character-String Handling.

In Model 20 PL/I, a picture specification always describes a numeric-character variable. In the statement

```
DCL NUMBER PICTURE '999V.99';
```

NUMBER is the numeric-character variable described by the picture specification '999V.99', which means that NUMBER may consist of 5 decimal digits and a decimal point to the left of the rightmost two digits. Arithmetic data associated with a picture specification can consist only of decimal digits, an (assumed) decimal point and, optionally, a plus or minus sign. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified. However, these characters are not 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. The picture specification can contain any of the picture specification characters listed in Figure 8.

You can use the picture characters in these groups in various combinations. These combinations depend upon the type of data being described by the specification. A detailed discussion of these types and how they can be described follows below.

A numeric-character variable can be considered to have two different types of value, depending upon its use. They are

1) its arithmetic value and

2) its character-string value.

| Category | Specification | Representing |
|---|---|---|
| Digit and decimal-point specifiers | 9 | any decimal digit |
| | V | assumed decimal point and subfield delimiter |
| Zero suppression characters | Z | digit or blank |
| | * | digit or * |
| Numeric signs and currency symbol (these are also drifting zero suppression characters) | $ | digit, $, or blank |
| | S | digit, ± sign, or blank |
| | - | digit, -, or blank |
| Insertion characters | , | comma |
| | . | decimal point |
| | B | blank |
| Credit, Debit, and Overpunched signs | CR | CR if field<0 |
| | DB | DB if field>0 |
| | T | digit overpunched by sign |
| | R | digit overpunched by - if field<0 |
| Exponent Specifier | E | E (start of exponent) |

Figure 8. Picture-Specification Character

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 arithmetic operation or in an assignment to a variable with either the FIXED or FLOAT attribute. In such cases, the arithmetic value of the numeric-character variable is converted to internal coded-arithmetic representation. The arithmetic value is also used in an assignment to another numeric-character variable.

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 operation or in an assignment to a character-string variable, or wherever a reference is made to a character-string variable that is defined on the numeric-character variable.

A picture specification can be made for fixed-point or floating-point data. The picture specification for a fixed-point value contains only one field, and this field can consist of two subfields: an integer subfield describing the digits to the left of the decimal point in the fixed-point value, and a fractional subfield describing the digits to the right of the decimal point.

```
DCL NUMBER PICTURE '999V.99';
```

A major requirement of the picture specification for numeric-character data is that 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 and *), also specify digit positions.

The picture specification for a floating-point value consists of two fields: a mantissa field and an exponent field. The mantissa field describes a fixed-point value, which when multiplied by ten raised to the power of the value described by the exponent field gives the actual value represented by the floating-point notation; the mantissa field is specified in the same way that a fixed-point field is specified. The exponent field describes a signed or integer power of ten.

```
DCL NUMBER PIC '9V.9999ES99';
```

For further details about picture specifications for floating-point values refer to The Exponent Specifier E in this section.

## Digit and Decimal-Point Specifiers

The picture characters 9 and V are the simplest form of numeric-character specifications you can use to represent fixed-point decimal values.

9 specifies that the associated field position 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 lost on the left, the result will be undefined. If no V character appears in the picture specification of a fixed-point decimal value or in the mantissa field of a picture specification of a floating-point decimal value, a V is assumed at the right end of the field specification. This causes 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.

Figure 9 gives examples of numeric-character specifications using the picture characters 9 and V.

| 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 lost on the left; the result shown, however, is that given for System/360 implementations.

Figure 9. Pictured Numeric-Character Examples

## Zero-Suppression Characters

The zero-suppression picture characters specify *conditional digit positions* in the character-string value. You may use them to cause leading zeros to be replaced by asterisks or blanks. *Leading zeros* are those that

1) occur in the leftmost digit positions of fixed-point numbers,

2) are to the left of the assumed position of a decimal point, and

3) are not preceded by any of the digits 1 through 9.

The leftmost non-zero digit in a number and all digits, zeros or not, to the right of it represent significant digits.

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 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, or R in a field.

*Note 1*: If one of the picture characters Z or * appears to the right of the picture character V, then all fractional digit positions in the specification, as well as all integer digit positions, must employ the Z or * picture character, respectively. When all digit positions to the right of the picture character V contain zero-suppression picture characters, fractional zeros of the value will be 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 will be replaced by blanks or asterisks if the fractional part contains any significant digit.

*Note 2*: Zero-suppression characters must not appear in pictures for floating-point data.

Figure 10 gives examples of the use of zero-suppression characters. In the figure, the letter *b* indicates a blank character.

| 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 |

[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 lost on the left; the result shown, however, is that given for System/360 implementations.

Figure 10. Examples of Zero Suppression

## Insertion Characters

The picture characters comma (,), point (.), and blank (B) are insertion characters; they cause the specified character to be inserted in 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 and point 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 specifies that a blank is to appear in the associated position.

Note: Insertion characters are applicable only to the character-string value. They have no influence on the arithmetic value of the data item.

, causes a comma to be inserted in the associated position of the numeric-character data when no zero suppression occurs. If zero suppression does occur, the comma is inserted only when an unsuppressed digit appears to the left of the comma position, or when a V appears immediately to the left of it and the fractional part contains any significant digits. In all other cases where zero suppression occurs, one of three possible characters is inserted in place of the comma. The choice of character to replace the comma depends upon the first picture character that both precedes the comma position and specifies a digit position:

- If this character is an asterisk, the comma position is assigned an asterisk.
- If this character is a drifting sign of a drifting currency symbol (discussed later), the drifting string is assumed to include the comma position, and the action taken is the same as that for drifting characters.
- If this character 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.

B specifies that a blank character is to be inserted in the associated position of the character-string value of the numeric-character field.

You can use the point (or the comma) in conjunction with the V to cause insertion of the point (or comma) 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 you may desire it in printing, since the V does not cause printing of a point. In this case, 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).

The insertion characters B, comma, and point must be preceded by a digit position in the same field.

Figure 11 gives examples of the use of insertion characters. In the figure, the letter b indicates a blank 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(4,2) | 67.89 | 9,999,999.V99 | 0,000,067.89 |
| 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 | 99.999.9 | 12.345.6 |
| FIXED(6) | 001234 | ZZ,ZZ,ZZ | bbb12,34 |
| FIXED(6) | 000000 | ZZ,ZZ,ZZ | bbbbbbbb |
| FIXED(6) | 000000 | **,**,** | ******** |
| FIXED(6) | 123456 | 99B99B99 | 12b34b56 |
| FIXED(3) | 123 | 9BB9BB9 | 1bb2bb3 |

[1] The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure 11. Examples of Insertion Characters

## Numeric 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.

You may use these picture characters 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, a single drifting character is always inserted (unless the entire field is suppressed) 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 (in the case of a minus sign character if the data value is less than or equal to zero) *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 blank, a sign, or the $ currency symbol.

A drifting character is specified by *multiple* use of that character in a picture field. Thus, if the field contains one currency symbol, it is interpreted as static; if the field contains more than one currency symbol, 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, or B. Any of the insertion characters 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, insertion characters, or a V. If a drifting string exists in a field, zero suppression characters (Z or *) must not appear in the same field.

The position in the data associated with the characters comma, point, and B appearing in a string of drifting characters will contain one of the following:

- comma, point, or blank if a significant digit has appeared to the left;
- the drifting symbol, if the next position to the right contains the leftmost significant digit of the field;
- blank, if the leftmost significant digit of the field is more than one position to the right.

If a drifting string contains the drifting character *n* times, then the string is associated with *n* - 1 conditional digit positions. The position associated with the leftmost drifting character can contain only the drifting character or blank, never a digit. If a drifting string is specified for a field, the other potentially drifting characters can appear only once in the field, i.e., the other character represents a static sign or currency symbol.

Only one type of sign character can appear in each field. An S or a minus (-) 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.

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.

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. 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 equal to or greater than zero, 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 minus-sign character (-) if the data value is less than zero,

| 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 | bbbbbbb |
| 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 | -123.45 |
| FIXED(5,2) | 123.45 | -999V.99 | b123.45 |
| FIXED(5,2) | 123.45 | 999V.99S | 123.45+ |
| 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 12. Examples of Numeric Signs and the Currency Symbol in Picture Specifications

otherwise it specifies a blank. The character may be drifting or static. The rules are identical to those for the currency symbol.

Note: $, S, and - cannot be drifting characters in floating-point picture specifications.

Figure 12 gives examples of the use of numeric signs and the currency symbol as picture characters. In the figure, the letter b indicates a blank character.

### Credit, Debit, and Overpunched-Sign Characters

The character pairs CR (credit) and DB (debit) specify the signs of fixed-point numeric character data items and usually appear in business report forms.

Any of the picture characters T or R specifies an overpunched sign in the associated digit position of a fixed-point numeric-character data item. 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. The overpunch character can appear only in the last digit position within a field.

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 | Z99R | 102J |
| FIXED(4) | 1021 | 999T | 102A |

[1] The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure 13. Examples of CR, DB, T, and R Picture Characters

R specifies that the associated position, on input, will contain a digit overpunched with - if the value is smaller than zero; 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 smaller than zero.

Note: You cannot use the picture characters CR, DB, T, and R with any other sign characters in the same field.

Figure 13 gives examples of the CR, DB, and overpunch characters. In the figure, the letter b indicates a blank character.

### The Exponent Specifier E

The picture character E delimits 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.

E specifies that the associated position contains the letter E, which indicates the beginning of the exponent field.

The value of the exponent is adjusted (that is, it is varied) 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 (see the first two examples of Figure 14).

Note: Drifting and zero-suppression characters are not allowed in floating-point pictures. Exponent-field pictures are restricted to the only format ES99.

Figure 14 gives examples of the use of exponent delimiters.

| Source Attributes | Source Data (in constant form) | Picture Specification | Character-String Value[1] |
|---|---|---|---|
| FLOAT(5) | .12345E06 | 9V.99999ES99 | 1.23450E+05 |
| FLOAT(5) | .12345E-06 | 9V.99999ES99 | 1.23450E-07 |
| FLOAT(5) | -123.45E12 | S999V.99ES99 | -123.45E+12 |
| FLOAT(5) | 001.23E04 | S99V.99ES99 | +12.30E+03 |

[1] The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure 14. Examples of Floating-Point Picture Specifications

## Edit-Directed Format Items

This section contains a description of each of the edit-directed format items that can appear in the format list of a GET, PUT or FORMAT statement.

There are three categories of format items:

- data format items,
- control format items, and
- remote format item.

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.

### 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. 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, whether as characters or as arithmetic values.

*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 the field into which the associated data item in character form is to be placed and describes the format that the value is to take.

Leading blanks are not inserted automatically to separate data items in the output stream. Character-string data is left-adjusted in the field whose width is specified. Arithmetic data is right-adjusted. Leading blanks will not appear in the stream 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 character-string items.

Figure 15 shows all data format items and their formats.

| Category | Data Format Item |
|---|---|
| Fixed-point | F(w,[d,[p]]) |
| Floating-point | E(w,d[,s]) |
| Character-string | A[(w)] |

Figure 15. Data Format Items

### Control Format Items

The control-format items apply to input and output files. They specify formatting of the data items coming from or going to the external medium.

Figure 16 shows all control format items and their formats.

A control format item has no effect unless it is encountered before the data list is exhausted.

The PAGE and SKIP format items have the same effect as the corresponding options of the PUT statement, except that the format items are executed only when they are

| Category | Control Format Item | |
|---|---|---|
| | for PRINT files | for non-PRINT files |
| Paging | PAGE | |
| Line skipping | SKIP [(w)] | |
| Record skipping | | SKIP [(w)] |
| Spacing | X(w) | X(w) |

Figure 16. Control Format Items

encountered in the format list, while the options of the PUT statement are executed before any data is transmitted.

## Remote Format Item

The remote format item specifies the label of a FORMAT statement: This statement contains a format list which replaces the remote format item in the GET or PUT statement.

The remote format item is:

```
R(statement-label-designator)
```

The statement-label designator is a label constant or an unsubscripted element label variable.

## Alphabetic List of Format Items

### The A Format Item

The A format item is:

```
For input:  A (w)
For output:  A [(w)]
```

where w is the number of characters to be transmitted.

The character-string format item describes the external representation of a string of characters. You must use it only for character strings. Character strings cannot be transmitted by any other format item. No conversion is performed.

General Rules:

1. The letter w must be a decimal integer constant, unsigned and greater than zero, but less than 256. It specifies the number of characters to be transmitted.

2. On input, the specified number of characters is obtained from the data stream and assigned to the associated variable in the data list. For input, you must always specify w. If apostrophes appear in the stream, they are treated as characters in the string.

3. On output, w need not be specified; in this case, the length of the associated string is used, and the data item completely fills the field. Enclosing apostrophes are not inserted.

### The E Format Item

The E format item is:

```
For input:  E (w,d)
For output:  E (w,d[,s])
```

where w is the field-width, d the number of fractional digits, and s the number of significant digits.

The floating-point format item E describes the external representation of decimal arithmetic data in floating-point format.

General Rules:

1. The letters w, d, and s must be unsigned decimal integer constants. The integer w specifies the total number of characters in the field. It must be less than 33. The integer d specifies the number of fractional digits, that is, the number of digits following the decimal point in the mantissa: s specifies the number of digits that must appear in the mantissa.

2. On input, the data item in the 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 the number is:

   $$[\pm]\ \text{mantissa} \left\{\begin{matrix}[E]\{\pm\}\\ E[\pm]\end{matrix}\right\} \text{exponent}$$

   The mantissa must be a fixed-point decimal constant.

   The number can appear anywhere in the specified field; blanks may appear before and after the number in the field. If the entire field is blank, the CONVERSION condition is raised. When no decimal point appears, the number of fractional digits (d) specifies the number of character positions to the right of the assumed decimal point of the mantissa. If a decimal point actually does appear in the data, it overrides d.

   The value expressed by w includes trailing blanks, the exponent position(s), the position for the optional plus or minus sign, the position for the optional letter E, and the position for the optional decimal point in the mantissa.

The exponent must be a decimal integer constant that does not exceed two significant digits. Leading zeros are allowed. Whenever the exponent and a preceding sign or the letter E are omitted, a zero exponent is assumed.

3. On output, the internal data is converted, if necessary, to floating-point numeric-character representation, and the external data item in the specified field has the following general form:

```
[-]{s-d digits}.{d digits}E{+|-}exponent
```

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 non-zero (provided that the mantissa is not zero, of course).

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 the Model 20 PL/I Compiler, the field width for negative or non-negative values of the data item must be greater than or equal to 6 plus the number of significant digits (although the sign of a positive digit is not written, it must be accounted for). However, if the number of fractional digits is zero, the decimal point is not written, and the above figure for the field width is reduced by 1.

When the internal data is converted to the output format, it is rounded as follows: if truncation causes a digit to be lost on the right and this digit is greater than or equal to 5, then 1 is added to the digit to the left of the lost digit.

Example:

```
DCL (A,B,C) FLOAT(15);
  A = -1234567
  B = -1.2345678E-10
  C =  1.2345678E+1

PUT FILE (OUT) EDIT (A,B,C)
   (E15,6),E(15,6,8),E(15,8)
```

When A,B and C are pointed, they will look as follows:

```
bb-1.234567E+06b-12.345678E-11
    b1.23456780E+01
```

## The F Format Item

The F format item is:

```
F (w[,d[,p]])
```

where w is the field width, d the number of fractional digits, and p the scale factor.

The fixed-point format item describes the external representation of a decimal arithmetic data item in fixed-point format.

General Rules:

1. The letters w, d, and p must be decimal integer constants. Only p can be signed; the others must be unsigned; w must be less than 33 and must, for output, account for the sign, even if it is blank.

2. On input, the data item in the 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. If the entire field is blank, it is interpreted as zero.

   If d is not specified, the number of fractional digits is assumed to be zero.

   If p is not specified and no decimal point appears in the field, d specifies the number of fractional digits, that is, the number of digits to the right of the assumed decimal point. If a decimal point actually does appear in the data, it overrides the specification d.

   If p is specified, it effectively multiplies the value of the data item in the stream by 10 raised to the power of the value of p. Thus, if p is positive, the number is treated as though the decimal point appeared p positions to the right of its given position. If p is negative, the number is treated as though the decimal point appeared p positions 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 specification 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, and the external data is the character representation of a decimal fixed-point number, right-adjusted in the specified field.

If only w is specified in the format item, only the integer portion of the number is written; no decimal point appears.

If both w and d are specified, both the integer and fractional portions of the number are written, and if d is greater than zero a decimal point is inserted before the leftmost d digits. Trailing zeros are supplied when the actual number of fractional digits is less than d (the value d must be less than the field width (w)). Suppression of leading zeros is applied to all digit positions (except the first) to the left of the decimal point.

When the internal data is converted to the output format, it is rounded as follows: if truncation causes a digit to be lost on the right and this digit is greater than or equal to 5, then 1 is added to the digit to the left of the lost digit.

When p is specified, the value of the associated element in the data list is effectively multiplied by 10 raised to the power of p before it is converted to its external character representation. When the number of fractional digits is zero, only the integer portion of the number is used.

If the value of the fixed-point number is less than zero, the external character representation is preceded by a minus sign. If it is greater than or equal to zero, a blank appears. Therefore, for all values of the fixed-point number, w must account for both the sign and a possible decimal point (the decimal point will not appear if there are no fractional digits).

### The PAGE Format Item

The PAGE format item is:

    PAGE

The paging format item PAGE specifies that printing is to continue on a new page.

General Rules:

The PAGE format item implies that printing is to continue on line 1 of the new page.

### The R Format Item

The R format item is:

    R (statement-label-designator)

The remote format item allows the use of format items specified in a FORMAT statement.

General Rules:

1. The statement-label designator is a label constant or a label variable whose value is the statement label of a FORMAT statement. The FORMAT statement includes a format list that is taken to replace the format item. The statement-label designator cannot be subscripted.

2. The R format item and the specified FORMAT statement must be contained within the same procedure.

3. A FORMAT statement must not contain an R format item.

### The SKIP Format Item

The SKIP format item is:

    SKIP [ (w) ]

where w specifies that writing or reading is to continue at the beginning of the wth line or record (for non-PRINT files) following the one just being written or read.

The skipping format item SKIP can be used with both PRINT and non-PRINT files, in GET as well as PUT statements. When used in a PUT statement for a PRINT file, it specifies that printing is to continue on a new line. When used in a GET statement, it specifies that a new record is to be read.

General Rules:

1. The letter w must be an unsigned decimal integer constant between 0 and 3 for PRINT files and 1 and 3 for non-PRINT files (SKIP(0) is not allowed for non-PRINT files). If w is not specified, 1 is assumed.

2. If w is greater than or equal to 1, w-1 blank lines or records will be inserted for output, or w-1 complete records will be skipped for input.

3. If SKIP(0) is specified for PRINT files, the effect is that of carriage return without line skipping. Characters previously written will be overprinted by the new characters. For

example, underscoring can be done in this form.

4. If the SKIP format item is not specified at the end of a line or record, SKIP(1) is assumed, that is, printing continues at the beginning of the following line (single spacing), or reading continues at the beginning of the following record.

5. If, for PRINT files, the specified line lies beyond the limit set by default (which is 60) or by the PAGESIZE option of the OPEN statement, the ENDPAGE condition is raised.

### The X Format Item

The X format item is:

```
X(w)
```

where _w_ is the _field width_.

The spacing format item controls the relative spacing in the data stream. It can be used in GET as well as PUT statements.

General Rules:

1. The letter _w_ must be an unsigned decimal integer constant less than 256.

2. On _input_, _w_ specifies the number of characters to be spaced over in the data stream, i.e., not to be transmitted to the program.

3. On _output_, the specified number of blank characters is inserted in the data stream.

# Built-In Functions and the Pseudo-Variable SUBSTR

This section contains a description of the built-in functions and the pseudo-variable SUBSTR available in Model 20 PL/I. These features are discussed in the following order:

1. Computational Built-In Functions

   a) String-handling built-in functions

   b) Arithmetic built-in functions

   c) Mathematical built-in functions

2. Other Built-In Functions

3. The Pseudo-Variable SUBSTR

The *computational built-in functions*, as shown above, provide string handling, arithmetic operations (absolute value, truncation, etc.), mathematical operations (trigonometric functions, square root, etc.).

The computational built-in functions are:

*String Handling*:
CHAR
HIGH
LOW
SUBSTR

*Arithmetic*:
ABS
CEIL
FLOOR
MAX
MIN
ROUND
TRUNC

*Mathematical*:
ATAN
COS
EXP
LOG
SIN
SQRT
TAN
TANH

*Other built-in functions* are:
ADDR
DATE

The section on the *pseudo-variable SUBSTR* gives a short discussion of the pseudo-variable. You find a more complete description in the discussion of the corresponding built-in function.

The built-in functions and the pseudo-variable are presented in alphabetical order under their proper headings.

## Computational Built-In Functions

### STRING HANDLING BUILT-IN FUNCTIONS

You may use the functions described in this section for manipulating character strings. The arguments you may use must be *expressions*.

### CHAR Built-In Function

*Definition*: CHAR is used to control the size of a character-string expression. In Model 20 PL/I, it is mainly used to convert a picture variable to a character string.

*Reference*: CHAR (expression [,size])

*Arguments*: The argument *expression* may be:

a. a character-string expression, or

b. a numeric-character expression

The argument *size*, when specified, must be a *decimal integer constant* giving the length of the result. If size is not specified, the length resulting from the character-string expression or numeric-character expression is taken.

*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.

*Example*:

```
   DCL X PIC'**9V.99';
          .
          .
          .
   PUT FILE(OUT)EDIT
      (CHAR(X))(A);
```

### HIGH Built-In Function

*Definition*: HIGH forms a character string of a specified length. Each character in the constructed string is the highest character in the collating sequence. For System/360 implementations, this character is stored as *hexadecimal FF*.

*Reference*: HIGH (i)

*Argument*: The argument *i* must be an unsigned 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 stored as hexadecimal FF.

Figure 17 illustrates the use of the built-in function HIGH.

Two sequential files with ascending keys shall be merged. When the first file is exhausted its key is set 'high'. Thus, only the records of the non-exhausted file will be copied.

```
MERGE: PROCEDURE OPTIONS(MAIN);
       DCL 1 S1 BASED (P1),
                2 KEY CHAR(5),....
           1 S2 BASED (P2),
                2 KEY CHAR(5),....
           1 S3 BASED(P),
                2 KEY........
           (P,P1,P2)POINTER,
           LBL LABEL INIT(START),
           F1 FILE SEQUENTIAL INPUT ....
           F2 FILE SEQUENTIAL INPUT ....
           FO FILE SEQUENTIAL OUTPUT ....
       /*FILE ACTIVATION*/
       OPEN FILE(F1),FILE(F2),FILE(FO);
       ON ENDFILE(F1) GO TO EOF1;
       ON ENDFILE(F2) GO TO EOF2;
READ1: READ FILE(F1) SET(P1); GO TO LBL;
START: LBL = COMP;
READ2: READ FILE(F2) SET(P2);
COMP : IF S1.KEY>=S2.KEY THEN P=ADDR(S2);
       ELSE P=ADDR(S1);
       WRITE FILE(FO) FROM(S3);
       IF P=ADDR(S2) THEN GO TO READ2;
       GO TO READ1;
EOF1 : IF S2.KEY=HIGH(5) THEN GO TO FINIS;
          S1.KEY=HIGH(5); GO TO COMP;
EOF2 : IF S1.KEY=HIGH(5) THEN GO TO FINIS;
          S2.KEY=HIGH(5); GO TO COMP;
FINIS: END;
```

Figure 17. Example for the Usage of a Built-in Function (HIGH)

### Low Built-In Function

*Definition*: LOW forms a character string of specified length from the lowest character in the collating sequence which, for System/360 implementations, is *hexadecimal 00*. Each character in the constructed string will be stored as hexadecimal 00.

*Reference*: LOW (i)

*Argument*: The argument *i* must be an unsigned decimal integer constant specifying the length of the string to be 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 which, for System/360 implementations, is hexadecimal 00.

*Example*: LOW(3) has the value X'000000'

### SUBSTR Built-In Function

*Definition*: SUBSTR extracts a substring of 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. This argument can be:

- a character string expression, or
- a numeric-character expression

The argument *i* represents the starting point of the substring relative to the beginning of the specified string, and the argument *j* represents the length of the substring. Argument *i* must be an expression that allows conversion to an integer; *j* must be a decimal integer constant.

Assuming that the length of *string* is *k*, the 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 1.
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*. Note that condition 1 is checked by the compiler; conditions 2 and 3 are not.

*Result*: The value returned by this function is a substring of length *j* beginning with the *i*th character of *string*.

*Example*: If AAA is a character string of length 8, the statements:

```
DCL AAA CHAR(8) INIT('ABCDEFGH')
              .
              .
              .
ITEM = SUBSTR(AAA, 3,5);
```

will cause a 5-character substring to be extracted from AAA. The extracted string is then returned to the point of invocation, after which it is assigned to ITEM (assuming ITEM is a character-string variable). It will have the following form:

```
CDEFG
```

## ARITHMETIC BUILT-IN FUNCTIONS

All values returned by the 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 it is converted to coded arithmetic before the function is invoked. Note, therefore, that in the function descriptions below, a reference to an argument always means the converted argument, if conversion was necessary.

The argument of an arithmetic built-in function may only be an expression. Unless specifically stated otherwise, the scale and precision of the returned value are determined according to the conversion rules for expression operands as given in Part I, under Expressions.

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 = 15 for FIXED and FLOAT DECIMAL values.

### ABS Built-In Function

Definition: The absolute value of a number is the number with the sign removed. Thus, the absolute values of 1.5 and -1.5 are the same, namely, 1.5.

ABS finds the absolute value of a given quantity and returns it to the point of invocation.

Reference: ABS (x)

Argument: x represents the value whose absolute value is to be found.

Result: The value returned by this function is the absolute value of x. The scale and precision are the same as those of x.

Example: To get the absolute value of a variable A, write

```
ABS(A)
```

Since the built-in function SQRT allows only positive arguments, it is advisable to write:

```
SQRT (ABS(A))
```

if you are not sure that A is positive.

### CEIL Built-In Function

Definition: CEIL determines and returns the next integer above x unless x is an integer, in which case it returns the value of x itself to the point of invocation.

Reference: CEIL (x)

Argument: x represents the value whose ceiling value is to be found.

Result: The value returned by this function is the smallest integer that is greater than or equal to x. The scale 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)
```

Examples:

```
CEIL (12.345) = 13
CEIL (345.99) = 346
CEIL (43.001) = 44
CEIL (-2.4)   = -2

CEIL (0056.34E02)   = 0056.34E02
CEIL (0012.37E-03)  = 1000.00E-03
CEIL (000.01E-02)   = 100.00E-02
CEIL (-00.1E00)     = 00.0E00
CEIL (000.01E-04)   = 100.00E-2
```

### FLOOR Built-In Function

Definition: FLOOR determines and returns the next integer below x unless x is an integer, in which case it returns the value of x itself.

Reference: FLOOR (x)

Argument: x represents the value whose floor value is to be found.

Result: The value returned by this function is the largest integer that does not exceed x. The scale 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)
```

Examples:

```
FLOOR (12.345) = 12
FLOOR (345.99) = 345
FLOOR (43.001) = 43
FLOOR (-2.4)   = -3

FLOOR (0056.34E20) = 0056.34E02
FLOOR (0012.37E-03 = 0000.00E00
FLOOR (000.01E-02) = 000.00E00
FLOOR (-000.1E0)   = -001.0E00
```

## MAX Built-In Function

Definition: MAX finds the expression with the highest value from a given set of two or more expressions and returns its value to the point of invocation.

Reference: MAX (x1, x2, ..., xn)

Arguments: Two or more arguments must be given. They must have identical scale and precision.

Result: The value returned by MAX is the value of the maximum-valued argument. The scale and precision is the same as of the arguments.

Example: Assume the following parameter list specified for MAX:

```
MAX (003.200, 042.356, NUMBER)
```

with NUMBER having the value 102.000. Then MAX returns the value 102.000.

## MIN Built-In Function

Definition: MIN finds the lowest-valued expression from a given set of two or more expressions and returns the value of this expression to the point of invocation.

Reference: MIN (x1, x2, ..., xn)

Arguments: Two or more arguments must be given. They must have identical scale and precision.

Result: The value returned by MIN is the value of the lowest-valued argument. The scale and precision of the result is the same as that of the argument.

## ROUND Built-In Function

Definition: ROUND rounds a given value at a specified digit position to the right of the decimal point and returns the rounded value to the point of invocation.

Reference: ROUND (expression,n)

Arguments: *expression* must be of fixed-decimal type. It is an expression representing the value to be rounded: *n* must be an unsigned decimal integer constant. It specifies the fractional digit position (to the right of the decimal point) at which the value of *expression* is to be rounded.

Result: *expression* is rounded at digit position *n* to the right of the decimal point. Spare digit positions are padded with zeros. The precision of the result is:

```
(MIN(p+1,N),q)
```

Note that if *expression* is negative, its *absolute value* is rounded. Its sign remains unchanged.

Example: If R is a fixed-point decimal variable of precision (7,5) containing the value 36.24976, and X, Y and Z are fixed-point decimal variables of precision (6,4), then after execution of the following statements:

```
X = R;
Y = ROUND (R,3);
Z = ROUND (R,4);
```

the value of X is 36.2497 (normal truncation due to precision deviation), the value of Y is 36.2500, and the value of Z is 36.2498.

## TRUNC 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 (ceiling): if the value is positive or equal to zero, TRUNC returns the largest integer that does not exceed that value (floor).

Reference: TRUNC (x)

Argument: *x* represents the value to be truncated.

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 scale of the result is the same as that of *x*. If *x* is floating-point, the precision remains the same. If *x* is a fixed-point value of precision (p,q), the precision of the value is:

```
(MIN(N,MAX(p-q+1)),0)
```

## MATHEMATICAL BUILT-IN FUNCTIONS

All arguments passed 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. Note, therefore, that in the function descriptions below, a reference to an argument always means the converted argument, if conversion was necessary.

An argument to a mathematical built-in function must be an expression. All of the mathematical built-in functions return coded arithmetic floating-point values. The precision of these values is always the same as those of the arguments.

Figure cc provides a survey of the mathematical built-in functions available in Model 20 PL/I.

### ATAN 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.

If y is specified, then the value whose arctangent is to be found is taken to be the expression x/y. In this case, x and y must not be equal to 0 both at the same time.

Result: When x alone is specified, the value returned by ATAN is the arctangent of x expressed in radians, where:

-pi/2<ATAN(x)<pi/2

If both x and y are specified, the possible values returned by this function are defined as follows:

1. For y > 0 and any x, the value is ATAN (x/y).

2. If x > 0 and y = 0, the value is (pi/2).

3. If x ≥ 0 and y < 0, the value is (pi+ ATAN (x/y)).

4. If x < 0 and y = 0, the value is (-pi/2).

5. If x < 0 and y < 0, the value is (-pi+ATAN(x/y)).

6. If x = 0 and y = 0, the ERROR condition is raised.

### COS 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 must be expressed in radians.

Result: The value returned by this function is the cosine of x.

### EXP 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. It must not be greater than 112.8.

Result: The value returned by this function is e raised to the power of x.

### LOG 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; it must not be less than or equal to 0.

Result: The value returned by this function is the natural logarithm of x.

### SIN 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.

### SQRT 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; it must not be less than 0.

Result: The value returned by this function is the positive square root of x.

### TAN 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.

### TANH 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 18 summarizes the mathematical built-in functions. In using it, you should be aware of the following:

1. All arguments must be coded arithmetic and floating-point scale, or such that they can be converted to coded arithmetic floating-point.
2. The value returned by each function is always floating-point.
3. The error conditions are those defined by the PL/I language. Additional error conditions detected by the Model 20 PL/I compiler can be found in Part III, under Practical Considerations Regarding Program Execution.

| Function Reference | Value Returned | Error Conditions |
|---|---|---|
| ATAN(x) | arctan(x) in radians<br>-(pi/2)<ATAN(x)<(pi/2) | - |
| ATAN(x,y) | see function description | error if<br>x=0 and y=0 |
| COS(x)<br>x in radians | cosine(x) | - |
| EXP(x) | $e^x$ | error if x>112.8 |
| LOG(x) | $\log_e(x)$ | error if x≤0 |
| SIN(x)<br>x in radians | sine(x) | - |
| SQRT(x) | $\sqrt{x}$ | error if x<0 |
| TAN(x)<br>x in radians | tangent(x) | - |
| TANH(x) | tanh(x) | - |

Figure 18. Mathematical Built-In Functions

## Other Built-In Functions

### ADDR Built-In Function

*Definition*: ADDR finds the location in main storage which has been allocated to a given variable and returns a pointer value to the point of invocation. This pointer value identifies the location allocated to the variable.

*Reference*: ADDR (x)

*Argument*: The argument, *x* is the variable whose location is to be found. It can be an element variable, an array variable, a structure variable, an element of an array, or an element of a structure. It can be of any data type and storage class.

*Result*: ADDR returns a pointer value identifying the main storage location allocated to *x*. 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 contains no value, the value returned by ADDR is undefined. For an example of the ADDR function refer to the example illustrating the HIGH built-in function in this section.

### DATE Built-In Function

*Definition*: DATE returns the current data to the point of invocation.

*Reference*: DATE

*Argument*: 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<br>
*mm* is the current month<br>
*dd* is the current day

*Example*: If the current date is February 29, 1970, execution of the statement

```
X = DATE;
```

will cause the character string '700229' to be returned to the point of invocation.

*Note*: If the DATE built-in function is used, DATE has to be declared with the BUILTIN attribute in a DECLARE statement.

## The Pseudo-Variable SUBSTR

*Reference*: SUBSTR (string,i,j)

*Description*: SUBSTR represents a substring of *string*. The value being assigned to SUBSTR is assigned to the substring of *string*, starting at the *i*th position of *string* and extending over a length of *j* positions as defined for the built-in function SUBSTR. The remainder of *string* remains unchanged.

*Example*:

```
DCL NAME CHAR (10)
         INIT 'JOHN SMITH';
SUBSTR (NAME,2,6) = 'ACK KE';
```

After execution of the assignment statement, NAME will contain the character string 'JACK KEITH'.

# ON-Conditions

The ON-conditions are those exceptional conditions for which you can make an action specification by means of an ON-statement. If a condition is enabled, the occurrence of the condition will result in an interruption of the program and 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 as given in that statement is either SYSTEM, in which case the standard system action for that condition is taken, or an ON-unit, in which case you have supplied your own action to be taken for that condition (i.e., either a null statement or a GO TO statement).

If a condition is not enabled (i.e., if it has been disabled), and the condition occurs, an interrupt will not take place, and errors may result.

ON-conditions are always enabled unless they have been explicitly disabled by condition prefixes.

Some of the ON-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 (see Part I, Exceptional Condition Handling for a discussion of the scope of condition prefixes).

The following conditions are always enabled and remain so for the duration of the program:

ENDFILE
ENDPAGE
ERROR
KEY
RECORD
TRANSMIT

## Groups of ON-Conditions

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 arise.

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 an ON-unit to handle that interrupt has not been specified.

5. Normal return -- the point to which control is returned as a result of a null ON-unit. A GO TO statement ON-unit is an abnormal ON-unit termination. Note that the conditions ENDFILE, KEY, and CONVERSION cannot have the null statement associated with them and, therefore, a normal return can never be made for these conditions.

The conditions are grouped as follows:

1. Computational conditions -- those conditions associated with data handling, expression evaluation, and computation. They are:

   CONVERSION
   FIXEDOVERFLOW
   OVERFLOW
   UNDERFLOW
   ZERODIVIDE

2. Input/Output conditions -- those conditions associated with data transmission. They are:

   ENDFILE
   ENDPAGE
   KEY
   RECORD
   TRANSMIT

3. System-action condition -- the condition (i.e., ERROR) that provides facilities to extend the standard system action that is taken after the occurrence of a condition.

## Computational Conditions

### The CONVERSION Condition

Description: The CONVERSION condition occurs whenever an illegal conversion is attempted. This attempt may be made internally or during an input/output operation.

All conversions of character-string data are carried out character-by-character in a left-to-right sequence and the condition occurs for the first illegal character. When such a character is encountered, an interrupt occurs (provided, of course, that CONVERSION has not been disabled by means of the condition prefix NOCONVERSION) and the current action specification for the condition is executed.

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.

Normal Return: A null ON-unit cannot be specified for this condition.

Example:

On input, an attempt is made to convert data that has been mispunched as

```
10R-01
```

to the floating-point format specified in the following GET statement:

```
GET FILE (CARD) EDIT (Y)(E(8,2));
```

The CONVERSION condition is raised, since 10R-01 does not have a valid floating-point format.

### 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.

FIXEDOVERFLOW can be disabled by the condition prefix NOFIXEDOVERFLOW.

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.

Normal Return: If a null ON-unit is specified 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 Model 20 PL/I, a floating-point number or intermediate result must be less than $10^{49}$. OVERFLOW can be disabled by the condition prefix NOOVERFLOW.

Result: When OVERFLOW has occurred, the value in the affected floating-point field is undefined.

Standard System Action: In the absence of an ON-unit, the system prints a message and raises the ERROR condition.

Normal Return: If a null ON-unit is specified for this condition, control returns to the point immediately following the point of interrupt.

### The UNDERFLOW Condition

Description: The UNDERFLOW condition occurs when the absolute value of a floating-point number is smaller than the permitted minimum. For Model 20 PL/I, the absolute value of a floating-point value must not bè less than $10^{-51}$, except that it may be zero.

UNDERFLOW does not occur when equal numbers are subtracted (often called significance error).

UNDERFLOW can be disabled by the condition prefix NOUNDERFLOW.

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.

Normal Return: If a null ON-unit is specified 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.

ZERODIVIDE can be disabled by the condition prefix specifying NOZERODIVIDE. However, in this case, division by zero results in a Model 20 hardware stop.

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.

Normal_Return: If a null ON-unit is specified 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-statements.

### The_ENDFILE_Condition

General_Format: ENDFILE (filename)

Description: The ENDFILE condition can be raised during a GET or READ operation; it is caused by an attempt to read past the end-of-file record of the file named in the GET or READ statement.

After ENDFILE has been raised, the file should be closed.

Standard_System_Action: In the absence of an ON-unit, the system prints a message and raises the ERROR condition.

Normal_Return: A null ON-unit cannot be specified for this condition.

### The_ENDPAGE_Condition

General_Format: ENDPAGE (filename)

The filename 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 maximum default page length (60 lines) or the page length specified in the PAGESIZE option of the OPEN statement for the file. The attempt to exceed the limit may be made during data transmission (including any format items specified in the PUT statement), or by the SKIP option. ENDPAGE is raised only once per page.

When ENDPAGE is raised, the current line number is one greater than that specified by the PAGESIZE option (or the default) so that it is possible to continue writing on the same page.

After ENDPAGE has been raised, a new page can be started by executing a PAGE option or a PAGE format item. If a new page is not started, the current line number may increase indefinitely.

Standard_System_Action: In the absence of an ON-unit, the system starts a new page.

Normal_Return: If ENDPAGE is raised during data transmission, then, on return from a null ON-unit, the data is written on the current line. If ENDPAGE results from a SKIP option, then, on return from a null ON-unit, the action specified by SKIP is ignored.

### The_KEY_Condition

General_Format: KEY (filename)

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 for a READ or REWRITE statement. In this case, the contents of the variable into which data is to be read is unpredictable.

2. An attempt is made to add a duplicate key by a WRITE statement.

3. The keys of a KEYED SEQUENTIAL OUTPUT file are not in ascending order.

4. No space is available to add the keyed record.

Standard_System_Action: In the absence of an ON-unit, the system prints a message and raises the ERROR condition.

Normal_Return: A null ON-unit cannot be specified for this condition.

### The_RECORD_Condition

General_Format: RECORD (filename)

Description: The RECORD condition can be raised only during a READ, WRITE, REWRITE, or LOCATE operation. It is raised in either of the following cases:

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.

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. Note that an ON-unit can only be specified for tape_input_files.

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 execution of a null ON-unit, execution continues with the statement immediately following the READ statement for which RECORD occurred. An ON-unit can only be specified for tape input files; in all other cases, the standard system action is executed.

The_TRANSMIT_Condition

General_Format: TRANSMIT (filename)

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 transmitted 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.

Standard_System_Action: In the absence of an ON-unit, the system prints a message and raises the ERROR condition.

Normal_Return: Upon execution of a null ON-unit, processing continues with the next data item for STREAM I/O, or with the next statement for RECORD I/O.

## System Action Condition

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 (for example, taking the SQRT of a negative value).

Standard_System_Action: In the absence of an ON-unit, a message is printed and control is returned to the DPS Monitor program.

Normal_Return: Upon execution of a null ON-unit, control is returned to the DPS Monitor program.

# Attributes

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 scale and precision attributes must be associated with the name. Associated attributes are those which you specifiy in a DECLARE statement or which are assumed by default.

This section discusses the different attributes. The attributes are grouped by function. Detailed discussions follow, in alphabetic order, showing the rules, default, and format for each attribute.

### SPECIFICATION OF ATTRIBUTES

Attributes specified in a DECLARE statement must be separated by blanks. However, in case of dimension, length, and precision attributes, blanks are not necessary. Except for the dimension, length, FILE, and precision attributes, they may appear in any order. The dimension attribute must immediately follow the array name; the length attribute must follow the CHARACTER attribute, and the precision attribute must follow the base or scale attribute; the FILE attribute must be the first attribute in every file declaration. 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. For example:

```
DCL MASTER FILE RECORD INPUT
            SEQUENTIAL other-attributes,
    A(10,10) FIXED DECIMAL,
    B DECIMAL FIXED (4,2),
    C CHAR(5) INIT('JOHN ');
```

### FACTORING OF ATTRIBUTES

Factoring is achieved by enclosing the names in parentheses, and following this by the set of attributes which apply. Names within the parenthesized list are separated by commas. All factored attributes must apply to all of the names. No factored attribute can be overridden for any of the names by a separate specification, but any name within the list may be given other attributes so long as there is no conflict with the factored attributes.

Except for the *dimension, INITIAL*, and *file-description attributes*, you can factor any attributes common to several names in a declaration to eliminate repeated specification of the same attribute for many identifiers. Note, however, that in Model 20 PL/I, a pair of factorization parentheses cannot contain more than 60 declarations and that factoring can be nested to a level of eight. (See the fourth example below for an illustration of nesting.)

*Note*: If elements of structures are factored, their level numbers must also be factored by preceding the parenthesized list. (See the third example below.)

*Examples*:

```
DECLARE (A,B,C) DECIMAL FIXED (4,2);
DECLARE (Y DECIMAL(6),F CHARACTER(10))
          STATIC;
DECLARE 1 A, 2(B,C,D) FIXED DECIMAL(4,2);
DECLARE((A,B) FIXED(10), C FLOAT (5))
          EXTERNAL;
```

## Data Attributes

### PROBLEM-DATA ATTRIBUTES

Attributes for problem data are used to describe arithmetic and character-string variables. An arithmetic variable has attributes that specify the base, scale and precision of the data item. A character-string variable has attributes that identify it as a character-string variable and specify its length.

The *arithmetic-data attributes* are:

```
DECIMAL
FIXED|FLOAT
(precision)
PICTURE
```

The string-data_attributes are:

```
CHARACTER
(length)
```

You can also declare other attributes for data variables. With the DEFINED attribute you can specify that the data variable is to occupy the same main-storage area as some other data variable. The storage class and scope attributes also apply to data. With the INITIAL attribute you specify initial values for data variables.

An attribute that applies only to array variables, is the dimension attribute. It specifies the number of dimensions and the bounds of an array.

### PROGRAM-CONTROL_DATA_ATTRIBUTES

Program-control data are labels and pointers. You can use them to control the execution of your program. The associated attributes are LABEL and POINTER.

### ENTRY-NAME ATTRIBUTES

An entry name is a special type of label, namely, the label identifying a PROCEDURE statement. With the entry-name attributes you specify that the associated name is an entry name and describe features of that entry point.

The entry_name_attributes are:

```
ENTRY
RETURNS
BUILTIN
```

All entry names of procedures that are invoked within a procedure must be declared in the invoking procedure with the ENTRY or RETURNS attribute. The RETURNS attribute has to be specified for a function returning a value that does not have the default attributes. It has to appear in both the invoking and the invoked procedure (function). In the function, the attributes of the value to be returned must be specified in the RETURNS attribute in the PROCEDURE statement. The PROCEDURE statement of a function is the only place where an attribute may appear outside a DECLARE statement. The BUILTIN attribute must be specified for the DATE built-in function.

## File Description Attributes

The file-description attributes establish an identifier as a file-name and describe characteristics for that file, for example, how the data of the file is to be transmitted, what the characteristics of the file are, etc. If the same filename is declared in more than one procedure, the file attributes must not conflict. A filename must always have the EXTERNAL attribute, either explicitly or by default. For file-description attributes see also Appendix_E._File_Attributes_and_Options.

The file-description_attributes are:

```
FILE
STREAM|RECORD
INPUT|OUTPUT|UPDATE
PRINT
SEQUENTIAL|DIRECT
BACKWARDS
ENVIRONMENT (option-list)
KEYED
```

## Scope Attributes

With the scope attributes you specify whether or not a name declared in a procedure is to be known only within or also beyond the scope of that procedure.

The scope_attributes are:

```
EXTERNAL
INTERNAL
```

For a discussion of the scope of names, see Part I, Recognition_of_Names.

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

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

The INTERNAL attribute specifies that the declared name is not known in any procedure other than the one in which it is declared. INTERNAL cannot be specified for a file or entry name.

You can declare the same identifier with the INTERNAL attribute in more than one procedure without regard to whether the attributes given in one procedure are consistent with the attributes given in another procedure, since such declarations refer to different variables.

## Storage Class Attributes

The storage-class attributes are used to specify the class of storage to be allocated to a data variable.

The storage-class_attributes are:

```
   STATIC
   AUTOMATIC
   BASED (pointer-variable)
```

## Alphabethic List of Attributes

Following is a list of attributes along wiht a detailed description of each attribute. 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.

AUTOMATIC,_STATIC,_and_BASED_(Storage-Class Attributes)

You use the storage-class attributes to specify the type of main-storage allocation for data variables.

AUTOMATIC specifies that storage is to be allocated upon each entry to the procedure in which the declaration has been made (either explicitly or by default). The storage area is freed (released) upon exit from the procedure.

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

The BASED_(pointer-variable) attribute specifies a variable that is a description of data that can be applied to different locations in a storage.

General Format:

STATIC|AUTOMATIC|BASED(pointer-variable)

General Rules:

1. AUTOMATIC and BASED variables can have INTERNAL scope only. STATIC variables may have either INTERNAL or EXTERNAL scope.

2. You must not specify storage-class attributes for entry names, file names, members of structures, DEFINED data items, or parameters.

3. For a structure_variable, you can specify a storage-class attribute only for the major-structure_name. The attribute then applies to all elements of the structure.

4. The following rules govern the use of based_variables:

   a) The pointer_variable must be explicitly declared with the POINTER attribute. The pointer variable must be an element variable; it cannot be an element of a structure, and it cannot have the BASED attribute.

   b) When reference is made to a based variable, the data attributes assumed are those of the based variable, while the associated pointer variable identifies the location of data.

   c) You can use a based variable to identify and describe existing data or to obtain storage in a buffer by use of the LOCATE statement.

   d) You cannot specify the EXTERNAL attribute with a based variable, but you can use a based variable with an EXTERNAL pointer variable.

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 is specified, AUTOMATIC is assumed.

BACKWARDS_(File-Description_Attribute)

With the BACKWARDS attribute you specify 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; you may not specify it for STREAM files.

2. The BACKWARDS attribute applies to tape files only.

3. The BACKWARDS attribute cannot be specified for variable-length records.

### BASED (Storage-Class Attribute)

See AUTOMATIC.

### BUILTIN (Entry Attribute)

In Model 20 PL/I, the BUILTIN attribute must be specified with the DATE built-in function. For the other built-in functions it may be specified, but is not necessary.

General Format:

```
   BUILTIN
```

General Rule:

The BUILTIN attribute has to be specified with the DATE built-in function. It must be the only attribute specified for the built-in function.

For example:

```
   DCL DATE BUILTIN;
```

### CHAR or CHARACTER (Character-String Data Attribute)

The CHARACTER attribute (abbreviated CHAR) is used to specify character-string variables. Together with the CHARACTER attribute you have to specify the length attribute.

General Format:

$$\left\{\begin{array}{l}\text{CHARACTER}\\ \text{CHAR}\end{array}\right\}\text{(length)}$$

General Rules:

1. With the length attribute you specify the length of the declared string. It must be a decimal integer constant, unsigned and greater than zero. The maximum length specification is 255.

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

### DECIMAL (Arithmetic-Data Attribute)

With the DECIMAL attribute you specify that the base of the data item represented by an arithmetic variable is decimal. In Model 20 PL/I, arithmetic variables can only have the DECIMAL base.

General Format:

```
   DECIMAL
```

General Rule:

The DECIMAL attribute cannot be specified with the PICTURE attribute.

Assumptions:

Identifiers that are not explicitly declared (or identifiers declared only with dimension, storage class, and scope attributes) are invalid if they begin with any of the letters I to N. If they begin with any other alphabetic character, they are assumed to be arithmetic variables with the default attributes FLOAT DECIMAL (6).

### DEF or DEFINED (Data Attribute)

The DEFINED attribute (abbreviated DEF) specifies that the variable being declared is to represent part or all of the same storage area as that assigned to another variable. The DEFINED attribute can be declared for element, array, or major-structure variables.

General Format:

$$\left\{\begin{array}{l}\text{DEFINED}\\ \text{DEF}\end{array}\right\}\text{base-identifier}$$

The *base identifier* is an unsubscripted variable whose location in main storage (or part of it) is also to be represented by the variable being declared.

Rules for Defining:

1. You cannot specify INITIAL, storage-class, and scope attributes for the *defined variable*. Note that although the base identifier can have the EXTERNAL attribute, the *defined variable* always has the INTERNAL attribute and cannot be declared with any scope attribute. If the base identifier is external, it will be known in all procedures in which it has been declared external, but the name of the defined variable will not. However, the value of the defined variable will be changed if the value of the base identifier is changed in any procedure. The defined variable cannot be a minor structure or an element of a structure.

2. The *base identifier* must always be known within the block in which the defined variable is declared. The base identifier cannot have the DEFINED attribute; it cannot be a based variable, a pointer variable or a parameter.

3. The base identifier cannot be a minor structure or an element of a structure. However, it can be a major structure, and it can also be an array.

There are two types of defining: simple defining and string-overlay defining. Simple defining is given if both the defined variable and the base identifier have identical formats. String-overlay defining is given when the formats are not identical.

### Simple Defining

Simple defining means that a reference to an element of the defined variable is interpreted as a reference to the corresponding element of the base identifier.

Corresponding structures must have the same structuring. Corresponding arrays must have the same number of dimensions and bounds. The elements of the base identifier and the elements of the defined item must have the same description.

### String-Overlay Defining

String-overlay defining means that the defined variable is to occupy part or all of the storage area allocated to the base identifier. In this way, changes to the value of either variable may be reflected in the value of the other. String-overlay defining is permitted between

a) character-string variables,

b) numeric-character variables, and

c) aggregates consisting of items a and b.

The extent of the defined variable must not be larger than the extent of the base identifier. The extent is calculated by summing the lengths of the elements of a variable, e.g., all individual elements of an array.

### Dimension (Array Attribute)

With the dimension attribute you specify the number of dimensions of an array and the bound of each dimension.

General Format:

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

General Rules:

1. The number of bounds specifies the number of dimensions in an array. As shown by the general format, the maximum number of dimensions allowed in Model 20 PL/I is three.

2. Each bound must be an unsigned decimal integer constant greater than zero. This number specifies the upper bound of the corresponding dimension. The lower bound is always assumed to be 1. Therefore, this number also specifies the extent of the corresponding dimension. For example, if a bound is 8, the extent of that dimension is 1, 2, ..., 8.

3. The dimension attribute must immediately follow the array name. Intervening blanks are optional. The dimension attribute cannot be factored.

Example:

```
DECLARE ARRAY (2,5,10);
```

The numbers 2, 5, and 10 are the bounds of a three-dimensional array of 100 elements. You obtain the number of elements in an array by multiplying the bounds with each other.

### DIRECT and SEQUENTIAL (File-Description Attributes)

With the DIRECT and SEQUENTIAL attributes you 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 file. DIRECT specifies that the records are to be accessed by use of a key. Each record of a direct file must, therefore, have a key associated with it.

General Format:

```
   SEQUENTIAL|DIRECT
```

General Rules:

1. DIRECT files must also have the KEYED attribute which, if not explicitly specified, is implied by DIRECT. SEQUENTIAL files may have the KEYED attribute only if the SEQUENTIAL attribute is associated with a file of INDEXED organization.

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

Assumption:

Default is SEQUENTIAL for RECORD files.

### ENTRY Attribute

With the ENTRY attribute you specify that the associated identifier is an entry name.

General Format:

```
   ENTRY
```

General Rules:

1. The ENTRY attribute must be specified for every entry name that is referred to in a procedure, unless RETURNS is specified, which implies the ENTRY attribute.

2. The ENTRY attribute must not be specified for built-in functions.

Assumptions:

The appearance of a name as a label of a PROCEDURE statement is an explicit declaration of that name as an entry name. However, if you want to refer to that entry name from within another procedure, you have to explicitly declare that identifier with the ENTRY attribute in the invoking procedure.

### ENV or ENVIRONMENT (File-Description Attribute)

The ENVIRONMENT attribute (abbreviated ENV) is an attribute that specifies various file characteristics that are not part of the PL/I language.

General Format:

```
  {ENVIRONMENT}(options-list)
  {ENV        }
```

The option list is defined individually for each implementation of PL/I. For Model 20 PL/I, it is as follows:

```
[CONSECUTIVE]
[INDEXED    ]
{F (blocksize [,recordsize])}
{V (maxblocksize)           }
{U (maxblocksize)           }
[BUFFERS (1|2)]
 MEDIUM (symbolic-device-address,
         device-type)
[CTLASA]
[LEAVE]
[NOTAPEMK]
[NOLABEL]
[VERIFY]
[NOWRITE]
[KEYLENGTH (decimal-integer-constant)]
[EXTENTNUMBER (decimal-integer-constant)]
[OFLTRACKS (decimal-integer-constant)]
[KEYLOC (decimal-integer-constant)]
[ALTTAPE]
```

General Rules:

1. Each file declaration must include the ENVIRONMENT attribute.

2. The options must be separated by one or more blanks.

### EXT or EXTERNAL and INTERNAL (Scope Attributes)

The EXTERNAL (abbreviated EXT) and INTERNAL attributes specify the scope of a name. INTERNAL specifies that the name is to be known only in the declaring procedure. EXTERNAL specifies that the name may be known in other procedures containing a declaration of the same name with the EXTERNAL attribute.

General format:

```
   EXTERNAL|EXT|INTERNAL
```

General Rules:

1. All file and entry names must be external. They cannot be declared as internal.

2. All external names are restricted, in Model 20 PL/I, to a length of six characters.

Assumptions:

INTERNAL is assumed for variables with any storage class. EXTERNAL (abbreviated EXT) is assumed for filenames and entry names.

### FILE Attribute

With the FILE attribute you specify that the identifier being declared is a filename.

General Format:

```
   FILE
```

General Rule:

The FILE attribute must be explicitly declared for each filename. It must be the first attribute in a file declaration. File declarations cannot be factored.

### FIXED and FLOAT (Arithmetic-Data Attributes)

With the FIXED and FLOAT attributes you 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:

You cannot specify the FIXED and FLOAT attributes with the PICTURE attribute.

Assumptions:

Identifiers that are not explicitly declared (or identifiers declared only with the dimension, storage class, and scope attributes) are invalid if they begin with any of the letters I to N. If they begin with any other alphabetic character, they are assumed to be arithmetic variables with the default attributes FLOAT DECIMAL(6).

### FLOAT (Arithmetic-Data Attribute)

See FIXED.

### INIT or INITIAL (Data Attribute)

With the INITIAL attribute (abbreviated as INIT) you can specify an initial value for a variable. The initial value is assigned to the variable at the time storage is allocated for it.

General Format:

```
{INITIAL}(item[,item]...)
{INIT   }
```

General Rules:

1. You may specify the INITIAL attribute for element variables and arrays. However, you cannot specify it for arrays of the storage class AUTOMATIC.

2. The variables you can initialize may either be arithmetic, character-string or label variables.

3. INITAL values *cannot* be declared for BASED variables, DEFINED variables, structures, parameters, STATIC LABEL variables and arrays, POINTER variables, file names and entry names. In a *structure declaration*, the INITIAL attribute can only be used in the declaration of *elementary names*.

4. Each *item* in the list following the INITIAL attribute may either be an *arithmetic constant*, a *character-string constant*, or an *iteration specification*.

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

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

   Iteration factors must be unsigned decimal integer constants. 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. For example:

   ```
   DCL B (5,5) DECIMAL FIXED STATIC INIT
      ((25) 0);
   ```

   In this DECLARE statement, the 25 elements of the array B are initialized to 0. 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; for example:

   ```
   DCL C (10,10) STATIC FLOAT INIT (1,(9)
      ((10)0,1));
   ```

   In this iteration specification, the first element is to be initialized to 1, then the item list ((10) 0,1) is to be repeated nine times. In this case, all diagonal elements of the matrix C are initialized to 1, while all other elements are initialized to 0. The iteration specification must not contain more elements than the array.

6. For initialization of character-string data, if only one parenthesized integer constant 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. Consequently, in order to cause initialization of more than one element in a character string, 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)
   ```

7. The depth of nested iteration factors in an INITIAL attribute is restricted to three in Model 20 PL/I. In the example

   ```
   DCL C (10,10) STATIC FLOAT INIT
      (1(9)((10) 0,1));
   ```

   the INITIAL attribute of the two-dimensional array C has a nested depth of two.

8. The INITIAL attribute cannot be factored.

### INPUT, OUTPUT, and UPDATE (File-Description Attributes)

The INPUT, OUTPUT, and UPDATE attributes indicate the function of the file. With INPUT you specify that data is to be transmitted from the file to the program. With OUTPUT you specify that data is to be transmitted from the program to the file. A new file is created, or, with INDEXED organization, an existing file may be extended at its end. With UPDATE you specify that data can be transmitted in either direction; that is, records of the file are read, updated, and rewritten. In case of INDEXED DIRECT files, file extension is possible.

General Format:

```
   INPUT|OUTPUT|UPDATE
```

General Rules:

1. For a file with the INPUT attribute you cannot specify the PRINT attribute.
2. A file with the OUTPUT attribute cannot have the BACKWARDS attribute.
3. For a file with the UPDATE attribute, you cannot specify 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, and then optionally, REWRITE.
4. For each file you have to specify one of the above attributes, unless you have declared the file with the PRINT attribute, in which case, OUTPUT is implied.
5. You cannot specify OUTPUT for INDEXED DIRECT files.

Assumption:

The PRINT attribute implies OUTPUT.

### INTERNAL (Scope Attribute)

See EXTERNAL.

### KEYED (File-Description Attribute)

With the KEYED attribute you specify that each record in the file has a key associated with it.

General Format:

```
   KEYED
```

General Rules:

1. A KEYED file can be read sequentially only if it has the INDEXED option and SEQUENTIAL attribute.
2. You can specify the KEYED attribute only for a file residing on a direct-access storage device.

Assumption:

The DIRECT attribute implies KEYED.

### LABEL (Program-Control Data Attribute)

With the LABEL attribute you specify that the identifier being declared is a label variable and is to have statement labels as values.

General Format:

```
   LABEL
```

General Rules:

1. The variable can have as values any of the statement labels known within the scope of the variable.
2. Label variables and arrays must not be contained in structures.
3. A label variable cannot be assigned an entry name as value.

### Length (Character-String Attribute)

See CHARACTER.

### OUTPUT (File-Description Attribute)

See INPUT.

### PIC or PICTURE (Data Attribute)

You use the PICTURE (abbreviated PIC) attribute to specify the internal and external formats of numeric-character data and to define the editing of data. Numeric-character data is data having an arithmetic value but stored internally in character form. Before arithmetic operations can be performed, numeric-character data is converted to coded arithmetic.

General Format:

$$\left\{\begin{matrix}\text{PICTURE}\\ \text{PIC}\end{matrix}\right\}\text{'numeric-picture-specification'}$$

The <u>numeric-picture specification</u> is composed of a string of picture-specification characters enclosed in apostrophes (as shown in the format).

You find a detailed description as well as a table of picture-specification characters in the section Picture-Specification Characters.

POINTER (Program-Control Data Attribute)

With the POINTER attribute you specify that the identifier being declared is a pointer variable and can be used to identify data declared with the BASED storage-class attribute.

General Format:

```
POINTER
```

General Rules:

1. You can specify the POINTER attribute for an identifier only in a DECLARE statement. Thus, you have to explicitly declare a pointer variable with the POINTER attribute.

2. There are two ways of assigning a value to a pointer variable:

   a. by pointer assignment, and

   b. by the SET option in a READ or LOCATE statement.

3. Pointer data cannot appear as an operand in an arithmetic expression, nor can conversion be performed between pointer data and other data types.

4. The only operators that can be used directly with pointer data are the comparison operators = and ¬=.

5. Pointer variables cannot be used with STREAM I/O.

6. A pointer variable cannot have the BASED attribute.

7. A pointer variable cannot be an element of a structure or of an array.

8. POINTER variables must not be defined.

Precision (Arithmetic-Data Attribute)

You use the precision attribute to specify the minimum number of significant digits to be maintained for the values of variables, and to specify, for fixed-point decimal variables, the scale factor (i.e., the assumed position of the decimal point).

General Format:

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

The number of digits and scale factor are unsigned decimal integer constants. The number of digits cannot be zero. 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), or base (DECIMAL) attribute at the same factoring level.

2. The number of digits specified is 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 is a decimal integer constant that states the number of digits to the right of the decimal point. It can be used only with fixed-point variables.

4. When the scale factor is not specified for fixed-point data, it is assumed to be zero; that is, the variable is to represent integers.

5. The maximum precision allowed in Model 20 PL/I is 15. The scale factor may range from 0 to 15.

Assumptions:

The defaults in Model 20 PL/I are as follows:

| | |
|---|---|
| (5,0) | for DECIMAL FIXED |
| (6) | for DECIMAL FLOAT |

PRINT (File-Description Attribute)

With the PRINT attribute you specify that the data of the file is ultimately to be printed. The PAGE and SKIP 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 the section Statements.

General Format:

```
PRINT
```

General Rules:

1. The PRINT attribute implies the OUTPUT and STREAM attributes.

2. The PRINT attribute causes the first data byte within each record to be reserved for an ASA printer-control

character. To account for this control character any length specification of the record must be 1 plus the length of the print line. The control characters are set by the PAGE and SKIP format items or options in the PUT statement.

## RECORD and STREAM (File-Description Attributes)

With the RECORD and STREAM attributes you 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 blocks, each of which consists of one or more records in any form. Each record is transmitted as an entity directly to or from a variable or directly to or from a buffer.

General Format:

```
RECORD|STREAM
```

General Rules:

1. A file with the STREAM attribute can be referenced only in the OPEN, CLOSE, GET, and PUT statements.

2. A file with the RECORD attribute can be referenced only in the OPEN, CLOSE, READ, WRITE, REWRITE, and LOCATE statements.

3. A file with the STREAM attribute cannot have any of the following attributes: UPDATE, DIRECT, SEQUENTIAL, BACKWARDS, and KEYED.

4. A file with the RECORD attribute cannot have the PRINT attribute.

Assumptions:

Default is STREAM.

## RETURNS (Entry-Name Attribute)

You must specify the RETURNS attribute when a function invoked by a function reference returns a value that has attributes other than the default attributes FLOAT DECIMAL (6). It must appear in the invoking as well as in the invoked procedure.

General Format:

```
RETURNS (attributes-list)
```

General Rules:

1. The RETURNS attribute must be specified as follows:

   In the invoking procedure:

   ```
   DECLARE entry-name [ENTRY] RETURNS
      (attributes-list);
   ```

   In the invoked procedure:

   ```
   entry-name: PROCEDURE [(parameter-
      list)] RETURNS (attributes-list);
   ```

2. The RETURNS attribute in the DECLARE statement implies the ENTRY attribute; hence, you can omit ENTRY in the DECLARE statement of the invoking procedure.

3. The attributes in the parenthesized list following the keyword RETURNS have to be separated by blanks. The attributes specified in the attributes list following the keyword RETURNS in the invoking as well as in the invoked procedure must be identical.

4. You can specify only arithmetic, character-string, PICTURE, or POINTER attributes with the RETURNS attribute.

5. The RETURNS attribute must not be specified for built-in functions.

Assumptions:

If the RETURNS attribute is not specified and the entry name referred to in the function reference does not start with any of the letters I to N, the value returned by the invoked function is assumed to have the default attributes FLOAT DECIMAL (6). If the entry name starts with any of the letters I to N, the RETURNS attribute has to be specified.

## SEQUENTIAL (File-Description Attribute)

See DIRECT.

## STATIC (Storage-Class Attribute)

See AUTOMATIC.

## STREAM (File-Description Attribute)

See RECORD.

## UPDATE (File-Description Attribute)

See INPUT.

# Statements

This section presents the PL/I statements in alphabetical order. 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 Model 20 PL/I program.

The_Assignment_Statement

Function:

The assignment statement evaluates expressions and assigns values to elements, arrays, or structures.

General formats:

The assignment statement has five general format types. They are shown in Figure 19.

Syntax Rules:

1. In Type_1, the variable in the receiving field (i.e., to the left of the equal sign) must represent a single element whose data type is arithmetic or character-string.

2. In Type_2, the variable in the receiving field must represent an array of arithmetic or character-string elements.

   If an expression appears to the right of the equal sign, the value of the expression is assigned to each element of the array in the receiving field.

   If an array name appears to the right of the equal sign, the array in the receiving field must have the same number of dimensions and identical bounds.

3. In Type_3, the variable in the receiving field must represent a structure and each element of the structure must be an arithmetic or character-string element.

```
Type_1:__Element_Assignment

 {element-variable}    = expression;
 {pseudo-variable }

Type_2:__Array_Assignment

  array-name            ={array-name}
                         {expression};

Type_3:__Structure_Assignment

  structure-name        = structure-name;

Type_4:__Statemet-Label_Assignment_(2_forms)

  a.  element-label-variable     ={label-constant         }
                                  {element-label-variable};

                        {label-constant        }
  b.  label-array       ={element-label-variable};
                        {label-array           }

Type_5:__Pointer_Assignment

  pointer-variable               = pointer-expression;
```

Figure 19. Assignment Statement Types

The structure name to the right of the equal sign must have the same relative structuring as the structure name to the left and corresponding elementary items of both structures must have the same attributes.

This means that you can assign minor structures to major structures and vice versa, if the relative structuring is the same and the corresponding elementary items in both the minor and the major structure have the same attributes. For example:

```
DCL 1 INDEX_REC,
       2 KEY CHAR (12),
       2 OTHER CHAR (63),
    1 OUTREC,
       2 CTL_CHAR CHAR (1) INIT ('W'),
       2 RECORD,
           3 KEY CHAR(12),
           3 OTHER CHAR(63);
                .
                .
                .
RECORD = INDEX_REC;
```

In this example, the major structure INDEX_REC is assigned to the minor structure RECORD. Except for the level numbers (which need not be identical), both structures have the same structuring and the same attributes.

4. In Type 4, item b, if a label constant or an element label variable appears on the right, then the constant or the value of the variable is assigned to every element in the label array in the receiving field.

   If a label array appears on the right, then the number of dimensions and the bound of each dimension of that array must be identical to those of the label array in the receiving field.

5. In Type 5, an element pointer expression is either an element pointer variable or a function reference that returns an element pointer value.

General Rules:

1. The assignment statement is evaluated as follows:

   a. For Types 1, 4, and 5, any expressions that appear in the receiving field, either in subscripts or in pseudo-variables, are evaluated from left to right. The expression on the right of the equal sign is evaluated and its value is assigned to the variable in the receiving field.

   b. For Types 2 and 3, the assignment statement is treated as a sequence of element assignment statements involving corresponding elements of the arrays or structures concerned. For arrays, the elements are assigned in row-major order; for structures, the elements are assigned in the order in which they were declared.

   c. Except for Type 3, the value of the expression on the right is, when necessary converted to the characteristics of the variable in the receiving field according to the rules given under Expressions.

2. When a variable in the receiving field is a character string, the expression on the right is evaluated as described in general rule 1, and the assignment is performed from left to right, starting with the leftmost character position. The following may also apply:

   a. If the value of the expression is longer than the character string, the value is truncated on the right to match the length of the string.

   b. If the value of the expression is shorter than the character string, the value is extended on the right with blanks.

3. Label-array assignment as shown in Type 4 follows the rules given for array assignment in general rule 1.

Example 1:

The following example illustrates array assignment:

```
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 = B;
```

After execution, A has the value:

```
                   1  5
                   7  8
                   3  4
                   6  3
```

Consider the assignment statement:

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

After execution, A has the value:

```
0  2
2  2
2  2
2  2
```

Note that the new value for A(1,1), which is 0, is used in evaluating all other elements.

Example 2:

The following example illustrates character-string assignment:

Given:

```
A is a string whose value is 'XZ/BQ'.
B is a string whose value is 'MAFY'.
C is a string of length 3.
D is a string of 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 'MAFYb'.
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'.
```

Example 3:

The following example (where A, B, and C are element variables) illustrates element assignment:

```
A = A + SIN(B) + C ** 2;
```

Example 4:

The following example illustrates structure assignment:

```
DECLARE 1 X, 2 Y, 2 Z, 2 R, 3 S, 3 P,
        1 A, 2 B, 2 C, 2 D, 3 E, 3 Q;

X = A;
```

The assignment statement is equivalent to:

```
X.Y = A.B;
X.Z = A.C;
X.S = A.E;
X.P = A.Q;
```

Example 5:

The following example illustrates statement-label assignment:

```
     DECLARE P LABEL;
     P = A;
     GO TO P;
       .
       .
       .
  A: X = Y ** 2;
```

This set of statements causes control to transfer to A when the GO TO P statement is executed.

Example 6:

The following example illustrates conversion of data defined by a picture description assigned to floating-point data, and vice versa:

```
DCL A FLOAT, B PIC '999V99';
A = B; (B is converted from fixed-point
       numeric character to
       floating-point)
B = A; (A is converted from floating-
       point to fixed-point numeric
       character)
```

The CALL Statement

Function:

The CALL statement invokes a procedure and causes control to be transferred to the entry point of that procedure.

General Format:

```
CALL entry-name (argument [,argument]...);
```

Syntax Rules:

1. The entry name (i.e., the label of the PROCEDURE statement) represents the entry point of the procedure being invoked. The entry point of a procedure is always the first executable statement in the invoked procedure.

2. An argument can be any expression except a based variable, a built-in function name, a file name, an entry name, or a label.

   Examples of valid arguments include minor structure names, pointer expressions, character-string constants, and array names.

   Note, however, that if the attributes of an argument are not consistent with those of its corresponding parameter, no conversion is performed and an error will result.

General Rule:

See the section Arguments and Parameters, for detailed descriptions of the interaction of arguments with the parameters that represent these arguments in the invoked procedure.

### The CLOSE Statement

Function:

The CLOSE statement de-activates the named file which was activated by a previous opening. It also dissociates from the specified file PAGESIZE, if specified in the OPEN statement for that file. However, all attributes explicitly specified for that file in a DECLARE statement remain in effect.

General Format:

```
CLOSE FILE (filename) [,FILE(filename)]...;
```

General Rules:

1. The filename in the FILE(filename) specification indicates the file to be closed. Since more than one such specification can be given in a CLOSE statement, more than one file can be closed by one CLOSE statement.

2. A closed file (except an INDEXED file) may be reopened after it has been closed.

3. Closing an unopened file, or a previously closed file, has no effect.

4. If a file is not closed by a CLOSE statement, it is automatically closed at the completion of the program in which it was opened.

### The DCL or DECLARE Statement

Function:

The DECLARE (abbreviated DCL) statement is the principal method for explicitly declaring attributes of names.

General Format:

```
{DECLARE}[level] identifier [attribute]...
{DCL    }[,[level] identifier
              [attribute]...]...;
```

Syntax Rules:

1. Level is a nonzero unsigned decimal integer constant which must not exceed 255. It can appear only in structure declarations; the major structure must have the level 1. A blank space must separate a level number from the identifier following it.

2. In general, attributes must immediately follow the identifier to which they apply (as shown in the general format). However, attributes common to several name declarations can be factored to eliminate repeated specification of the same attribute for many identifiers. Factoring is achieved by enclosing the involved declarations (non-common attributes included) in parentheses and following this by the set of common attributes. In the case of factored elements of structures, the level number must precede the parenthesized list (a blank is not required between the factored level number and the left parenthesis). For example:

   ```
   DCL 1 A, 2(B,C,D) CHAR(20);
   ```

   Dimension INITIAL, and file-description attributes cannot be factored. Factoring can be nested up to a level of eight. For more examples of factoring see Factoring of Attributes in the section Attributes.

General Rules:

1. A major-structure identifier or an identifier not contained within a structure can be specified in only one DECLARE statement within a particular procedure. All attributes given explicitly for that identifier must be declared together in one DECLARE statement.

2. Attributes of external names, in separate procedures and compilations, must be consistent.

3. Labels may be prefixed to DECLARE statement, however, such labels are treated as comments and, hence, have no meaning.

4. File names must be explicitly declared, and the first attribute in a file declaration must be FILE.

5. All entry names (except built-in function names) referred to in a procedure through a CALL statement or a function reference must be explicitly declared.

6. The built-in function DATE must be explicitly declared.

### The DISPLAY Statement

Function:

The DISPLAY statement is used to display a one-byte message on the CPU console, usually to the operator. It is used, together with the REPLY specification, to allow the operator to communicate with the program. REPLY permits the operator to return a one-byte message. Execution of the program is

```
Type 1:

   DO;

Type 2:

   DO variable = expression1 [TO expression2][BY expresion3];
```

Figure 20. General Format of DO Statement

suspended until the operator has entered his reply.

General Format:

DISPLAY (expression) REPLY (character-string variable)

General Rules:

1. The DISPLAY statement displays one character in the T-R register. The E-S register contains standard information. The REPLY specification returns one character from the DPS operator communication byte.

2. REPLY must be specified.

3. The expression must result in a character string.

## 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:

There are two format types for the DO statement as shown in Figure 20.

Syntax Rules:

1. In both types, the DO statement is used in conjunction with the END statement to delimit the DO-group. Type 2 provides for iterative execution of the statements within the group, Type 1 does not.

2. The variable in Type 2 must be arithmetic and must represent a single element; it cannot be subscripted. For example:

   ```
   DO COUNTER = 1 TO 10 BY 2;
   ```

   In this example, COUNTER is the control variable, 1 is expression1, 10 is expression2, and 2 is expression3. COUNTER must either be explicitly declared to be an arithmetic variable or it is given the default attributes FLOAT DECIMAL(6).

3. Each expression in a DO statement must be an element expression.

4. If BY expression3 is omitted, and if TO expression2 is included, expression3 is assumed to be 1.

5. If TO expression2 is omitted, iterative execution continues until it is terminated 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.

General Rules:

1. In Type 1, the DO statement only delimits the start of a DO-group; it does not provide for iterative execution.

2. In Type 2, the DO statement delimits the start of a DO-group and provides for controlled iterative execution as defined by the following:

   ```
   LABEL: DO variable = expression1
             TO expression2 BY expression3;
         statement-1
             .
             .
             .
         statement-m
   LABEL1: END;
   NEXT: statement
   ```

   The above is exactly equivalent to the following expansion:

   ```
   LABEL:  e1 = expression1;
           e2 = expression2;
           e3 = expression3;
           v  = e1;
   LABEL2: IF e3 >= 0
              THEN IF v > e2
                  THEN GO TO NEXT;
                  ELSE;
              ELSE IF v < e2
   ```

```
                 THEN GO TO NEXT;
         statement-1
               .
               .
               .
         statement-m
LABEL1:  v = v + e3;
         GO TO LABEL2;
NEXT:    statement
```

In the above expansion, e1, e2, and e3 are compiler-created work areas having the attributes of expression1, expression2, and expression3, respectively; *v* is synonymous with variable.

3. *Expression1* represents the initial value of the control *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 as soon as the value of the control variable is outside the range defined by expression1 and expression2. When execution of the DO-group is terminated, control passes to the statement following the DO-group.

4. If both options, TO and BY, are present in an iterative specification, TO must occur first.

5. The control variable must not be changed during the execution of an iterative specification other than by the iterative specification itself.

6. Control may be transferred from outside the DO-group into the DO-group (i.e., to a statement in the group other than the DO statement) only if the DO-group is delimited by a DO statement of Type 1; that is, only if iterative execution is not specified. See also *The DO Statement* in Part I of the manual, under *Statement Classification*.

## The END Statement

Function:

The END statement terminates DO-groups and procedures.

General Format:

```
   END;
```

General Rules:

1. The END statement always terminates that DO-group or procedure headed by the nearest preceding DO or PROCEDURE statement for which there is no corresponding END statement.

2. If control reaches an END statement for a procedure, it is treated as a RETURN statement.

## The FORMAT Statement

Function:

With the FORMAT statement you specify a format list that is to be used in 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 the section *DataTransmission*.

2. You must specify at least one *label* for a FORMAT statement. In general, one of the labels (or a label variable having the value of one of the labels) is the statement-label designator specified in the remote format item.

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 preceded by the statement label specified with R. An R format item cannot appear in the format list of a FORMAT statement.

2. You have to specify the remote format item and the FORMAT statement in the same procedure.

3. The format list in a FORMAT statement may contain nested iteration factors. However, the depth of the nest is limited to 2.

4. If the format list of a GET or PUT statement contains a remote format item (R(statement-label-designator)) contained in an iteration nest, it must not be at a depth greater than 2.

## The GET Statement

Function:

The GET statement is a STREAM transmission statement which you can use in either of the following ways:

- It can cause the assignment of data from an external source (that is, from a file) to one or more internal receiving fields (that is, to one or more variables).

- 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 {FILE (filename)                   }
    {STRING (character-string-variable)}
        data-specification;
```

Syntax Rules:

1. The data_specification is as described in Part I, under Data_Transmission.

2. The data specification must follow the FILE or STRING option one of which must be specified.

3. The character-string_variable refers to the character string that is to provide the values to be assigned to the variables in the data specification.

4. The filename is the name of a file that will provide the values to be assigned to the variables in the data specification. It must have the STREAM and INPUT attributes.

General Rules:

1. If the FILE option refers to an unopened file, the file is opened implicitly.

2. If the STRING option has been specified, the internal GET operation always starts at the beginning of the specified string. If the number of characters in this string is less than the total number of characters required by the variables in the data specification, the ERROR condition is raised. Note that the variables in the data specification do not have to be character strings; the internal assignment is the same as the transmission from the stream to internal storage, the only difference being that the character-string variable is considered to be the input stream.

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 incactive procedure.

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 iterative execution, unless the GO TO terminates a procedure invoked from within the DO-group or unless the GO TO is an ON-unit given control from within the DO-group.

4. If a GO TO statement transfers control from within a procedure to a point not contained within that procedure, the procedure is terminated. Also, if the transfer point is contained in a procedure that did not directly activate the procedure being terminated, all intervening procedures in the activation sequence are also terminated (See Part I, Flow_of_Control_and_Storage_Allocation, for examples and details). When one or more procedures are terminated by a GOTO statement, conditions are reinstated and automatic variables are freed just as if the procedures had been terminated in the usual fashion.

5. When a GOTO 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.

6. If the GO TO statement is an ON-unit, the specified label must be unsubscripted.

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 or a DO-group. It must, however, not be a PROCEDURE, a DECLARE, or FORMAT statement.

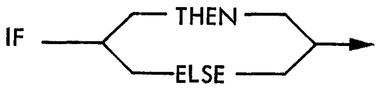
2. The IF statement itself is not terminated by a semicolon; however, each *unit* specified must be terminated by a semicolon.

3. Each unit may be labeled.

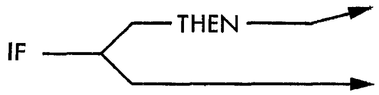
General Rules:

1. The *expression* following the keyword IF must contain one and only one comparison operation.

2. If the comparison is true, the THEN clause is executed. After execution of the THEN clause, control branches around the ELSE clause and execution continues with the next statement. If the comparison is not true, control branches around the THEN clause, and the ELSE clause is executed. Control then continues normally.

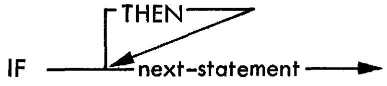


Note that the THEN clause or the ELSE clause can contain a GOTO statement or some other control statement that can cause a different transfer of control.



3. 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.

This may be illustrated by the following diagram:



In the kind of IF statement illustrated above, the alternatives are "execute the THEN clause" or "do not execute the THEN clause". In either case, the next sequential statement is executed. If the expression tested is not true, control continues through the logical flow of execution. If the expression is true, the THEN clause is executed, and control returns to exactly the same point where it would have been if the expression had not been true. In this kind of IF statement the word ELSE *must not* appear since the ELSE clause would be skipped whenever the THEN clause it executed.

4. IF statements may be nested; that is, either unit, or both, may itself be an IF statement. In Model 20 PL/I, the number of IF and DO statements in one nest must not exceed 20. However, the number of DO statements alone must not exceed 12 per nest. Any IF statement, at any level, may have a DO-group as either or both of its alternative units. An ELSE clause is always associated with the innermost unmatched IF in the same DO-group; an ELSE with a null statement may be required to specify a desired sequence of control.

*Example*:

```
IF A > B
   THEN IF C = 0 THEN X = SQRT(A-B);
                 ELSE X = 1;
```

In this example, the statement X = 1; is executed if A is greater than B and C is not equal to 0.

If the statement X = 1; is to be executed if A is not greater than B, regardless of the value of C, you may write the example in the following way:

```
IF A > B
   THEN IF C = 0 THEN X = SQRT (A-B);
                 ELSE;
   ELSE X = 1;
```

## The LOCATE Statement

Function:

The LOCATE statement is a RECORD transmission statement which you can use only for CONSECUTIVE output files. It allocates storage for a based variable in an output buffer to allow the creation of a record for that based variable. The record is created by assigning values to the based variable within the buffer. The record is not transmitted to the external medium until immediately before the next WRITE, LOCATE, or CLOSE statement (or implicit close operation) is executed for the specified file.

General Format:

```
LOCATE based-variable FILE (filename)
          SET (pointer-variable);
```

Syntax Rules:

1. The FILE and SET specifications must appear in the order shown in the general format.

2. The *based variable* must be an unsubscripted based variable that is not a minor structure or an element of a structure. It may, however, be an array name or a major-structure name.

3. The *pointer variable* must be an element pointer variable.

4. The *filename* is the name of the file that has been activated (by opening) and that will eventually receive the record. The file must have the SEQUENTIAL and OUTPUT attributes. It must, however, not be an INDEXED file.

General Rules:

1. The based variable is used to determine the length of the buffer area to be reserved. When the LOCATE statement is executed, the pointer variable in the SET specification is set to identify the location in the buffer at which the based variable is to be allocated.

2. The record identified by the based variable is written from the buffer into the output file, immediately before the next WRITE, LOCATE, or CLOSE operation (implicit or explicit) for that file. For blocked records, the record is not written until the whole block is completed.

3. The FILE specification must refer to a previously opened file.

### The Null Statement

Function:

The null statement causes no action and does not modify sequential statement execution. The null statement is represented by the semicolon (;).

General Format:

```
   ;
```

General Rule:

The null statement must not be specified for an ON statement whose condition is CONVERSION, ENDFILE, or KEY.

### The ON Statement

Function:

With the ON statement you specify what action is to be taken when an interrupt results from the occurrence of the specified exceptional condition.

General Format:

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

For a description of the ON statement and the ON-conditions that may be specified with it see the section *ON-Conditions*.

### The OPEN Statement

Function:

The OPEN statement *opens* (activates) a file by associating a file with a file declaration.

General format:

```
OPEN FILE (filename) [option]
   [,FILE (filename) [option]]...;
```

where *option* is

```
   PAGESIZE (element-expression).
```

Syntax Rules:

1. The FILE specification must appear first.

2. The *filename* is the name of the file that is to be activated. Several files can be opened by one OPEN statement.

General rules:

1. The opening of an already open file does not affect the file. In such cases, any expressions in the option, if specified, are evaluated, but they are not used.

2. The PAGESIZE option can be specified only for a file having the STREAM and PRINT attributes. The expression is evaluated and converted to an integer, which represents the maximum number of lines to a page. This integer must be greater than zero and less than 256. During subsequent transmission to the PRINT file, a new page may also be started by use of the PAGE format item or by an option in the PUT statement. For the Model 20 PL/I Compiler, the default for PAGESIZE is 60.

3. When a PRINT file is opened, a new page is started.

4. The OPEN statement is mandatory for RECORD files and optional for STREAM files.

## The PROCEDURE Statement

Function:

The PROCEDURE statement has the following functions:

a. It heads a procedure and defines its entry point.

b. It specifies the parameters, if any, for the entry point.

c. It may specify certain special characteristics that a procedure can have.

d. It specifies the attributes of the value that is returned by the procedure when it is invoked as a function.

General Format:

```
[(condition-prefix):]
entry-name: PROCEDURE
           ⎧ OPTIONS(option-list)                ⎫
           ⎨⎧[(parameter[,parameter)]...]⎫       ⎬;
           ⎩⎩[RETURNS(data-attributes)]  ⎭       ⎭
```

where, for the Model 20 PL/I Compiler, the *option list* is defined as:

```
MAIN[,ONSYSLOG]
```

Syntax Rules:

1. The *data attributes* represent the attributes of the value returned by the procedure when it is invoked as a function. Only arithmetic, string, PICTURE, and POINTER attributes are allowed.

2. OPTIONS is a special procedure specification. Two options can be specified in the OPTIONS attribute which must be specified for only one external procedure in the program:

   a. MAIN must be specified if and only if the procedure is the initial procedure of the program.

   b. ONSYSLOG specifies that diagnostic output is on SYSLOG instead of SYSLST. ONSYSLOG can only be specified together with MAIN.

3. One and only one entry name must appear in a PROCEDURE statement. The entry name must not exceed 6 characters.

4. The maximum number of parameters that can be specified for one procedure is 12.

General Rules:

1. When a 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, under *Arguments and Parameters*.

2. The MAIN option specifies that the procedure for which it is specified is the initial procedure and will be invoked by the programming system as the first step in the execution of the program. The ONSYSLOG option specifies that all output resulting from actions derived from ON conditions will be printed on the device assigned to SYSLOG. No other options are permitted. The procedure declared with the OPTIONS attribute remains active for the duration of the program and hence cannot be called by other procedures. For the Model 20 PL/I Compiler, only one procedure must have the OPTIONS(MAIN) designation.

3. The data attributes specify the attributes of the value returned by the procedure when it is invoked as a function. For details see the section *Arguments and Parameters* in Part I.

4. The entry name of a procedure is an external name and as such is restricted in Model 20 PL/I to a length of six characters.

5. The name of a procedure must not be redeclared within that procedure.

## The PUT Statement

Function:

The PUT statement is a STREAM transmission statement which can be used in either of the following ways:

1. It can cause the values in one or more main-storage locations to be transmitted to a file on an external medium. Related to this, it can control the format of a PRINT file.

2. It can cause the values in one or more main-storage locations to be assigned to an internal receiving field (represented by a character-string variable).

General Format:

with the STRING option:

```
PUT STRING (character-string-variable)
     data-specification;
```

with the FILE option:

```
PUT FILE (filename)
  ⎧EDIT data-specification       ⎫
  ⎪PAGE                          ⎪
  ⎨SKIP (expression) ]           ⎬
  ⎪PAGE EDIT data-specification  ⎪ ;
  ⎩SKIP [ (expression) ]         ⎭
     data-specification
```

Syntax Rules:

1. Either the FILE or the STRING option must be specified in the PUT statement.

2. The FILE option specifies transmission to a file on an external medium. The file name in this option is the name of the file that has been activated (by implicit or explicit opening) and that is to receive the values. This file must have the OUTPUT and STREAM attributes.

3. The STRING option specifies transmission from main-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" cannot be a pseudo-variable.

4. The data specification is as described in *Part I, Data Transmission*.

5. The FILE or STRING option must always be the first option. If the data specification appears, it must be the last option. At least one of the options PAGE, SKIP, or data specification must appear. Note that the options PAGE and SKIP must not appear both in one PUT statement.

General Rules:

1. If the FILE option is specified, and the *filename* refers to an unopened file, the file is opened implicitly.

2. If the STRING option is specified, the PUT operation begins assigning values to the beginning of the string (that is, at the leftmost character position), after appropriate conversions have been performed. Blanks and delimiters are inserted as usual. If the string is not long enough to accommodate the data, the ERROR condition is raised. Note that the variables in the data specification do not have to be character strings; the internal assignment is the same as the transmission from main storage to the output stream, the only difference being that the character-string variable is considered to be the output stream.

3. The option PAGE or SKIP can be given only for PRINT files. If specified, they take effect before the transmission of the values defined by the data specification takes place.

4. The PAGE option causes printing to continue on a new page. If a data specification is present, the transmission of values occurs after the definition of a new current page. A new current page implies line 1.

5. The SKIP option causes a new current line to be defined for the file. The *expression* is the number of lines (w) to be skipped and may range from zero to three inclusive. If *w* is greater than zero, w - 1 blank lines will be inserted in the data stream. If *w* is equal to zero, the effect is that of a carriage return, the characters previously printed will be overprinted. (SKIP(0) may be used to underline parts of the printed line by use of the break character.) If *w* is not specified, 1 is assumed. If less than *w* lines remain on the current page (where the number of lines is determined by the PAGESIZE option of the OPEN statement or by default), the ENDPAGE condition is raised.

## The READ Statement

Function:

The READ statement is a RECORD transmission statement that you can use to transmit a record from an INPUT or UPDATE file to a variable or buffer in main storage.

General format:

```
READ FILE (filename)
    ⎧INTO(variable)          ⎫
    ⎨SET(pointer-variable)   ⎬ ;
    ⎩INTO(variable)          ⎭
       KEY(expression)
```

Syntax rules:

1. The FILE specification must appear first. Either INTO or SET must be specified.

2. The *filename* is the name of the file from which the record is to be read. The file must have the RECORD attribute and must also have either the INPUT or UPDATE attributes.

3. The *variable* of the INTO option is the variable into which the record is to be read. It must be an unsubscripted variable not contained in a structure. It cannot be a label or pointer vari-

able or a parameter and it cannot have the DEFINED attribute.

4. KEY can be specified only if INTO is specified.

General Rules:

1. The file appearing in the FILE specification must have been opened previously.

2. The KEY option must appear if the file has the DIRECT attribute. The _expression_ is the key that determines which record will be read. (See Part III, _Input/Output_, for a discussion of keys.

   The KEY option may also appear for files of INDEXED organization 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.

   If the key specified in the KEY option of a READ statement for an INDEXED SEQUENTIAL file is not found in the file the KEY condition is raised.

3. The SET option can only be specified for CONSECUTIVE files; it cannot be specified for any file having the KEYED attribute. The SET option specifies that the record is to be read into a buffer and the _pointer variable_ is to be set to point to the location of that record within the buffer. The description of the record is determined by a based variable. The value of the pointer variable is valid until the next READ statement is executed or until the file is closed.

## The RETURN Statement

Function:

The RETURN statement terminates execution of a procedure and returns control to the invoking procedure. It may also return a value to the invoking procedure.

General Format:

RETURN [(expression)];

General Rules:

1. If the _expression_ is not specified, the RETURN statement can only terminate a procedure that has not been invoked as a function. When such a statement is executed, control is returned to the invoking procedure at the point logically following the point of invocation. If a RETURN statement is executed in the initial procedure, program execution is terminated.

2. If you have specified an expression, the procedure from which control is to be returned, must be a function procedure. When such a statement is executed, control is returned to the invoking procedure at the point of invocation; the value returned to this point is the value of the expression. If this value does not conform to the explicit or default attributes specified for the procedure being terminated, the value is converted to these attributes before it is actually returned.

## The REWRITE Statement

Function:

You can use the REWRITE statement only for UPDATE files to replace an existing record in a file.

General Format:

REWRITE FILE (filename) [FROM(variable)
          [KEY (expression)]];

Syntax Rules:

1. The FILE specification must appear first. KEY cannot be specified without FROM.

2. The _filename_ 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 represents the record that will replace the existing record in the specified file. It must be an unsubscripted variable; it cannot be contained in a structure; it cannot be a parameter; it cannot be a label or pointer variable; and it cannot have the DEFINED attribute.

General Rules:

1. The file whose name appears in the FILE specification must have been opened previously.

2. If the file has the DIRECT attribute, you have to specify the KEY option. The expression must be a character string. This character string is the source key that determines which record is to be rewritten.

3. The FROM option must be specified for UPDATE files of INDEXED organization having either the DIRECT attribute or the SEQUENTIAL attribute.

4. The FROM option can be omitted only for SEQUENTIAL UPDATE files of CONSECUTIVE organization. When this is the case, the record rewritten is the record in the buffer. Hence, this record must be the last record that was read and it should have been read by a READ statement with the SET option. (The record will be updated by whatever assignments were made to it in the buffer). If the record had been read by a READ with the INTO option, it would be rewritten unchanged.

## The WRITE Statement

Function:

The WRITE statement is a RECORD transmission statement that transfers a record from a variable in main storage to an OUTPUT or UPDATE file.

General Format:

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

Syntax Rules:

1. The FILE specification must appear first.

2. The filename specifies the file into 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 contains the record to be written. It must be an unsubscripted variable; it cannot be a parameter; it cannot be contained in a structure, it cannot be a label or pointer variable; and it cannot have the DEFINED attribute.

4. The KEYFROM option must be specified for DIRECT files, it must also be specified for INDEXED SEQUENTIAL files, but not for any other files.

General Rules:

1. The file must have been opened previously.

2. If the KEYFROM option is specified, the expression is the source key that specifies the logical location in the file where the record is written. (See Part III, Input/Output, for a discussion of source keys). In Model 20 PL/I the source key automatically replaces the recorded key whose length is determined by the KEYLENGTH option and whose location in the record is specified in the KEYLOC option in the ENVIRONMENT attribute.

# Part III

# Model 20 PL/I as Part of the Disk Programming System

# PL/I in the Model 20 Disk Programming System

## INTRODUCTION

Model 20 PL/I is part of the System/360 Model 20 Disk Programming System. It requires a minimum of 16,384 bytes of main storage. Model 20 PL/I consists of a compiler and a set of subroutines, all operating under the control of the DPS System Control programs.

The Model 20 PL/I language is a subset of the full PL/I language and, except for support of input/output devices that can only be attached to a Model 20, is upward compatible with the System/360 DOS PL/I subset language.

## SYSTEM CONFIGURATIONS

### MINIMUM SYSTEM CONFIGURATION

The minimum system configurations for both compilation and execution of PL/I programs under the System/360 Model 20 Disk Programming System are as follows:

#### Submodel 2

An IBM 2020 Central Processing Unit Model D2 (16,384 bytes of main storage);

an IBM 2311 Disk Storage Drive Model 11 or 12;

one of the following card reading devices:

IBM 2501 Card Reader Model A1 or A2,
IBM 2520 Card Read-Punch Model A1,
IBM 2560 Multi-Function Card Machine (MFCM) Model A1;

one of the following printers:

IBM 1403 Printer Model N1, 2, or 7,
IBM 2203 Printer Model A1.

#### Submodel 4

An IBM 2020 Central Processing Unit Model D4 (16,384 bytes of main storage);

an IBM 2311 Disk Storage Drive Model 12;

an IBM 2560 MFCM Model A2;

an IBM 2203 Printer Model A2.

#### Submodel 5

An IBM 2020 Central Processing Unit Model D5 (16,384 bytes of main storage);

an IBM 2311 Disk Storage Drive Model 11 or 12;

one of the following card reading devices;

IBM 2501 Card Reader Model A1 or A2,
IBM 2520 Card Read-Punch Model A1,
IBM 2560 Multi-Function Card Machine (MFCM) Model A1;

one of the following printers:

IBM 1403 Printer Model N1, 2, or 7,
IBM 2203 Printer Model A1.

### MAXIMUM SYSTEM CONFIGURATION

#### Submodel 2

An IBM 2020 Central Processing Unit Model D2 (16,384 bytes of main storage);

two IBM 2311 Disk Storage Drives Model 11 or 12 (both must be the same model);

an IBM 2415 Magnetic Tape and Control Unit Model 1 through 6;

an IBM 2501 Card Reader Model A1 or A2;

an IBM 1442 Card Punch Model 5;

one of the following card units:

IBM 2520 Card Read-Punch Model A1,
IBM 2520 Card Punch Model A2 or A3,
IBM 2560 MFCM Model A1;

one of the following printers:

IBM 1403 Printer Model N1, 2, or 7,
IBM 2203 Printer Model A1;

an IBM 2152 Printer-Keyboard.

#### Submodel 4

An IBM 2020 Central Processing Unit Model D4 (16,384 bytes of main storage);

two IBM 2311 Disk Storage Drives Model 12;

an IBM 2560 MFCM Model A2;

an IBM 2203 Printer Model A2.

an IBM 2152 Printer-Keyboard.

Submodel 5

An IBM Central Processing Unit Model E5 (32,768 bytes of main storage);

four IBM 2311 Disk Storage Drives Model 11 or 12;

an IBM 2415 Magnetic Tape Unit and Control Model 1 through 6;

an IBM 2501 Card Reader Model A1 or A2;

an IBM 1442 Card Punch Model 5;

one of the following card units:

IBM 2520 Card Read-Punch Model A1,
IBM 2520 Card Punch Model A2 or A3,
IBM 2560 MFCM Model A1;

one of the following printers:

IBM 1403 Printer Model N1, 2, or 7,
IBM 2203 Printer Model A1;

an IBM 2152 Printer-Keyboard.

To show how programs written in Model 20 PL/I are compiled and executed under the Disk Programming System (DPS), the main components of the DPS must first be described.

The programs that form the Disk Programming System can be grouped into five categories as listed below:

1. Control programs
2. Service programs
3. Language translators
4. Several Utility programs
5. User-written programs

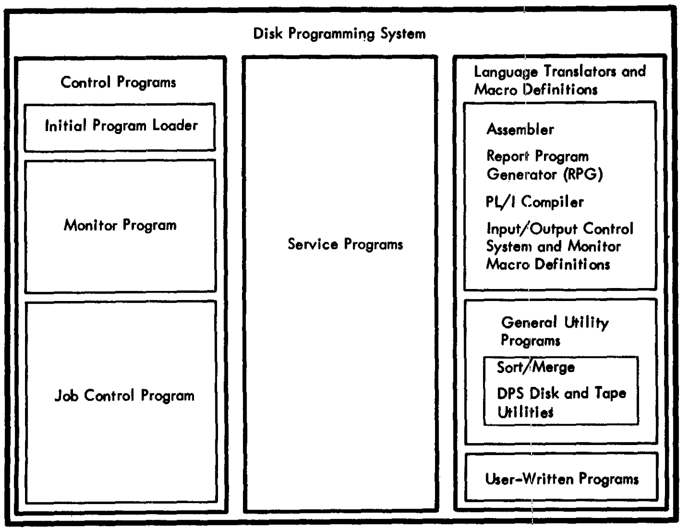


Figure 21. Schematic Representation of the Disk Programming System

### Control Programs

The Initial Program Loader loads the Monitor program into main storage and transfers control to it, causing it to load the Job Control Program. After execution of the Job Control program, control is returned to the Monitor program. The Job Control program is used before each job (1) to assign actual input/output addresses to the symbolic addresses used in the programs, (2) to specify other environmental data (e.g., the date or the storage capacity), and (3) to communicate the name of the next program to be executed to the Monitor program.

### Service Programs

The Service programs are a group of programs that create and maintain the system libraries. They are executed under control of the Monitor.

The system libraries are:

- the core image library, which contains, for example, the Job Control Program, the PL/I Compiler, and executable user-programs.
- the macro library, which contains macro definitions.
- the relocatable area, which is used to temporarily hold compiler output that is to be executed or cataloged immediately.

### Language Translators

One of the DPS language translator programs is the PL/I compiler which compiles PL/I source programs, and links together separately compiled PL/I procedures and library routines into an executable object program.

For the execution of PL/I programs, the most important parts of the programming system are the Monitor program, the Job Control program, and the PL/I compiler, all of which are discussed below.

Figure 21 gives a schematic representation of the Disk Programming System.

## Monitor Program

The Monitor program, which is loaded by the Initial Program Loader, is the main control program of the DPS. It provides functions and contains information needed by all programs. Therefore, it must be in main storage throughout a system run. The Monitor program controls the loading and execution of object programs, that is, programs that have already been compiled (i.e., translated into machine language) by the PL/I compiler.

## Job Control Program

The Job Control program, which is the first program to be loaded by the Monitor program, provides for automatic job-to-job transition in a system run. It prepares the system for the execution of the next job by reading and processing a set of job-control statements punched into cards by the user. The job-control statements must contain all the information required to run a job. They have to indicate the start and name of a job, specify the jobs that are to be executed, and to define input/output requirements of the programs. In response to the job-control statements, the Job Control program allocates the input/output units required, and then requests the Montior program to initiate the execution of the specified job (or batch). After execution of a job, the Job Control program prints the error statistics if specified and obtains information about the next job from a new set of control statements.

## Model 20 PL/I Compiler

The Model 20 PL/I compiler is a program that

1. translates (i.e., _compiles_) a PL/I source program into a set of System/360 Model 20 machine instructions, and

2. _link-edits_ the set of machine instructions into a form suitable for execution.

The set of machine instructions produced by a _compilation_ is termed a _compiled object module_.

The set of machine instructions produced by _link-editing_ is termed an _executable object program_.

A compiled object module is not executable. It must first be link-edited by the PL/I compiler, that is, linked with other user and IBM-supplied modules or -- if the compiled object module is a complete program -- only with IBM-supplied modules to form an executable object program.

The Job Control program is executed before each job to prepare the system for the job to be executed, and it is called subsequently each time the end of the job is reached. Control statements supply the information required by the Job Control program.

This section gives a detailed description of the most important job-control statements you need to compile and execute your programs.

Figure 22 is an illustration of the I/O device assignments used for compilation and execution.

Note: SYSRDR and SYSIPT may be assigned the same card-reading device, in which case the cards read on SYSIPT follow those read on SYSRDR in the hopper. If the additional work areas WORK2 or WORK3 are used, the two (three) extents WORK1, WORK2, (and WORK3) must reside on different disks. Note also that the symbolic device addresses specified in this figure must be used for the work files. The work file WORKL is used in connection with the option LSORT.

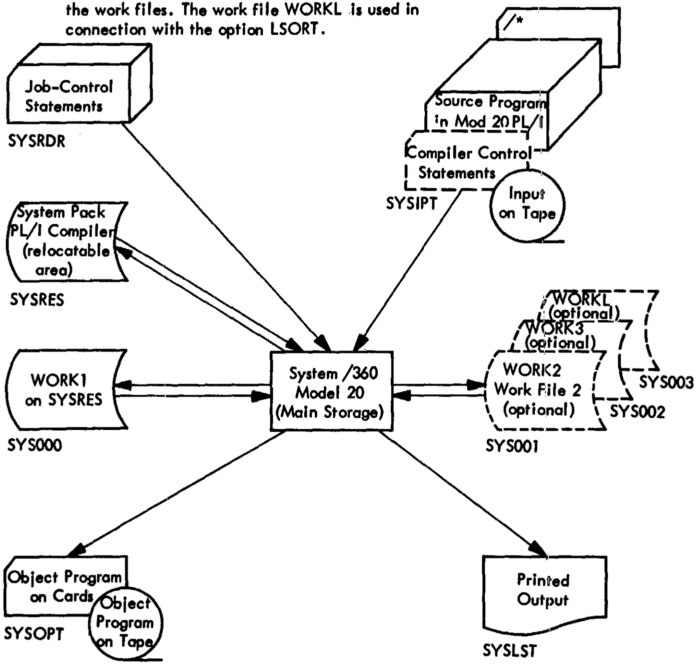


Figure 22. I/O Device Assignments Used for Compilation and Execution

## I/O Device Assignment

The I/O devices used by the system for program compilation and execution are referred to by symbolic, not actual device addresses. This means that, when writing your programs, you can disregard the actual device assignments of the system configuration you use. The symbolic device addresses you can use are listed in Figure 23. No other addresses may be used.

| Symbolic Device Address | Refers to |
|---|---|
| SYSRDR | Card reading device for control statements. Not used by PL/I compiler or object programs. |
| SYSIPT | Input device (card reader or tape drive) from which the input for the compiler is read. |
| SYSOPT | Card punching device or magnetic-tape drive on which the object program of the compiler is written. |
| SYSLST | Printer for printing listings and diagnostic messages. |
| SYSLOG | Printer used to print operator messages. The device may be the same as the one referred to by SYSLST. |
| SYS000- SYS019 | User-program I/O devices (disk and magnetic tape devices only). |

Figure 23. Symbolic Device Addresses

Symbolic device addresses can be assigned to actual devices

1. when building the system, or
2. by means of the ASSGN statements.

## Job Control Statements

Job-control statements identify a job and define its requirements and options. They serve as input to the Job Control program and enable it to provide automatic job-to-job transition. Figure 24 lists the DPS job-control statements and their functions.

| Operation Specification | Function |
|---|---|
| ASSGN | Changes or deletes I/O assignments |
| CONFG | Specifies main-storage capacity (either 16K, 24K, or 32K bytes) |
| DATE | Specifies date, i.e., the day of the year and the year |
| DELET | Causes permanent labels to be deleted |
| DLAB | Supplies disk label information for individual file |
| DSPLY | Causes listing of all permanent labels |
| EXEC | Indicates end of control statements |
| FILES | Positions magnetic tape reel by skipping specified number of tape marks |
| JOB | Specifies name of job and the type of operation to be performed |
| LOG | Causes listing of control statements on SYSLOG |
| NOLOG | Causes listing of control statements to be discontinued |
| OPTN | Indicates that the printout of tape error statistics is required by the job and/or indicates that the execution of the job is not to be interrupted by an inquiry program |
| PAUSE | Causes immediate halt |
| TPLAB | Supplies tape label information for individual file |
| VOL | Specifies name of file to be processed and the symbolic address of the drive on which it is mounted |
| XTENT | Defines the extents used by a file on a disk pack and specifies the symbolic address of the drive on which the pack is mounted |

Figure 24. Summary of Job Control Statements and their Functions

## ORDER OF INPUT

Job-control statements are read by the Job Control program on a device whose symbolic address is SYSRDR. Normally, the first job-control statement for a particular job is a JOB statement. Only PAUSE, LOG, and NOLOG statements may precede a JOB statement. The last job-control statement must be an EXEC statement.

Except where noted, the remaining job-control statements may be arranged in any order between the JOB and the EXEC statements.

## FORMAT OF JOB-CONTROL STATEMENTS

The general format of the job-control statements is as follows:

| Name | Operation | Operand(s) | Comments |
|---|---|---|---|
| // | operation | [operand] [,operand]... | |

// identifies a statement as a job-control statement. The slashes must appear in the first two columns of the job-control statement and must be followed by at least one blank.

operation indicates the function of the job-control statement. For example, the word ASSGN indicates that the control statement specifies an I/O device assignment. The operation field can be up to five characters long and must be followed by at least one blank.

operands supply additional information about the job-control statement. For example, the operands of the ASSGN statement specify the symbolic device address and the characteristics of the actual device. The operand field may be blank or may contain one or more operands, separated by commas, with no intervening blanks. A blank to indicate the end of the field must follow the last operand in the field. The field must not extend beyond column 71 of the punched card. With a DLAB statement where not all operands can be accommodated on one card, a continuation card must be used.

programmer's comments may be inserted in all control statements used in conjunction with the Model 20 DPS Control and Service programs. They must, however, follow the rightmost operand of the statement.

When preparing a control statement that contains comments, observe the following rules:

1. If the control statement has one or more operands, the comments must be separated from the last operand by at least one blank column.

2. If the statement does not permit the use of an operand, the comments must be separated from the operation entry by at least one blank column.

3. If an operand has been omitted from the statement, the absence of the operand must be indicated by a comma and the comments must be preceded by at least one blank column.

Comments are printed but have no effect on the program. They must not extend beyond column 71 of the statement.

### JOB Control Statement

The JOB Control statement indicates to the Job Control program that a set of job-control statements follows. The format of the JOB control statement indicates whether the program is to be executed or compiled only or compiled and executed. The JOB control statement has the following formats:

| Name | Operation | Operand(s) | Comments |
|---|---|---|---|
| // | JOB | program-name[1] | |
| // | JOB | PL1 | |
| // | JOB | PL1, program-name[2] | |
| [1]may be any name<br>[2]must be name of root segment | | | |

program-name
The name of the program to be executed. If the name appears by itself, it is the name of either a program in the core-image library or an executable object program stored on cards or magnetic tape.

If the name is preceded by the word PL1, the input is to be compiled, link-edited, and executed. If only precompiled object modules are used, compilation is skipped.

If the word PL1 appears alone, the input is to be compiled and/or link-edited, but not immediately executed.

There is no limit on the length of the program-name in any JOB control statement. However, if the length exceeds six characters, only the leftmost six characters are recognized.

Figure 25 gives an example of using job-control statements.

### DATE Control Statement

The DATE control statement must be included in the control statements for the first job after initial program loading. Further DATE statements need only be supplied if the date is to be changed.

The format of the DATE control statement is:

| Name | Operation | Operand(s) | Comments |
|---|---|---|---|
| // | DATE | yyddd | |

yy -- the last two digits of the year; e.g., 70 for 1970.

ddd -- the day of the year; e.g., 002 for January 2nd. (Any of the numbers 001 - 366 may be used).

The year and the day of the year are used for label checking. The date can be used for dating output reports of problem programs by use of the PL/I DATE built-in function.

### CONFG Control Statement

The CONFG control statement specifies the capacity of main storage. It must be included only if the capacity of main storage stored in the Monitor program is to be changed. The CONFG statement must precede any label information for tape files (e.g., VOL, TPLAB).

The format of the CONFG control statement is:

| Name | Operation | Operand(s) | Comments |
|---|---|---|---|
| // | CONFG | 16 | |
| // | CONFG | 24 | |
| // | CONFG | 32 | |

16 -- indicates a storage capacity of 16, 384 bytes.

24 -- indicates a storage capacity of 24, 576 bytes.

32 -- indicates a storage capacity of 32, 768 bytes.

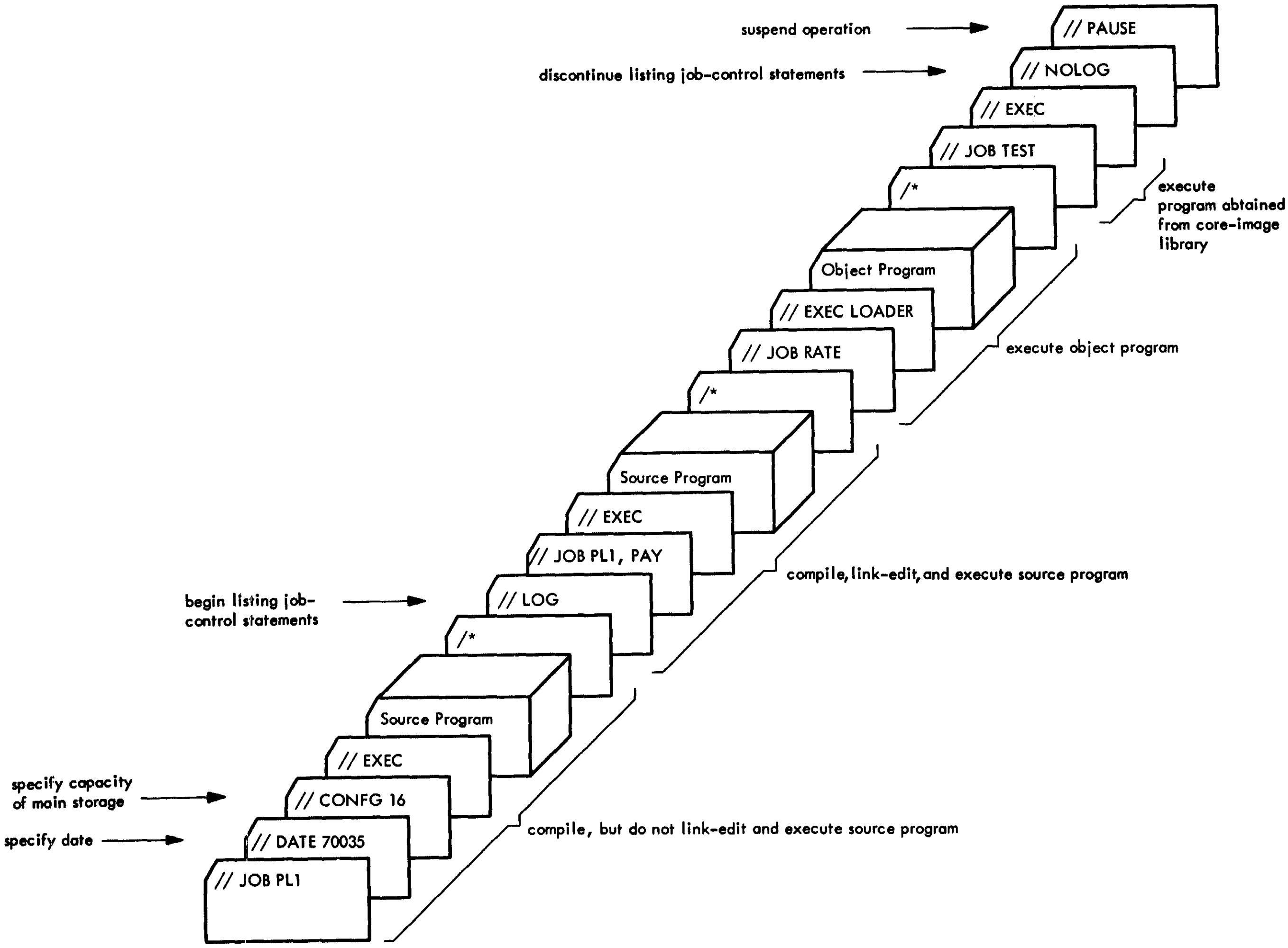

suspend operation
// PAUSE
discontinue listing job-control statements
// NOLOG
// EXEC
// JOB TEST
execute program obtained from core-image library
/*
Object Program
// EXEC LOADER
// JOB RATE
execute object program
/*
Source Program
// EXEC
// JOB PL1, PAY
compile, link-edit, and execute source program
begin listing job-control statements
// LOG
/*
Source Program
// EXEC
specify capacity of main storage
// CONFG 16
compile, but do not link-edit and execute source program
specify date
// DATE 70035
// JOB PL1

Figure 25. Example of Using Job Information Control Statements

OPTN Control Statement

The OPTN control statement has the following format:

| Name | Operation | Operand(s) Comments |
|---|---|---|
| // | OPTN | NOINQ, TES |

NOINQ
No program may be executed as an inquiry program until the next Job Control run has been completed. For information about inquiry programs, refer to the section Special Programming Information.

TES
Indicates that tape error statistics are to be kept during the job that follows and printed before the next Job Control run. When TES is specified, LOG must also be specified. You may specify one or both operands, in any order, separated by a comma.

LOG Control Statement

The Job Control program does not normally list job-control statements. However, when a LOG control statement is encountered, it begins to list all job-control statements for all jobs until a NOLOG statement is encountered. The LOG statement is the first statement that appears in the listing. The format of the LOG control statement is:

| Name | Operation | Operand(s)          Comments |
|---|---|---|
| // | LOG | |

Note: You may place the LOG statement either before the JOB statement or anywhere between the JOB and EXEC statements, but not between the VOL, DLAB, XTENT, or VOL, TPLAB control statements. The LOG statement is only effective if a printer is assigned to SYSLOG.

## NOLOG Control Statement

The NOLOG control statement stops the listing of job-control statements for all jobs. It is the last statement to appear in the listing. Listing of control statements is resumed only if another LOG control statement is encountered. The format of the NOLOG statement is:

| Name | Operation | Operand(s)          Comments |
|---|---|---|
| // | NOLOG | |

Note: You may place the NOLOG statement either before the JOB statement or anywhere between the JOB statement and the EXEC statement, but not between the VOL, DLAB, XTENT or VOL, TPLAB control statements.

## PAUSE Control Statement

The PAUSE control statement is used to interrupt processing. It merely suspends operation and does not affect the contents of main storage. The operator need only press the Start key on the CPU console to resume operation. The PAUSE control statement has the format:

| Name | Operation | Operand(s)          Comments |
|---|---|---|
| // | PAUSE | |

The Job Control program executes the PAUSE statement as soon as it encounters it. You may place it anywhere among the control statements, but not between the VOL, DLAB, XTENT or VOL, TPLAB control statements.

## EXEC Control Statement

The EXEC control statement indicates to the Job Control program that the reading of a set of job-control statements has been completed and that control is to be returned to the Monitor. The EXEC control statement has the following formats:

| Name | Operation | Operand(s)          Comments |
|---|---|---|
| // | EXEC | |
| // | EXEC | R |
| // | EXEC | LOADER |
| // | EXEC | LOADER,R |

R
The operand R as the only operand (2nd format) is used exclusively in EXEC statements for core-image maintenance. It indicates that the input to the CMAINT (Core-Image Maintenance) program is read from the relocatable area on the system disk pack.

LOADER
The operand LOADER used as the only operand (3rd format) indicates that the execute-loader function is used, i.e., your executable program is read from the card reading device or tape drive assigned to SYSIPT and then excuted.

LOADER,R
The operand R used in conjunction with the operand LOADER (4th format) indicates that the execute loader function is to be used and that the program you want to be executed is read from the relocatable area and executed. The name in the JOB statement is then ignored.

Note: When the execute-loader function is used, the CMAINT program must be contained in the core-image library of your disk-resident system.

Figure 26 shows examples for the usage of the JOB and the EXEC control statement.

## ASSGN Control Statement

In the PL/I program itselt, you refer to I/O devices in the MEDIUM option of the ENVIRONMENT attribute by symbolic device addresses, and not by specifying the actual device address. The Job Control program assigns actual I/O device addresses to the symbolic device address. the actual device address is assigned to the symbolic one by means of an ASSGN control statement.

Figure 27 shows an example of I/O device assignment.

Permanent device assignments are made by means of ASSGN macro instructions when generating the Monitor. These permanent device assignments are written onto the system disk pack and loaded into main storage together with the Monitor at IPL time.

| | |
|---|---|
| 1. // JOB CMAINT<br>// EXEC[R] | Input from SYSIPT = Card or Tape unless R is specified in which case the program in the relocatable area is included in the core-image library. |
| 2. // JOB program-name<br>// EXEC | Fetch executable object program from core-image library and execute. |
| 3. // JOB program-name<br>// EXEC LOADER | Load executable object program from SYSIPT and execute.<br>Input (program) in card image form from SYSIPT = Card or Tape. |
| 4. // JOB program-name<br>// EXEC LOADER,R | Load program from relocatable area and execute. |
| 5. // JOB PL1, program-name<br>// EXEC | Compile, link-edit, and execute program.<br>Input from SYSIPT = Card or Tape. |
| 6. // JOB PL1<br>// EXEC | Compile, or compile-and-link, or link depending on compiler-control statements.<br>Input from SYSIPT = Card or Tape.<br>Output onto SYSOPT = Card or Tape and onto the relocatable area. |

Figure 26. Examples for the Usage of JOB and EXEC Control Statements

Permanent device assignments loaded into main storage apply to all jobs, unless you alter them by inserting ASSGN control statements in your problem program. These new device assignments then remain in effect for all subsequent jobs unless changed again by other ASSGN control statements.

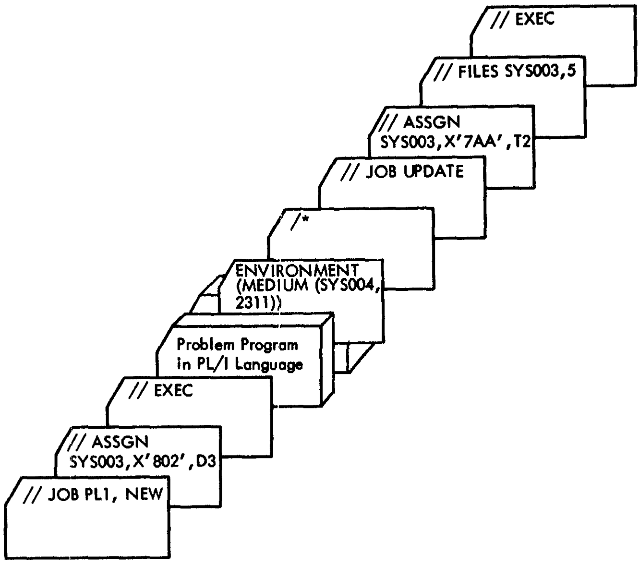

// EXEC
// FILES SYS003,5
// ASSGN SYS003,X'7AA',T2
// JOB UPDATE
/*
ENVIRONMENT (MEDIUM (SYS004, 2311))
Problem Program in PL/I Language
// EXEC
// ASSGN SYS003,X'802',D3
// JOB PL1, NEW

Figure 27. Example of I/O Device Assignment

If the operator reloads the Monitor into main storage through an IPL run, the permanent device assignments written on the system disk pack are reloaded into main storage and take effect, cancelling any assignments that have been changed by ASSGN control statements.

Changing, at execution time, assignments for card devices, printer and printer-keyboard has no effect on the device addresses specified in the MEDIUM options in the source program. These device types can only be changed by changing the device type in the MEDIUM option of the ENVIRONMENT attribute and recompiling the program.

The formats of the ASSGN control statement are:

| Name | Operation | Operands Comments |
|---|---|---|
| // | ASSGN | symbolic-device-address, actual-device-address, type, specification |
| // | ASSGN | symbolic-device-address, actual-device-address,type |
| // | ASSGN | symbolic-device-name, UA |

symbolic-device-address
One of the following names:

| | |
|---|---|
| SYSRDR | Card-reading device for job-control statements |
| SYSIPT | Card-reading device or magnetic-tape drive for input |
| SYSOPT | Card-punching device or magnetic-tape drive for output |
| SYSLST | Printer for output listings |
| SYSLOG | Printer for listing job-control statements |
| SYS000-<br>SYS019 | User I/O devices (tape and disk devices only) |

*Note 1*: In the case of magnetic tape and disk units, the introduction of symbolic device names allows to change the actual device address by means of an ASSGN job-control statement.

*Note 2*: If SYSRDR and SYSIPT are both assigned to the same 2560 device, two /* cards must follow the source input on SYSIPT.

*actual-device-address*

The device address of the unit chosen by the machine operator. The attachment point and the unit must be specified in the form X'cuu' where

*c*, the *attachment point*, is:

| Attachment Point | for Device Type |
|---|---|
| 1 | IBM 2501 Card Reader |
| 2 | IBM 2520 or 2560 (all models) |
| 3 | IBM 1442 Card Punch |
| 4 | IBM 1403 or 2203 Printer |
| 7 | IBM 2415 Magnetic Tape Unit |
| 8 | IBM 2311 Disk Storage Drive |

*uu*, the *unit*, is:

| Unit | for Device Type |
|---|---|
| 01, 02, 03, and 04 | disk |
| 08 through FD | tape |
| 00 | all other equipment |

*type*

Code specifying the device type:

| Code | for Device Type |
|---|---|
| D3 | 2311 Disk Storage Drive Model 11 |
| D4 | 2311 Disk Storage Drive Model 12 |
| L1 | 1403 Printer |
| L3 | 2203 Printer |
| P2 | 1442 Card Punch |
| P3 | 2520 Card Punch |
| R4 | 2501 Card Reader |
| R5 | 2520 Card Read Punch |
| R6 | 2560 MFCM Primary Feed |
| R7 | 2560 MFCM Secondary Feed |
| T1 | 2415 7-track Tape Drive |
| T2 | 2415 9-track Tape Drive |

*specification*

Code indicating device specifications for 7-track tape and 9-track phase-encoded tape. This field is not used for other types of equipment. The codes for 7-track tape are shown in Figure 28.

| Code | Bytes per Inch | Parity | Translate Feature | Convert Feature |
|---|---|---|---|---|
| X'10' | 200 | odd | off | on |
| X'20' | 200 | even | off | off |
| X'28' | 200 | even | on | off |
| X'30' | 200 | odd | off | off |
| X'38' | 200 | odd | on | off |
| X'50' | 556 | odd | off | on |
| X'60' | 556 | even | off | off |
| X'68' | 556 | even | on | off |
| X'70' | 556 | odd | off | off |
| X'78' | 556 | odd | on | off |
| X'90' | 800 | odd | off | on |
| X'A0' | 800 | even | off | off |
| X'A8' | 800 | even | on | off |
| X'B0' | 800 | odd | off | off |
| X'B8' | 800 | odd | on | off |

Figure 28. Codes for 7-Track Tapes

For 9-track tape, specifications are not required unless phase-encoded tape with the compatibility feature is used.

The codes for 9-track phase-encoded tape are:

```
X'C0' -- 1600 bytes per inch
X'C8' --  800 bytes per inch
```

*UA*

Indicates that the device is to be unassigned.

The following examples illustrate the use of the ASSGN control statement.

1. The following control statement assigns SYS001 to a 2415 9-track magnetic tape drive whose unit number is 08, specification 1600 bytes per inch.

```
// ASSGN  SYS001,X'708',T2,X'C0'
```

2. The following control statement assigns SYS002 to a 2311 Disk Storage Drive Model 11, whose unit number is 02.

```
// ASSGN  SYS002,X'802',D3
```

3. The following control statement releases SYS003 from a device assignment.

```
// ASSGN  SYS003,UA
```

## FILES Control Statement

The FILES control statement is used to position the magnetic tape at the beginning of any file on a multi-file reel. If, in a set of job-control statements, a FILES control statement and an ASSGN control statement refer to the same symbolic I/O address, the FILES statement must follow the ASSGN statement. The format of the FILES control statement is:

| Name | Operation | Operand(s) | Comments |
|---|---|---|---|
| // | FILES | device-name,skip | |

device-name
The symbolic name of the tape drive on which the reel of tape to be positioned is mounted. (A complete list of device names is given in the discussion of the ASSGN control statement).

skip
The number of tape marks to be skipped (1-999), counted from the load point, in order to position the tape. (On unlabeled files, a tape mark follows each file. On labeled files, another tape mark follows the labels).

An example showing the use of the FILES control statement is shown in the section Input/Output, under the heading Positioning of Unlabeled Tape Files.

## Label Information Processing

The Job Control program prepares for the writing and checking of standard labels for disk and magnetic tape files.

Control statements are required only for labeled tape and disk files. Two control statements must be supplied for each labeled tape file. At least three control statements must be supplied for each disk file (a minimum of four is required for indexed files). The types of control statements are:

1. VOL (volume)
2. TPLAB (tape label)
3. DLAB (disk label)
4. XTENT (disk extent)

Each file requires a VOL control statement and either a TPLAB or DLAB control statement. XTENT control statements are required only for disk files.

The control statements must be read in the order indicated below.

1. The VOL control statement for a file.
2. Either the TPLAB or the DLAB control statement for that file.
3. One or more XTENT control statements if the file is in disk storage. If there is more than one XTENT control statement, they must be arranged in the order in which the areas they define are to be used.

The label control statements for tape and disk are further discussed in the section Input/Output.

Figure 29 shows another example illustrating the use of job-control statements.

```
               // LOG THIS JOB STREAM ILLUSTRATES OPERATIONAL
               // LOG    ASPECTS OF PL/I AND THE JOB CONTROL LANGUAGE.
               // LOG FIRST IS AN EXAMPLE OF PL/I COMPILE AND EXECUTE
               // JOB PL1,PROG1
               // DATE 70001
If different   // CONFG 16
from the       // ASSGN SYS000,X'801',D3      2311 DISK DRIVE, MODEL 11
Monitor        // ASSGN SYSIPT,X'100',R4      2501 READER
Assignments    // ASSGN SYSLST,X'400',L1      1403 PRINTER
               // ASSGN SYSOPT,X'200',R7      2560 SECONDARY
               // ASSGN SYSLOG,X'400',L1      1403 PRINTER
               // OPTN NOINQ
               // DELET
               // VOL SYS000, WORK1
               // DLAB'DPS 16K DISK WORK FILE1   1202020',     X
                          0001,70001,70001                 col.72
               // XTENT  1,000,0090000,0102009,'202020',SYS000
               // EXEC
               +  COPTN LINK,GODECK
               +  SEGMENT PROGA
               +  PROCESS  XREF,ATR,LIST
                 /* PL/I SOURCE DECK */
                         .
                         .
                         .
               /* DELIMITER CARD
               DATA CARDS
Two /* -->     /*
cards if       // PAUSE END OF JOB. INSERT OBJECT IN JOB STREAM BELOW.
SYSRDR =       // LOG PL/I PROGRAM IS STILL IN RELOCATABLE AREA, AND IT IS
SYSIPT and     // LOG    POSSIBLE TO CATALOG IT INTO THE CORE IMAGE LIBRARY,
assigned to    // LOG    AS SHOWN IN THE FOLLOWING JOB.
the same       // JOB CMAINT
2560           // EXEC R
If SYSRDR -->  // CATAL
≠ SYSIPT,      // END
CATAL cards    // PAUSE END OF JOB.
have to be     // LOG IT IS ALSO POSSIBLE TO CATALOG THE PROGRAM INTO
included for   // LOG    THE LIBRARY FROM THE OBJECT DECK, AS FOLLOWS.
each program   // JOB CMAINT
segment        // EXEC   , NOTE--FOLLOWING IS COMPILER OUTPUT FROM FIRST JOB!
               // CATAL
                  PHASE PROGA,A,X'1200' GENERATED DURING LINK-EDITING.
               12
                2 ESD
                9
OBJ            12
DECK            2 TXT
                9
               12
                2 END
                9
               /*
               // END
```

Figure 29. Example of the Sequence of Job Control Statements, Part 1 of 2

```
  // PAUSE END OF JOB
  // LOG THE OBJECT PROGRAM IS STILL IN THE RELOCATABLE
  // LOG    AREA AND CAN BE AGAIN EXECUTED, AS FOLLOWS ...
  // JOB ANYNAME
  // EXEC LOADER,R
  DATA CARDS
  /*
  // PAUSE
  // LOG OR THE OBJECT DECK CAN BE EXECUTED, AS FOLLOWS.
  // JOB ANYNAME
  // PAUSE INSERT OBJECT DECK AFTER EXEC CARD
  // EXEC LOADER
    OBJECT DECK GOES HERE
  /*
 ┌DATA┐
 └/*  ┘
  // PAUSE
  // LOG THE PROGRAM CAN ALSO BE EXECUTED FROM THE
  // LOG    CORE IMAGE LIBRARY, SINCE IT HAS BEEN
  // LOG    PREVIOUSLY CATALOGED.
  // JOB PROGA
  // EXEC
 ┌DATA CARDS┐
 └/*        ┘
  // PAUSE END OF JOB
  // LOG FOLLOWING IS AN EXAMPLE OF PL/I COMPILE ONLY.
  // JOB PL1 OTHER JCL PROVIDED BY FIRST JOB
  // EXEC
  +  PROCESS
     /* IN THIS EXAMPLE, ALL COMPILER OPTIONS ARE ASSUMED */
     /* PL/I SOURCE DECK GOES HERE */
  /*
  // PAUSE
```

Figure 29. Example of the Sequence of Job Control Statements, Part 2 of 2

The Model 20 PL/I compiler is a component of the Model 20 Disk Programming System.

This section describes the Model 20 PL/I compiler, its options, the listings it produces, and its diagnostic features. (At the present stage, this section contains only preliminary information.)

Let us briefly recall the essential functions of the control programs described so far:

After initial program loading, the Monitor receives control and loads the Job Control program, which reads and processes the job-control statements. It then returns control to the Monitor, which loads the PL/I compiler and transfers control to it.

The compiler is the program that translates source programs into machine language and performs the necessary link-editing to transform compiled object modules into an executable object program.

In addition to translating the source program, the PL/I compiler produces listings useful for documentation and program check-out. If it detects errors, it prints the corresponding diagnostic messages. Syntax errors are printed with the source program listing. Errors detected within data declarations are printed together with the symbol table listing. Other errors are printed at the end of compilation with an extra listing.

## Input to Compiler

When the Job Control program has read and processed the job-control statements preceding a job, it transfers control to the compiler. The compiler then reads the input -- which may be on cards or on magnetic tape in card-image format -- from SYSIPT to perform any of the following:

- compile
- compile-and-link
- compile-link-and-execute
- link
- link-execute

The input read from SYSIPT may consist of

- one or more PL/I source modules
- one or more precompiled object modules or
- PL/I source module(s) and precompiled object module(s)

A _module_ is a procedure that is either part of a program or a complete program. Precompiled object modules are modules that have been compiled, but not link-edited, or assembled by the PL/I compiler or the Assembler, respectively, and are used again as compiler input.

If more than one module is processed by the compiler in one job, we speak of _batch compilation_.

### PL/I_SOURCE_MODULE_INPUT

Source modules are always read from SYSIPT. They may either be on cards or on magnetic tape, depending on whether a card reading device or a tape drive is assigned to SYSIPT.

The first statements processed by the compiler are always the compiler-control statements, with their options.

Compiler-control statements and options are furnished to the compiler on cards either for the whole job or separately, for each procedure.

For the Model 20 PL/I compiler, we have the following control statements: COPTN, PROCESS, SEGMENT, and COPY. The COPTN control statement is valid for the whole job. The PROCESS statement is furnished once per procedure.

The statement immediately preceding a source module must always be the PROCESS statement. The PROCESS statement must be specified for every source module to be compiled. For a compile-link-and-execute run in which several procedures are to be compiled and link-edited, the first compiler-control statement read from SYSIPT must always be the COPTN statement with the LINK or GODECK option specified. For source modules that are to be compiled only, the COPTN statement is optional. If specified, the COPTN statement must precede all other compiler-control statements.

If overlay is used, SEGMENT control statements must precede each segment.

### Compilation of PL/I Source Modules

To compile two source modules without immediate execution and/or cataloging and obtain the output (i.e., the compiled object modules) on cards, use the job and compiler-control statements in the sequence shown in Figure 30.

```
// JOB PL1
     .
     .  (other job-control statements)
     .
// EXEC
[+  COPTN compiler-options]
+ PROCESS DECK
     .
     .  (source program 1)
     .
+ PROCESS DECK
     .
     .  (source program 2)
     .
/*
```

Figure 30. Compilation of PL/I Source Modules

The option DECK in the PROCESS control statements specifies that the compiler output (i.e., the compiled object modules) is to be punched or written on tape. Since we want the output to be punched into cards, we have to assign SYSOPT to a card-punching device. (The complete set of options of the PROCESS control statement is discussed in the section Options and Control Statements).

### Compilation and Linkage

If the two source modules are to be linked as well as compiled, that is, to have an executable object program available for execution at a later date, your control statements would have to look as shown in Figure 31.

```
// JOB PL1
     .
     .  (other job-control statements)
// EXEC
+ COPTN LINK,GODECK
+ PROCESS DECK
     .
     .  (source program 1)
     .
+ PROCESS DECK
     .
     .  (source program 2)
/*
```

Figure 31. Compilation and Linkage of PL/I Source Modules

### Compilation, Linkage and Execution of PL/I Source Modules

For a compile-link-and-execute run, you would need the control statements shown in Figure 32.

```
// JOB PL1,program-name
     .
     .  (other job-control statements)
     .
// EXEC
+ COPTN LINK
+ PROCESS
     .
     .  (PL/I source program 1)
     .
+ PROCESS
     .
     .  (PL/I source program 2)
     .
/*
data cards
/*
```

Figure 32. Compilation, Linkage, and Execution of PL/I Source Modules

Compilation starts with the compiler-control statement + COPTN LINK. The option LINK indicates that the program is to be link-edited into an executable object program.

Precompiled object modules produced by the Model 20 PL/I compiler cannot be handled by the DPS Linkage Editor program, nor can they be loaded immediately by the Fetch routine of the Monitor.

### Compilation, Linkage and Cataloging of a PL/I Source Module

To compile-link-and-catalog a PL/I source module, that is, to include it in the core-image library as a permanent entry, use the set of control statements shown in Figure 33.

Compilation starts with the + COPTN LINK control statement, LINK specifying that the PL/I source module is to be compiled and link-edited.

If you want to compile-link-and-catalog and then immediately execute the same program, specify the job-control statements

```
// JOB program-name
// EXEC
```

following the CMAINT control statements (see Figure 33).

```
// JOB PL1
      .
      .  (other job-control statements)
      .
// EXEC
+  COPTN LINK
+  SEGMENT
+  PROCESS
      .
      .  (source module)
/*
// JOB CMAINT
// EXEC R
// CATAL
// END
```

Figure 33. Compilation, Linkage, and Cataloging of PL/I Source Modules

## PRECOMPILED OBJECT MODULE INPUT

A precompiled object module is a source module that has been compiled or assembled but not link-edited and that is used again as input to the PL/I compiler.

Precompiled object decks may be output from the Model 20 PL/I compiler or from the DPS Assembler. If subroutines assembled with the DPS Assembler are used, none of the following Assembler options are allowed in the AOPTN control statements: NOESD, NORLD, ENTRY. The assembler language must not contain any supervisor macros or input/output programming, nor the XFR pseudo instruction.

Control statements for the DPS Linkage Editor Program must not be contained in the precompiled deck.

### Linkage and Execution of Precompiled Object Modules

Suppose you have several precompiled object modules which you want to be linked into one executable PL/I object program and executed immediately. The set of control statements for this job would have to look as shown in Figure 34.

The job-control statements for this run indicate that the program is to be executed immediately after compilation -- in this case, after link-editing. The link-editing of the precompiled object modules starts with the COPTN LINK card and ends with the first end-of-file (/*) record.

Link-editing produces an object program that can be executed immediately.

```
 // JOB PL1,program-name
       .
       .  (other job-control statements)
       .
 // EXEC
 +  COPTN LINK
 +  COPY
       .
       .  (precompiled object module 1)
       .
[+  COPY]
       .
       .  (precompiled object module 2)
       .
[+  COPY]
       .
       .  (precompiled object module 3)
       .
 /*
       .
       .  (data)
       .
 /*
```

Figure 34. Linkage and Execution of Precompiled Object Decks

### Linkage of Precompiled Object Modules

If you want to only link the same precompiled object programs and have them available for execution at a later date, use a set of control statements as shown in Figure 35.

```
 // JOB PL1
       .
       .  (other job-control statements)
       .
 // EXEC
 +  COPTN LINK,GODECK
 +  COPY
       .
       .  (precompiled object program 1)
       .
[+  COPY]
       .
       .  (precompiled object program 2)
       .
[+  COPY]
       .
       .  (precompiled object program 3)
       .
 /*
```

Figure 35. Linkage of Precompiled Object Modules

Note: You have to specify GODECK in the COPTN control statement in addition to assigning SYSOPT to a card-punching device or a magnetic-tape drive.

### PL/I SOURCE MODULE AND PRECOMPILED OBJECT MODULE INPUT

#### Compilation and Execution

If several PL/I source modules are to be compiled and several precompiled object modules are to be link-edited with the compiled object modules into one executable object program that is to be executed immediately, you must use the set of control statements needed for such a job as shown in Figure 36.

```
// JOB PL1,program-name
     .
     .  (other job-control statements)
     .
// EXEC
+  COPTN LINK
+  COPY
     .
     .  (precompiled object module 1)
     .
+  PROCESS
     .
     .  (source module 1)
+  PROCESS
     .
     .  (source module 2)
     .
+  COPY
     .
     .  (precompiled object module 2)
/*
┌(data)┐
└/*    ┘
```

Figure 36. Compilation and Linkage of PL/I Source Modules and Pre-Compiled Object Modules for Immediate Execution

The precompiled object decks must be headed by a COPY control statement, each source program by a PROCESS control statement.

#### Compilation, Link-Editing and Cataloging

To compile, link-edit, and catalog the set of programs shown in Figure 36, omit the ,program-name in the first job-control statement and the data cards followed by /*.

The following control statements would be needed to catalog the job after compilation:

```
// JOB CMAINT
// EXEC R
// END
```

These control statements must follow the input deck if the program is to be cataloged from the relocatable area.

## Output from Compiler

Output from the compiler may be

- a compiled object module and/or
- an executable object program

The executable object program is always in the relocatable area and on the device assigned to SYSOPT if the option GODECK has been specified in the COPTN control statement.

The compiled object module is on the device assigned to SYSOPT. Note, however, that you must specify the option DECK in the PROCESS control statement and assign a card-punching device or a tape drive to SYSOPT. Otherwise, the output is lost.

### COMPILED OBJECT MODULES

A compiled object module is an output module that has been compiled but not link-edited by the PL/I compiler. It cannot immediately be executed or cataloged. Before the object module can be linked, linked-and-executed, or linked-and-cataloged, it must be read again from the device assigned to SYSIPT like a PL/I source module, but with different and additional control statements and options. Such an object module, which is used again as compiler input, is referred to as a precompiled object module. (For the necessary control statements, refer to the section Input to the Compiler).

Compiled object module output may be on cards or on magnetic tape.

#### Card Output

The output is on cards if the DECK option has been specified in the PROCESS control statement, and if a card-punching device has been assigned to SYSOPT.

#### Tape Output

The output is on magnetic tape if the DECK option has been specified in the PROCESS control statement and if a magnetic-tape drive has been assigned to SYSOPT. The output tape contains the compiled object module(s) in card-image format.

SYSOPT must be assigned if PROCESS DECK is specified.

### EXECUTABLE OBJECT PROGRAMS

Executable object programs are programs that have been compiled and link-edited. They are in the relocatable area and can immediately be executed and/or cataloged into the core-image library by the CMAINT program.

Note: Executable object programs cannot be used again as compiler input.

If the GODECK option has been specified in the COPTN control statement, the executable object-program output is also on a device assigned to SYSOPT. Depending on whether a card punching device or a magnetic-tape drive is assigned to SYSOPT, the output is either on cards or on magnetic tape.

For the job-control statements necessary to execute an executable object program, refer to the section Job Control.

For the control statements necessary to catalog an executable object program, refer to the section Cataloging.

## Compiler-Control Statements and Options

The control statements for the Model 20 PL/I compiler are COPTN, PROCESS, SEGMENT, and COPY.

The general format of the compiler-control statements is:

| Name | Operation | Operand(s) |
|---|---|---|
| + | operation | [operand [,operand]...] |

name
: identifies the statement as a compiler-control statement. The plus (+) sign must appear in the first card column of the control statement and must be followed by at least one blank column.

operation
: indicates the function of the control statement. For example, the word COPY specifies that the deck following the COPY card is a precompiled object deck. The operation field must be followed by at least one blank.

operand(s)
: supply additional information about the compiler-control statement. For example, the operand DECK in the PROCESS control statement specifies that the compiled object module is to be punched on cards or written on tape. The operand field may be blank or may contain one or more operands, separated by commas, with no intervening blanks. A blank to indicate the end of the field must follow the last operand in the field. The field must not extend beyond column 71 of a punched card. Operands of a statement may be referred to as options whenever a choice can be made among them.

Whenever a job is to be compiled and link-edited for execution, the first statement read from the device assigned to SYSIPT must always be COPTN, with the option LINK or GODECK. In all other cases the use of the COPTN statement is optional.

The control statement immediately preceding the PL/I source deck must always be the PROCESS statement. The PROCESS statement is mandatory for PL/I source modules. If overlay is used, SEGMENT control statements must precede each segment. If precompiled object modules are used in the input deck, they must be preceded by a COPY statement.

### COPTN Compiler Control Statement (Optional)

Options valid for the whole job are specified in the COPTN control statement which has the following format:

| Name | Operation | Operand(s) |
|---|---|---|
| + | COPTN | option [,option]... |

Note that you may specify one or more COPTN control statements for a job. However, they must precede the whole job and follow each other. In case of conflicting options, the option specified last is the one that is valid.

The possible options shown in Figure 37 (default values underlined) are:

| Option | Function |
|---|---|
| DUMP | causes the contents of main storage and registers to be listed on SYSLST or SYSLOG in case of abnormal termination of the job. |
| NODUMP | suppresses the DUMP option. |
| LINK | causes the source module(s) to be compiled and link-edited. |
| NOLINK | suppresses the LINK and GODECK options. The program is only compiled, not link-edited. |
| WORK1 | specifies that the work file is located on one disk drive. |
| WORK2 | specifies that the work files are located on two disk drives. |
| WORK3 | specifies that the work files are located on three disk drives. |
| LSORT | indicates that, for batch compile mode, the specified listings are to be collected for each procedure. |
| NOLSORT | indicates that, for batch compilation, the specified listings are to be collected for each list type: first all source module listings, then all cross-reference listings, etc. |
| GODECK | specifies that the executable object program is punched or written on SYSOPT. If GODECK is specified, LINK must also be specified. |
| NOGODECK | suppresses the GODECK option. |

Figure 37. Option of COPTN Control Statement

### PROCESS Compiler-Control Statement (for PL/I Source Modules)

Options used for the compilation of a single procedure are specified in the PROCESS statement, which has the following format:

| Name | Operation | Operand(s) |
|---|---|---|
| + | PROCESS | [option [,option]...] |

The possible options shown in Figure 38 (default values underlined) are:

| Option | Function |
|---|---|
| DECK | punches or writes a compiled object module on SYSOPT if no irrecoverable errors have been detected during compilation. |
| NODECK | suppresses the DECK option. |
| SOURCE | lists the source module on SYSLST. |
| NOSOURCE | suppresses the SOURCE option. |
| XREF | causes the PL/I Compiler to list the cross-reference table. |
| NOXREF | suppresses the XREF option. |
| ATRO | lists the offsets of labels and variables and the length of automatic storage. |
| NOATRO | suppresses the ATRO option. |
| ATR | lists the attributes of all variables, entry names, and file names (Symbol-table listing). |
| NOATR | suppresses the ATR option. |
| LIST | lists the object module in symbolic form on SYSLST. |
| NOLIST | suppresses LIST option. |
| WARNING | lists all detected errors and warning messages. |
| ERROR | lists errors and severe errors. |
| SEVERE | lists only severe errors. |
| CHAR48 | informs the compiler that the source module is written in the 48-character set in EBCDIC notation. |
| CHAR60 | informs the compiler that the source module is written in 60-character set in EBCDIC notation. |
| STMT | permits the listing of "the last statement entered" in case of an object time error message. |
| NOSTMT | suppresses the STMT option. |
| EXTREF | prints all external references, names with the attributes EXTERNAL, and library names of this procedure. |

Figure 38. Options of PROCESS Control Statement, Part 1 of 2

| | |
|---|---|
| NOEXTREF | suppresses the EXTREF option. |
| OFFSET | lists the offsets of the beginning of each statement from the beginning of the procedure. This listing is a subset of the LIST option, and cannot be used together with LIST. If the STMT option would take up too much main storage, use the OFFSET option for debugging. It permits the number of the statement that caused the error to be determined by simple computation. |
| NOOFFSET | suppresses the OFFSET option. |

Figure 38. Options of PROCESS Control Statement, Part 2 of 2

## Printed Listings and Diagnostic Aids

(This section contains only preliminary information).

### DIAGNOSTIC CAPABILITIES DURING COMPILATION

PL/I source module diagnostic messages will be given at three levels:

1. Syntax errors will be printed with the PL/I source module listing.
2. Errors detected with data declarations will be printed together with the symbol-table listing (ATR).
3. Other errors will be printed at the end of the compilation with an extra listing.

### DIAGNOSTIC CAPABILITIES DURING EXECUTION

The following diagnostic capabilities will be provided at execution time:

1. A dynamic dump facility that can be called by the statement CALL DYNDUMP (variable-name [,variable-name]...);. This statement dumps data in hexadecimal format, including statement-label and pointer variables. This feature will require a minimum of main storage.

2. Object time error conditions will be printed on a device assigned to SYSLST or SYSLOG, giving an error code, the address where the error occurred, the chain of active procedures and, if applicable, the address of the file that caused the error.

3. If the ERROR condition has been raised, the standard system action must be taken. If the DUMP option was specified at compile time, a hexadecimal main storage dump will be taken before the end of job is called. All files are closed automatically before the end of the job.

4. In case of a hardware stop, the operator can start the same action as described under 3.

5. If the STMT option was specified at compilation time, actions 2 to 4 above are extended by printing the statement number of the last PL/I statement entered.

## Executing a Simple PL/I Program

Figure 39 shows a simple, but complete PL/I program which reads cards and prints them. It is shown with the complete set of job-control statements needed to run the program. The job-control statements used are discussed below in the sequence in which they appear.

//_JOB_PL1,FIRST: indicates to the Job Control program that (1) a set of job-control statements and a PL/I program are to follow, that (2) the PL/I Compiler is to translate the source program into an object module, and that (3) the PL/I program is to be executed immediately.

The JOB statement must be specified for every program. It indicates to the Monitor what kind of job is to be done.

//_DATE_70002: specifies the day of the year (January 2nd, 1970). The DATE control statement must appear in the first job following initial program loading.

The DATE statement is used for label checking and for dating output reports.

//_ASSGN_SYS000,X'801',D3: specifies that 2311 Disk drive, Model 11, (identified by D3) will hold the work area for the compilation of the program. SYS000 is the symbolic device address to be used by the compiler to refer to the address of the disk device holding the work area whose location (actual device address) is 01.

A work area must always be assigned for a compilation.

//_ASSGN_SYSIPT,X'100',R4: specifies that the PL/I source program is to be read from the 2501 card reader (identified by R4) whose symbolic device address is SYSIPT.

Input to the compiler is always read from SYSIPT.

//_ASSGN_SYSLST,X'400',L1: assigns the 1403 printer to the symbolic device address SYSLST on which the output of the PL/I compiler is to be printed.

It is assumed that SYS000, SYSIPT, and SYSLST had other assignments in the previous job or, if this is the first job, that they had other standard assignments. Otherwise, the ASSGN statements would not be needed.

//_VOL_SYS000,WORK1: The VOL (volume), DLAB (disk label), and XTENT cards must be supplied because a disk work file is used. The VOL control statement indicates the symbolic address of the device (SYS000) and name (WORK1) of the work file to be used.

//_DLAB... contains the standard IBM disk label for the file (WORK1) named in the VOL control statement that precedes the DLAB control statement.

//_XTENT... specifies the purpose of the extent, the extent sequence number, the location of the extent etc.

For a full description of the label control statements VOL, DLAB, and XTENT see the section Input/Output.

//_EXEC: indicates to the Job Control program that the reading of the control statements has been completed and that control is to be returned to the Monitor. The Monitor then transfers control to the PL/I compiler, which begins reading and compiling the source program from the card reader.

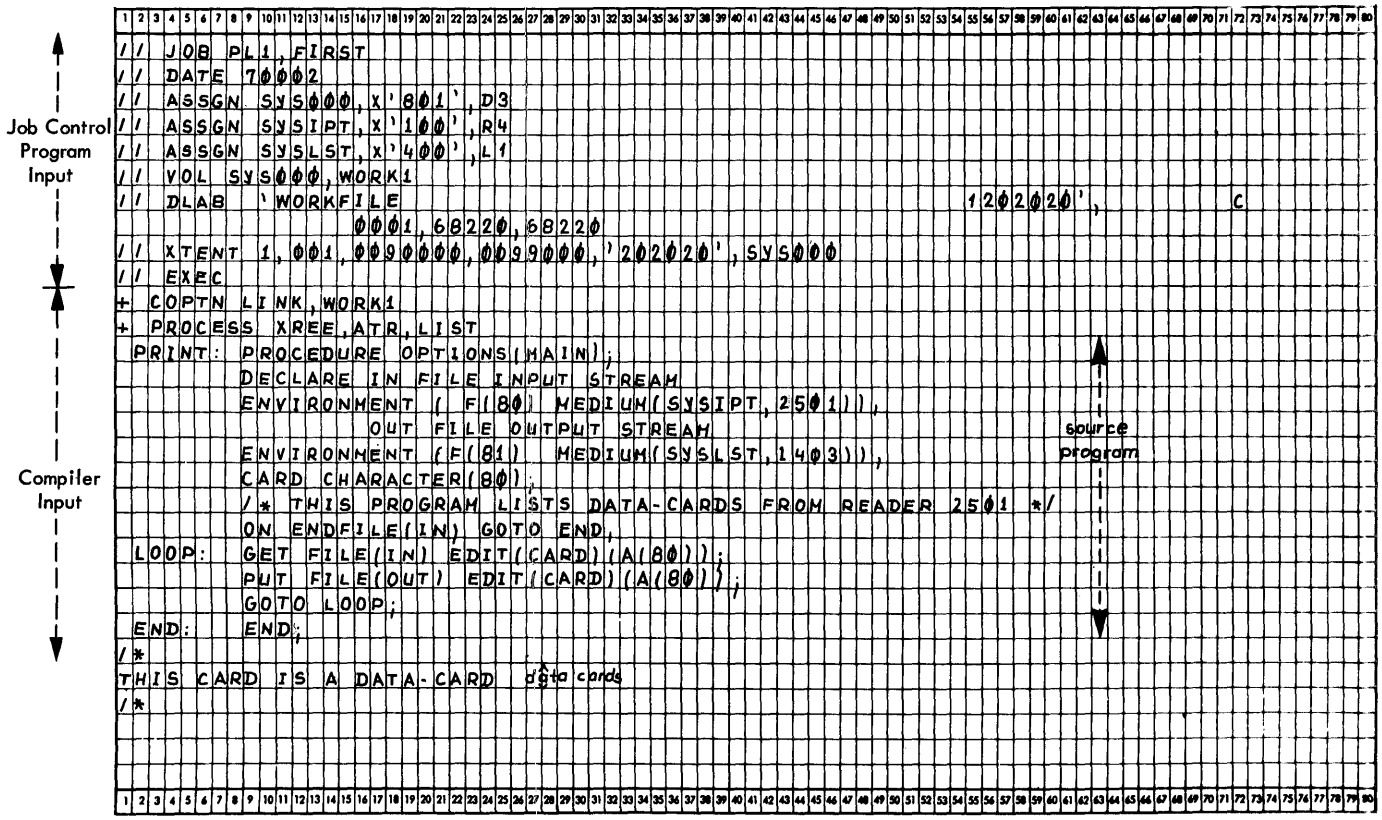
+_COPIN_LINK,WORK1: specifies that the source program is to be compiled and link-edited (LINK). The option WORK1 specifies that one work file is to be used. Since the option NOLINK is applied by default, LINK must always be specified if the job is to be compiled and link-edited to form an executable object program.

+_PROCESS_XREF,ATR,LIST: causes the compiler to print the cross-reference table (XREF), the source program (by default), the symbol table (ATR), and the object module (LIST) on the device assigned to SYSLST. The input on SYSIPT will be (by default) in 60-character set EBCDIC notation.

/* signifies the end of the source program.

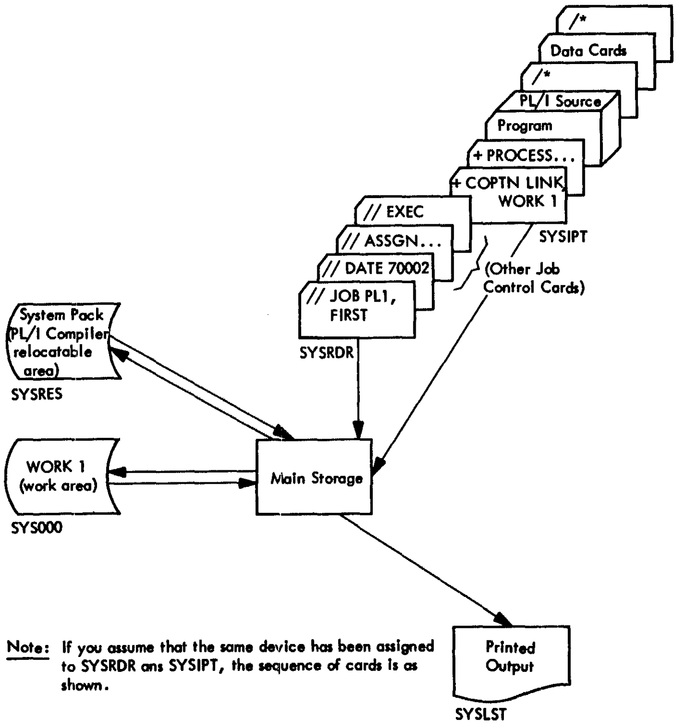
/* signifies that no more data cards follow.

Figure 40 illustrates the arrangement of Input/Output devices that would be needed for compilation and execution of the program shown in Figure 39.


Job Control Program Input
Compiler Input
source program
data cards

```
// JOB PL1,FIRST
// DATE 7ØØØ2
// ASSGN SYSØØØ,X'8Ø1',D3
// ASSGN SYSIPT,X'1ØØ',R4
// ASSGN SYSLST,X'4ØØ',L1
// VOL SYSØØØ,WORK1
// DLAB  'WORKFILE                                    12Ø2Ø2Ø',        C
               ØØØ1,6822Ø,6822Ø
// XTENT 1,ØØ1,ØØ9ØØØØ,ØØ99ØØØ,'2Ø2Ø2Ø',SYSØØØ
// EXEC
+ COPTN LINK,WORK1
+ PROCESS XREE,ATR,LIST
 PRINT: PROCEDURE OPTIONS(MAIN);
        DECLARE IN FILE INPUT STREAM
        ENVIRONMENT ( F(8Ø) MEDIUM(SYSIPT,25Ø1)),
                OUT FILE OUTPUT STREAM
        ENVIRONMENT (F(81)  MEDIUM(SYSLST,14Ø3)),
        CARD CHARACTER(8Ø);
        /* THIS PROGRAM LISTS DATA-CARDS FROM READER 25Ø1 */
        ON ENDFILE(IN) GOTO END,
 LOOP:  GET FILE(IN) EDIT(CARD)(A(8Ø));
        PUT FILE(OUT) EDIT(CARD)(A(8Ø));
        GOTO LOOP;
 END:   END;
/*
THIS CARD IS A DATA-CARD
/*
```

Figure 39. PL/I Program, with a Complete Set of Job-Control Statements



Note: If you assume that the same device has been assigned to SYSRDR ans SYSIPT, the sequence of cards is as shown.

Figure 40. Arrangement of I/O Devices for the PL/I Program shown in Figure 39

(At the present stage, this section contains only preliminary information).

## Overlay Facility

With the Model 20 PL/I, you can divide (segment) large programs into segments that can all be executed one segment at a time in one job. This feature allows you to save main storage and reduce the overall requirements of a program.

Suppose you have a program consisting of many procedures that would not fit into main storage without partitioning. By means of SEGMENT compiler-control statements, which you must include between procedures in the input deck, you can partition your program into segments that can -- at execution time -- be successively called into main storage and executed.

A segment may consist of one or more procedures. Segments are called into main storage by the calling (fetching) procedure with a statement of the format CALL OVERLAY ('segment-name'). The called segments successively overlay each other, except the first, the root segment, which must remain active and reside in storage throughout execution of the problem program.

When overlay is used, the second segment occupies the main storage immediately above the first segment. The third, fourth, etc., segments start, when they are overlaid, at the same location where the second segment started after loading. Each segment, including the first (root) segment, must be preceded by the SEGMENT compiler control statement, which has the following format:

| Name | Operation | Operand(s) |
|---|---|---|
| + | SEGMENT | name [,option] |

operand(s)

name is the segment name, which has the same syntax as an external PL/I name, except that break characters are not allowed. The segment name is independent of any procedure name.

option is a decimal integer constant specifying the load address of the root segment. It may optionally follow the name of the root segment. If this option is not specified or, if the program is not segmented, the SEGMENT statement is missing, the executable object program will be loaded immediately behind the end of the Monitor.

### CREATING A SEGMENTED PROGRAM

To segment a program, include SEGMENT control statements between procedures in your input deck; one or more procedures may follow each SEGMENT card.

Figure 41 illustrates how you might segment a program to be compiled, link-edited, and executed:

```
// JOB PL1,EXAMPLE
     .
     .  (other job-control statements)
     .
// EXEC
+   COPTN LINK
+   SEGMENT ROOT
+   COPY
     .
     .  (precompiled object module)
     .
+   PROCESS NODECK
     .
     .  (PL/I source module)
     .
+   COPY
     .
     .  (precompiled object module)
     .
+   SEGMENT Y
+   COPY
     .
     .  (precompiled object module)
     .
+   SEGMENT X
+   PROCESS NODECK,NOXREF
     .
     .  (PL/I source module)
     .
/*
```

Figure 41. Segmenting a PL/I Program

### LOADING SEGMENTS

During program execution, the individual segments are loaded by the fetching procedure(s) with a statement of the format

CALL OVERLAY ('segment-name');

CALL OVERLAY does not automatically invoke a procedure in the newly loaded segment. This has to be done by a normal CALL statement or function reference.

The fetching procedure must always be contained in the root segment, never in one of the segments that are overlaid during program execution.

However, within a segment, one procedure may call another procedure. Figure 42 illustrates how segments are fetched and executed.

```
// JOB PL1,EXAMPLE
     .
     .  (other job-control statements)
     .
// EXEC
+   COPTN LINK
+   SEGMENT ROOT
+   PROCESS NODECK
    A: PROCEDURE OPTIONS (MAIN);
       DCL (data-items) EXTERNAL;
       ON ENDFILE (filename) action;
    BEGIN: CALL OVERLAY ('SEGM1');
           CALL B1;
           CALL OVERLAY ('SEGM2');
           CALL B2;
           CALL B4;
             .
             .  (PL/I source statements)
             .
    END;
+   SEGMENT SEGM1
+   PROCESS NODECK
    B1:PROCEDURE;
       DCL (data-items) EXTERNAL;
              .
              .  (PL/I source statements)
    RETURN;
    END;
+   SEGMENT SEGM2
+   PROCESS NODECK
    B2:PROCEDURE;
       DCL  (data-items) EXTERNAL;
              .
              .  (PL/I source statements)
              .
       CALL B3;
       END;
+   PROCESS
    B3:PROCEDURE;
       DCL (data-items) EXTERNAL;
              .
              .  (PL/I source statements)
       END;
+   PROCESS
    B4:PROCEDURE;
       DCL (data-items) EXTERNAL;
              .
              .  (PL/I source statements)
       END;
/*
```

Figure 42. Fetching PL/I Program Segments

In this example, the main procedure A of the program (contained in the root segment) first loads SEGM1 and calls procedure B1 (contained in SEGM1) for execution. After B1 has returned control to A, A loads SEGM2 and successively calls the two procedures B2 and B4 for execution. However, before B2 returns control to A, it calls B3 for execution, which returns control to B2. B2, B3, and B4 are contained in SEGM2.

The structure of the overlay scheme of the above example is shown in Figure 43.

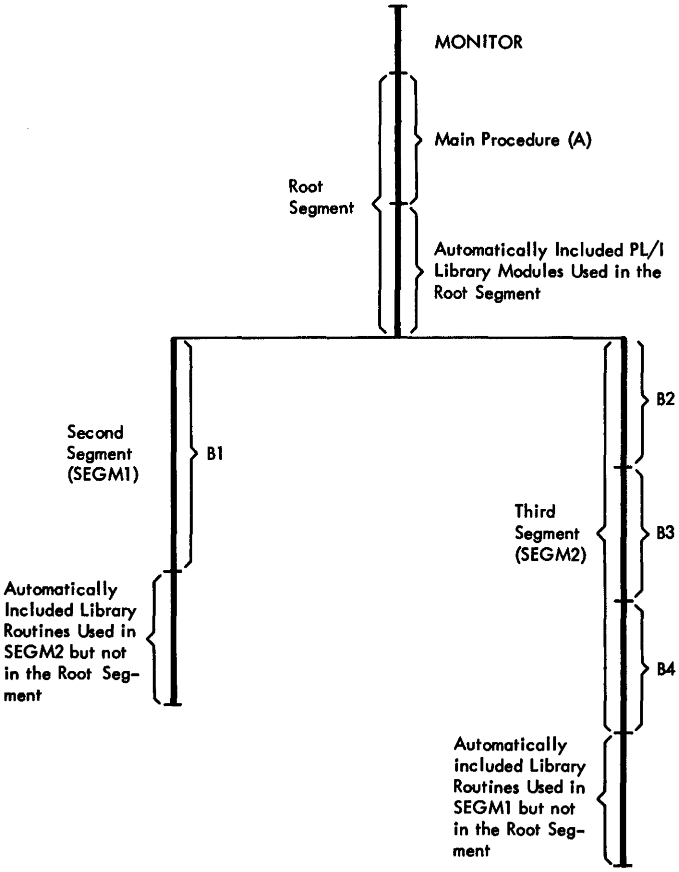

MONITOR
Main Procedure (A)
Root Segment
Automatically Included PL/I Library Modules Used in the Root Segment
Second Segment (SEGM1)
B1
Automatically Included Library Routines Used in SEGM2 but not in the Root Segment
B2
Third Segment (SEGM2)
B3
B4
Automatically included Library Routines Used in SEGM1 but not in the Root Segment

Figure 43. Structure of Overlay Scheme

## USE OF FILES AND STATIC STORAGE IN SEGMENTS

### Use of Files

If you use files in the different segments of your program, observe the following rules:

If (1) a file is declared in a procedure that is not contained in the root segment, (2) this file is opened in this procedure, and (3) the segment in which this procedure is contained is about to be overlaid with another segment, close this file before the segment is overlaid. However, this restriction does not apply if the file is not only declared in the fetched segment in which it is opened, but also in the root segment.

### Use of Static Storage

If you have data that is to be used in more than one segment, you must give the data the EXTERNAL attribute (which implies the STATIC attribute) or transmit them through argument lists of the CALL statement. For larger volumes of data, the use of the EXTERNAL attribute generally requires less storage than argument transmission. These data must be declared EXTERNAL in the root segment as well. However, where the arguments change, argument transmission is normally more useful.

## RULES FOR USING OVERLAY

This section gives a summary of the rules to be observed when using overlay:

1. After a segment has been fetched into main storage, the procedure(s) contained in it may be activated by means of a call to the name of the procedure or, if more than one procedure is contained in the segment, by a call to one of the procedure names.

2. The segment name is independent of any procedure name. It is assigned by means of the SEGMENT compiler-control statement contained in the input deck.

3. CALL OVERLAY statements must always be contained in the root segment.

4. The root segment cannot be overlaid. It is in storage throughout execution of the problem program.

5. Any procedure of a segment may be activated at any time after the segment has been loaded, provided it has not been overlaid.

6. Fetching a segment already fetched into main storage causes the segment to be loaded again. The variables contained in the segment that are in static storage have no known values or the values specified in an INITIAL attribute.

7. Names of segments to be fetched must be unique. Names of procedures in the whole program must also be unique.

8. A library subroutine is automatically included in every segment in which it is used if (a) it is used in a segment other than the root segment and (b) it is not in the root segment.

9. Data referred to in more than one segment must either be given the EXTERNAL attribute in each segment in which they are used and in the root segment or they must be transmitted as arguments with the CALL statement.

10. Files declared and opened in a segment below the root segment but not declared in the root segment also must be closed before the segment is overlaid by another segment.

# Input/Output

In this section, we will discuss and illustrate how to create a file and how to relate a file on an external medium to your program.

The section headed *Data Transmission* in Part I is prerequisite for the understanding of this section.

Let us briefly repeat what a file is and how you can relate it to your program.

A *file* is an organized collection of related data external to a program. A file is stored on an external medium, generally referred to as a *volume*, such as a reel of magnetic tape or a disk pack. One reel of magnetic tape or one disk pack, for example, i.e., a valume, may contain one or more files or parts of a file. A volume that contains more than one file, is called a *multi-file volume*. If more than one volume is needed to hold one file, we speak of a *multi-volume file*.

The individual data items in a file are arranged in distinct physical groupings called *blocks*. For processing purposes, each block consists of one or more logical parts called *records*, each of which can contain one or more related 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 parts, the logical parts are called *logical records*.

When a physical record contains two or more logical records, we say the records are *blocked*. Blocking permits a more compact and efficient use of external storage and a faster access to the logical records.

To be able to deal with the data items of a file, that is, to read them into main storage or to write them onto an external storage medium, a relationship has to be established between a file and a program. You do this by means of a *file declaration*, that is, by declaring a *file name* for the file to be processed in your program and by specifying *attributes* that describe the file and the manner in which it will be handled. You establish the connection between the device on which the file resides and the program in main storage by means of the MEDIUM option in the ENVIRONMENT attribute in the file declaration and, if necessary, with ASSGN control statements. (For the ASSGN control statement, see also the section *Job Control*).

Unlike a file, however, a file declaration has significance only in a program. For example, if you use the same file again, you may specify a different file name and some different attributes for it.

## File Organization

The organization of a file determines how data is recorded in a file and how data may be retrieved from a file to be transmitted to a program for processing.

For the Model 20 PL/I, a file may be of *CONSECUTIVE organization*, in which case logical records can be stored in and retrieved from a file in sequential order, or it may be of *INDEXED organization*, in which case logical records may be retrieved from or stored in a file either in sequential or direct order, that is, on the basis of key values specified in the data- transmission statements.

Files of CONSECUTIVE organization may be read or written in either stream- or record-oriented transimission. INDEXED files may be read or written in record-oriented transmission only.

### *Record Formats*

Logical records can appear in one of three formats: *fixed-length* (format F), *variable-length* (format V), or *undefined-length* (format U). One of these record formats must be specified. They provide flexibility in the design of files and allow you to take advantage of the fixed-length and variable-length features of specific input/output devices.

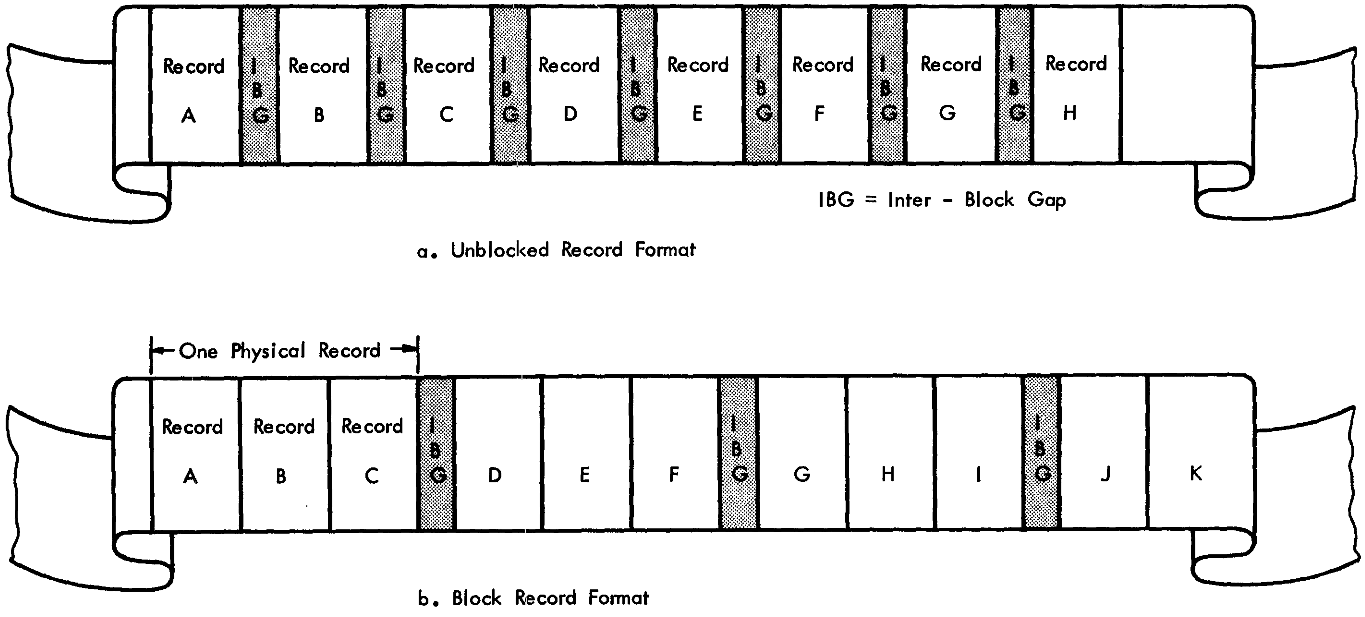
The block size and the record size are specified as the number of bytes in a block or record, respectively. For *format-F records*, if the record size is not specified in the ENVIRONMENT attribute, the records are assumed to be unblocked. The block size must be specified. The record size may be specified for format F records only. Blocking and deblocking is handled automatically.

With format F records, blocking is based on the stated record size. The block size must be an integer multiple of the record size.

With format_V records, deblocking is based on information at the beginning of each block and at the beginning of each logical record. Four bytes are used at the beginning of each block to specify block length, and another four bytes are used at the beginning of each record to specify the length of that record. Although insertion of this length information is done automatically by the system when the file is created, you must include the number of control bytes when you specify the length of the block in the ENVIRONMENT attribute. When format V files are created, records are always blocked if their lengths allow two or more to be placed into a block smaller than or equal to the maximum that is specified.

With format-U records, each block consists of only one record. The blocks (records) are of varying lengths. No system control bytes appear anywhere within the block. All processing of record is your responsibility. If you include a length specification in the record, you must insert it yourself, and must retrieve the information yourself.

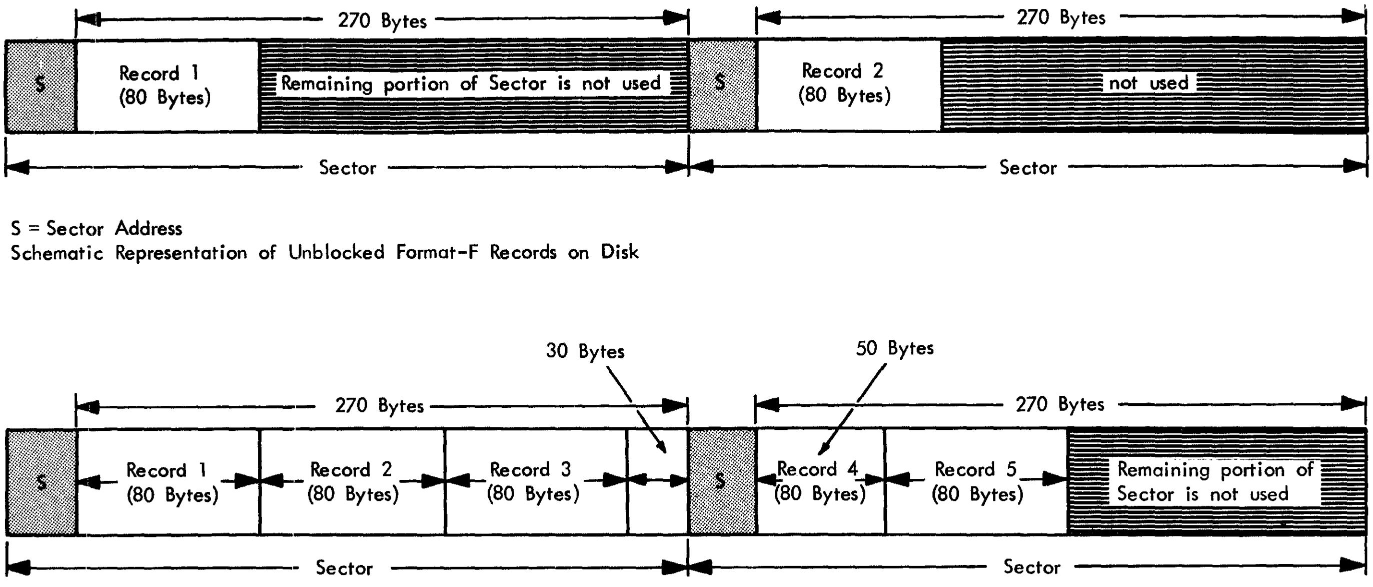
Figures 44 through 46 illustrate record formats.



a. Unblocked Record Format

b. Block Record Format

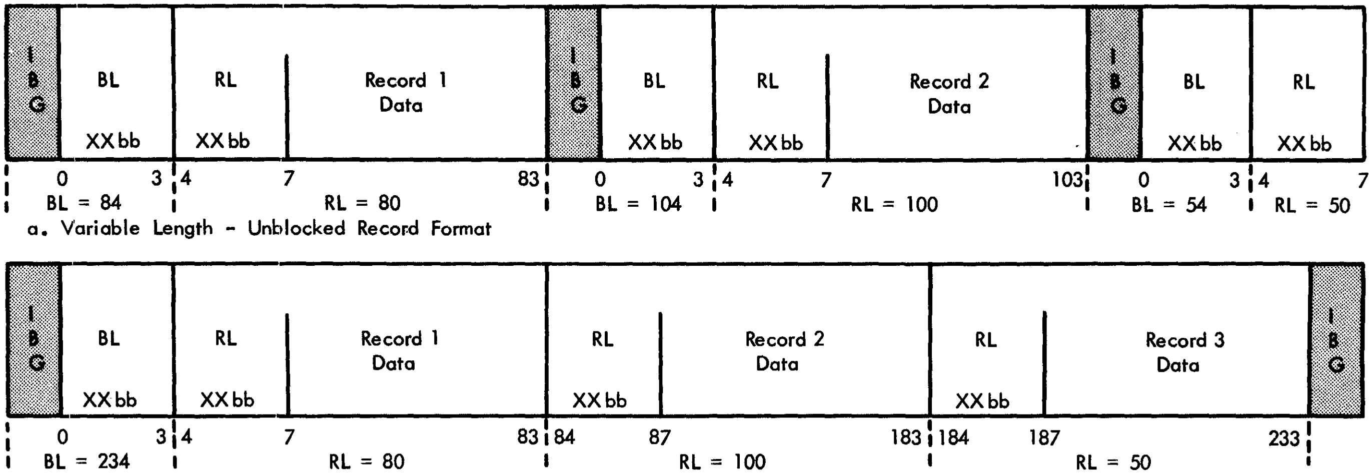
Figure 44. Example of Format-F Records on Magnetic Tape



S = Sector Address
Schematic Representation of Unblocked Format-F Records on Disk

S = Sector Address
Schematic Representation of Blocked Format-F Records on Disk, assuming five Records per Block.

Figure 45. Example of Format-F Records on Disk



a. Variable Length - Unblocked Record Format

b. Variable Length - Blocked Format

BL is Block Length
RL is Record Length
IBG is Inter-block Gap

Figure 46. Example of Format-V Records on Magnetic Tape

## CONSECUTIVE FILE ORGANIZATION

In a CONSECUTIVE file, the logical records are organized on the basis of their successive physical positions, such as they appear on magnetic tape. Records can only be retrieved in sequential order. Therefore, the associated files must either be declared with the SEQUENTIAL attribute or be STREAM files.

### Attributes and Options

The file attributes, which are part of the PL/I language, are not discussed in detail here. They are explained in Part I, in the section *Data Transmission*, and in Part II, in the section *Attributes*.

The *attributes* you may specify for CONSECUTIVE files are:

FILE<br>
RECORD<br>
STREAM<br>
SEQUENTIAL<br>
INPUT<br>
OUTPUT<br>
UPDATE<br>
PRINT<br>
BACKWARDS<br>
ENVIRONMENT (options-list)

The *options* you may specify in the ENVIRONMENT attribute for CONSECUTIVE files are:

```
     [CONSECUTIVE]
    (F (blocksize [,recordsize]))
    <V (maxblocksize)           >
    (U (maxblocksize)           )
    [BUFFERS ({1|2})]
     MEDIUM (symbolic-device-address,
             device-type)
    [CTLASA]
    [LEAVE]
    [NOTAPEMK]
    [NOLABEL]
    [VERIFY]
    [ALTTAPE]
```

The individual options -- which are not part of the PL/I language, but Model 20 compiler keywords -- are discussed in the two programming examples shown in Figures 47 and 48 in the sequence in which they appear.

A complete list of all options you may use for CONSECUTIVE as well as INDEXED files is given later in this section under the heading *The ENVIRONMENT Attribute and its Options*. See also *Appendix E. File Attributes and Options*.

### Record Formats

Records in consecutive files may be of format F, format V, or format U. The format V may be used only for record-oriented I/O, and only with tape units. However, format-V records cannot be read backwards. The format U may only be used for magnetic tape and printer-keyboard I/O. It must be used with printer-keyboard I/O. With magnetic tape I/O you may use format U with record-oriented transmission only.

### Input/Output Devices

Input/output devices permitted for CONSECUTIVE files include magnetic-tape drives, card readers and punches, a printer-keyboard, disk-storage devices, and printers.

### Creating CONSECUTIVE Files

Following now are two simple programming examples that illustrate the use of files of CONSECUTIVE organization, their attributes and options.

The *first example*, shown in Figure 47, demonstrates the use of the ASA control character for record-oriented output. With RECORD OUTPUT files that are to be printed, you cannot use the SKIP or PAGE format items or options, which are used for printer-carriage control with STREAM PRINT files (see second example, Figure 48). With the CTLASA option specified in the ENVIRONMENT attribute, you can, however, achieve the same effect with RECORD OUTPUT files that are to be printed. The first program uses multi-valume tape input and printed output.

*Explanation*:

1 *Assignments*: The alternate tape drive must always be assigned to a symbolic device address that is one higher than that specified in the MEDIUM option.

2 *Labels*: For an explanation of file labels refer to *File Labels* in this section.

3 *Relating the Program to the Input File*: The program LISTTAPL -- which uses multi-volume file tape input -- is related to the input file by means of the FILE declaration which associates the name INP with the file and declares it to be an INPUT file that is to be transmitted in RECORD-oriented mode. The MEDIUM option of the ENVIRONMENT attribute in the file declaration specifies that the file is located on a 2415 9-track tape drive assigned to the symbolic device address SYS004. The records have a fixed length of 120 bytes and are blocked 20 logical records to a block. On input, the blocked records are automatically deblocked so that the program deals only with

```
    // JOB PL1,LISTTAPE
    // ASSGN SYS004,X'780',T2,X'C0'
 1  // ASSGN SYS005,X'782',T2,X'C0' ALTERNATE TAPE UNIT
    // VOL SYS004,INP
 2  // TPLAB 'TRANSACTION FILE 00111100010001000700 70010 71009'
    // EXEC
    +  COPTN LINK,WORK2
    +  PROCESS LIST,CHAR60,ATR,NODECK
     LIST: PROCEDURE OPTIONS(MAIN);
 3         DCL INP FILE RECORD INPUT ENV (MEDIUM (SYS004,2400)
                      F(2400,120) ALTTAPE),
 4         PRT FILE RECORD OUTPUT ENV (MEDIUM (SYSLST,2203)
 5                    F(121) CTLASA),
           1 INRCD BASED (P),
             2 KEY CHAR (8),
             2 FILLER CHAR (112),
           INSTREAM BASED(P) CHAR (120),
           1 OUTRCD,
 6           2 CTLASA_CONTROL CHAR (1),
             2 TEXT CHAR (120),
           (LINE FIXED (3) INIT (57), OLDKEY CHAR (8), P POINTER) STATIC;
 7         OPEN FILE (INP), FILE (PRT);
 8         ON ENDFILE (INP) GOTO FINIS;
     LOOP: READ FILE (INP) SET (P);
           IF LINE > 56 THEN DO;
 9                    CTLASA_CONTROL = '1';
                      LINE = 1;
                      END;
                      ELSE IF KEY = OLDKEY THEN DO;
10                         CTLASA_CONTROL = ' ';
                           LINE = LINE + 1;
                           END;
                                 ELSE DO;
11                                 CTLASA_CONTROL = '0';
                                   LINE = LINE + 2;
                                   END;
           TEXT = INSTREAM;
           WRITE FILE (PTR) FROM (OUTRCD);
           OLDKEY = KEY;
           GO TO LOOP;
     FINIS: END;
    /*
```

Note: The numbers to the left of Figure 47 are used for explanation purposes only. They are not part of the coding.

Figure 47. Programming Example Illustrating the Use of the ASA Control Characters for RECORD OUTPUT

logical records. Since a multi-volume tape file is used, the option ALTTAPL must be specified in the ENVIRONMENT attribute. ALTTAPE indicates that an alternate tape drive is assigned to enable automatic switching from one drive to the other.

4 Relating the Program to the Output File: The connection between the program and the device on which the output file is to be stored is established by means of the FILE declaration for PRT which declares PRT to be an OUTPUT file that is to be printed on a 2203 printer assigned to the symbolic device address SYSLST. The records are unblocked with a length of 121 bytes, the first being reserved for the ASA control character. An ASSGN statement for SYSLST is not needed since the device assignments for printers are taken from the device specification in the MEDIUM option only.

7 Opening the Files. Both the input file INP and the output file PRT are opened (activated) which means that the file declarations for the two files are associated with the respective actual files on the external storage medium and that processing of the records can begin.

5 *Use of CTLASA*. The option CTLASA in the ENVIRONMENT attribute of a RECORD file specifies that the first character of a logical record transmitted to the output file is to be interpreted as an ASA control character. When RECORD output is to be on a printer or on cards, the ASA carriage control character is used for printer carriage control and for punched card stacker selection. (The character codes that you can use with CTLASA are shown in *The ENVIRONMENT Attribute and Its Options* in this section).

6 In the structure declaration for the output record OUTRCD, the first character is reserved for the ASA control character.

9 Depending on the result of comparison operations, the ASA control character is either set to '1' which is interpreted as "skip to channel 1 before printing", when the output record is printed,

10 or set ot ' ' (blank) which is interpreted as "space one line before printing",

11 or to '0' which is interpreted as "space two lines before printing".

8 *De-Activating the Program*: The processing loop ends automatically when the ENDFILE condition is raised for the input file INP. Control is transferred to the statement labeled FINIS by means of the ON statement. When control is returned to the Monitor by execution of the last END statement of the program, both files are automatically closed.

The *second programming example*, shown in Figure 48, illustrates printer-carriage control for STREAM files. Printer-carriage control for STREAM files is achieved by means of the SKIP option in the PUT statement.

The second program (LISTBACK) uses tape input that is on a multi-file tape volume which is to be read backwards in record-oriented transmission mode.

*Explanation*:

1 The job-control statement // JOB PL1, LISTBACK specifies that the program LSTBACK is to be compiled, link-edited, and executed immediately. For an explanation of the other job-control statements, see the section *Job Control* and the paragraph headed *File Labels* in this section. For the compiler-control statements, see the section *The Compiler*.

2 The input file that is to be processed by LISTBACK is related to LISTBACK by means of the FILE declaration which declares the file to be an INPUT file that is to be read BACKWARDS from a

```
      // PAUSE  LOAD REEL 118 ON UNIT 81
      // JOB PL1,LISTBACK
      // ASSGN SYS007,X'781',T2
  1   // VOL SYS007,BACKT
      // TPLAB 'BACKWARDS EXAMPLE0001180001000200010l 70111 70113'
      // FILES SYS007,6
      // EXEC
      +  COPTN LINK,WORK2
      +  PROCESS LIST,ATR,NODECK
       LISTB: PROCEDURE OPTIONS (MAIN);
              DCL BACKT FILE INPUT RECORD BACKWARDS ENV
  2                 (MEDIUM (SYS007,2400) F (1500,100) LEAVE),
                 PRT FILE OUTPUT PRINT ENV (MEDIUM (SYS003, 2400)
  3                 F(121) NOLABEL NOTAPEMK),
                 1 INRCD BASED (P), 2 KEY CHAR(8), 2 TEXT CHAR(92),
                 (SKIP FIXED(1), P POINTER, OLDKEY CHAR(8))STATIC;
  4           OPEN FILE (BACKT), FILE (PRT) PAGESIZE (48);
              ON ENDFILE (BACKT) GO TO FINIS;
  5           OLDKEY = HIGH(8);
       LOOP:  READ FILE (BACKT) SET (P);
              IF KEY = OLDKEY THEN SKIP = 1; ELSE SKIP = 2;
  6           PUT FILE (PRT) SKIP (SKIP) EDIT (KEY, TEXT) (A,X(5),A);
              OLDKEY = KEY;
              GO TO LOOP;
       FINIS: END;
      /*
```

Figure 48. Programming Example Showing Printer-Carriage Control for STREAM Files

multi-file tape volume. The file BACKT is on the multi-file tape volume assigned to the symbolic-device address SYS007. (It is the file following the sixth tapemark on the volume (FILES control statement).)

The records are of blocked format, with 15 logical records, each of a length of 100 bytes. On input, the records are automatically deblocked, so that the program deals only with the logical records.

The LEAVE option in the ENVIRONMENT attribute is used to specify that no rewind operation is to be performed when the file BACKT is opened. You should always specify it for files that have the BACKWARDS attribute to ensure proper positioning of the file.

3 The file PRT that is to be created by LISTBACK is declared as an OUTPUT tape file that is to be printed. The attribute PRINT specifies that the first byte of each logical record of the STREAM file is to be reserved for an ASA control character needed for printer-carriage control. The records are unblocked and of fixed length. Since output is stream-oriented, in this example, the records must not be blocked.

The options NOLABEL and NOTAPEMK in the ENVIRONMENT attribute specify that no label processing is to be done for the file. On output, a tapemark is automatically written as the first record on the tape, unless NOTAPEMK is specified in the ENVIRONMENT attribute.

4 Both the input and the output file are opened by means of the OPEN statement which specifies the names of the files to be opened. The PAGESIZE option specifies that, before writing beyond the 48th line, the ENDPAGE condition is raised. Since no appropriate ON statement is given, standard system action, i.e., advancing to a new page, is taken.

5 OLDKEY is assigned an initial value of 16 hexadecimal Fs.

6 A printer-carriage control character is created preceding each output record on tape.

### Accessing a CONSECUTIVE File

An existing CONSECUTIVE file can be accessed in two ways: either as an INPUT or as an UPDATE file.

Reading of an INPUT file may be either stream- or record-oriented. If the file is on magnetic tape, it may be read either forward or backwards.

Transmission of an UPDATE file must always be record-oriented. An UPDATE file cannot have the STREAM, BACKWARDS, or PRINT attributes. The attribute UPDATE specifies that the record is to be read, processed, and rewritten into its previous place in the existing file. Only disk files may be updated.

It is not possible to insert additional records into an existing CONSECUTIVE file or to extend it by adding records at its end.

## DISK ORGANIZATION

A file of INDEXED organization is always on a direct-access device. For the Model 20 System, this is a 1316 Disk Pack used with the 2311 Disk-Storage Drive Model 11 or 12.

Before we come to the discussion of INDEXED file organization, we will, therefore, first deal with the organization of the IBM 1316 Disk Pack, as shown in Figure 49.

### The IBM 1316 Disk Pack

The 1316 disk pack is the medium on which data is stored externally. It is mounted on a 2311-11 or 2311-12 disk-storage drive. Let us consider the physical characteristics of the 1316.

A 1316 disk pack consists of six disks. The top surface of the upper disk and the bottom surface of the lowest disk are not used, which leaves ten surfaces for storage of data. Each disk surface has 203 concentric tracks of which, on a 2311-11, you may use tracks 004 to 202 whereas, on a 2311-12, you may use only tracks 004 to 102. Track 1, 2, 3, etc., on each surface is physically located below or above track 1, 2, 3, etc., of the other surfaces. Each group of ten vertically aligned tracks can therefore be said to form an imaginary cylinder. 200 or 100 cylinders, respectively, are used for actual recording depending on the disk pack used; the remaining three are reserved. (Refer to Figure 49).

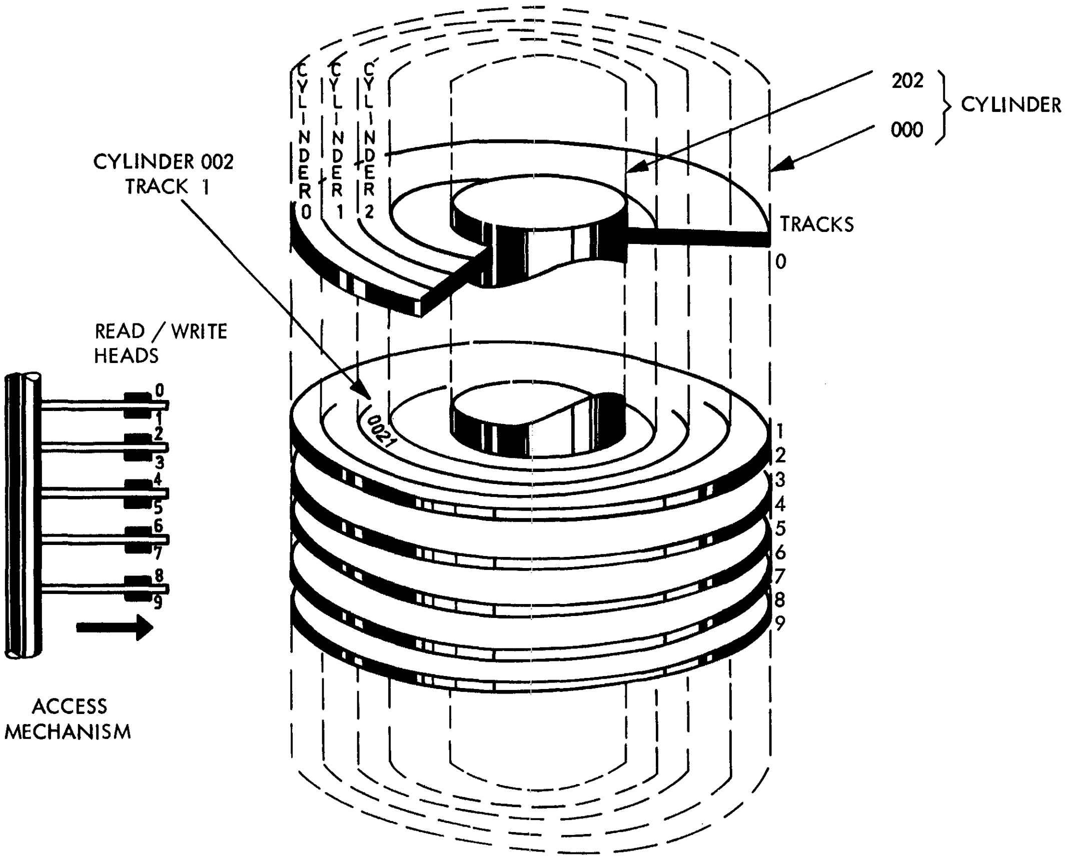


Figure 49. 1316 Disk Pack and Access Mechanism

The 2311 has one access arm, equipped with ten read/write heads. These heads are aligned vertically so that data contained in one and the same cylinder can be accessed without any mechanical movement of the arm from track to track. This, however, makes it necessary to switch from surface to surface within a cylinder when a file being written overflows from track to track or when one track has been read and the file continues on the next. When a cylinder is filled with data, writing continues on the first track of the next cylinder. This technique reduces the time needed to move the access arm.

Thus, a disk pack is thought of as consisting of 200 or 100 cylinders, each cylinder in turn consisting of ten tracks. A consecutive set of tracks or cylinders set aside for usage of a specific file is referred to as an *extent*.

Each track consists of ten *sectors*. Each sector can accommodate 270 bytes of data. When reading records from, or writing them onto disk, one or more complete sectors are always read or written. Parts of sectors cannot be transferred. A single read or write operation may transfer the contents of one 270-byte sector, of several sectors of a single track, or of several sectors of two or more tracks within the same cylinder. A single read or write command must not exceed cylinder limits.

The fact that only one or more *complete* sectors can be read or written is important in dealing with disk-file organization.

All records written on disk must be of fixed length (format F). They may be blocked or unblocked. Only one logical record at a time is available for processing by the problem program. However, data may be transferred between a disk file and main storage in blocks of two or more logical records.

For example, assume that a file is specified to contain unblocked logical records of 160 bytes in length. In this case, the first 160 bytes of each sector contain one record. Note that 110 bytes of each sector are wasted. The input/output area that is automatically established in main storage is 270 bytes in length to accommodate one complete sector, even though only 160 bytes of each sector are utilized by a logical record. This is obviously wasteful. In this example, unblocked records result in poor utilization of disk capacity. An additional disadvantage is a delay in retrieval of records (slower retrieval times as compared with retrieval of blocked records).

Figure 50 shows the possibilities of blocking records to achieve good utilization of the disk area.

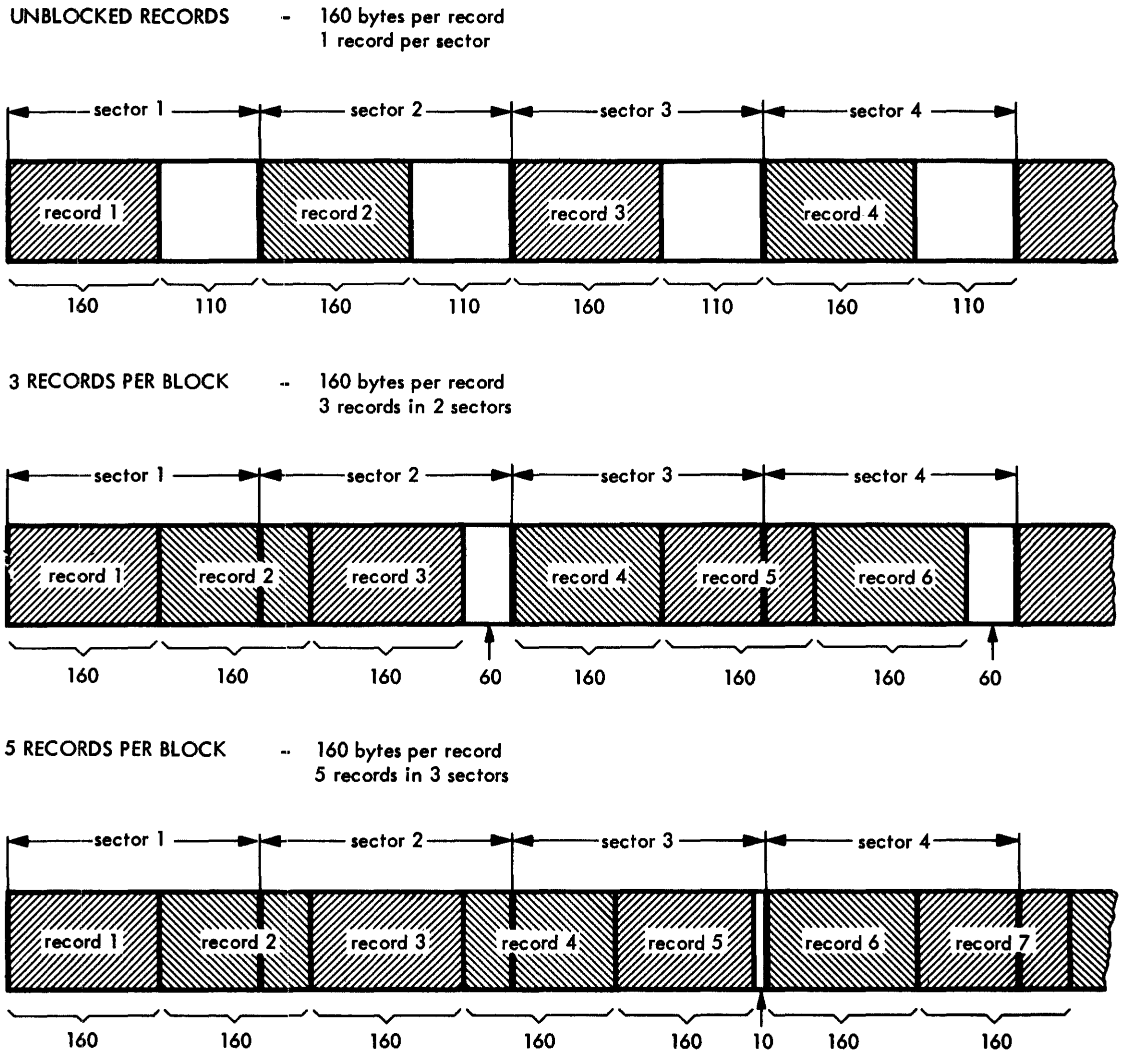


Figure 50. Comparison of Blocked and Unblocked Records

## INDEXED FILE ORGANIZATION

For the Model 20, a file of INDEXED organization is always on a 1316 disk pack mounted on the 2311-11 or the 2311-12 disk-storage drive. The structure of an INDEXED file is basically sequential. Its records are arranged in logical sequence according to keys that are associated with every record. The key must be a character string that represents an item within the record, such as a date or a name. Each key must be unique. Records may be retrieved either sequentially or directly.

INDEXED file organization offers the following additional features not provided by CONSECUTIVE file organization:

- Sequential retrieval of records within specified limits in the file.
- Direct retrieval.
- Extension of the file at its end.
- Insertion of new records without copying the whole file.

When an INDEXED file is created (or loaded), it must be created as an INDEXED SEQUENTIAL OUTPUT file. INDEXED DIRECT files cannot be output files. The records of the file to be written must be presorted to the required sequence. Once an INDEXED file is created, it may be retrieved in either sequential or direct access mode.

### Sequential Retrieval

Sequential retrieval permits all records of the file to be read into main storage, beginning with the record that is lowest in sequence. However, you may also read only a portion of the file by specifying the key of the lowest record of that portion and then processing as many records as required. The lowest record is retrieved in direct access and the file is then processed sequentially by READ statements without source keys until the upper limit is reached.

### Direct Retrieval

For direct retrieval, you must specify a source key in every READ statement.

### File Extension

An INDEXED file may be extended at its end, once it has been created. For file extension, the records that are to be added must all have keys that are higher in sequence than the key of the last record of the current file. Note, however, that extension is not possible if a record has been inserted on the last active track of the file in a previous run. In this case, the file must be entirely rewritten.

### Insertion of New Records

You may also insert records with new keys into the file. For the insertion of new records, you must reserve overflow areas within the file.

### Deletion of Records

Deletion is not accomplished by physically removing the records from the file. If you want to delete records, you must specify this by writing a deletion code in each record to indicate that it is to be removed when the file is eventually reorganized by a reorganization program that you must supply yourself.

### Indexes

The ability to read and write records anywhere in an INDEXED file is provided by indexes that are part of the file. Two types of indexes are automatically constructed whenever a file is created or extended: a cylinder index for the entire file, and a track index for each cylinder. An entry in a cylinder or track index contains the identification of a specific cylinder or track and the highest key associated with that cylinder or track. The indexes are automatically retrieved and searched when their use is required, that is, when a READ, WRITE, or REWRITE statement with a source key is encountered during execution of a program.

When new records are added to the file, as insertions or extensions, the indexes are automatically modified to account for the new records.

### Overflow Area

In addition to the prime data area, whose tracks initially receive the records of an INDEXED file, there are overflow areas for records forced off their original tracks by the insertion of new records. An overflow area called a cylinder overflow area, may be designated for each cylinder of the prime data area. If a large number of additions is anticipated, an independent overflow area may be designated to supplement or replace the cylinder-overflow areas. This area must be defined as a separate extent.

When a new record is to be inserted in a track that is already full, the records already on the track with keys that are higher in sequence than the key of the record to be added are removed and -- after insertion of the new record -- written back

Page Missing From Original Document

first 45 bytes of each card are processed, even though a complete card is fed through the card reader each time a READ statement is executed.

4 The format of the records to be written on the INDEXED file is described in the structure declaration DATA. The key, which, in this example, has also the name KEY, starts with the eighth character of the record as specified by the KEYLOC option in the file declaration. The length of the key, 12 characters, must be the same as specified in the KEYLENGTH option.

2 EXTENTNUMBER (3) serves for either (a) one index extent, one prime data extent, and one independent overflow area, or (b) one index extent and two prime data extents. Of which kind the last extent actually is, has to be specified by the third job control XTENT statement for this file (not shown in this example). Two tracks per cylinder are set aside for later insertion of overflow records by the OFLTRACKS option. The records of the file to be loaded are blocked with a blocking factor of 6. Since the file is to be saved for later usage, the VERIFY option has been specified to assure error detection during writing of blocks on the disk pack.

3 The printer will be used only in exceptional cases, that is, when the KEY or the ENDFILE condition is raised. Therefore, the use of only one buffer will not impair performance significantly.

5 Even though the key is already in its final position in the record, the KEYFROM option must be specified with the WRITE statement.

6 This point is reached each time a key is out of sequence. The program need not be terminated; it proceeds with reading the next record.

7 The program is terminated by detecting an end-of-file statement, /*, in card columns 1 and 2 in the input file. A termination message is printed.

In the second example, shown in Figure 53, the file created in the previous example is updated and records are added to

```
     ADDUPD: PROCEDURE OPTIONS (MAIN);
             DCL INPUT FILE RECORD INPUT ENV(MEDIUM(SYSIPT,2520)F(45)),
  1              INDEX FILE RECORD UPDATE DIRECT ENV(MEDIUM(SYS008,2311)
                                                       F(270,45)
                       INDEXED KEYLENGTH(12) KEYLOC(8) EXTENTNUMBER(3)
                       OFLTRACKS(2) VERIFY),
                 1 DATA BASED (P),
                    2 PERS_NO PIC'(7)9',
                    2 INKEY CHAR(12),
                        .
                        .
                        .
  2              1 FDATA,
                    2 PERS_NO PIC'(7)9',
                    2 FKEY CHAR(12),
                        .
                        .
                        .
                 P POINTER STATIC;
             /* FILE ACTIVATION*/ OPEN FILE(INPUT), FILE (INDEX);
              ON ENDFILE (INPUT)GO TO END_OF_JOB;
              ON KEY (INDEX) GO TO NEW_RECORD;
     /* PROCESSING LOOP*/
     LOOP: READ FILE(INPUT) SET (P);
  3        READ FILE (INDEX) INTO (FDATA) KEY (INKEY);
           /* UPDATE RECORD IN 'FDATA' BY 'DATA'*/

  4     REWRITE FILE(INDEX) FROM (FDATA) KEY (FKEY);
        GO TO LOOP;
        /* ADD NEW RECORD TO INDEXED FILE */
  5  NEW_RECORD: WRITE FILE (INDEX) FROM (DATA) KEYFROM (INKEY);
        GO TO LOOP;
     END_OF_JOB: END;
```

Figure 53. Updating of and Addition of Records to an INDEXED File

the file. The distinction between updating and adding is made by use of the KEY condition. The KEY condition is raised when searching for a record with a READ statement is unsuccessful. In this case, adding instead of updating is performed.

Explanation:

1 The attribute DIRECT implies the attribute KEYED.

3 The key (INKEY), which is used to determine whether a record is to be updated or added, is provided by the card file INPUT whose records are read into the structure DATA.

2 Since both the card record and the old record from the INDEXED file are needed for updating, an extra structure, FDATA, is used for holding the records of the INDEXED file.

5 If a record with a key specified in INKEY is not yet on the INDEXED file, control branches automatically to the statement label NEW_RECORD where the new records are added to the INDEXED file.

4 The updated record is written back into its previous location on the INDEXED file.

In the third example, shown in Figure 54, sequential regions of an INDEXED file (SAMPLE) are punched into cards with an output file (PUNCH). The limits of the regions to be punched are defined by upper keys and lower keys read from a third file (CONTRL). The block length of the INDEXED file is 525 bytes occupying two disk sectors, so that 15 bytes of the second sector are left unused. The output file (PUNCH) uses the ASA control character W, which causes the cards from the secondary hopper of the MFCM to be fed into stacker 4. All cards receive running numbers in columns 76 to 80.

```
    INDXPC: PROCEDURE OPTIONS(MAIN);
            DCL SAMPLE FILE RECORD INPUT KEYED SEQUENTIAL
                ENV(MEDIUM(SYS006,2311) INDEXED KEYLENGTH(12)
                KEYLOC(1) EXTENTNUMBER(3) F(525,75));
            DCL CONTRL FILE RECORD INPUT ENV(MEDIUM (SYSIPT,2560P)
                F(24) BUFFERS(1));
            DCL PUNCH FILE RECORD OUTPUT ENV(MEDIUM(SYSOPT,2560S)
                F(81) CTLASA);
            DCL 1 INDEX_REC,
                  2 KEY CHAR(12),
                  2 OTHER CHAR(63),
                1 OUTREC,
                  2 CTL_CHAR CHAR(1) INIT ('W'),
                  2 RECORD,
                    3 KEY CHAR(12),
                    3 OTHER CHAR(63),
                  2 NUMBER PIC'(5)9',
                1 CTL_REC,
                  2(LOWER,UPPER) CHAR(12),
                NO FIXED DECIMAL (5,0) INIT(1);
            /* FILE ACTIVATION */
            OPEN FILE(SAMPLE), FILE(CONTRL), FILE (PUNCH);
            ON ENDFILE (CONTRL) GO TO EOJ;
            /* CONTROL-FILE LOOP */
    CLOOP:  READ FILE (CONTRL) INTO (CTL_REC);
 1          READ FILE (SAMPLE) INTO (INDEX_REC) KEY (LOWER);
            GO TO E_2;
            /* SAMPLE-FILE LOOP */
 2  SLOOP:  READ FILE (SAMPLE) INTO (INDEX_REC);
 3  E_2:    IF INDEX_REC.KEY > UPPER THEN GO TO CLOOP;
 4          RECORD = INDEX_REC;
            NUMBER = NO;
            WRITE FILE (PUNCH) FROM (OUTREC);
            NO = NO + 1;
            GO TO SLOOP;
    EOJ:    END;
```

Figure 54. Example Illustrating (a) Sequential Retrieval of Records Within Specified Limits From an INDEXED File and (b) Stacker Selection

Explanation:

1 The READ statement for the first record of each sequential region to be punched requires the KEY option to position the file to the specified lower limit.

2 Further READ statements to transmit records of the same sequential region do not require the KEY option. The next sequential record is read automatically.

3 The IF statement tests whether the upper limit of the region to be punched has been exceeded. If it has, a branch to CLOOP is taken, where the next control key from the file CNTRL is read.

4 This is a structure assignment. Note that both structures have the same structuring and description, but no identical level numbers. Level numbers need not be identical.

### Accessing INDEXED Files

Records in INDEXED files are arranged on the basis of keys that are associated with each record. INDEXED files may be transmitted only in record-oriented mode. They must be created as INDEXED SEQUENTIAL OUTPUT files. Once they have been created, they can be accessed only as INPUT or UPDATE files, for either sequential or direct retrieval.

To retrieve the entire file sequentially, you may read it just like a CONSECUTIVE file, without specifying a key. To read only a given portion of the file, you must specify the key of the lowest record of that portion and then test whether the key of the highest record of that portion has been reached. The records between the lowest and the highest record can be retrieved without specifying source keys in the READ statements.

To retrieve records from an INDEXED file in random order, you must specify a source key with every READ statement to identify the record to be retrieved from the file. And, in the file declaration, you must specify the attribute DIRECT, together with the option INDEXED.

For an INDEXED file that is to be retrieved sequentially, you must specify the attributes SEQUENTIAL and KEYED, together with the option INDEXED in your file declaration.

## The ENVIRONMENT Attribute and Its Options

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

The options list for the Model 20 PL/I compiler is as follows:

```
[CONSECUTIVE]
[INDEXED    ]
{F (blocksize[,recordsize])}
{V (maxblocksize)          }
{U (maxblocksize)          }
[BUFFERS ({1|2})]
 MEDIUM (symbolic-device-address,
      device-type)
[CTLASA]
[LEAVE]
[NOTAPEMK]
[NOLABEL]
[ALTTAPE]
[VERIFY]
[NOWRITE]
[KEYLENGTH
      (decimal-integer-constant)]
[EXTENTNUMBER
      (decimal-integer-constant)]
[OFLTRACKS
      (decimal-integer-constant)]
[KEYLOC (decimal-integer-constant)]
```

General Rules:

1. Each file declaration must have an associated ENVIRONMENT attribute.

2. The options must be enclosed in parentheses and separated by one or more blanks.

3. The MEDIUM option and one of the record format options (F, V, and U) must always be specified. The others are optional. Their use depends upon the associated files that are to be processed.

The individual options are discussed in the sequence in which they are listed above.

### The CONSECUTIVE Option

The option CONSECUTIVE indicates that the logical records of the associated file are arranged on the basis of their successive physical positions, such as they appear on tape.

General Rules:

1. A CONSECUTIVE file must either have the SEQUENTIAL attribute or be a STREAM file.

2. If neither the CONSECUTIVE nor the INDEXED option is specified in the ENVIRONMENT attribute, CONSECUTIVE is assumed by default.

3. Once a CONSECUTIVE file has been created, it may be opened only as an INPUT or UPDATE file. Such a file may be read either forward or backwards if it is on magnetic tape. In order to be read backwards, it must have the BACKWARDS attribute, and the LEAVE option specified in the ENVIRONMENT attribute.

4. All three kinds of record formats -- F, V, and U -- are allowed for CONSECUTIVE files. The last two formats, V, and U, may be used only for RECORD I/O and only with magnetic tape. As an exception to this rule, printer-keyboard files must always consist of format-U records. Format-V records, however, cannot be read backwards.

5. I/O devices permitted for CONSECUTIVE files include magnetic tape units, card readers and punches, disk-storage drives, printer-keyboards printers.

### The INDEXED Option

The INDEXED option specifies that the records of the associated file has or will have its records arranged in logical sequence, according the keys associated with every record.

General Rules:

1. An INDEXED file may be created only as an INDEXED SEQUENTIAL OUTPUT file. Once it has been created, it may be processed as an INDEXED SEQUENTIAL OUTPUT file only if it is to be extended, in which case the keys of the extended part must be higher in the collating sequence than the keys of the already existing part. Although you can extend an INDEXED SEQUENTIAL file at its end, you cannot insert new, additional records into it.

   INDEXED DIRECT files may be opened only for INPUT or UPDATE activity. You cannot extend an INDEXED DIRECT file at its end, but you can replace old records and insert new, additional ones.

2. Only record-oriented retrieval is permitted with INDEXED files.

3. The key associated with each logical record must always be a character string of not more than 60 characters. The length of the recorded key must be specified in the ENVIRONMENT option KEYLENGTH(n). The high-order position of the key -- which is always embedded in the actual data part of the logical record -- is specified in the ENVIRONMENT option KEYLOC(n).

4. Only format-F records (blocked or unblocked) may be used with INDEXED file organization.

5. LOCATE and READ with the SET option are not allowed for INDEXED files.

### The Options F, V, and U

The options F, V, and U are used to describe physical records. F specifies fixed-length records, V specifies variable-length records, and U specifies records of undefined length.

Records may be blocked or unblocked. Block size and record size are specified in number of bytes, as unsigned decimal integer constants.

General Rules:

Format-F records:

1. If the record size of format-F records is not specified in the ENVIRONMENT attribute, the records are assumed to be unblocked. In case of blocked records, record and block size must be specified. The record size can be specified for format-F records only. Blocking and deblocking is handled automatically. It is dependent upon the stated record size. The quotient of block size divided by record size must be an integer. Fixed-length blocked records are constructed if, on output, both block size and record size are specified. The blocking factor is the block size divided by the record size.

2. For INDEXED files, only format-F records (blocked or unblocked) are allowed. The maximum block size for INDEXED files is 16200, the maximum record size 4090 bytes.

3. For STREAM files, only fixed-length, unblocked records are allowed, with the exception of printer-keyboard files which require format-U records.

Format-V records:

1. Blocking and deblocking of format-V records is dependent on the information at the beginning of each block and at the beginning of each logical record. Four bytes are used at the beginning of each block to specify block size, and another four bytes at the beginning of each logical record to specify the size of that logical record. Although the block and record sizes are inserted automatically in these four bytes when the file is created, you must include the number of these bytes when you spe-

cify the length of the block in the ENVIRONMENT attribute. When format-V files are created, records are always blocked if their sizes allow two or more records to be placed into a block smaller than or equal to the maximum that is specified.

2. Format-V records cannot be read backwards.

Format-U records:

1. With format-U records, each block consists of only one record. The blocks (records) are of varying lengths. No system-control bytes appear anywhere within the block. All processing of records is your responsibility. If you include a length specification in the record, you must insert it yourself, and you must also retrieve this information yourself.

2. The two formats V and U may be used only with record-oriented magnetic tape I/O, printer-keyboard files must consist of format-U records.

3. The minimum and maximum physical record sizes (or block sizes) that are allowed with the various I/O devices are shown in Figure 55.

| Device | Block Size in Bytes | |
|---|---|---|
| | Min | Max |
| Card | 1 | 80 |
| Printer | 1 | 1 line |
| Typewriter | 2 | 511 |
| Magnetic Tape | 18* | 4095 |
| 2311 DASD (for CONSECUTIVE files) | 1 | 26730 |
| 2311 DASD (for INDEXED files) | 1 | 16200 |
| *The minimum record size must also be 18. | | |
| Add 1 to the above values if an ASA control character is used. | | |

Figure 55. Block Sizes Permitted Depending Upon the I/O Devices Used

4. When reading files with format V or U records, you normally decide from some control field in the record you must have inserted yourself, which type of record you are processing. In this case, you should use the READ statement with the SET option to avoid raising of the RECORD condition.

## The BUFFERS(n) Option

The BUFFERS(n) option is used to specify that buffer storage area is used for data transmission. If you specify BUFFERS(2), input/output activity can run concurrently with internal processing.

General Rules:

1. The number n must either be 1 or 2.

2. If the BUFFERS(n) option is not specified, the number of buffers is assumed to be 2, except for INDEXED DIRECT files.

3. BUFFERS(n) may be used with both STREAM and RECORD files.

## The MEDIUM Option

The MEDIUM option in the ENVIRONMENT attribute -- and, if necessary, the ASSGN statement of the Model 20 Job Control -- are used to associate a file on an external storage medium with a program. The format is:

```
MEDIUM (symbolic-device-address,
      device-type)
```

The symbolic-device-address entry relates a device to a particular file. The device-type entry defines the type of I/O device on which the file is located.

General Rules:

1. The symbolic device address is specified as SYSnnn, where nnn can be IPT (system input device), LST (system output device), used for listing), PCH or OPT (system output device used for punching or writing on tape), or 000 through 019 (as selected by the programmer).

2. OPT should not be used for Model 20 PL/I programs that are expected to run under DOS/TOS as well.

3. The device-type entry is a four or five-character code identifying the type of I/O device to be used. For example, for the IBM 2415 Magnetic Tape Unit, the code is 2400, for the IBM 2311 Disk Unit, the code is 2311. Figure 56 shows how the individual device types are specified.

| Type | Device | Device-Type Specification |
|---|---|---|
| Card Readers and Punches | IBM 1442 | 1442 |
| | IBM 2520 | 2520 |
| | IBM 2560[1] | 2560P |
| | IBM 2560[2] | 2560S |
| | IBM 2501 | 2501 |
| Printers | IBM 1403 | 1403 |
| | IBM 2203 | 2203 |
| Magnetic Tape Units | IBM 2415 (9-track) | 2400 |
| | IBM 2415 (7-track) | 2400 |
| Disk Storage | IBM 2311 | 2311 |
| Printer-Keyboard | IBM 2152 | 2152 |

[1] Primary feed
[2] Secondary feed

Figure 56. Device Types and Corresponding Specifications

4. The device types listed in Figure 56 may be assigned to the logical unit names SYSIPT, SYSLST, SYSPCH, SYSOPT, or SYS000 - SYS019, as shown in Figure 57.

| Symbolic Device Address | Device Type Specification |
|---|---|
| SYSIPT | 2501<br>2520<br>2560P<br>2560S<br>2400<br>2152 |
| SYSLST | 1403<br>2203<br>2152 |
| SYSOPT, SYSPCH | 1442<br>2520<br>2560P<br>2560S<br>2400 |
| SYS000 through SYS019 | 2311 and 2400 and any of the above devices |

Figure 57. Device Type Specifications that may be Associated with Symbolic Device Addresses

5. The system automatically associates the symbolic device address with the device type specified in the MEDIUM option of the ENVIRONMENT attribute. Only if standard assignments are used, no additional ASSGN job-control statements are needed.

   For magnetic tape and disk storage, you may have to change the permanent device assignments of the system. This means that, in addition to specifying the symbolic device address and the device type in the MEDIUM option, you may have to furnish ASSGN statements. These ASSGN job-control statements must contain the actual device address and device type following the SYSnnn. The device specification in the MEDIUM option must correspond with the device specification in the ASSGN statement. (For a complete discussion of the ASSGN job-control statement, refer to the section Job_Control.)

6. All files in Model 20 PL/I must be explicitly declared with the MEDIUM option in the ENVIRONMENT attribute. ASSGN statements are required only if you want to change the previous device assignments for tape and disk files.

7. With card, printer, and magnetic tape, only one file must be open at the same time for the same device. An exception is the 2560 multifunction card machine. With the 2560, one file may be open at the same time for each hopper. Disk files that are open at the same time must not have overlapping extents.

   With the 2152 printer-keyboard, one input and one output file may be open at the same time. For synchronisation of these files, see Use_of_the Printer-Keyboard in the section Special_Programming_Considerations.

## The_CTLASA_Option

The CTLASA option may be used for card and printer RECORD OUTPUT files. It indicates that the first character of a record is to be interpreted as an ASA control character for printer-carriage control and for punched-card stacker selection.

General Rules:

1. It is your responsibility to provide the correct control character. For an example of its use, refer to Creating_Files_of_CONSECUTIVE_Organization in this section.

2. The character codes you can use with CTLASA are listed in Figure 58.

| Code | Interpretation |
|---|---|
| (blank) | Space one line before printing |
| 0 | Space two lines before printing |
| - | Space three lines before printing |
| + | Suppress space before printing |
| 1 | Skip to channel 1 before printing |
| 2 | Skip to channel 2 before printing |
| 3 | Skip to channel 3 before printing |
| 4 | Skip to channel 4 before printing |
| 5 | Skip to channel 5 before printing |
| 6 | Skip to channel 6 before printing |
| 7 | Skip to channel 7 before printing |
| 8 | Skip to channel 8 before printing |
| 9 | Skip to channel 9 before printing |
| A | Skip to channel 10 before printing |
| B | Skip to channel 11 before printing |
| C | Skip to channel 12 before printing |
| V | Select stacker 1 (5 if 2560S) |
| W | Select stacker 2 (4 if 2560S) |

Figure 58. Character Codes that can be used with Printer Carriage Control for Record Output Files

### The LEAVE Option

The LEAVE option is used for file positioning. It is used to specify, that no rewind operation is to be performed when a tape file is opened or closed.

General Rules:

1. LEAVE can only be used with tape I/O files, either RECORD or STREAM.

2. LEAVE should be specified for files that have the BACKWARDS attribute to ensure proper positioning of the file.

### The NOTAPEMK Option

The NOTAPEMK option for tape files enables you to prevent a leading tapemark from being written ahead of the data records in unlabeled tape files.

General Rules:

1. NOTAPEMK may be used for tape OUTPUT files with NOLABEL specified.

2. NOTAPEMK may be used with STREAM and RECORD files.

### The NOLABEL Option

The NOLABEL option is used to indicate that no label processing is to be done for the associated file.

General Rules:

1. If the NOLABEL option is specified for output files, a tapemark is automatically written as the first record on the tape unless, in addition to NOLABEL, NOTAPEMK is specified in the ENVIRONMENT attribute. Labels are not processed.

2. If your program is to run as an inquiry program the NOLABEL option must be specified for all tape files.

### The ALTTAPE Option

The ALTTAPE option is used to indicate that an alternate tape drive can be assigned to enable automatic switching from one tape drive to another.

General Rules:

1. The ALTTAPE option is used for multivolume tape files.

2. The alternate tape drive is always assigned a symbolic device address that is one higher than the symbolic device address specified in the MEDIUM option. For example, if the symbolic device address specified in the MEDIUM option is SYS004, the symbolic device address for the alternate tape drive is SYS005.

### The VERIFY Option

The VERIFY option causes a check to be performed after every WRITE operation.

General Rules:

1. The VERIFY option is allowed only for files that are associated with a 2311 disk-storage device.

2. It is recommended to use VERIFY if the disk file is to be retained for re-use at a later date.

### The NOWRITE Option

The NOWRITE option is used to specify that no new records will be added to the file by the associated program. This results in loading a smaller library routine, thus saving main storage space.

General Rule:

This option may only be used with DIRECT UPDATE files.

### The KEYLENGTH Option

The KEYLENGTH(n) option is used to specify the length of the recorded key for I/O operations.

General Rules:

1. This option may be used only for INDEXED files.

2. The length of the recorded key, specified by n, must not exceed 60 characters.

3. n must be an unsigned decimal integer constant.

### The EXTENTNUMBER Option

The EXTENTNUMBER(n) option is used to specify the number of extents used for INDEXED files.

General Rules:

1. EXTENTNUMBER(n) must be specified for INDEXED files. It is not permitted with CONSECUTIVE files.

2. The minimum number that may be specified is two: one extent for the prime data area, and one for the cylinder index.

3. The number n must be a decimal integer constant.

4. EXTENTNUMBER(n) must include all prime data area extents, the cylinder index extent, and the independent-overflow area extent, if any.

### The OFLTRACKS Option

The OFLTRACKS(n) option is used to specify the number of tracks to be reserved on each cylinder for the addition of new records.

General Rules:

1. This option is specified only for INDEXED SEQUENTIAL OUTPUT and INDEXED DIRECT UPDATE files. It is meaningless for INPUT and SEQUENTIAL UPDATE files. If specified for these files, the option is ignored.

2. The number n -- which must be a decimal integer constant -- must be from 0 to 8.

### The KEYLOC Option

The KEYLOC(n) option is used to specify the high-order position of the key field within the data part of each logical record in INDEXED files. For example, if the key field is to start in the second byte of the record, KEYLOC(2) has to be specified.

General Rules:

1. The KEYLOC(n) option is used with INDEXED files only. It must be specified.

2. The number n must be a decimal integer constant.

3. At least three bytes in the logical record must follow the last byte of the key.

## File Labels

A reel of tape may be, whereas a disk pack must be identified to determine whether it contains the file to be processed. Besides a sticker on the pack or reel that identifies a volume to the operator there are also internal labels. The internal label is written, like other data, on the tape or on the disk and provides programmed identification of the files contained in the volume.

Internal labels are also used to protect files on a volume from being destroyed. For example, labels are checked to determine whether the correct volume has been mounted. Another use of labels is to specify the location of the data files. Labels also contain other information concerning files, such as expiration dates. They are processed whenever an OPEN or CLOSE statement is executed for a particular file.

There are two types of internal labels: volume labels and file labels.

*Volume labels* are used to identify the volume (tape reel or disk pack). They are written by IBM-supplied Utility programs when a tape reel or a disk pack is prepared for use. They are automatically checked each time a file is opened.

*File labels* are used to identify a file on a volume and to specify how long a file is to be retained on the volume. When an OPEN statement is encountered for a file, the information contained in the file labels is compared against the information supplied by the job-control statements. If a mismatch is found, a message is displayed to the operator and execution of the program may be discontinued.

In our discussion, we will be concerned mainly with file labels. Volume labels are of minor interest to the programmer.

## PUNCHED CARD AND PRINT FILES

Punched card, printer-keyboard, and print files cannot be labeled.

## TAPE FILES

Tape files may be labeled or unlabeled.

### Unlabeled Tape Files

In the case of unlabeled tape files, neither the volume nor the file is associated with an identification. The operator must make sure that the correct volume is mounted by checking external labels. When working with multi-file volumes, the operator must also correctly position the volume at the beginning of the file in question (// FILES card).

The first record on a tape containing unlabeled files may or may not be a tapemark; however, every unlabeled tape file is followed by a tapemark. The last file on a volume is always followed by two tapemarks.

If the first record on a tape containing unlabeled files is not a tapemark, the record is assumed to be a data record.

If no labels are specified for an output tape file, no label checking is performed, and any labels on the output tape are destroyed. A tapemark is automatically written on the output file unless, in addition to the option NOLABEL, the option NOTAPEMK has been specified in the ENVIRONMENT attribute of the associated file declaration.

### Labeled Tape Files

There are two types of labels: header labels and trailer labels. The header label precedes each file and defines it. The trailer label is written at the end of the file. It furnishes the information required to determine whether the end of the file has been reached or whether the file is continued on another volume.

Label information for tape files is furnished to the Job Control program in two consecutive control cards: The VOL and the TPLAB card.

The VOL card is simply a header card for the TPLAB card. It contains the file name and the symbolic device address and serves a diagnostic function. It is used to check whether the device assigned to the symbolic address in the TPLAB card is really a tape drive. The TPLAB card contains the file identification, file sequence number, and all other information used by the Open routine to position and identify the file.

Only one VOL and one TPLAB statement are needed for each logical file, regardless of the number of reels on which the file is recorded.

## DISK FILES

Disk files must be labeled, that is, the volume and the actual file must both be identified. However, these labels do not precede or follow the individual file on the disk. The file labels are contained in a special region referred to as VTOC (Volume Table of Contents).

Disk labels are also automatically written, checked, and updated by IBM-supplied programs. The correct mounting of the packs is checked by means of the volume serial number. The positioning at the beginning of the actual data on the volume is done by means of the file identification. In your program, you need only specify the filename and the symbolic device address. Switching from volume to volume for multi-volume files is automatic without any programming effort.

All information necessary to locate the data on the disk pack is specified in the job-control statements. A set of three related, consecutive control statements is used: the VOL, DLAB, and XTENT statements. The VOL and DLAB statements have the same function for disk files as the corresponding control statements have for labeled tape files. The XTENT statements contain the symbolic device address to locate the drive, the volume serial number to determine whether the correct pack is mounted, and the extent limits.

## LABEL-CONTROL STATEMENTS

The label-control statements for tape and disk must precede the // EXEC statement in the following sequence:

for tape files:

```
    // VOL
    // TPLAB
```

for disk files:

```
    // VOL
    // DLAB
    // XTENT
    // XTENT
   [// XTENT]
         .
         .
         .
```

### VOL Control Statement

A VOL (volume) control statement is required for each labeled tape and disk file used in your problem programs. It indicates the symbolic address of the device to which the volume is assigned and identifies the file by name. A VOL control statement must be followed by a TPLAB control statement if the volume is a reel of magnetic tape, or by a DLAB control statement and one or more XTENT control statements if the volume is a disk pack. The format of the VOL control statement is:

| Name | Operation | Operand(s) Comments |
|---|---|---|
| // | VOL | symbolic-device-address,<br>filename |

symbolic-device-address
The symbolic address identifies the device on which the associated file is to be read or written. (A complete list of symbolic device addresses is given in the discussion of the ASSGN control statement).

filename
The name of the file to be processed.

Note: If the symbolic device address refers to a tape drive, a TPLAB control statement is expected as the next control statement. If it refers to a disk drive, a DLAB statement is expected to follow.

### TPLAB Control Statement

The TPLAB (tape label) control statement contains a portion of the tape header label for the file named on the VOL control statement that precedes the TPLAB control statement. The contents of the tape file label are shown in Figure 59. The information in the TPLAB control statement is used to check input-file tape labels or write output-file tape labels. The format of the TPLAB control statement is:

| Name | Operation | Operand(s) Comments |
|---|---|---|
| // | TPLAB | 'label-information' |

'label-information'
The information in fields 3 through 10 of the standard IBM tape label (see Figure 59) for the associated file. The label information must be enclosed in apostrophes. The following information is required:

| Field | Contents | No. of Characters |
|---|---|---|
| 3 | File identification | 17 |
| 4 | File serial number | 6 |
| 5 | Volume sequence number | 4 |
| 6 | File sequence number | 4 |
| 7 | Generation number (if used) | 4 |
| 8 | Version number (if used) | 2 |
| 9 | Creation date (yyddd preceded by one blank) | 6 |
| 10 | Expiration date (yyddd preceded by one blank) | 6 |

Figure 59. Label Information for Tape Files

The file-label fields required in the operand field of a TPLAB control statement must be contiguous and not separated by commas. The operand requires 51 columns (two apostrophes with 49 columns of information between them).

The standard IBM tape-file label format and contents are as shown in Figure 60.

| Field | Name and Length | Description |
|---|---|---|
| 3 | file_identification<br>17 bytes, EBCDIC | uniquely identifies the entire file; may contain only printable characters. |
| 4 | file_serial_number<br>6 bytes, EBCDIC | uniquely identifies a file/volume relationship. This field is identical to the Volume Serial Number in the volume label of the first or only volume of a multi-volume file or a multi-file set. This field will normally be numeric (000001 to 999999) but may contain any six alphameric characters. |
| 5 | volume_sequence number<br>4 bytes, EBCDIC | indicates the order of a volume in a given file or multi-file set. The first must be numbered 0001 and subsequent numbers must be in proper numeric sequence. |
| 6 | file_sequence_number<br>4 bytes, EBCDIC | assigns numeric sequence to a file within a multi-file set. The first must be numbered 0001. |
| 7 | generation_number<br>4 bytes, EBCDIC | uniquely identifies the various editions of the file. May be from 0001 to 9999 in proper numeric sequence. |
| 8 | version_number_of generation<br>2 bytes, EBCDIC | indicates the version of a generation of a file. |
| 9 | creation_date<br>6 bytes, EBCDIC | indicates the year and the day of the year the file was created;<br>Position   Code   Meaning<br>1   blank   none<br>2-3   00-99   year<br>4-6   001-366   day of year<br>(e.g., January 31, 1970 would be entered as 70031). |
| 10 | expiration_date<br>6 bytes, EBCDIC | indicates the year and the day of the year when the file may become a scratch tape. The format of this field is identical to that of Field 9. On a multi-file reel, processed sequentially, all files are considered to expire on the same day. |

Figure 60. Label Information for Tape Files

DLAB_Control_Statement

The DLAB (disk label) control statement contains the disk-label information for the file named in the VOL control statement that precedes the DLAB control statement. The information in the DLAB control statement is used to check input-file disk labels or write output-file disk labels. The format of the DLAB control statement is:

| Name | Operation | Operand(s) Comments |
|---|---|---|
| // | DLAB | label-information |

label-information
Information contained in the disk label for the associated file. Fields 1-3 of the disk file label (a 51-character string) are punched into the DLAB statement as they appear in the label. The contents of the file label are shown in Figure uc. The character string must be enclosed in apostrophes and followed by a comma. If the label information is to be cataloged permanently, the comma must be followed by the code P followed by a blank (See Permanent_and_Temporary_Disk Labels in this section). The remainder of the required label information, beginning with the volume sequence number, is punched into a continuation card. Therefore, column 72 of the DLAB control statement must contain a continuation punch (any EBCDIC character except blank). The information in the continuation card must begin in column 16. Columns 1-15 must be left blank. The fields in the continuation card are

| Field | Name and Length | Description |
|---|---|---|
| 1 | identification<br>44 bytes, EBCDIC | this field serves as identifier of the file. Each file must have a unique filename. Duplication of filenames will cause retrieval errors. The Model 20 Disk Programming System compares the entire filename field against the filename given in the DLAB card. |
| 2 | format identifier<br>1 byte, EBCDIC | 1 |
| 3 | file serial number<br>6 bytes, EBCDIC | uniquely identifies a file/volume relationship. It is identical to the volume serial number of the first or only volume of a file. |
| 3a | code for permanent label<br>2 bytes, EBCDIC | specifies that the label is to be cataloged permanently (optional). |
| 4 | volume sequence number<br>4 bytes | indicates the order of a volume relative to the first volume on which the data file resides. |
| 5 | creation date<br>5 bytes | indicates the year and the day of the year the file was created. It is of the form yyddd, where yy signifies the year (0-99) and ddd the day of the year (1-366). |
| 6 | expiration date<br>5 bytes | indicates the year and the day of the year the file may be deleted. The form of this field is identical to that of Field 5. |
| 7 | system code<br>13 bytes | uniquely identifies the programming system. The character codes that can be used in this field are limited to 0-9, A-Z, or blanks. |

Figure 61. Label Information for Disk Files

separated by commas. The last field (system code, 13 characters) is optional, and if used, must be enclosed in apostrophes. The following fields are used:

| Field | Contents | No. of Characters |
|---|---|---|
| 1 | File Identification | 44 |
| 2 | Format (always coded as 1) | 1 |
| 3 | File serial number | 6 |
| 3a | Code for permanent label (optional) | 2 |
| 4 | Volume sequence number | 4 |
| 5 | Creation Date (yyddd) | 5 |
| 6 | Expiration date (yyddd) | 5 |
| 7 | System code (optional) | 13 |

The standard IBM disk-file label (see Figure 61) is common to all data files on disk:

### XTENT Control Statement

In addition to the VOL and DLAB control statements, the user must provide at least one XTENT control statement for SEQUENTIAL files and at least two XTENT control statements (prime data area plus cylinder index) for an INDEXED file. The format of the XTENT control statement is:

| Name | Operation | Operand(s) Comments |
|---|---|---|
| // | XTENT | type,sequence,lower-limit,<br>upper-limit,<br>'volume-serial-number',<br>symbolic-device-address |

type
: A code indicating the purpose of the extent:

1 for prime data area
2 for independent overflow area
4 for cylinder index area

Note 1: At least one type-1 XTENT statement must be among the file definition statements.

Note 2: Only one XTENT statement is allowed for types 2 and 4. If type 2 is used, it must be the last XTENT statement for that file.

Note 3: If an independent overflow extent exists, a cylinder-index extent must also be specified.

Note 4: If a cylinder-index extent is specified, the prime-data extents must be on cylinder boundaries (i.e., lower limit track 0, upper limit track 9). The index extent itself may not occupy more than ten tracks.

sequence
The three-digit extent sequence number (ESN), indicating the position of the extent within a multi-extent file. The ESN may be any of the numbers 000-255. The extent sequence numbers must be specified in ascending order, but not necessarily consecutive.

lower-limit
The address of the beginning (lowest disk address) of the area. The address is given in the form cccchhh, where cccc is the four-digit cylinder number (0000-0202) and hhh the three-digit head number (000-009). The lower-limit address must not be cylinder 0, track 1. It should at least be cylinder 4, track 0.

upper-limit
The address (form cccchhh) of the end (highest disk address) of the area.

Note: A file must not occupy the volume label area (cylinder 0, track 1) or the alternate track area (cylinders 1-3).

'volume-serial-number'
An EBCDIC character string (6 characters) enclosed in apostrophes.

Note 1: The file serial number of the DLAB statement and the volume-serial-number operand of the first XTENT statement must be identical.

Note 2: If several volume serial numbers occur in a group of XTENT statements, the statements with identical volume serial numbers must be grouped together.

Note 3: A volume serial number is assigned to each disk pack by the Initialize Disk Utility program.

symbolic-device-address
The symbolic address of the device on which the volume containing the extent is mounted.

Notes:

- The symbolic-device-address operand in the VOL and first XTENT statement must be assigned to the same actual device address.

- The symbolic-device-address operand in XTENT statements with the same volume serial number must be assigned to the same actual device address.

- The volume-serial-number operand of the XTENT statement is used for controlling multi-volume processing. For INDEXED files located on more than one disk pack, multi-drive operation is mandatory.

- One disk volume may contain several files, each with its own extent. The volume serial number in the XTENT statement must be the same for all these files. The symbolic device address in the VOL statements may be different for these files, but the actual device addresses in the ASSGN statement must be the same.

- One-drive users may also use different files with different volume serial numbers assigned in the XTENT statements. They must then change disk packs before a file with a different volume serial number is opened or processed.

- If different extents are used for one set of VOL and DLAB statements the symbolic device addresses and the volume serial numbers in the XTENT statements must refer to the correct disk pack.

- All information for the XTENT statement can be accommodated on one card; the information must not be continued on a continuation card.

For examples of label-information control statements refer to Figure 62.

```
1       1         2         3         4         5         6         7 7       8
        0         0         0         0         0         0         0 2       0
// JOB   FIRST
// ASSGN SYS001,X'708',T2
// FILES SYS001,1
// VOL   SYS001,KEY
// TPLAB 'KEY TAPE FILE   00038400010001000102 70085 71085'
// EXEC

// JOB   SECOND
// ASSGN SYS002,X'801',D3
// VOL   SYS002,INPUT       MULTI-VOL, SINGLE DRIVE
// DLAB  'INPUT DATA FILE                             1SSSSSS',        C
              0001,70032,71033,'0000000000000'
// XTENT 1,001,0011003,0011006,'SSSSSS',SYS002
// XTENT 1,003,0201000,0202009,'TTTTTT',SYS002
// XTENT 1,008,0050008,0060006,'UUUUUU',SYS002
// EXEC

// JOB   THIRD
// ASSGN SYS019,X'802',D3
// VOL   SYS002,OUTPUT      MULTI-VOL, TWO DRIVES
// DLAB  'OUTPUT DATA FILE                            1SSSSSS',P       C
              0001,70032,71033,'0000000000000'
// XTENT 1,004,0023001,0023001,'SSSSSS',SYS002
// XTENT 1,099,0050008,0060006,'A12BCZ',SYS019
// EXEC
```

Figure 62. Example of Using Label Information Control Statements

## Multi-File Volumes

The handling of multi-file volumes requires no special programming effort. All information required for locating the correct file is contained in the job-control statements.

### TAPE FILES

Files on a multi-file tape reel may be labeled or unlabeled. Labeled tape files are identified by file header and trailer labels. Unlabeled tape files are identified by tapemarks, that is, a tapemark separates one file from the next.

### Positioning of Labeled Tape Files

If a reel of tape contains more than one file and BACKWARDS reading is not specified, the label processing routines use the file-sequence number to position the file correctly. The file-sequence numbers in the header labels are checked against the file sequence number in the TPLAB statement, and the files on the tape are bypassed until a match is found or the end of the tape is reached. If the tape is positioned beyond the desired file when a search is started or if the file is not contained on the tape, the operator is notified.

To position labeled tape files, you may also use the FILES card. Note, however, that three tapemarks are associated with every tape file that is labeled.

For tapes that are to be read backwards, you must specify the attribute BACKWARDS and the option LEAVE for the file to be processed to prevent the normal rewinding of the tape when the file is opened. You must also use the FILES statement to properly position the file, if it has not been properly positioned by a previously executed program.

Figure 63 shows the format of a labeled multi-file reel.

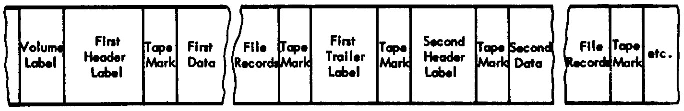


Figure 63. Format of Labeled Tape File (Multi-File Reel)

### Positioning of Unlabeled Tape Files

Unlabeled tape files are positioned on a multi-file reel in accordance with the entries of the FILES statement, which has the following format:

| Name | Operation | Operand(s) Comments |
|---|---|---|
| // | FILES | symbolic-device-address,<br>skip |

symbolic-device-address
   The name of the tape drive on which the tape to be positioned is mounted. (A complete list of symbolic-device addresses is given in the discussion of the ASSGN statements in the section Job Control).

skip
   The number of tapemarks to be skipped (1 - 999, counted from the load point) in order to position the tape.

Note that a tapemark may or may not precede the first file on the tape, while a tapemark must always follow each file.

The use of the FILES statement is illustrated by the following example.

Example:

Suppose you have a tape reel that contains unlabeled files. Each file on the tape is delimited by tapemarks as shown in Figure 64.

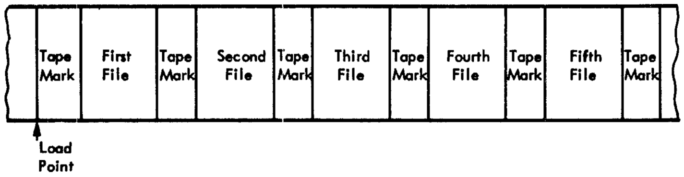


Figure 64. Unlabeled Files and Tape Marks

Case 1:

To read the first file on this tape forwards, you need not specify a FILES statement at all. To read the first file on the tape backwards, however, you must use the following FILES statement:

```
   // FILES SYSnnn,2
```

When this statement is executed, the first two tapemarks are skipped and the tape is positioned immediately behind the second tapemark. The file is then processed backwards, if you have specified the BACKWARDS attribute and the LEAVE option in the ENVIRONMENT attribute of the associated file declaration.

Case 2:

To read the third file on the tape forward, your FILES statement must look as follows:

```
   // FILES SYSnnn,3
```

The first three tapemarks are skipped and the tape is positioned at the beginning of the third file.

If the first file on the tape is not preceded by a tapemark, the FILES statement has to read as follows:

```
   // FILES SYSnnn,2
```

### DISK FILES

The handling of multi-file volumes on disk storage requires no special preparation, since all file labels are available when the file is opened. The positioning at the beginning of a file on a disk volume is done automatically by means of the file identification.

## Multi-Volume Files

Generally, the handling of multi-volume files requires no special programming effort either. All information required for processing multi-volume files must be contained in the job-control statements and in the ENVIRONMENT attribute of the associated file declaration. The only exception is with unlabeled multi-volume tape input files.

### TAPE FILES

Multi-volume tape files should always be labeled because end-of-volume conditions raised for unlabeled multi-volume tape input files are not handled automatically. You must handle them in your program as described below under the heading Positioning of Unlabeled Tape Files. Note that multi-volume tape files cannot be read backwards.

### Positioning of Labeled Tape Files

A multi-volume labeled tape file is always automatically positioned at the beginning of the first reel, if correctly mounted by the operator. The labels are checked by the system.

To enable automatic switching between tape units with labeled multi-volume files, you must specify the option ALTTAPE in the ENVIRONMENT attribute for the file to be processed. For the alternate tape, you must assign (by means of the ASSGN control statement) a symbolic device address that is one higher than the one specified in the MEDIUM option of the ENVIRONMENT attribute for the associated file. The ALTTAPE option saves you extra mounting time when an end-of-volume condition arises. (For an example illustrating the use of the ALTTAPE option, refer to Figure 62.

### Positioning_of_Unlabeled_Tape_Files

An unlabeled multi-volume tape file is always automatically positioned at the beginning of the first reel.

End-of-volume conditions that arise during the processing of unlabeled multi-volume tape input files, are not handled automatically by the system. You may, however, handle end-of-volume conditions for these files in your program.

On input, the ENDFILE condition is raised when the end of volume is reached. You may now use a control field you have inserted in your file to decide whether the end of file or the end of volume has been reached. When an end-of-volume has been reached, you must close the file and reopen it when the new reel has been mounted.

For unlabeled multi-volume tape output files, you may specify the ALTTAPE option to achieve automatic switching between volumes. For unlabeled multi-volume tape input files, the ALTTAPE option cannot be used.

## DISK_FILES

All information needed to position a multi-volume disk file correctly is contained in the job-control statements. The XTENT statements supply the necessary information regarding each extent occupied by a file. Switching from one volume to the next is automatic. It requires no programming effort on your part. However, with an INDEXED file, all volumes used by the file must be mounted when the file is opened.

### PERMANENT_AND_TEMPORARY_DISK_LABELS

For every disk file to be opened, the pertinent label information must be available to the system when the file is opened.

The required label information may either be supplied in the job-control cards preceding the program to be executed, or it may already be contained as a permanent entry in the label-information area of the system disk pack where it is checked by the Open routine when the file is opened. Permanent label information may be added to the label-information area in any Job Control run.

#### CATALOGING LABEL INFORMATION

To identify permanent label information, the DLAB control statement must contain the code P followed by a blank in the two-character field following the file serial number. For example:

```
// DLAB 'OUTFILEbbb...1SSSSSS',Pb...     C
        \_________________________/      ↑
            label information         Col.72
            (51 characters)
```

Note: If both permanent and temporary label information is specified with the same Job Control run, the control statements providing the permanent information must precede those providing temporary label information.

When processing a disk file whose label information is already cataloged, you can omit the VOL, DLAB, and XTENT statements from the job-control cards preceding the program to be executed.

The ability to catalog label information of disk files is of special importance for running inquiry programs (see Inquiry_Programs in this section).

#### DELETING CATALOGED LABEL INFORMATION

You can delete permanent labels from the label information area with the following statement:

| Name | Operation | Operand(s) |
|---|---|---|
| // | DELET | [filename1,filename2,...] |

If the operand field is blank, all permanent labels will be deleted from the label-information area.

If the operand field contains a file name, the permanent label information for that file will be deleted.

The DELET control statement enables you to make room for new labels if the label information area is full, or if the file to which the label refers has expired.

The control statement may be inserted anywhere between the JOB and the EXEC control statements but (1) it must not interrupt the sequence VOL, DLAB, XTENT or VOL, TPLAB, (2) disk-label information statements may not appear anywhere before a DELET statement.

### DISPLAYING PERMANENT LABELS

You may display all permanent labels by using the following control statement:

| Name | Operation | Operand |
|---|---|---|
| // | DSPLY | |

The DSPLY statement may be placed anywhere between the JOB and EXEC statements, but it must not inerrrupt the sequence VOL, DLAB, XTENT (or VOL, TPLAB).

The DSPLY statement is executed as soon as it is encountered. The information displayed consists of the filename, the expiration date, and the extents of the pertinent file.

The DSPLY statement enables you to determine which label information is permanent and to check the expiration date of your files.

## Inquiry Programs

Inquiry programs which are initiated by pressing the Request key on the printer-keyboard, do not begin with a Job Control run. Therefore, all disk-label information required by the inquiry program must be provided in an earlier Job Control run as permanent labels. Labeled tape files cannot be processed by inquiry programs.

## Program-Label Communication

Figure 65 shows the communication between a PL/I source program, the object program, job-control statements, and a 2311 disk drive with a 1316 disk pack.

The FILE table (I/O control) produced by the PL/I compiler somewhere contains the filename as a character string. The communication between this table and the actual file extent(s) is established by storing the extent information in the table during execution of the OPEN statement.

The VOL statement in the deck of control statements used during opening the file contains the same filename as stored in the character string of the FILE table. The symbolic device address is taken from the VOL and XTENT cards. The actual device address -- in this case that of a 2311 disk drive -- is then determined from the standard assignment or from // ASSGN statements, respectively. The serial-number field of the XTENT statement is compared against the volume label of the 1316 disk pack to determine whether the right pack has been mounted.

The remaining action depends on the type of file. For INPUT or UPDATE files, the VTOC on the disk pack is searched for a label matching the file identification entered in the DLAB statement (EMPLOYEE MASTER FILE in Figure 65). When a matching label is found, the remaining file information is checked against the label information in the VTOC, and the extent information is passed to the FILE table to allow proper addressing of the blocks to be transferred.

In the case of OUTPUT files, all existing labels in the VTOC are checked to determine whether the extents of any existing, unexpired file overlap with the extents of the file to be created. The file is opened only if there is no overlap with any unexpired file. The new label is then written into the VTOC.

If an unexpired file with the same identification as the file to be created already exists on the disk pack, the action taken depends on the file organization. If the new file is a CONSECUTIVE file, it cannot be opened. If it is an INDEXED file, it can be opened only to be extended. (Note that all extents that are required for creation of an INDEXED file and all extents that might be required later when new records are to be added, must be specified for the initial creation of the file).

In the case of CONSECUTIVE multi-volume files, one volume will be opened at a time, i.e., the second volume is opened when the last extent of the first volume has been processed, etc. Opening of the second and following volumes is automatic. Thus, no explicit OPEN statement need be given. For INDEXED multi-volume files, all volumes will be opened at once. Therefore, all volumes to be processed must be mounted at the same time.

The handling of tape-label information is similar.

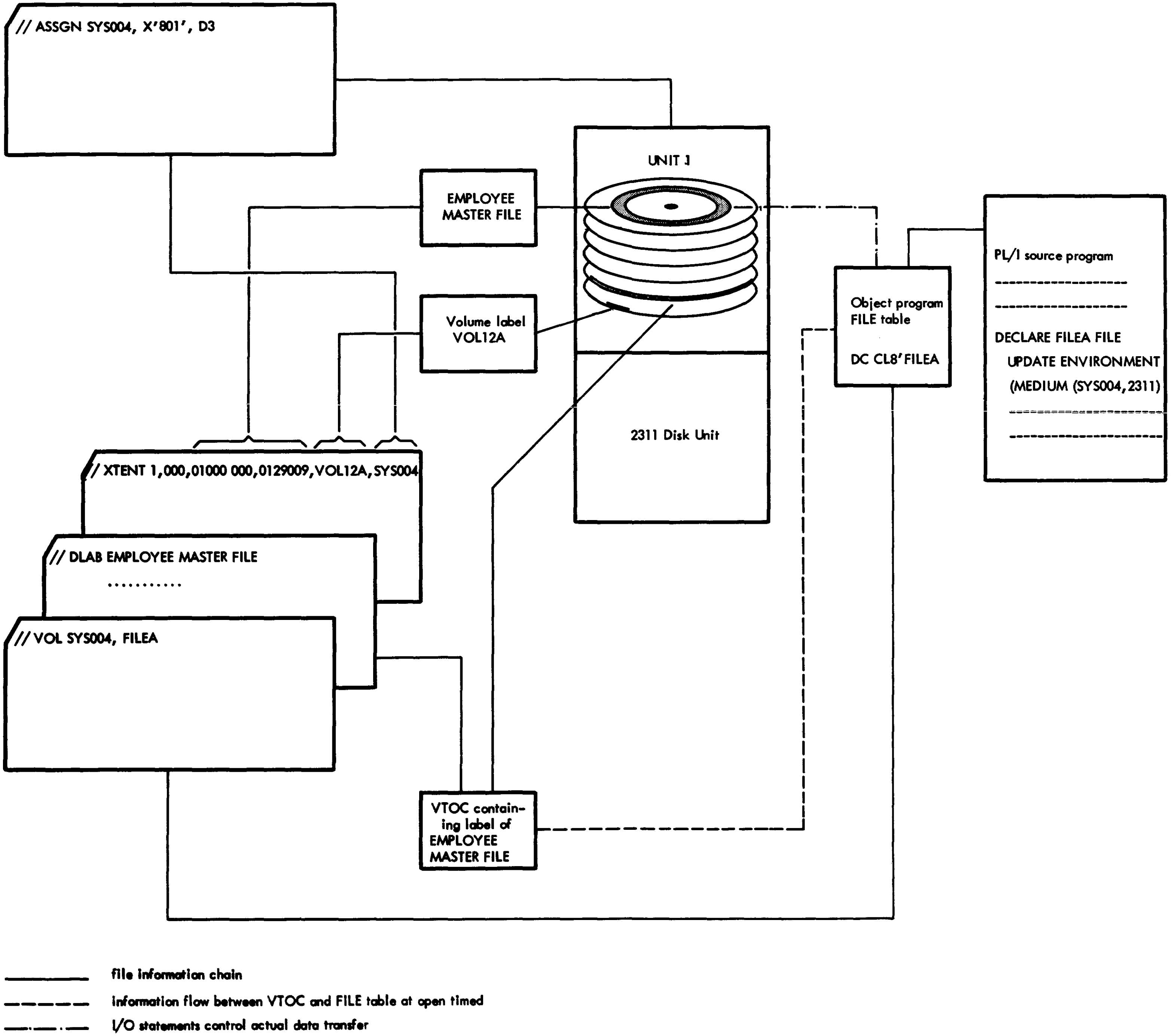

// ASSGN SYS004, X'801', D3
UNIT 1
EMPLOYEE MASTER FILE
Volume label VOL12A
2311 Disk Unit
Object program FILE table
DC CL8'FILEA
PL/I source program
DECLARE FILEA FILE
UPDATE ENVIRONMENT
(MEDIUM (SYS004,2311)
// XTENT 1,000,01000 000,0129009,VOL12A,SYS004
// DLAB EMPLOYEE MASTER FILE
............
// VOL SYS004, FILEA
VTOC containing label of EMPLOYEE MASTER FILE
file information chain
information flow between VTOC and FILE table at open timed
I/O statements control actual data transfer

Figure 65. Program - Label Communication

# Cataloging

If you have a PL/I program you frequently use, you may catalog this program into the core-image library as a permanent entry and call it each time you want it to be executed. This greatly reduces the time required for card or tape reading. When you no longer need the cataloged program, you can delete it from the core-image library and use the same storage space for other purposes.

The program used to catalog a PL/I program into or to delete it from the core-image library, is the CMAINT (Core-Image Library Maintenance) program. Therefore, the first control statement needed for cataloging or deleting a PL/I program is always the JOB control statement with the operand named CMAINT:

| Name | Operation | Operand |
|---|---|---|
| // | JOB | CMAINT |

CMAINT
Identifies the operation as a maintenance operation for the core-image library.

For cataloging a PL/I program, the CMAINT program uses the information furnished to the compiler by the SEGMENT cards to identify the segments to be added to the core-image library. Thus, a program to be cataloged must always be preceded by a SEGMENT card, even if it consists of only one segment. For a compile-and-execute run of a one-segment program, however, no SEGMENT card is needed.

You must make sure that the name of a segment to be cataloged is different from the name of any segment already cataloged in the core-image library, except if you want to replace an old segment.

## Cataloging a PL/I Program into the Core-Image Library

The control statement used to cause CMAINT to add a program segment to the core-image library is the CATAL control statement, which has the following format:

| Name | Operation | Operand |
|---|---|---|
| // | CATAL | |

A CATAL statement, encountered on SYSRDR, specifies that the next program segment is to be read and cataloged from either the device assigned to SYSIPT or the relocatable area. If read from SYSIPT, it must be an executable object program stored on either cards or magnetic tape.

When setting up the control statements for cataloging from SYSIPT, you must consider whether or not the same device is assigned to SYSIPT and to SYSRDR. Another posibility is to catalog from the relocatable area on disk. In the following, all three methods are described:

1. Cataloging from SYSIPT, with SYSIPT=SYSRDR
2. Cataloging from SYSIPT, with SYSIPT≠SYSRDR
3. Cataloging from the Relocatable Area.

Cataloging from SYSIPT (=SYSRDR)

Suppose you have a precompiled object program consisting of any number of program segments, which you want to be cataloged into the core-image library. The input is to be read from the device assigned to SYSIPT, and the same device is assigned to SYSRDR. The control statements required for such a job are shown in Figure 66.

```
// JOB CMAINT
// ASSGN SYSIPT,...
// EXEC
   .
   .  (executable object program con-
      sisting of any number of program
      segments)
   .
// END
```

Figure 66. Cataloging Program Segments from SYSIPT (=SYSRDR)

Note that you have no CATAL cards here. CATAL cards are provided automatically by the compiler, one card per segment, if the option GODECK has been specified in the COPTN control statement preceding the program.

A // END card must follow the input read from the device assigned to SYSIPT.

### Cataloging from SYSIPT (≠ SYSRDR)

The control cards needed to catalog an executable object program consisting of three segments from SYSIPT, if different devices are assigned to SYSIPT and SYSRDR are shown in Figure 67.

```
                ┌─────────────────────────────┐
Control cards   │// Job CMAINT                │
on SYSRDR       │// ASSGN SYSIPT,...          │
                │// EXEC                      │
                │// CATAL                     │
                │// CATAL                     │
                │// CATAL                     │
                │// END                       │
                │(executable object deck,     │
Input read      │consisting of three          │
from SYSIPT     │segments)                    │
                │/*                           │
                └─────────────────────────────┘
```

Figure 67. Cataloging from SYSIPT (≠SYSRDR)

For each segment in your executable object deck program you must supply a // CATAL statement on the device assigned to SYSRDR. The CATAL statements automatically provided by the compiler are ignored by CMAINT.

A // END statement must follow the control statements on SYSRDR, /* and a statement has to follow the input read from the device assigned to SYSIPT.

### Cataloging from the Relocatable Area

In order to be cataloged from the relocatable area, your program must have been compiled previously. The set of control statements needed to catalog such a program is shown in Figure 68:

```
┌─────────────────────────────────────────┐
│ // JOB CMAINT                           │
│ // EXEC R                               │
│ // CATAL                                │
│[// CATAL]                               │
│[// CATAL]                               │
│      .                                  │
│      .  (additional CATAL statements,   │
│      .  as required)                    │
│                                         │
│ // END                                  │
└─────────────────────────────────────────┘
```

Figure 68. Cataloging from the Relocatable Area

For each program segment contained in the relocatable area, you must supply one CATAL statement on the device assigned to SYSRDR. Note that, when cataloging from the relocatable area, an R must be specified in the // EXEC statement.

## Executing a Cataloged PL/I Program

The set of job-control statements needed to fetch a PL/I program from the core-image library for execution, looks as follows:

```
// JOB program-name
// EXEC
```

program-name
: The name of the root segment of the program to be executed.

When the root segment of a multi-segment program to be executed has been called into main storage by the above set of control statements, the remaining program segments are successively called into main storage and executed one at a time. This is done automatically by a procedure in the root segment, with a statement having the format:

```
CALL OVERLAY ('segment-name');
```

Thus, with the above set of control statements, you can call a program consisting of any number of segments from the core-image library into main storage for execution. (Refer to Overlay Facility in the section Practical Considerations Regarding Program Execution).

## Deleting a PL/I Program from the Core-Image Library

When you no longer need a program contained in the core-image library, you can delete it with the DELET statement, which has the following format:

| Name | Operation | Operand |
|---|---|---|
| // | DELET | segment-name |
| // | DELET | name.ALL |

segment-name
: The name of the segment to be deleted from the core-image library.

name.ALL
: Indicates that all segments whose names begin with the characters specified in the left-hand portion of the operand (name) are to be deleted from the core-image library. The name may contain one to six characters.

If, for example, all segments in your program start with the same three characters, you can delete all segments with one DELET statement, instead of specifying a DELET statement for each segment to be deleted.

Segments whose names begin with SYS or $$$ must not be deleted. If $.ALL, SY.ALL, S.ALL, or $$.ALL is specified, a halt occurs when a segment name beginning with the characters SYS or $$$ is encountered.

The operand ALL must not be the only operand of the DELET statement in CMAINT programs.

Example:

Suppose you want to delete from the core-image library a program that consists of five segments whose names all start with the same two characters PS, but no other segments start with the two characters. The set of control statements needed to delete all five segments of the program would look as shown in Figure 69.

```
// JOB CMAINT
// EXEC
// DELET PS.ALL
// END
```

Figure 69. Deleting a PL/I Program from the Core-Image Library

The following sections provide you with the information you need to fully use all of the facilities of PL/I under the Disk Programming System.

(At the present stage, this section contains only preliminary information).

## Inquiries on the IBM 2152 Printer-Keyboard

The IBM 2152 Printer-Keyboard can be used as an inquiry device. It is used for fast retrieval of information from disk files, with only a brief interruption of processing in progress. You just enter the name of your inquiry program on the printer-keyboard. The job in progress is interrupted while the inquiry is made.

The Model 20 DPS support for the printer-keyboard as an inquiry device consists of interrupt routines. Through the use of these IBM-supplied routines, an inquiry request entered on the printer-keyboard interrupts the job in progress, saves the status of that job, and initiates the processing of a specified user-written program. After this program has been processed as an inquiry program the interrupted program is returned from the system disk pack to main storage and its execution is resumed. The intervening program, which must be in the core-image library, is called into main storage by entering the program name on the printer-keyboard. It can occupy all of main storage, except for the part required for the Monitor, and can use any I/O devices, provided such use does not inerfere with the status of the interrupted program. This versatility enables it to perform the operations required of any program executed on the Model 20. The interrupted program is referred to as the mainline program.

### INQUIRY PROGRAMS

You can write an assortment of inquiry programs and store them in the core-image library. An inquiry program specifies the operations to be performed, using the inquiry file and any other files that are necessary. These normally include only disk files from which records designated in a printer-keyboard INPUT file are to be retrieved. Note that an inquiry may be requested during the processing of a wide variety of mainline programs. If these mainline programs are operating with tape or card devices that are also accessed by the inquiry program, it may not be possible to restore the interrupted mainline program to its original status.

Some Model 20 DPS programs and certain routines in such programs must not be interrupted by an inquiry program. These programs and routines are listed in the SRL publication _IBM System/360 Model 20, Disk Programming System, Operating Procedures_, Form C33-6004.

The inquiry program, like any other program, requires that label information be supplied in order to open disk files to be used in the program. The disk files, of course, must be on-line at the time the inquiry is requested. The label information submitted to the Job Control program in VOL, DLAB, and XTENT statements must have been cataloged permanently in a previous Job Control Run into the label-information area. (Refer to _Permanent and Temporary Disk Labels_ in the section _Input/Output_.

All programs compiled with the Model 20 PL/I compiler can be used as mainline programs as well as inquiry programs. However, programs running as inquiry programs must not use labeled tape files. To protect a PL/I program from being interrupted by an inquiry program, the job-control statement // OPTN NOINQ must be used. The PL/I compiler itself runs only as mainline program.

_Note_: The inquiry input and output area of the Monitor and the inquiry record are not used by PL/I. Input/output on the IBM 2152 Printer-Keyboard is through normal PL/I file processing.

### USE OF THE IBM 2152 PRINTER-KEYBOARD

Even though all types of file processing available with Model 20 PL/I are supported by the IBM 2152 Printer-Keyboard, its main purpose is the transfer of messages between the program and the operator.

Since messages are mostly of variable length, undefined record format has been introduced as the only record format for the printer-keyboard. If, however, with RECORD transmission, an input record does not have the same length as the variable into which it is read, the RECORD condition is raised.

The transfer of messages between the program and the operator requires that the input and the output file are synchronized in such a way that the listing on the keyboard reflects the sequence in which the input and output statements have been executed. This synchronization is guaranteed if both files have the option BUFFERS(1) in the ENVIRONMENT attribute. Otherwise, read requests are issued to the printer-keyboard before PL/I GET or READ statements are executed.

The following set of additional rules applies to files that are read from or written on the printer-keyboard:

1. For STREAM files, the actual length of an output record is defined as follows:

   - By an explicit occurrence of a SKIP option or format item.
   - By an implicit occurrence of SKIP( 1) when the maximum record length (declared in the U option of the ENVIRONMENT attribute) is being exceeded.
   - When a GET or READ statement for a printer-keyboard file is executed, any output produced by previous PUT statements that has not been written out, is printed before the keyboard is unlocked, that is, before the input data can be typed in.

2. SKIP(0) is not supported for printer-keyboard files since over-printing is not possible.

3. CTLASA is meaningless for printer-keyboard files.

4. Since pages are not defined on a printer-keyboard, the PRINT attribute and the PAGESIZE option are ignored for printer-keyboard files. The PAGE option or format item is treated as SKIP(3). The ENDPAGE condition cannot be raised.

## Linking PL/I Programs with Assembler Procedures

In order to increase the capability and/or efficiency of your programs, you may combine programs written in PL/I with procedures written in Assembler language (in the following referred to as Assembler procedures).

For a PL/I program consisting of several procedures, the compiler automatically generates the code required for the activation and de-activation of the procedures. When writing Assembler procedures that are to be linked with PL/I procedures, however, you must write the code required for the activation and deactivation of other procedures yourself. The code you must supply corresponds exactly to the code generated automatically by the compiler for PL/I procedures.

Following, now, is a description of the code that will be of interest for you when writing Assembler procedures that are to be included in PL/I programs.

Figure 70 shows the code generated by the compiler for each procedure of a PL/I program.

```
entry-name    DC X'FFFF')
               .       (
               .       >         (for n addresses of parameters)
               .       (
              DC X'FFFF')
              DC X'FFFF'         (for address of function value)
              DC H'0'            (end of parameter list)
              DC Y(ONBLOCK)      (address of On-Block)
               .
               .                 (code generated for the statements of the procedure)
               .
ONBLOCK       DC X'FF00'
              DC X'...'          (length of dynamic-storage area)
              DC C'...'          (entry-name, six bytes in length)
               .
               .                 (On-Block entries)
               .
              DC X'7FFF'         (end of On-Block)
```

Figure 70. Code Generated by the Compiler for Each Procedure of a PL/I Program

```
            DC X'4D80'       (BAS instruction to invoke the Library Call Routine)
            DC Y(PL1SCA)
            DC Y(...)        (address of called procedure)
            DC Y(...)*  )
             .          (
             .          >    (addresses of arguments)
             .          (
            DC Y(...)*  )
            DC Y(...)*       (address of function value)
            DC H'0'          (end of argument list)
---------------------------------------------------------------------------------
*All addresses marked by an asterisk may also consist of a register and a displacement.
```

Figure 71. Code for a CALL Invoking Another Procedure

Call

Whenever, during program execution, one procedure calls another procedure, the Call routine PL1SCA, a member of the PL/I library, is entered.

This routine

- provides the Dynamic Storage Area (DSA) [1]
- stores the register contents and the return address of the calling procedure in the DSA
- provides On-Block[1] information
- transmits the addresses of arguments
- causes the branch to the called procedure

[1] DSA and On-Block are discussed below in this section.

To include in a PL/I program an Assembler procedure that calls a PL/I procedure, you must yourself supply the code needed for the call.

The code generated automatically for the CALL statement is shown in Figure 71.

Arguments and Parameters

Arguments are passed to the invoking procedure by transferring the addresses of the arguments to the invoked procedure. In the invoked procedure, the arguments are referred to by names called parameters.

During transmission of the addresses, a test is performed to determine whether the addresses are absolute. If they are not, they are converted to absolute addresses. Also passed to the invoked procedure is the address of a function value that might be returned by the invoked procedure. This address is specified even if no function value is returned.

Note: The number of arguments of a call must be identical to the number of parameters in the called procedure.

On-Block

The On-Block holds all information needed about ON-conditions during activation of a procedure. Its address is contained in the code for the called procedure.

For Assembler procedures, the On-Block must only contain the following:

```
DC X'FF00'
DC X'length of DSA'
DC C'entry-name'   (6 bytes)
DC X'7FFF'
```

Note: The On-Block must begin on a halfword boundary.

Dynamic Storage Area

Since a procedure needs working storage for its own purposes, storage is dynamically allocated for it during activation by the Call routine; This dynamically allocated storage is freed again by the Return routine PL1SCAR, a member of the PL/I library, upon deactivation of the procedure.

The DSA is used to save register contents and the return address of the calling procedure and to hold any intermediate results used only by the called procedure. Its length is given in the On-Block of a procedure. It must have a length of at least 24 bytes. These 24 bytes are needed for internal use. To use additional dynamic storage for an Assembler procedure, you can increase the length of the DSA up to 4096 bytes. You can address the DSA via register 13. This register is set by the Call routine and reset by the Return routine.

Note: If you use register 13 for other purposes in an Assembler procedure, you must save it and restore it later. You must not use register 13 for other purposes during activation of another PL/I procedure.

### Return

When a procedure returns control to the calling procedure, this is done via the Return routine PL1SCAR which is entered. This routine

- restores the register contents
- frees the Dynamic Storage Area
- passes control back to the calling procedure

The code generated for a return is a branch to PL1SCAR and looks as follows:

```
DC X'47F0'
DC Y(PL1SCAR)
```

For an Assembler procedure, you have to supply this return code yourself.

Notes:

1. When combining PL/I procedures with Assembler procedures, the main procedure must always be written in PL/I; it must not be coded in Assembler.
2. To combine Assembler procedures with PL/I procedures, include the already assembled procedures in your PL/I source program or precompiled object program. The assembled object deck(s) (as well as any precompiled PL/I object deck(s), must be headed by a + COPY card. (For more information about the COPY card, refer to the section entitled The Compiler).

### Rules and Restrictions for Writing Assembler Procedures

When writing Assembler procedures to be combined with PL/I procedures, you must observe the following rules and restrictions:

Rules:

1. An Assembler procedure may have more than one entry point. All entry names of an Assembler procedure have to be declared with the ENTRY (or START or CSECT) statement.
2. If a PL/I procedure is called from within an Assembler procedure, the name of the PL/I procedure must be declared with the EXTRN statement in the Assembler procedure. Names declared with the EXTRN statement must appear only within DC Y(...) constants.

Restrictions:

1. An Assembler procedure used in combination with PL/I procedures must not contain any Monitor or IOCS macro instructions.
2. It must not contain any CIO, XIO, or TIOB instructions.
3. It must not change the communication region and must not contain an XFR Assembler instruction.
4. The Assembler procedure must be relocatable; that is, it must not contain the NOESD or NORLD options in the AOPTN control statement, nor contain the Linkage-Editor ENTRY statement.
5. When linking PL/I with Assembler procedures, the main procedure must always be written in PL/I.

### Programming Example

In order to illustrate the use of an Assembler procedure in combination with PL/I procedures, a small programming example consisting of three procedures is given in Figure 72.

The problem dealt with in the program is to determine whether the current year is a leap year or not.

The main procedure ALPHA is written in PL/I. ALPHA calls the procedure LEAP, which returns, as a function value, the letter Y or N, depending on whether the current year is a leap year (Y) or not (N). The returned letter is displayed in the E-S-T-R registers on the pannel. However, before LEAP can return its function value, it calls the function YEAR which returns to LEAP a two-digit number specifying the current year. The procedure YEAR is also written in PL/I, whereas the procedure LEAP is written in PL/I as well as in Assembler language. The two procedures are equivalent.

The Assembler and the PL/I coding for the procedure LEAP are shown in parallel in Figure 72 for comparison.

```
+---------------------------------------------------------------------------+
|ALPHA: PROCEDURE OPTIONS(MAIN);                                            |
|       DCL LEAP RETURNS CHAR(1), A CHAR(1);                                |
|       DISPLAY (LEAP) REPLY(A); END;                                       |
+--------------------------+------------------------------------------------+
|LEAP:  PROCEDURE CHAR(1); |LEAP    START 0                                 |
|       DCL V,W FIXED (2,0),|        BASR  8,0                               |
|           YEAR RETURNS PIC'99';|  EXTRN PL1SCA        CALL ROUTINE        |
|                          |        EXTRN PL1SCAR       RETURN ROUTINE      |
|                          |        EXTRN YEAR          PL/I PROCEDURE TO BE|
|                          |                            CALLED              |
|                          |RA      EQU   9             WORKING REGISTER    |
|                          |L0      DC    X'FFFF'       HALFWORD FOR ADDRESS OF |
|                          |                            FUNCTION VALUE      |
|                          |        DC    X'0000'                           |
|                          |        DC    Y(ONBLOCK)    POINTER TO ONBLOCK  |
+--------------------------+------------------------------------------------+
|       V=YEAR;            |        DC    X'4D80'       BAS INSTRUCTION TO  |
|                          |        DC    Y(PL1SCA)     CALL PROC YEAR      |
|                          |        DC    Y(YEAR)       ADDRESS OF CALLED   |
|                          |                            PROCEDURE YEAR      |
|                          |        DC    Y(A)          ADDR OF FUNCTION VALUE |
|                          |        DC    X'0000'                           |
|                          |        PACK  B(3),A(2)     PACK VALUE RETURNED FROM |
|                          |                            YEAR                |
+--------------------------+------------------------------------------------+
|       W=V/4;             |        DP    B(3),V(1)     DIVIDE IT BY 4      |
|       IF V=W*4 THEN      |        CLI   B+2,X'0C'     TEST REST FOR 0     |
|                          |        BNE   L1            IF YES, LEAP-YEAR   |
+--------------------------+------------------------------------------------+
|       RETURN('Y');       |        LH    RA,L0         ELSE,NO LEAP-YEAR   |
|       ELSE               |        MVI   0(RA),C'Y'    RETURN C'Y' FOR RESULT |
|                          |        DC    X'47F0'                           |
|                          |        DC    Y(PL1SCAR)    BRANCH TO RETURN ROUTINE |
+--------------------------+------------------------------------------------+
|       RETURN('N');       |L1      LH    RA,L0         LOAD ADDR OF RETURN VALUE |
|                          |        MVI   0(RA) C'N'    RETURN C'N' FOR RESULT |
|                          |        DC    X'47F0'                           |
|                          |        DC    Y(PL1SCAR)    BRANCH TO RETURN ROUTINE |
|                          |V       DC    X'4C'         CONSTANT 4          |
|                          |A       DS    CL2           VALUE RETURNED FROM YEAR |
|                          |B       DS    CL3           WORKING STORAGE     |
+--------------------------+------------------------------------------------+
|       END;               |ONBLOCK DC    X'FF00'       BEGIN OF ONBLOCK    |
|                          |        DC    H'24'         LENGTH OF DSA       |
|                          |        DC    C'LEAPbb'     PROCEDURE NAME      |
|                          |        DC    X'7FFF'       END OF ONBLOCK      |
|                          |        END                                     |
+--------------------------+------------------------------------------------+
|YEAR:  PROCEDURE CHAR(2);                                                  |
|       DCL DATE BUILTIN;                                                   |
|       RETURN (DATE);                                                      |
|       END;                                                                |
+---------------------------------------------------------------------------+
```

Figure 72. Programming Example Illustrating the Use of an Assembler Procedure in Combination With PL/I Procedures

## Sterling Currency Processing Routines

Model 20 PL/I does not support Sterling. However, the Model 20 Sterling Currency Processing Routines (260-LM-015) can be linked to PL/I object programs. The linkage of these routines to PL/I programs must be done like the linkage of Assembler procedures.

For detailed information, refer to the publication _IBM System/360 Model 20, Sterling Currency Processing Routines_, Form C26-3605.

## The DYNDUMP Routine

Model 20 PL/I provides you with a dynamic dump facility that can be called at execution time with the statement

```
CALL DYNDUMP
   (variable-name [,variable-name]...);
```

This statement may be used to display the items listed in the variable list in hexadecimal notation. The variable list may contain up to 12 variables and include label and pointer variables.

The following example shows the use of the DYNDUMP routine:

```
DCL A FIXED(5,2),(B(20),C)CHAR(2);
     .
     .
     .
CALL DYNDUMP (A,B,C);
```

The three items A, B, and C are displayed: A as three bytes (five hexadecimal digits + sign), B as 40 bytes (80 hexadecimal digits), and C as two bytes (four hexadecimal digits).

This feature requires a minimum of main storage.

## Data Storage Mapping

This section discusses the location of a data item in storage in relation to other data items.

### Alignment of Data Items in Storage

The Model 20 PL/I compiler aligns all data items -- except label variables and pointer variables -- on one-byte boundaries, that is, the data items in storage are aligned one after the other without any storage space being lost between the individual items. Only label variables and pointer variables are aligned on two-byte boundaries, that is, the first byte of a label or pointer variable is in a storage position where the address is divisible by two.

### STORAGE MAPPING OF ARRAYS

The storage requirement of an array is the sum of the requirements of the individual data items contained in the array. For example, the storage requirement of an array declared in the statement

```
   DCL A(5,4,3) CHAR(2);
```

can be calculated as follows: The number of data items in the array is 5 * 4 * 3 = 60. Each data item requires two bytes as declared. Thus, the total storage requirement of the array is 2 * 60 = 120 bytes.

The individual items of an array are stored in row-major order. For the above example, this means that the items are stored as follows:

```
   A(1,1,1)
   A(1,1,2)
     ...
   A(5,4,2)
   A(5,4,3)
```

### STORAGE MAPPING OF STRUCTURES

With structures, storage is required only for data on the elementary structure level. Names of major and minor structures require no storage space. Elementary items are stored in the order in which they are declared within the structure, the length of each individual item being dependent on the attributes declared for it. The length of a structure is equal to the sum of the lengths of all elements contained in it.

The starting point of a major or minor structure is the starting point of its first element.

Note: The pointer and label variables cannot be elements of structures. Thus, no boundary problems can appear in structure mapping.

Figure 73 illustrates the storage mapping of the structure declared as follows:

```
DECLARE 1 A,
          2 B,
          2 C,
            3 D,
            3 E,
          2 F,
            3 G,
            3 H,
              4 I,
              4 J,
                5 K,
                5 L;
```

```
┌───┬───┬───┬───┬───┬───┬───┐
│ B │ D │ E │ G │ I │ K │ L │
└───┴───┴───┴───┴───┴───┴───┘
                      J-------
                 H------------
             F----------------
     C------------------------
A-----------------------------
```

Figure 73. Storage Mapping of Structures -- Example 1

| Variable Type | Stored Internally as | Storage Requirements | Alignment Requirements |
|---|---|---|---|
| CHARACTER(n) | 1 Byte per Character | n | Byte |
| PICTURE | 1 Byte for each PICTURE Character except for V | Number of PICTURE Characters | Byte |
| DECIMAL FIXED (w,d) | 1/2 Byte Per Digit Plus 1/2 Byte for Sign | CEIL $\frac{w + 1}{2}$ | Byte |
| DECIMAL FLOAT (w) w < 7 | Short Floating Point | 5 | Byte |
| DECIMAL FLOAT (n) 6 < w < 16 | Long Floating Point | 9 | Byte |
| LABEL | --- | 4 | Half-Word |
| POINTER | --- | 2 | Half-Word |

Figure 74. Summary of Data Alignment Requirements

Explanation: Only the elementary data items B, D, E, G, I, K, and L actually occupy storage space. A, the major structure name, and the minor structure names C, F, H, and J occupy no storage space. The range of the structure names is indicated by the vertical lines in Figure 73. For the length of each data item, see Figure 74.

Figure 75 shows storage mapping of a fully declared structure. The structure has been declared as follows:

```
1 A,
  2 B PICTURE '999V99',
  2 C,
    3 D CHAR (3),
    3 E FIXED (5,2),
  2 F FLOAT (6),
  2 G CHAR (1);
```

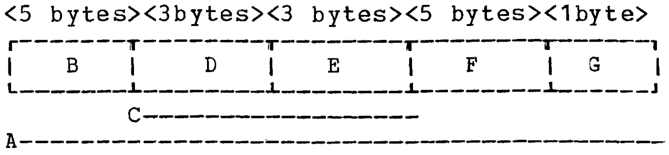


Figure 75. Storage Mapping of Structures -- Example 2

Explanation: B, D, E, F, and G are elementary data items and occupy actual storage space according to the rules listed in Figure 74. A and C are structure names and require no storage space. Their range is indicated by the vertical lines in Figure 75.

# Two Programming Examples

This section discusses two complete programming examples:

(1) an example for scientific application, and

(2) an example for commercial application.

## Example for Scientific Application

The example shown in Figure 76 illustrates numerical integration of an ordinary differential equation, using the method of Runge and Kutta. Since this method can be used for numerical integration of more than one type of ordinary differential equation, a main procedure (RUKU) was written for the Runge-Kutta method, and a separate procedure (YPRIME) to evaluate a special differential equation y' = f(x,y).

Both procedures are to be compiled together and executed immediately. In addition, two decks are to be produced: a compiled object deck from the main procedure, and an executable object deck from the complete program.

The compiled object deck can -- in later applications with other equations -- be used again as compiler input (headed by a + COPY card) together with a PL/I source module for any new equation.

The executable object deck can be used to solve the same problem again with different parameters.

The parameters are the starting values for x0, y0, the end value xn, and the step width h. They are read from a card. The result of each step is printed.

The differential equation used in this example is:

y' = 2 * y * cotan x

Explanation:

1 The first two job-control statements specify that the program is to be compiled, link-edited, and executed immediately. No other job-control statements are used; it is assumed that the VOL, DLAB, and XTENT statements for the two work files WORK1 and WORK2 are cataloged as permanent labels.

2 The COPTN statement is a compiler-control statement. The LINK option is required so that the program can be link-edited and executed immediately. The option WORK2 specifies that two work files are to be used to achieve faster compilation. The option GODECK indicates that an executable object deck is to be provided by the compiler.

3 The PROCESS compiler-control statement signals that a PL/I source deck follows. The option DECK specifies that a compiled object deck is to be provided. The option SOURCE specifies that the source program is to be printed.

4 Comments in the form of headings are used in the program to document the requirements for the parameters as well as to indicate the main program steps.

5 Although the parameters X0, Y0, H and XN need not be declared explicitly since their attributes are the default attributes FLOAT DECIMAL (6), they are explicitly declared for documentation purposes.
YPRIME is explicitly declared as an entry point by the RETURNS attribute.

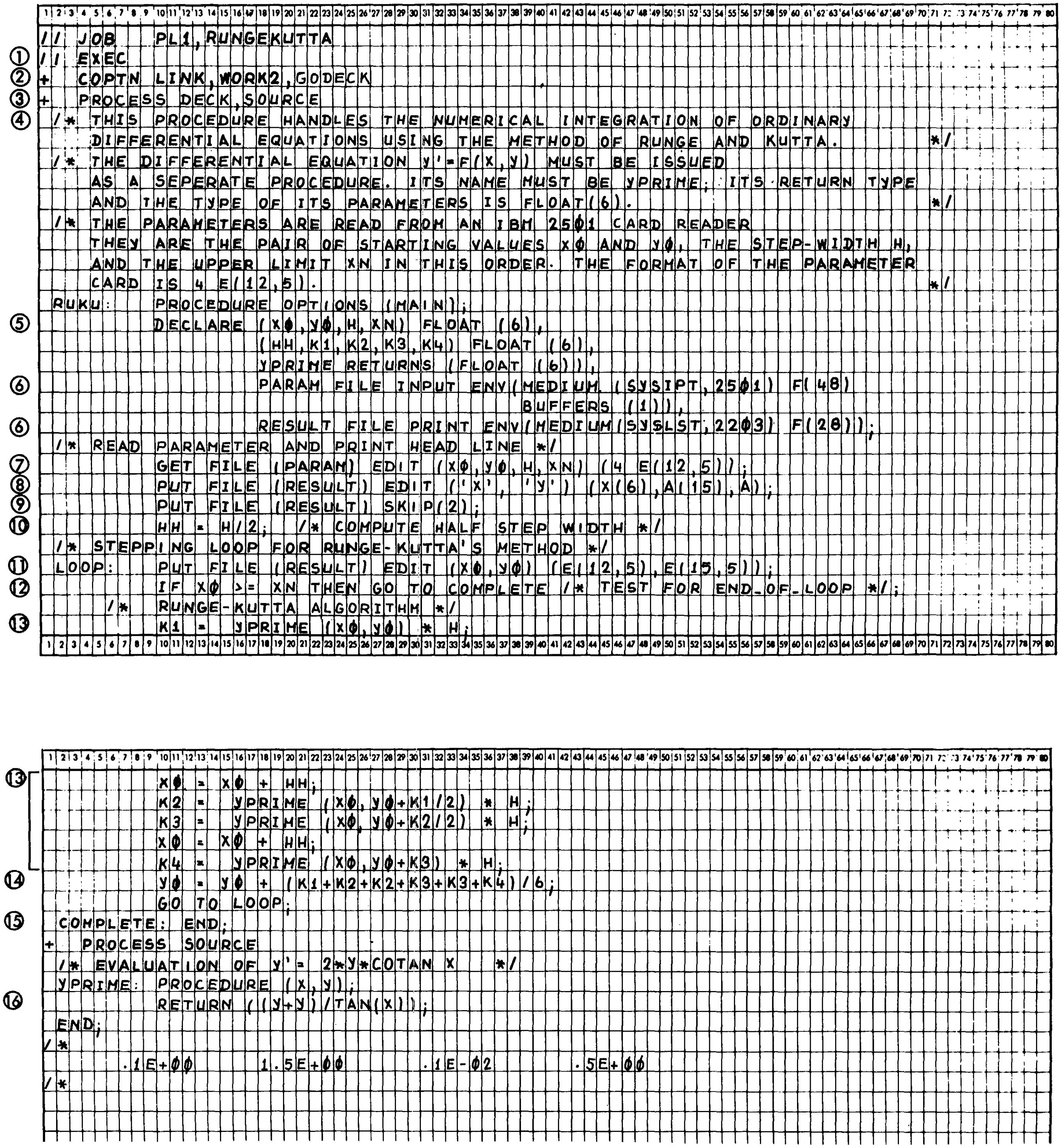
6 PARAM is the file from which the parameters are read. In this case, it is one card read from a 2501 card reader assigned to SYSIPT. PARAM is by default a STREAM file. RESULT is the output file, which is printed on a 2203 printer assigned to SYSLST. It is a STREAM file, which is implied by the attribute PRINT.

7 As the first action, the parameters are read in.

8 Then, a heading is printed.

9 Following the heading, one line is left blank.

10 The step length divided by two is used within the loop that follows. To avoid repeated computation of this value (HH), it is computed once outside the loop. This saves execution time.



```
    // JOB    PL1,RUNGEKUTTA
①   // EXEC
②   +  COPTN LINK,WORK2,GODECK
③   +  PROCESS DECK,SOURCE
④    /* THIS PROCEDURE HANDLES THE NUMERICAL INTEGRATION OF ORDINARY
        DIFFERENTIAL EQUATIONS USING THE METHOD OF RUNGE AND KUTTA.        */
     /* THE DIFFERENTIAL EQUATION Y'=F(X,Y) MUST BE ISSUED
        AS A SEPERATE PROCEDURE. ITS NAME MUST BE YPRIME; ITS RETURN TYPE
        AND THE TYPE OF ITS PARAMETERS IS FLOAT(6).                        */
     /* THE PARAMETERS ARE READ FROM AN IBM 25Ø1 CARD READER
        THEY ARE THE PAIR OF STARTING VALUES XØ AND YØ, THE STEP-WIDTH H,
        AND THE UPPER LIMIT XN IN THIS ORDER. THE FORMAT OF THE PARAMETER
        CARD IS 4 E(12,5).                                                 */
     RUKU:    PROCEDURE OPTIONS (MAIN);
⑤             DECLARE (XØ,YØ,H,XN) FLOAT (6),
                       (HH,K1,K2,K3,K4) FLOAT (6),
                       YPRIME RETURNS (FLOAT (6)),
⑥                      PARAM FILE INPUT ENV(MEDIUM (SYSIPT,25Ø1) F(48)
                                             BUFFERS (1)),
⑥                      RESULT FILE PRINT ENV(MEDIUM(SYSLST,22Ø3) F(28));
     /* READ  PARAMETER AND PRINT HEAD LINE */
⑦             GET FILE (PARAM) EDIT (XØ,YØ,H,XN) (4 E(12,5));
⑧             PUT FILE (RESULT) EDIT ('X', 'Y') (X(6),A(15),A);
⑨             PUT FILE (RESULT) SKIP(2);
⑩             HH = H/2;  /* COMPUTE HALF STEP WIDTH */
     /* STEPPING LOOP FOR RUNGE-KUTTA'S METHOD */
⑪    LOOP:    PUT FILE (RESULT) EDIT (XØ,YØ) (E(12,5),E(15,5));
⑫             IF XØ >= XN THEN GO TO COMPLETE /* TEST FOR END_OF_LOOP */;
           /* RUNGE-KUTTA ALGORITHM */
⑬             K1 =  YPRIME (XØ,YØ) * H;
```

```
⑬ ┌           XØ = XØ + HH;
   │           K2 =  YPRIME (XØ,YØ+K1/2) * H;
   │           K3 =  YPRIME (XØ,YØ+K2/2) * H;
   │           XØ = XØ + HH;
   └           K4 =  YPRIME (XØ,YØ+K3) * H;
⑭             YØ = YØ + (K1+K2+K2+K3+K3+K4)/6;
              GO TO LOOP;
⑮    COMPLETE: END;
    +  PROCESS SOURCE
     /* EVALUATION OF Y'= 2*Y*COTAN X   */
     YPRIME:  PROCEDURE (X,Y);
⑯             RETURN ((Y+Y)/TAN(X));
     END;
    /*
          .1E+ØØ     1.5E+ØØ      .1E-Ø2      .5E+ØØ
    /*
```

Figure 76. Programming Example for Scientific Application

11 The PUT statement that edits the values of the individual integration steps precedes the integration loop so that also the initial values of x0 and y0 are printed. Since the record length (28) specified in the ENVIRONMENT attribute for the PRINT file is equal to the length of the two fields for x0 and y0 (12 and 15) plus one character for printer-carriage control, a new line is printed each time the PUT statement is encountered, even though no SKIP format item is specified.

12 A test for completion is made. > = is used instead of =, since round-off errors in floating-point arithmetic may prevent an equal compare.

13 The formulas used for the integration from the point (xi,yi) to

$(x_{i+1} = x_i + h)$ are:

$$k_1 = f(x_i, y_i) * h$$

$$k_2 = f(x_i + h/2, y_i + k_2/2) * h$$

$$k_3 = f(x_i + h/2, y_i + k_2/2) * h$$

$$k_4 = f(x_i + h, y_i + k_3) * h$$

$$y_{i+1} = y_i + (k_1 + 2k_2 + 2k_3 + k_4)/6$$

Since the values of xi,yi are no longer needed when xi+1,yi+1 have been computed, the same variables used for the old values are used for the new values, i.e., there is no need to declare x and y as arrays. Since the argument xi + h/2 is used twice, namely in the computation of k2 and k3, xi is incremented once by h/2 before the computation of k2 and k3, thus avoiding duplicate computation of this expression. X0 itself can be replaced by X0+HH since the old value of X0 is no longer needed.

14 K2+K2 has been chosen instead of 2 * K2, since addition is normally faster than multiplication.

15 When the END statement of the main procedure is executed, all files are automatically closed and control is returned to the Monitor.

16 The RETURN statement is the only executable statement in the procedure YPRIME. Since YPRIME, X, and Y have default attributes, no DECLARE statement and no RETURNS attribute are required. YPRIME uses the TAN built-in function. Note that tan(x) = 1/cotan(x).

## Example for Commercial Application

The programming example shown in Figure 78 is a simplified inventory problem. Different departments of a factory order different quantities of different parts (screws, nuts, nails, etc.) from a central store. If orders cannot be fulfilled because there are not enough parts available from the store or if, after fulfilling the order, less than the required minimum number of parts remains in the store, a purchase order is issued automatically.

The main file is an inventory file, called MASTER, residing on disk. It contains all information about the individual parts in the store, like part-number, number of parts currently in store, price, minimum number to be held in store, etc. This file may very well also be used by other inventory programs.

The orders coming in from the factory departments are on punched cards (DETAIL file). They are not sorted in ascending sequence by part number. Therefore, the MASTER file is accessed as an INDEXED DIRECT file, since the DETAIL records coming in in random order require random processing of the MASTER inventory file. (For rather large detail files it may be advisable to sort the detail file with the DPS Sort/Merge program and then process the sorted output against a SEQUENTIAL UPDATE INDEXED or CONSECUTIVE inventory file. However, this case is not handled in the sample program).

Records for parts that are no longer held in store are marked by a character code and deleted from the MASTER file when it is eventually reorganized.

The program produces three output files: (1) a transaction report file (REPORT), which is initially written on tape and is used later to produce a printed output report of the orders fulfilled, (2) a card file (REORDR) in the form of reorders to refill the store, and (3) an exception file (EXCEPT), which is printed immediately and contains detail records for which no MASTER records exist or for which the MASTER records have been marked for deletion, as well as any orders that cannot be fulfilled.

Since the printer is used for printing the EXCEPT file, the REPORT file is stored intermediately on magnetic tape. The REPORT file might be used by other programs for the printing of orders or the computation of product costs, etc. Therefore, the REPORT file is labeled to make it easily identifiable. Figure 77 shows the configuration used for this example.

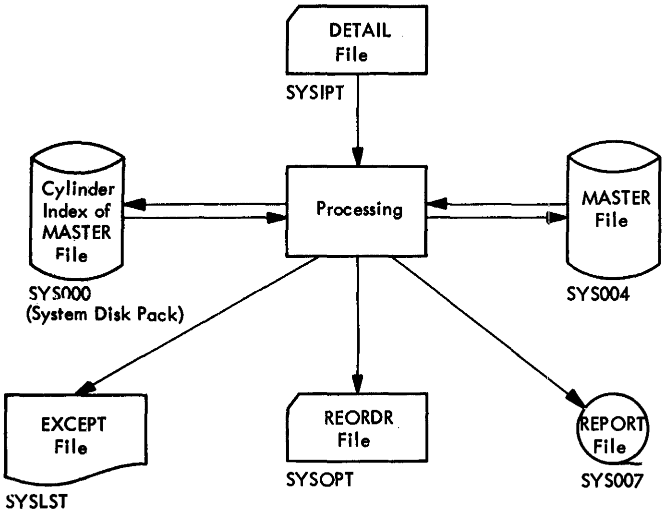


Figure 77. Configuration Used for Inventory Problem

Explanation:

Note that the numbers to the left of the programming example are used for reference purposes only. They are not part of the coding.

1 The first job-control statement specifies that the program is to be compiled, link-edited, and executed immediately.

2 NOINQ in the OPTN control statement specifies that the execution of this program must not be interrupted by any inquiry program. TES specifies that the tape error statistics are to be printed after program execution. NOINQ is used to ensure that the position of the tape and card file is not changed by an inquiry program.

3 When TES is specified, LOG must also be specified.

4 The 2311 disk drive, which holds the prime data extent of the MASTER file, is assigned the symbolic-device address SYS004 and the unit number 02.

5 The index area of the MASTER file is on the system disk pack with the symbolic device address SYS000.

6 MASTER INVENTORY FILE is the file identification. The numbers in columns 55 through 61 are the format (always coded as 1) and the file serial number which must correspond to the file serial number of the XTENT statement (the first in this example) for the cylinder index area of the MASTER file. The numbers in the continuation card are the volume sequence number (0000), the creation date (January 2nd, 1970), and the expiration date.

7 While the prime data extent is on an extra pack with the volume label INV002, the index area is on another pack, the system disk pack, to minimize disk-arm movement time when reading the cylinder index.

8 The tape used for intermediate storage of the REPORT file is a 9-track magnetic tape with a density of 1600 bytes per inch.

9 Label information for the REPORT file is the file identification INVENTORY REPORT (44-character field padded with blanks) followed by the file serial number (6 characters) the volume sequence number (4 characters), the file sequence number (4 characters), the generation number (4 characters) indicating that this is the first edition of the file, the version number (2 characters) indicating that this is the first version of the generation of this file, the creation date (Jan. 23rd, 1970), and the expiration date.

10 The COPTN compiler-control statement indicates that two work files are used (WORK2), that the source program is to be compiled and link-edited (LINK), and that the executable object deck is to be produced on the device assigned to SYSOPT (GODECK). No label control statements have been provided for the two work files. We assume that these labels are permanent.

11 The PROCESS compiler-control statement indicates that no compiled object deck is to be produced on SYSOPT (NODECK, specified for documentation purposes; it need not have been specified, since it is applied by default anyway). ATRO causes the printing of the offsets of labels and variables and the length of automatic storage. ATR causes the listing of all attributes of all variables, entry names, and file names.

12 Columns 73 to 80 are ignored by the compiler. In this case, they are used for sequential numbering of the cards, so that -- in case of accidental scattering -- they can easily be sorted again.

13 The main procedure of the program, that is, the procedure initially invoked by the system, must always have OPTIONS(MAIN) specified in the PROCEDURE statement.

14 The MASTER file is declared to be an INDEXED DIRECT UPDATE file. Note that DIRECT and UPDATE are file attributes while INDEXED is an option of the ENVIRONMENT attribute. Since no new records are added to MASTER when it is updated, NOWRITE has been specified to cause loading of a smaller library routine, which saves main storage space.
The record length 41 is the number of bytes actually needed to hold all the information required for one part. It is the length of the structure MAST REC. The block size of 533 was chosen when the file was built so that a minimum of the disk extent (7 bytes per 2 sectors) is unused.

The key of each record has a length of eight bytes and starts in the second byte of each record (KEYLOC(2)). Two extents are used (see XTENT control statements), one for the prime data area, and one for the cylinder index. One track per cylinder is reserved for overflow records.

VERIFY is specified to ensure error detection when the updated records are rewritten onto disk.

15 The DETAIL file contains the orders coming in from the different factory departments. Only the first 25 bytes of each card are used.

16 Following are the structure declarations for the MASTER record, the DETAIL record, the REORDR record, and the REPORT record (TRANSACT). In order to have the individual data items of the TRANSACT record separated by blanks when it is finally printed, "fillers" initialized to blanks, have been introduced in the record.

17 An invalid part number specified in the records of the file DETAIL will raise the KEY condition when the file MASTER is read. The raising of the KEY condition will result in a transfer of control to an error-handling routine. Note that, because internal names may be up to 31 characters in length, labels and variables may have explanatory names, like the labels END OF_JOB and NO_RECORD_FOUND.

18 The part number specified in each record of the file DETAIL is used to select the associated record from the file MASTER.

19 Even though building of the transaction-report record (TRANSACT) starts later -- with card ST00150 -- OLD_SIZE and OLD_VALUE are assigned values here, since the appropriate variables are updated before TRANSACT is built.

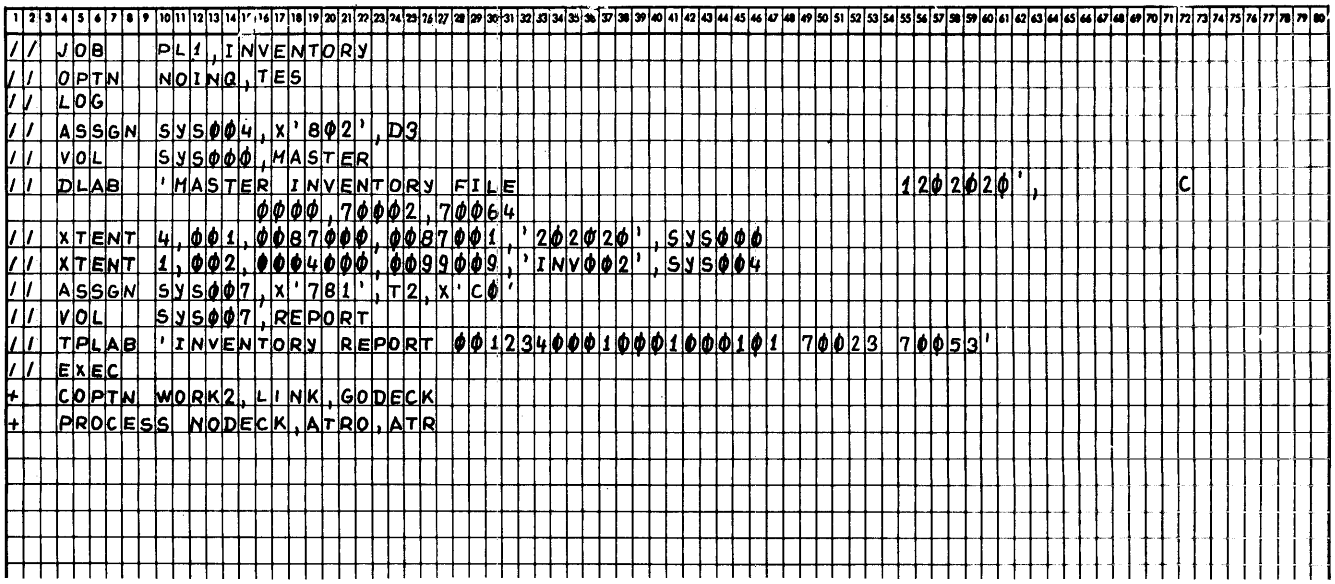
20 The auxiliary variable WANTED has been introduced to minimize the number of conversions, saving space and execution time.

21 In the second format item (A(15)), a length of 15 has been specified to add three blank characters to the right of the output field. This specification is equivalent to the specification (A,X(3)).

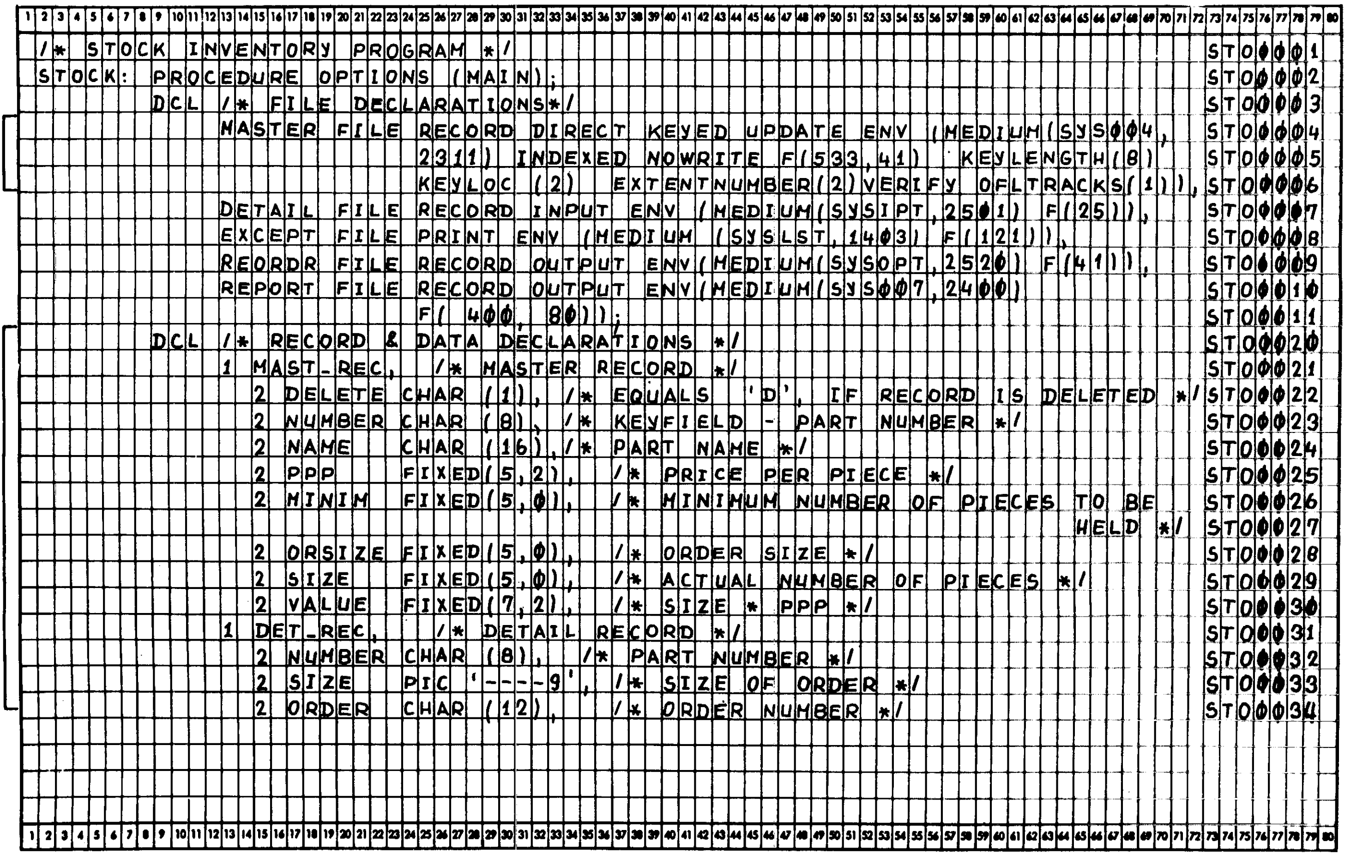
22 The size of the purchase order (REORDER REC.SIZE) has been determined in either of two ways before: If the department order exceeds the number of pieces in store, the order size is computed with the formula shown card ST00121, otherwise the standard order size is taken (see card ST00126).

23 This statement is preceded by two label prefixes.

24 This is not an error message but merely a comment that the list is complete.

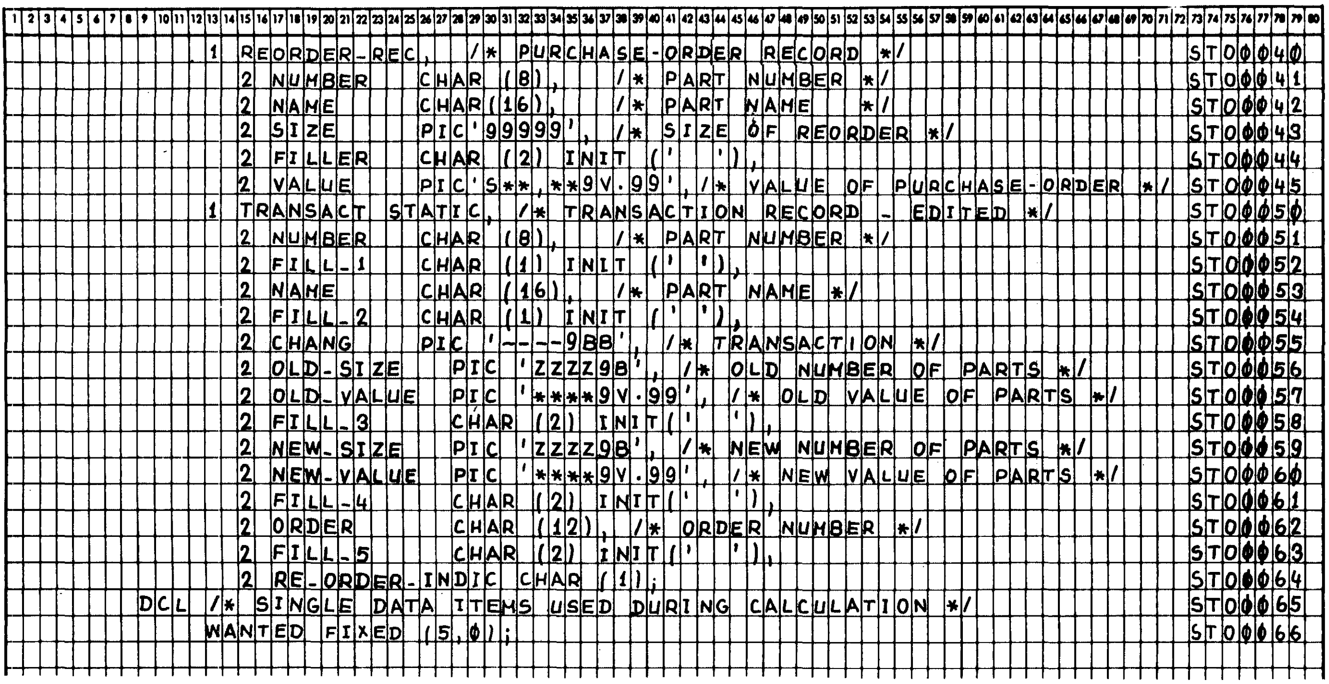


```
①   // JOB   PL1,INVENTORY
②   // OPTN  NOINQ,TES
③   // LOG
④   // ASSGN SYSØØ4,X'8Ø2',D3
⑤   // VOL   SYSØØØ,MASTER
⑥   // DLAB  'MASTER INVENTORY FILE                       12Ø2Ø2Ø',        C
                   ØØØØ,7ØØØ2,7ØØ64
⑦   // XTENT 4,ØØ1,ØØ87ØØØ,ØØ87ØØ1,'2Ø2Ø2Ø',SYSØØØ
    // XTENT 1,ØØ2,ØØØ4ØØØ,ØØ99ØØ9,'INVØØ2',SYSØØ4
⑧   // ASSGN SYSØØ7,X'781',T2,X'CØ'
    // VOL   SYSØØ7,REPORT
⑨   // TPLAB 'INVENTORY REPORT ØØ1234ØØØ1ØØØ1ØØØ1Ø1 7ØØ23 7ØØ53'
    // EXEC
⑩   +  COPTN WORK2,LINK,GODECK
⑪   +  PROCESS NODECK,ATRO,ATR
```

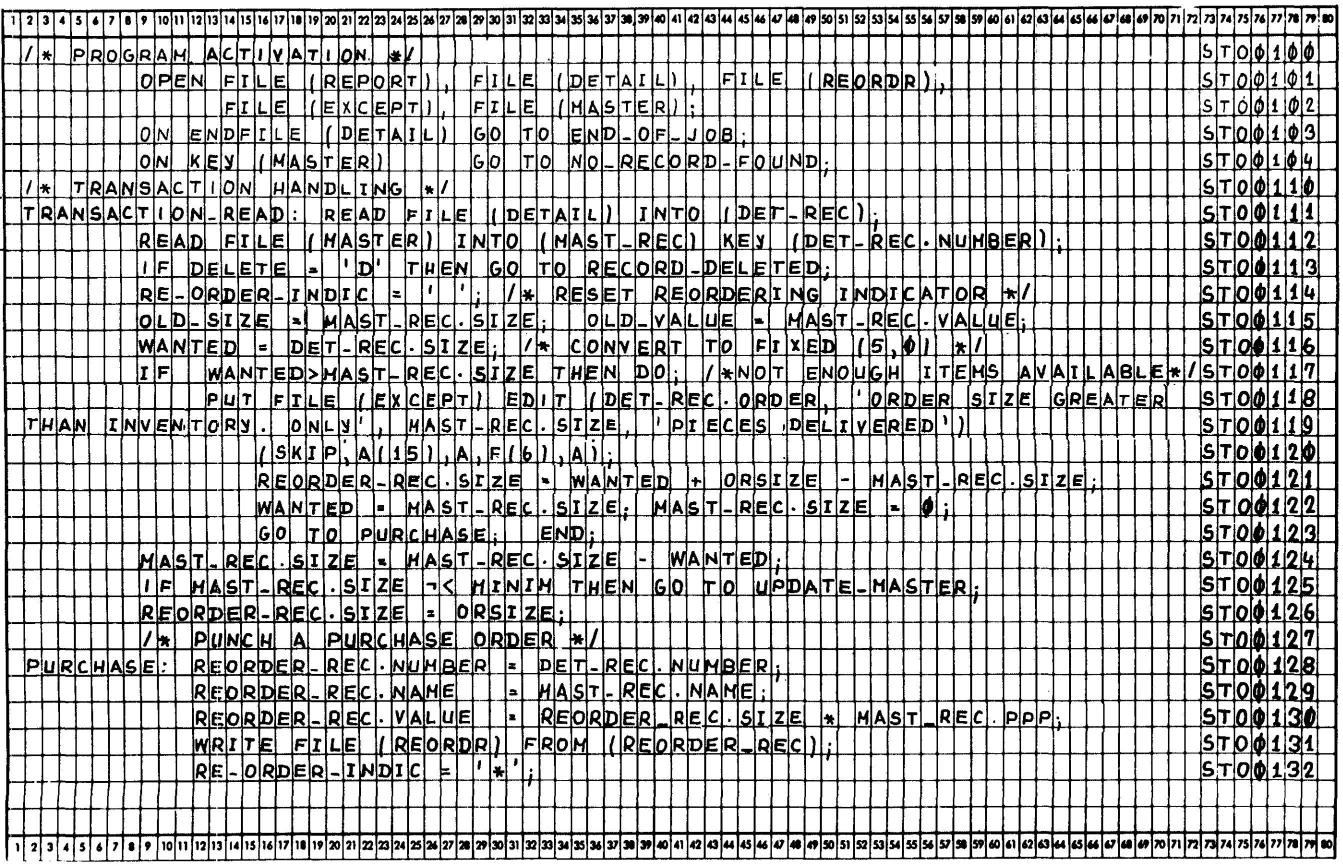


```
     /* STOCK INVENTORY PROGRAM */                                          STOØØØ1
⑬    STOCK: PROCEDURE OPTIONS (MAIN);                                       STOØØØ2
            DCL /* FILE DECLARATIONS*/                                      STOØØØ3
                MASTER FILE RECORD DIRECT KEYED UPDATE ENV (MEDIUM(SYSØØ4,   STOØØØ4
⑭                           2311) INDEXED NOWRITE F(533,41)  KEYLENGTH(8)   STOØØØ5
                            KEYLOC (2)  EXTENTNUMBER(2)VERIFY OFLTRACKS(1))), STOØØØ6
⑮               DETAIL FILE RECORD INPUT ENV (MEDIUM(SYSIPT,25Ø1) F(25)),   STOØØØ7
                EXCEPT FILE PRINT ENV (MEDIUM (SYSLST,14Ø3) F(121))),       STOØØØ8
                REORDR FILE RECORD OUTPUT ENV(MEDIUM(SYSOPT,252Ø) F(41)),   STOØØØ9
                REPORT FILE RECORD OUTPUT ENV(MEDIUM(SYSØØ7,24ØØ)           STOØØ1Ø
                            F( 4ØØ, 8Ø));                                   STOØØ11
⑯           DCL /* RECORD & DATA DECLARATIONS */                            STOØØ2Ø
                1 MAST_REC,    /* MASTER RECORD */                          STOØØ21
                  2 DELETE CHAR (1), /* EQUALS  'D', IF RECORD IS DELETED */STOØØ22
                  2 NUMBER CHAR (8),  /* KEYFIELD - PART NUMBER */          STOØØ23
                  2 NAME   CHAR (16),/* PART NAME */                        STOØØ24
                  2 PPP    FIXED(5,2),   /* PRICE PER PIECE */              STOØØ25
                  2 MINIM  FIXED(5,Ø),   /* MINIMUM NUMBER OF PIECES TO BE  STOØØ26
                                                          HELD */           STOØØ27
                  2 ORSIZE FIXED(5,Ø),   /* ORDER SIZE */                   STOØØ28
                  2 SIZE   FIXED(5,Ø),   /* ACTUAL NUMBER OF PIECES */      STOØØ29
                  2 VALUE  FIXED(7,2),   /* SIZE * PPP */                   STOØØ3Ø
                1 DET_REC,     /* DETAIL RECORD */                          STOØØ31
                  2 NUMBER CHAR (8),    /* PART NUMBER */                   STOØØ32
                  2 SIZE   PIC '----9',  /* SIZE OF ORDER */                STOØØ33
                  2 ORDER  CHAR (12),    /* ORDER NUMBER */                 STOØØ34
```

Figure 78. Programming Example for Commercial Application, Part 1 of 3

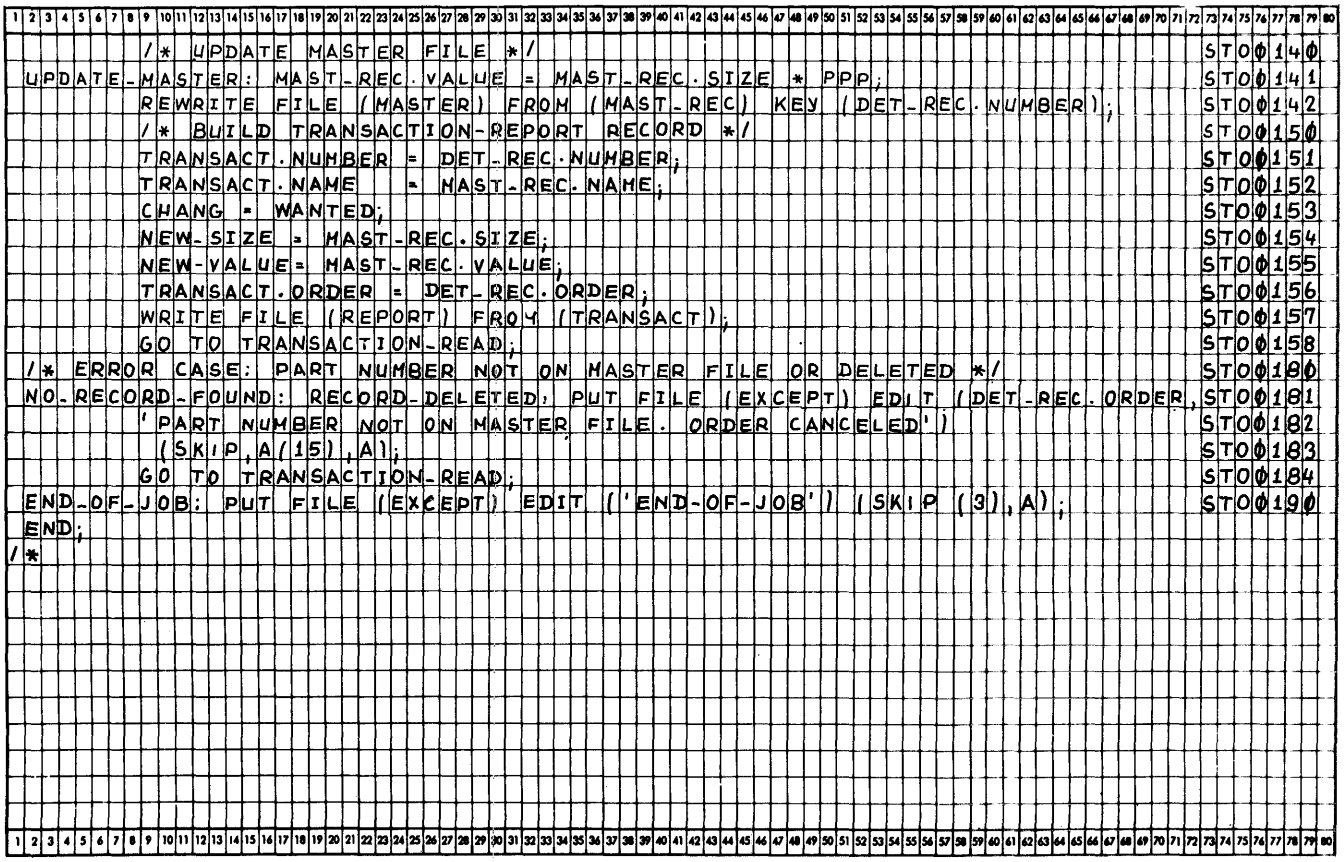


```
⑯             1 REORDER-REC,     /* PURCHASE-ORDER RECORD */              ST0ØØ4Ø
                2 NUMBER   CHAR (8),   /* PART NUMBER */                  ST0ØØ41
                2 NAME     CHAR(16),   /* PART NAME   */                  ST0ØØ42
                2 SIZE     PIC'99999', /* SIZE OF REORDER */              ST0ØØ43
                2 FILLER   CHAR (2) INIT ('  '),                          ST0ØØ44
                2 VALUE    PIC'S**,**9V.99', /* VALUE OF PURCHASE-ORDER */ ST0ØØ45
              1 TRANSACT STATIC, /* TRANSACTION RECORD - EDITED */        ST0ØØ5Ø
                2 NUMBER   CHAR (8),   /* PART NUMBER */                  ST0ØØ51
                2 FILL-1   CHAR (1) INIT (' '),                           ST0ØØ52
                2 NAME     CHAR (16),  /* PART NAME */                    ST0ØØ53
                2 FILL-2   CHAR (1) INIT (' '),                           ST0ØØ54
                2 CHANG    PIC '---9BB', /* TRANSACTION */                ST0ØØ55
                2 OLD-SIZE    PIC 'ZZZZ9B', /* OLD NUMBER OF PARTS */     ST0ØØ56
                2 OLD-VALUE   PIC '***9V.99', /* OLD VALUE OF PARTS */    ST0ØØ57
                2 FILL-3      CHAR (2) INIT('  '),                        ST0ØØ58
                2 NEW-SIZE    PIC 'ZZZZ9B', /* NEW NUMBER OF PARTS */     ST0ØØ59
                2 NEW-VALUE   PIC '***9V.99', /* NEW VALUE OF PARTS */    ST0ØØ6Ø
                2 FILL-4      CHAR (2) INIT('  '),                        ST0ØØ61
                2 ORDER       CHAR (12), /* ORDER NUMBER */               ST0ØØ62
                2 FILL-5      CHAR (2) INIT('  '),                        ST0ØØ63
                2 RE-ORDER-INDIC CHAR (1);                                ST0ØØ64
          DCL /* SINGLE DATA ITEMS USED DURING CALCULATION */             ST0ØØ65
              WANTED FIXED (5,Ø);                                         ST0ØØ66
```



```
    /* PROGRAM ACTIVATION */                                              ST0Ø1ØØ
           OPEN FILE (REPORT), FILE (DETAIL), FILE (REORDR),              ST0Ø1Ø1
                 FILE (EXCEPT), FILE (MASTER);                            ST0Ø1Ø2
           ON ENDFILE (DETAIL) GO TO END-OF-JOB;                          ST0Ø1Ø3
⑰         ON KEY (MASTER)      GO TO NO-RECORD-FOUND;                     ST0Ø1Ø4
    /* TRANSACTION HANDLING */                                            ST0Ø11Ø
    TRANSACTION-READ: READ FILE (DETAIL) INTO (DET-REC);                  ST0Ø111
⑱         READ FILE (MASTER) INTO (MAST-REC) KEY (DET-REC.NUMBER);        ST0Ø112
           IF DELETE = 'D' THEN GO TO RECORD-DELETED;                     ST0Ø113
           RE-ORDER-INDIC = ' '; /* RESET REORDERING INDICATOR */         ST0Ø114
⑲         OLD-SIZE = MAST-REC.SIZE;  OLD-VALUE = MAST-REC.VALUE;          ST0Ø115
⑳         WANTED = DET-REC.SIZE; /* CONVERT TO FIXED (5,Ø) */             ST0Ø116
           IF WANTED>MAST-REC.SIZE THEN DO; /*NOT ENOUGH ITEMS AVAILABLE*/ST0Ø117
                 PUT FILE (EXCEPT) EDIT (DET-REC.ORDER, 'ORDER SIZE GREATER ST0Ø118
    THAN INVENTORY. ONLY', MAST-REC.SIZE, 'PIECES DELIVERED')             ST0Ø119
㉑                (SKIP,A(15),A,F(6),A);                                  ST0Ø12Ø
                  REORDER-REC.SIZE = WANTED + ORSIZE - MAST-REC.SIZE;     ST0Ø121
                  WANTED = MAST-REC.SIZE; MAST-REC.SIZE = Ø;              ST0Ø122
                  GO TO PURCHASE;  END;                                   ST0Ø123
           MAST-REC.SIZE = MAST-REC.SIZE - WANTED;                        ST0Ø124
           IF MAST-REC.SIZE ¬< MINIM THEN GO TO UPDATE-MASTER;            ST0Ø125
           REORDER-REC.SIZE = ORSIZE;                                     ST0Ø126
           /* PUNCH A PURCHASE ORDER */                                   ST0Ø127
㉒  PURCHASE: REORDER-REC.NUMBER = DET-REC.NUMBER;                         ST0Ø128
              REORDER-REC.NAME    = MAST-REC.NAME;                        ST0Ø129
              REORDER-REC.VALUE   = REORDER_REC.SIZE * MAST_REC.PPP;      ST0Ø13Ø
              WRITE FILE (REORDR) FROM (REORDER_REC);                     ST0Ø131
              RE-ORDER-INDIC = '*';                                       ST0Ø132
```

Figure 78. Programming Example for Commercial Application, Part 2 of 3



```
        /* UPDATE MASTER FILE */                                        STOØ14Ø
 UPDATE_MASTER: MAST_REC.VALUE = MAST_REC.SIZE * PPP;                   STOØ141
        REWRITE FILE (MASTER) FROM (MAST_REC) KEY (DET_REC.NUMBER);     STOØ142
        /* BUILD TRANSACTION-REPORT RECORD */                           STOØ15Ø
        TRANSACT.NUMBER = DET_REC.NUMBER;                               STOØ151
        TRANSACT.NAME   = MAST_REC.NAME;                                STOØ152
        CHANG = WANTED;                                                 STOØ153
        NEW_SIZE = MAST-REC.SIZE;                                       STOØ154
        NEW-VALUE= MAST_REC.VALUE;                                      STOØ155
        TRANSACT.ORDER = DET_REC.ORDER;                                 STOØ156
        WRITE FILE (REPORT) FROY (TRANSACT);                            STOØ157
        GO TO TRANSACTION_READ;                                         STOØ158
 /* ERROR CASE: PART NUMBER NOT ON MASTER FILE OR DELETED */            STOØ18Ø
 NO-RECORD-FOUND: RECORD-DELETED, PUT FILE (EXCEPT) EDIT (DET-REC.ORDER,STOØ181
        'PART NUMBER NOT ON MASTER FILE. ORDER CANCELED')               STOØ182
         (SKIP,A(15),A);                                                STOØ183
        GO TO TRANSACTION_READ;                                         STOØ184
 END-OF-JOB: PUT FILE (EXCEPT) EDIT ('END-OF-JOB') (SKIP (3),A);        STOØ19Ø
 END;
/*
```

Figure 78. Programming Example for Commercial Application, Part 3 of 3

# Appendix A. Definition of Terms

Absolute_address: (1) an address that is permanently assigned by the machine to a storage location. (2) a pattern of characters that identifies a unique storage location without further modification.

access_method: any of the data management techniques available to the user for transferring data between main storage and an input/output device.

action_specification: in an ON statement, the ON-unit or single keyword SYSTEM, either of which specifies the action to be taken whenever an interrupt results from the raising of the named condition.

activation: (1) initiation of execution of a procedure. A procedure is activated when it is invoked at its entry point; (2) opening of a file to access it.

active: the state in which a procedure or file is said to be after activation and before termination;

actual_device_address: address specified in the ASSGN job-control statement. Includes attachment point, unit, and I/O device type.

additive_attributes: file attributes for which there are no defaults and which, if required, must always be stated explicitly;

address: (1) a specific storage location at which a data item can be stored, (2) I/O device address.

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: file attributes that may be chosen from groups of two or more alternatives. If none is specified, a default is assumed.

alternative_drive: when two drives are given for one multi-volume file, the first volume is mounted onto the primary drive and the second volume onto the alternative drive. If no alternative drive is given, all reels or packs must be read or written on the primary drive.

ambiguous_reference: name with insufficient qualification to make the name unique.

argument: an expression, a constant, or variable passed to an invoked procedure as part of the procedure or function reference.

arithmetic_data: data that has the characteristics of base, scale, 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.

Assembler: a program which prepares an object program by producing absolute or relocatable machine code from a source program of statements containing symbolic operation codes and symbolic operands.

attribute: a descriptive property associated with a name or expression to describe a characteristic of a data item or a file that the name may represent.

automatic_storage: storage that is allocated at the activation of a procedure and released at the termination of that procedure.

base: the number system in terms of which a written value is represented. Examples are the decimal base (ten), the binary base (two), and the hexadecimal base (sixteen). For Model 20 PL/I, the base is decimal.

based_variable: a variable declared to have the BASED (pointer-variable) attribute specification. The pointer variable associates the description with an allocation of storage.

based_storage: storage that is allocated for based variables. This allocation is dependent upon the manipulation of a pointer variable.

*batch compilation*: the processing of more than one procedure by the compiler in one job.

*batched-job processing*: a technique that permits multiple job definitions to be grouped (stacked) for presentation to the computing system, which automatically recognizes the jobs, and executes them one after the other.

*bit*: The smallest unit of information in System/360. It can have either of the two values: zero or one.

*block*: distinct physical grouping of data items within a file.

*blocked record*: logical record grouped together with other logical records to form a physical record (or block).

*blocking factor*: block size divided by record size. Applies to fixed-length records only.

*bound*: the upper or lower limit of an array dimension. The lower limit is always assumed to be 1.

*buffer*: an intermediate main-storage 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.

*byte*: The basic unit of information in IBM System/360. Every byte consists of 8 bits, each having a value of zero or one. (See Bit).

*call*: the invocation of a procedure by means of the CALL statement.

*cataloging*: insertion of an executable object program or program segments, or of label information as a temporary or permanent entry into the core-image library or the label information area, respectively.

*character*: An 8-bit (1-byte) code that can be manipulated in the main storage of the central processing unit.

*character set*: a collection of 48 or 60 characters used to write PL/I source programs.

*character string*: a string composed of one or more characters from the data character set.

*closing (a file)*: de-activation of a file by means of a CLOSE statement for the associated file or by an END or RETURN statement of the main procedure of a program.

*coded arithmetic data*: internal representation of arithmetic data whose characteristics are given by the base, scale, and precision attributes. The types for Model 20 PL/I are packed decimal and short or long floating point form.

*collating sequence*: the relative order of alphameric characters upon which sorting or merging is based.

*comment*: a string of characters, used for documentation, which is preceded by /* and terminated by */.

*comparison operators*: the operators ¬<, <, <=, ¬=, =, >=, >, ¬>.

*compilation*: translation of a PL/I source module into an object module (i.e., machine language).

*compiled object module*: the set of machine instructions produced by compilation.

*compile time*: the time during which a source program is translated into an object module.

*compiler*: a translator that converts a PL/I source program into a compiled and/or executable object program.

*compiler control statement*: any one of the control statements in the input stream that defines the requirements and options of a job to the compiler.

*composite operator*: operator composed of two operator symbols (e.g.,¬=).

*compound statement*: a statement that contains other statements. IF and ON are the only compound statements.

*concatenation*: the operation that connects two character 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 procedure statement by a colon and preceding the entry name. 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.

consecutive file organization: organization of records on the basis of their successive physical positions. Records of a consecutive file can be accessed only sequentially.

constant: (1) an arithmetic or character string data item that does not have a name, (2) a statement label.

contextual declaration: the appearance of an identifier to the left of an assignment symbol or in the data list of a GET statement, or as a built-in function name (except DATE).

control format item: description of page and spacing operations.

control programs: a set of system programs that control the execution of the compiler and of user-programs. They are the Monitor, the Job Control program, and the Initial Program Loader.

control statement: any of the statements in the input stream that define the requirements of the job, its options, or control its actions.

control variable: variable used to control the iterative execution of a DO-group.

conversion: the transformation of a value from one representation to another.

Core-Image Library: a disk area containing the Job Control program, other IBM-supplied programs (except the Monitor and the IPL), and user's problem programs. Permits retrieval of programs and/or segments by the Monitor.

Core-Image Library Maintenance Program (CMAINT): a DPS Service program. Updates the core-image library and directory. Is used to add and/or delete program segments.

cylinder: a group of ten vertically aligned tracks on a 1316 disk pack.

cylinder index: table containing the identification and the highest key associated with each cylinder occupied by an INDEXED file.

cylinder overflow area: overflow area designated for each cylinder of the prime data area of an INDEXED file.

data: representation of information in the form of digits and characters that have certain characteristics called attributes.

data character set: all of those characters whose bit configuration is recognized by the computer in use.

data file: a collection of related records organized in a specific manner. For example, a payroll file (one record for each employee, showing his rate of pay, deductions, etc.) or an inventory file (one record for each inventory item, showing the cost, selling price, number in stock, etc.).

data format item: description of data in the stream that specifies whether the data items are characters or arithmetic values in character form.

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, and from which data items are to be written, during output. (On input, the list may contain only variables).

data specification: the portion of an edit-directed data transmission statement that specifies the mode of transmission (EDIT) and includes the data list and the format list.

data transmission: the transfer of data from an external storage medium to main storage and vice versa.

deblocking (of records): segregating physical records into their logical parts. Deblocking is done automatically by the system so that a program deals only with logical records.

decimal: the number system based on the value 10.

decimal fixed-point data item: data item consisting of one or more decimal digits and an optional decimal point. If no decimal point appears, the point is assumed to be immediately to the right of the rightmost digit.

decimal floating-point data item: decimal number followed by an integer exponent. The exponent specifies the assumed position of the decimal point, relative to the position in which it actually appears. The data item is written as one or more digits, the mantissa, followed by the letter E, followed by the exponent.

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.

defined item: item having the DEFINED attribute.

delimiter: any valid special character or combination of special characters used to separate identifiers and constants, or statements from one another.

device address: see symbolic device address and actual device address.

device independence: the ability to request input/output operations without regard to the physical characteristics of the input/output devices.

digit: the ten decimal digits 0 through 9.

dimensioning: the number of bound specifications associated with an array. It cannot be greater than three.

direct access: random processing of the logical records of a file by use of a key.

disabled (condition): the state in which the occurrence of a particular condition will not result in a program interrupt.

disk-resident system: contains the Monitor, the disk-resident portion of the IPL, and the Job Control program. May contain any IBM-supplied and/or user-written programs and/or macro definitions as well as the relocatable area.

DO-group: a sequence of statements headed by a DO statement and closed by its corresponding END statement.

DPS Control Programs: a collective term used to refer to the Initial Program Loader, the Monitor program, and the Job Control program.

DPS Linkage Editor: a system service program. Relocates programs or phases and links separately assembled programs or phases. Cannot be used for Model 20 PL/I programs.

drifting picture character: multipally specified picture character ($, S, or -) that causes leading zeros to be suppressed and a $ symbol, a sign, or a blank to appear in the rightmost position of the suppressed field of the character-string value of an arithmetic data item.

dummy argument: a compiler-assigned variable for an argument that has no programmer-assigned name.

dump: (1) print-out of total main storage or of parts of main storage, (2) print-out of disk areas.

dynamic storage: storage that is allocated when execution of a procedure begins and that is freed when execution of a procedure is terminated. See automatic storage.

EBCDIC: (Extended Binary Coded Decimal Interchange Code) a specific set of eight-bit codes standard throughout System/360.

edit-directed transmission: STREAM transmission; both a data list and a format list are specified.

editing character: a picture character in a numeric-picture specification that causes the specified picture character to appear in the character-string value of the numeric-picture specification.

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.

enabled (condition): that state in which the occurrence of a particular condition will result in a program interrupt.

entry name: a label of a PROCEDURE statement.

entry point: that point in a procedure at which it may be invoked by reference to the entry name. (For Model 20 PL/I only the PROCEDURE statement).

epilogue: those processes which occur at the termination of a procedure.

established action: any action specified to take place when an enabled condition arises and causes an interrupt.

exceptional condition: an occurrence, which can cause a program interrupt, or 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.

executable object program: the set of machine instructions produced by compilation and link-editing.

execute-loader_function: reading of executable object program and its execution.

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.

explicit_file_opening: opening of a file by means of an OPEN statement. RECORD files always have to be explicitly opened.

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.

extent: area of a disk file specified by an upper limit and a lower limit; and reserved for or occupied by a particular file.

external_declaration: an explicit or implicit declaration of the EXTERNAL attribute for an identifier. Such an identifier is known in all other procedures for_which_such_a_declaration_exists.

external_name: an identifier which has the EXTERNAL attribute.

external_storage_medium: the medium on which data may be stored; for example, cards, magnetic tape, or disk.

external_symbol: a control section name, entry point name, or external reference; a symbol contained in the external symbol dictionary.

factoring_(of_attributes): enclosing of names having the same attributes in paren theses. Following the parenthesized list is the set of attributes that apply, in order to eliminate repeated specification of the same attributes for more than one name.

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.

file: the collection of related records organized in a specific manner on an external storage medium.

file_declaration: the association of attributes with a filename in a program.

file_label: label containing information applicable to a given data file or portion of a data file stored on a particular volume.

filename: the symbolic reference, within a program, to a file.

file_reorganization: a term used to describe the process of writing a new file from an indexed file, purging records that are tagged for deletion, and merging records in the overflow area in their sequential positions in the prime data area.

fixed-length_record: a record having the same length as all other records with which it is logically or physically associated.

fixed-point_data_item: see decimal_fixed-point_data_item.

floating-point_data_item: see decimal floating-point_data_item.

format_item: a specification used in edit-directed transmission to describe the representation of a data item in the stream or to control the spacing and the format of a printed page.

format_list: a list of format items required for an edit-directed data specification.

fractional_digit: digit to the right of the decimal point.

function: a procedure that is invoked by the appearance of its entry name in a function reference.

function_reference: the appearance of an entry name in an expression, usually in conjunction with an argument list.

group: a DO group;

halfword: two adjacent bytes where the left byte is on a halfword boundary.

halfword_boundary: even-numbered position in main storage.

header_label: file label preceding a labeled tape file and defining it.

hexadecimal: a character representation for a set of four bits. The values 0 - 15 are represented by the digits 0 - 9 and the alphabetic characters A - F.

high-order_digit: leftmost digit of a decimal number.

*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.

*implicit declaration*: association of attributes with an identifier that is not explicitly or contextually declared.

*implicit file opening*: opening a file by means of a GET or PUT statement when no previous OPEN statement for that file has been executed. Only STREAM files can be implicitly opened.

*inactive procedure*: a procedure that has not been activated or that has been terminated.

*independent overflow area*: overflow area designated to supplement or replace cylinder-overflow areas.

*index*: see *cylinder index* and *track index*.

*indexed-file organization*: organization of records in a disk file on the basis of keys that are associated with each logical record. Indexed files may be accessed sequentially as well as directly.

*infix operator*: an operator that defines an operation between two operands.

*initial procedure*: a 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.

*Initial Program Loader (IPL)*. A DPS Control program. Loads Monitor into main storage. Is used to assign physical I/O device addresses to symbolic addresses SYSRES and SYSRDR. Required for the initialization of the disk-resident system.

*initial value*: value assigned to a variable at the time storage is allocated to it.

*inquiry programs*: inquiry programs are initiated by pressing the Request key on the printer-keyboard and typing in the name of the program. The mainline program is rolled out on the system disk pack; then the inquiry program is loaded and processed; after execution is completed, the mainline program is rolled back into main storage and resumes processing. Inquiry programs can be executed only under control of a Monitor that supports inquiry facilities. The execution of inquiry programs is not preceded by a Job Control run.

*input/output*: the transfer of data between an external storage medium and (main) storage.

*I/O time*: (1) the time interval between the instant at which data is called for from an external storage device and the instant delivery is completed i.e., the read time. (2) the time interval between the instant at which data is requested to be stored and the instant at which storage is completed, i.e., the write time.

*insertion character (in numeric picture specifications)*: a picture character that causes a character to appear in the character-string value of a numeric character data item. There are three picture character in Model 20 PL/I: (.) causing a decimal point to be inserted; (,) causing a comma to be inserted and (B) causing a blank to be inserted.

*integer digit*: digit to the left of the decimal point.

*internal name*: an identifier that has the INTERNAL attribute.

*inter-record gap*: a blank space on magnetic tape that separates physical records.

*interrupt*: the suspension of normal program activities as the result of the occurrence of an enabled condition.

*invoke*: to activate a procedure at its entry point.

*invoked procedure*: a procedure that has been activated at its entry point.

*invoking procedure*: a procedure containing a statement that activates another procedure.

*iteration factor*: a constant 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.

*job*: a unit of work for the programming system that is externally identified by one set of job-control statements.

*Job Control program*: a DPS Control program. Resides in main storage between jobs and provides for automatic job-to-job transition. Performs I/O device assignment. Causes Monitor to load next program.

job-control_statement: any one of the control statements read from the device assigned to SYSRDR that identifies a job or defines its requirements and options.

key: see source_key and recorded_key.

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 procedure in which it has been declared.

Label: (1) a physical identification record on magnetic tape located either preceding or following a data file, or both. If a data file extends beyond a single reel of tape, a label can be placed preceding and following the data on each reel; (2) a physical identification record on disk which identifies the volume or file, (3) statement label.

label_constant: see statement_label.

Label_Information_Area_(LIA): an area on the system disk pack into which each disk file label information, as contained in the VOL, DLAB, and XTENT statements, is placed by the Job Control program. This information is used by the label processing routines.

label_prefix: an unparenthesized identifier prefixed to a statement by a colon.

language_translator: a general term for any assembler, compiler, or other routine that accepts statements in one language and produces equivalent statements in another language.

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.

library-management_programs: collective term for four system service programs: Core-Image Maintenance, Macro Maintenance, Directory Service, and Library Allocation Organization programs.

link-editing: combining of compiled object module(s) with other compiled object module(s) and/or PL/I library routines into an executable object program.

load: to read an executable object program into main storage preparatory to executing it.

logical_record: see record.

loop: sequence of statements executed successively more than one time.

Low-order_digit: rightmost digit of a decimal number.

main_procedure: see initial_procedure.

main_storage: the internal storage area of the central processing unit (CPU) that controls all internal manipulation of data.

mainline_program: program whose execution is interrupted by an inquiry program and whose execution is continued when the inquiry program has been processed.

major_structure: a structure whose name is declared with level number 1.

major-structure_name: name used to refer to the entire structure must be declared with level number 1.

minor_structure: a structure whose name is declared with a level number greater than 1.

module: the input to, or output from, a single execution of an assembler or compiler; a source, object, or load module; hence, a program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading.

Monitor_Program. The main DPS control program. Resident in main storage throughout a system run. Loads programs into main storage and causes their execution.

multi-extent_disk_file: file stored on a disk pack defined by more than one extent.

multi-file_volume: volume that contains more than one file.

multi-reel_tape_file: a file stored on more than one tape reel.

multi-volume_disk_file: a disk file stored on more than one disk pack.

multi-volume_tape_file: see multi-reel tape_file.

multiple_declaration: two or more declarations of the same identifier in the same procedure without different qualifications, or two or more EXTERNAL declarations of the same identifier as different names within a single program. Multiple declaration is in error.

*name*: an identifier that has been declared.

*nesting*: 1. the occurrence of a DO-group within another DO-group. 2. the occurrence of an IF statement in a THEN clause or an ELSE clause. 3. the occurrence of a function reference as an argument of another function reference.

*null statement*: represented by a semicolon; indicates that no action is to be taken.

*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.

*object module*: the output of an assembler or a compiler. An object module consists of (1) one or more control sections in relocatable, though not executable, form or (2) one executable object program. See *compiled object module* and *executable object module*.

*on-line*: pertaining to equipment or devices under direct control of the central processing unit.

*ON-unit*: the action to be executed upon the occurrence of the ON-condition named in the containing ON statement.

*opening (a file)*: activating a file by means of an OPEN or GET or PUT statement associated with that file.

*operand*: the representation of (1) information that must be supplied to define a selective function to the program, (2) selection of a value on which an operation is to be performed.

*operation*: a selective function to be performed by the program.

*operational expression*: expression containing operators.

*operator*: a symbol specifying an operation to be performed. See *arithmetic 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.

*organization of a file*: see *consecutive file organization* and *indexed file organization*.

*overflow area*: tracks designated to accommodate records that are forced off the prime-data tracks by the insertion of new records.

*overlay*: to place a segment of an executable object program into main storage locations occupied by another program or segment that has already been processed.

*packed-decimal*: storage technique whereby two digits or one digit and a sign are stored per byte.

*parameter*: a name in an invoked procedure that is used to represent an argument passed to that procedure.

*permanent disk label*: disk label that has been cataloged as a permanent entry into the label information area.

*phase-encoded tape*: tape of the magnetic tape drive models 2415-4,-5,-6.

*physical record*: the block (or the unit) of data that is physically transmitted to and from a volume.

*picture*: a character-by-character specification describing the composition and attributes of numeric character data. It allows editing.

*point alignment*: alignment of arithmetic data in a variable depending upon the location of the decimal point as specified by the precision and scale attributes or the picture character V in a numeric-picture specification.

*point of invocation*: the point in the invoking procedure at which the procedure reference to the invoked procedure appears.

*pointer variable*: a variable that identifies the storage to be used when referring to a based variable.

*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 point.

*precompiled object module*: module that has been compiled or assembled by the PL/I Compiler or Assembler, respectively, but not link-edited, and that is used again as compiler input to be link-edited.

*prefix*: (1) a label connected by a colon to the beginning of a statement; (2) a parenthesized list of condition names connected by a colon to the beginning of a procedure statement.

*prefix operator*: an operator that precedes, and is associated with, a single operand. The prefix operators are + and -.

*prime data area*: disk areas where the records of an INDEXED file are initially stored.

*problem data*: character 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 it. A procedure may contain any statement except another PROCEDURE statement.

*procedure reference*: the appearance of a procedure name in a CALL statement.

*program*: a set of one or more 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. Label data and pointer data are the types of program control data.

*prologue*: those processes that occur at the activation of a procedure.

*pseudo-variable*: the built-in function SUBSTR that can be used as a receiving field.

*qualified name*: a sequence of names of structure members connected by periods, to uniquely identify a component of a structure.

*random access*: see *direct access*.

*receiving field*: any field to which a value may be assigned.

*record*: a general term for any unit of data that is distinct from all others when considered in a particular context.

*record format*: see *fixed-length record*, *variable-length record*, and *undefined-length record*.

*record-oriented I/O*: the transmission of collections of data, called records, one record at a time. The external representation of the data is an exact copy of the internal, and vice versa.

*recorded key*: a character string recorded in a logical record of an INDEXED file to identify that record.

*relocatable area*: an area on the system disk pack to temporarily hold an object module, thus permitting the assembly or compilation and the execution of a program or program segment in one job.

*relocation*: the modification of address constants required for a change of origin of a module or control section.

*remote format item*: specification of a label of a separate statement that contains the format list to be used.

*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.

*Report Generator and Report Program Generator (RPG)*: A program which constructs reports or report-writing programs in accordance with input specifications of the data file and of the desired report.

*reserved word*: a keyword that has a specific meaning and may only be used in a specific context.

*returned value*: the value returned by a function to the point of invocation.

*root segment*: segment containing the main procedure of a segmented program. Must remain in storage throughout execution of a program. Must be the 1st segment of the program.

*scale*: fixed-or floating-point representation of an arithmetic value.

*scale factor*: the number of digits to the right of the decimal point.

*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.

*sector*: one tenth of a track (270 bytes).

segment: the smallest addressable unit in the core-image library of a disk-resident system.

sequential access (of a file): consecutive transfer of either the whole file or part of the file.

service programs: a group of programs that create and maintain the system libraries and the relocatable area.

significant digit: a digit that contributes to the accuracy or precision of a numeric value. The number of significant digits is counted beginning with the digit contributing the most value, called the most significant digit, and ending with the one contributing the least value, called the least significant value.

simple statement: see statement

source key: a character string or a numeric character data item referred to in a RECORD transmission statement that identifies a particular record within an INDEXED file. The source key is a character string to be compared with, or written as, a recorded key to identify the record.

source module: a set of source statements forming a complete control section or procedure.

source program: the program that is translated and link-edited by the compiler.

special character: each non-alphameric character of the 60 or 48 character set.

stacked-job processing: see batched-job processing.

standard system action: action taken by the system when an interrupt occurs for which no other action has been specified.

statement: a basic element of a PL/I program that is used to delimit a portion of a 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 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.

storage allocation: Association of a storage area with a variable.

stream: data being transferred from or to an external medium represented as a continuous string of data items in character form.

stream-oriented input/output: transmission of data items as a continuous stream of characters that are, on input, automatically converted to conform to the attribute of the variables to which they are assigned and, on output, are automatically converted to character representation.

string: a connected sequence of characters that is treated as a single data item.

string operator: the string operator is ||, denoting concatenation of character strings.

structure: a hierarchical set of names that refers to an aggregate of data items that may 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.

subscript: expression enclosed in parentheses following an array variable. It specifies the relative position, within the array, of a particular data element.

substructure: structure declared one or more levels below the major-structure level.

symbolic device address: a symbol used in IBM-supplied and user-written programs to refer to an I/O device (e.g., SYSRES, SYSIPT, SYS005).

system-disk pack: the disk pack on which the user's disk-resident system is stored.

Tape Error Recovery routine (TER): a routine to control the execution of error recovery procedures in the case of magnetic tape I/O errors.

Tape Error Statistics routine (TES). a routine to analyze the interrupts and magnetic tape I/O errors occurring during the execution of a program.

tapemark: a special record that can be read from, or written onto, magnetic tape. Used to distinguish the end of a file or file segment, and to separate labels from the data.

*termination*: cessation of execution of a procedure and the return of control to the activating procedure by means of a RETURN or END statement, or the transfer of control to the activating procedure or some other active procedure by means of a GO TO statement. A return of control to the programming system via a RETURN or END statement in the initial procedure. See *epilogue*.

*track*: concentric circle on a disk surface used for the storage of data. One track may accommodate 2,700 bytes of data.

*track index*: table containing the identification and the highest key associated with each track occupied by a file.

*trailer label*: file label following a file or part of tape file furnishing the information required to determine whether the end of file has been reached or whether the file is continued on another volume.

*truncation*: loss of digits to the right or left of a data item.

*undefined-length records*: records of varying length, where by each block consists of only one record. The system does not insert any length-specifying records into the block.

*Utility program(s)*: a program or a set of programs which assist in the operation of a computer, i.e., storage clearing, intermediate data transmission, dump program, file organization routines, etc.

*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.

*variable-length records*: logical records whose number of bytes is not fixed, but may vary within prescribed limits. Variable-length logical records may be blocked into physical records. Deblocking is dependent upon length-specifying bytes at the beginning of each logical and each physical record. The length bytes are inserted automatically by the system.

*volume*: that portion of a single unit of storage media that is accessible to a single read/write mechanism. For example, a reel of magnetic tape for a 2415 magnetic tape drive, or one 1316 Disk Pack for an IBM 2311 Disk Storage Drive.

*volume label*: the volume label indentifies and protects the entire volume (disk pack or magnetic tape reel). It is fixed in length and format, and lies in a fixed location within the volume. The volume label contains the volume serial number. In addition, the disk volume label contains the address of the volume table of contents.

*Volume Table of Contents (VTOC)*: a number of records on a disk pack, composed of disk file labels, specifying the extents of, and identifying all files on the pack.

*zero-suppression character (in a picture specification)*: picture character (Z or *) used to suppress leading zeros.

# Appendix B. Upward Combatibility

(This appendix contains only preliminary information).

Source programs written in Model 20 PL/I are upward compatible to DOS/TOS PL/I and yield identical results on other System/360 models with the following exceptions:

1. Floating-point arithmetic is implemented in Model 20 PL/I in decimal arithmetic. This means that internal precision and round-off errors are slightly different.

2. The difference mentioned above causes a file incompatibility when using FLOAT variables in PL/I record-oriented I/O. This is due to the difference in the representation of data. A method for reading such files with a DOS/TOS PL/I program will be described in the third part of this publication at a later date.

3. The IBM 2560 Multi-Function Card Machine, the 2203 Printer, and the 2152 Printer-Keyboard are not supported on other System/360 models. Minor changes in the ENVIRONMENT attribute are necessary to change to other devices.

4. In the MEDIUM option, the symbolic-device address SYSOPT has been introduced for Model 20 DPS. Programs that are expected to run on both Model 20 and DOS/TOS PL/I should use SYSPCH instead of SYSOPT. SYSPCH and SYSOPT are synonymous in Model 20 PL/I.

# Appendix C. Character Sets with EBCDIC and Card-Punch Codes

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 |

* not used in Model 20 PL/I

| Character | Card-Punch | 8-Bit Code |
|---|---|---|
| 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<br>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 |

Note: When using the 60-character set the following rule should be observed: The composite symbols of the 48-character set (see 48-Character Set in this Appendix) must not be used as keywords or delimiters with the 60-character set. The alphabetic character combinations would be interpreted as identifiers; the special character combinations would cause errors. Exceptions are the character combinations for comments. They are identical in both character sets.

## 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 | ¬= |
| ,. | 0-8-3, 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 | */ |

*Note*: When using the 48-character set, the following rules should be observed:

1. 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.

2. The composite symbols of the 48-character set are reserved words in the 48-character set; i.e., these identifiers must not be used as variables, entry names, or filenames.

# Appendix D. Model 20 PL/I Keywords

| Keyword | Use of Keyword |
|---|---|
| ABS(x) | built-in function |
| ADDR(x) | built-in function |
| ATAN(x[,y]) | built-in function |
| AUTOMATIC | attribute |
| BACKWARDS | attribute |
| BASED (pointer-variable) | attribute |
| BUILTIN | attribute |
| BY | clause of DO statement |
| CALL | statement |
| CEIL(x) | built-in function |
| CHAR(value[,size]) | built-in function |
| CHAR(length) | attribute |
| CHARACTER(length) | attribute |
| CLOSE | statement |
| CONVERSION | condition |
| COS(x) | built-in function |
| DATE | built-in function |
| DCL | statement |
| DECIMAL | attribute |
| DECLARE | statement |
| DEF | attribute |
| DEFINED | attribute |
| DIRECT | attribute |
| DISPLAY | statement |
| DO | statement |
| EDIT | STREAM I/O transmission mode |
| ELSE | clause of IF statement |
| END | statement |
| ENDFILE | condition |
| ENDPAGE | condition |
| ENTRY | attribute |
| ENV | attribute |
| ENVIRONMENT | attribute |
| ERROR | condition |
| EXP(x) | built-in function |
| EXT | attribute |
| EXTERNAL | attribute |
| FILE | attribute |
| FILE(file-name) | option of GET and PUT, specification of RECORD I/O statement |
| FIXED | attribute |
| FIXEDOVERFLOW | condition |
| FLOAT | attribute |
| FLOOR(x) | built-in function |
| FORMAT(format-list) | statement |
| FROM | option of REWRITE or WRITE statement |
| GET | statement |
| GO TO,GOTO | statement |
| HIGH(i) | built-in function |
| IF | statement |
| INIT | attribute |
| INITIAL | attribute |
| INPUT | attribute |
| INTERNAL | attribute |
| INTO(variable) | option of READ statement |
| KEY(file-name) | condition |
| KEY(x) | option of READ and REWRITE statement |
| KEYED | attribute |
| KEYFROM(x) | option of WRITE and LOCATE statement |
| LABEL | attribute |
| LOCATE | statement |

| Keyword | Use of Keywords |
|---|---|
| LOG(x) | built-in function |
| LOW(i) | built-in function |
| MAIN | option of PROCEDURE statement |
| MAX(arguments) | built-in function |
| MIN(arguments) | built-in function |
| NOCONVERSION | condition prefix identifier, disables CONVERSION |
| NOFIXEDOVERFLOW | condition prefix identifier, disables FIXEDOVERFLOW |
| NOOVERFLOW | condition prefix identifier, disables OVERFLOW |
| NOUNDERFLOW | condition prefix identifier, disables UNDERFLOW |
| NOZERODIVIDE | condition prefix identifier, disables ZERODIVIDE |
| ON | statement |
| ONSYSLOG | option of PROCEDURE statement |
| OPEN | statement |
| OPTIONS(list) | option of PROCEDURE statement |
| OUTPUT | attribute |
| OVERFLOW | condition |
| PAGE | format item, option of PUT statement |
| PAGESIZE(w) | option of the OPEN statement |
| PIC | attribute |
| PICTURE | attribute |
| POINTER | attribute |
| PRINT | attribute |
| PROCEDURE | statement |
| PUT | statement |
| READ | statement |
| RECORD | attribute |
| RECORD(file-name) | condition |
| REPLY(c) | option of DISPLAY statement |
| RETURN | statement |
| RETURNS | attribute |
| REWRITE | statement |
| ROUND(x,n) | built-in function |
| SEQUENTIAL | attribute |
| SET | option of READ and LOCATE statements |
| SIN(x) | built-in function |
| SKIP[(x)] | format item, option of PUT statement |
| SQRT(x) | built-in function |
| STATIC | attribute |
| STREAM | attribute |
| STRING(string-name) | option of GET and PUT statements |
| SUBSTR(string,i,j) | built-in function, pseudo-variable |
| SYSTEM | action specification of the ON statement |
| TAN(x) | built-in function |
| TANH(x) | built-in function |
| THEN | clause of IF statement |
| TO | clause of DO statement |
| TRANSMIT | condition |
| TRUNC(x) | built-in function |
| UNDERFLOW | condition |
| UPDATE | attribute |
| WRITE | statement |
| ZERODIVIDE | condition |

# Appendix E. File Attributes and Options

| | TYPE OF FILE | STREAM | | | | | | | | | | RECORD | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | INPUT | | | OUTPUT | | | | OUTPUT PRINT | | | SEQUENTIAL | | | | | | | | | | | DIRECT | |
| | | | | | | | | | | | | CONSECUTIVE | | | | | | | | INDEXED | | | | |
| | | | | | | | | | | | | CARD PRINT | | TAPE | | | DISK | | | | | | | |
| | FILE ATTRIBUTES AND OPTIONS | CARD | TAPE | DISK | CARD | PRINTER | TAPE | DISK | PRINTER | TAPE | DISK | INPUT | OUTPUT | INP. FORW | INP. BACK | OUTPUT | INPUT | OUTPUT | UPDATE | INPUT | OUTPUT | UPDATE | INPUT | UPDATE |
| ATTRIBUTES | filename 1-6 characters | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S |
| | FILE | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S |
| | RECORD | | | | | | | | | | | S | S | S | S | S | S | S | S | S | S | S | S | S |
| | STREAM | D | D | D | D | D | D | D | D | D | D | | | | | | | | | | | | | |
| | SEQUENTIAL | | | | | | | | | | | D | D | D | D′ | D | D | D | D | D | D | D | | |
| | DIRECT | | | | | | | | | | | | | | | | | | | | | | S | S |
| | KEYED | | | | | | | | | | | | | | | | | | | S | S | S | S | S |
| | INPUT | S | S | S | | | | | | | | S | | S | S | | S | | | S | | | S | |
| | OUTPUT | | | | S | S | S | S | D | D | D | | S | | | S | | S | | | S | | | |
| | UPDATE | | | | | | | | | | | | | | | | | | S | | | S | | S |
| | PRINT | | | | | | | | S | S | S | | | | | | | | | | | | | |
| | BACKWARDS | | | | | | | | | | | | | | S | | | | | | | | | |
| | ENVIRONMENT ( | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S |
| OPTIONS | MEDIUM ( | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S | S |
| | SYSIPT | C | C | C | | | | | | | | C | | C | | | C | | | | | | | |
| | SYSOPT/SYSPCH | | | | C | | C | C | | | | | C | | | C | | C | | | | | | |
| | SYSLST | | | | | C | | | C | | | | C | | | | | | | | | | | |
| | SYSnnn (nnn = 000 - 019) | C | C | C | C | C | C | C | C | C | C | C | C | C | S | C | C | C | S | S | S | S | S | S |
| | 2501/2520/2560P/2560S/2152 | C | | | | | | | | | | C | | | | | | | | | | | | |
| | 2520/2560P/2560S/1442 | | | | C | | | | | | | | C | | | | | | | | | | | |
| | 1403/2203/2152 | | | | | C | | | C | | | | C | | | | | | | | | | | |
| | 2400 | | S | | | | S | | | S | | | | S | S | S | | | | | | | | |
| | 2311) | | | S | | | | S | | | S | | | | | | S | S | S | S | S | S | S | S |
| | CONSECUTIVE | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | | | | | |
| | INDEXED | | | | | | | | | | | | | | | | | | | S | S | S | S | S |
| | KEYLENGTH | | | | | | | | | | | | | | | | | | | S | S | S | S | S |
| | KEYLOC | | | | | | | | | | | | | | | | | | | S | S | S | S | S |
| | OFLTRACKS | | | | | | | | | | | | | | | | | | | | O | | | O |
| | EXTENTNUMBER | | | | | | | | | | | | | | | | | | | S | S | S | S | S |
| | U (maxblocksize) | T | | | T | | | | T | | | T | T | C | C | C | | | | | | | | |
| | F (blocksize) | S | S | S | S | S | S | S | S | S | S | S | S | C | C | C | C | C | C | C | C | C | C | C |
| | F (blocksize, recsize) | | | | | | | | | | | | | C | C | C | C | C | C | C | C | C | C | C |
| | V (maxblocksize) | | | | | | | | | | | | | C | | C | | | | | | | | |
| | BUFFERS (1) | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | D | D |
| | BUFFERS (2) | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | | |
| | LEAVE | | O | | | | O | | | O | | | | O | O | O | | | | | | | | |
| | NOLABEL | | O | | | | O | | | O | | | | O | O | O | | | | | | | | |
| | VERIFY | | | | | | | O | | | O | | | | | | | O | O | | O | O | | O |
| | NOTAPEMK | | | | | | O | | | O | | | | | | O | | | | | | | | |
| | ALTTAPE | | O | | | | O | | | O | | | | O | | O | | | | | | | | |
| | CTLASA | | | | | | | | | | | | O | | | | | | | | | | | |
| | NOWRITE) | | | | | | | | | | | | | | | | | | | | | | | O |

Symbols used:
S = Attribute must be specified
D = Default attribute, if not specified
O = Optional attribute. Specify if applicable
C = Choice must be made among these attributes
T = For 2152 U must be used instead of F
No entry is permitted, where a blank appears

# Appendix F. Valid I/O Statements

| VALID INPUT/OUTPUT STATEMENT FORMATS | APPLICABLE ON-CONDITIONS | STREAM | | | RECORD | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | SEQUENTIAL | | | | | | DIRECT | |
| | | | | | CONSECUTIVE | | | INDEXED | | | | |
| | | INPUT | OUTPUT NOT PRINT | OUTPUT PRINT | INPUT | OUTPUT | UPDATE | INPUT | OUTPUT | UPDATE | INPUT | UPDATE |
| OPEN | FILE (filename) | O | O | O | M | M | M | M | M | M | M | M |
| | FILE (filename) PAGESIZE (n) | | | O | | | | | | | | |
| CLOSE | FILE (filename) | O | O | O | O | O | O | O | O | O | O | O |
| GET | FILE (filename) EDIT (data) (format)... | O | | | | | | | | | | |
| PUT | FILE (filename) EDIT (data) (format)... | | O | O | | | | | | | | |
| | FILE (filename) PAGE | | | O | | | | | | | | |
| | FILE (filename) SKIP (n) | | | O | | | | | | | | |
| | FILE (filename) PAGE EDIT (data) (format)... | | | O | | | | | | | | |
| | FILE (filename) SKIP (n) EDIT (data) (format)... | | | O | | | | | | | | |
| READ | FILE (filename) INTO (variable) | | | | O | | O | O | | O | | |
| | FILE (filename) SET (pointer) | | | | O | | O | | | | | |
| | FILE (filename) INTO (variable) KEY (expression) | | | | | | | O | | O | O | O |
| REWRITE | FILE (filename) | | | | | | O | | | | | |
| | FILE (filename) FROM (variable) | | | | | | O | | | O | | |
| | FILE (filename) FROM (variable) KEY (expression) | | | | | | | | | O | O | |
| LOCATE | variable FILE (filename) SET (pointer) | | | | | O | | | | | | |
| WRITE | FILE (filename) FROM (variable) | | | | | O | | | | | | |
| | FILE (filename) FROM (variable) KEYFROM (expression) | | | | | | | | O | | | O |
| ON-CONDITIONS WHICH MAY OCCUR | CONVERSION | O | | | | | | | | | | |
| | ENDFILE (filename) | O | | | O | | O | O | | O | | |
| | ENDPAGE (filename) | | | O | | | | | | | | |
| | KEY (filename) | | | | | | | O | O | O | O | O |
| | RECORD (filename) | | | | O | | | | | | | |
| | TRANSMIT | O | O | O | O | O | O | O | O | O | O | O |

Symbols used: M = Use of this statement is mandatory
O = For I/O statements: Use of this statement format is optional
For ON conditions: This condition may occur

# READER'S COMMENT FORM

IBM System/360 Model 20
Disk Programming System
PL/I

Form C33-6007-1

- How did you use this publication?

  As a reference source .............................. □
  As a classroom text .................................. □
  As a self-study text ................................. □

- Based on your own experience, rate this publication . . .

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| As a reference source: | ........ | .......... | ........ | ........ | ........ |
|  | Very Good | Good | Fair | Poor | Very Poor |
| As a text: | ........ | .......... | ........ | ........ | ........ |
|  | Very Good | Good | Fair | Poor | Very Poor |

- What is your occupation? ...................................................................................................................

- We would appreciate your other comments; please give specific page and line references where appropriate. If you wish a reply, be sure to include your name and address.

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

## YOUR COMMENTS, PLEASE . . .

This SRL manual is part of a library that serves as a reference source 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.

CUT ALONG THIS LINE

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 BP

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®

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]