SC33-0009-2

Program Product

OS PL/I Checkout and Optimizing Compilers: Language Reference Manual

Program Number 5734-PL1
(This program product is available
as part of composite package 5734-PL3)

Program Number 5734-PL2

IBM

Third Edition (September 1972)

This is a reprint of SC33-0009-1, incorporating changes released in the following Technical Newsletter:

SN33-6036 (dated November 5th,1972)

This edition applies to Release 1.0 of the OS PL/I Optimizing Compiler, and all subsequent releases until otherwise indicated in new editions or Technical Newsletters.

Changes are continually made to the information in this publication; before using it in connection with operation of IBM systems, consult the latest IBM System/360 and System/370 Bibliography SRL Newsletter, Order No. GN20-0360, for the editions that are applicable and current.

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

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

Preface

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

The publication is in two parts. Part I contains discussions of concepts of the language. Part II contains detained rules and syntactic descriptions.

Although implementation information is included, the book is not a complete description of any implementation environment. In general, it contains information needed to write a program that will be processed by the OS PL/I Optimizing Compiler or the OS PL/I Checkout Compiler. It does not contain all the information needed to execute programs. For further information on executing a program refer to either the compiler's programmer's guide (for batch processing only) or its Time Sharing Option publication (for processing in a TSO System).

The following restrictions apply to the information given in this publication if the PL/I Optimizing Compiler is being used under Releases 19.6 or 20.0 of the IBM Operating System:

- Extended precision floating point arithmetic is not available under Release 19.6.
- ASCII data sets are not supported under Releases 19.6 and 20.0.
- Conversational processing is not available under Releases 19.6 and 20.0.
- PL/I Teleprocessing facilities are not available under Releases 19.6 and 20.0.

In order to execute programs processed by these compilers, subroutine libraries are required. The subroutines are provided by the OS PL/I resident library (optimizing compiler only) and the OS PL/I transient library (both compilers).

The OS PL/I Optimizing and Checkout Compilers require an MFT or MVT version of the IBM Operating System. Programs that have been compiled by the PL/I Optimizing Compiler and which utilize PL/I multitasking facilities can be executed only under the MVT version of the operating system.

USE OF THIS PUBLICATION

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

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

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

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

In the same manner, he would read chapter 4, "Expressions and Data Conversions" in Part I for a discussion of the concepts of data conversion, but he would use section F, "Data Conversion and Expression Evaluation" in Part II, to determine the exact results of a particular type of conversion.

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

REQUISITE PUBLICATIONS

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

OS PL/I Optimizing Compiler: Programmer's Guide, Order No. SC33-0006

OS PL/I Checkout Compiler: Programmer's Guide, Order No. SC33-0007

OS Time Sharing Option: PL/I Optimizing Compiler, Order No. SC33-0029

OS Time Sharing Option: PL/I Checkout Compiler, Order No. SC33-0033

RECOMMENDED PUBLICATIONS

The subjects covered in the following publications include the compiler facilities, the optimization or checkout features (whichever are applicable), methods of implementing the various language features, and comparisons of the language implemented by the OS PL/I Optimizing or Checkout Compilers with that implemented by the PL/I (F) Compiler.

OS PL/I Optimizing Compiler: General Information,
Order No. GC33-0001

OS PL/I Checkout Compiler: General Information,
Order No. GC33-0003

OS PL/I Optimizing Compiler: Execution Logic,
Order No. SC33-0025

OS PL/I Checkout Compiler: Execution Logic,
Order No. SC33-0032

AVAILABILITY OF PUBLICATIONS

The availability of a publication is indicated by its use key, the first letter in the order number. The use keys for publications referred to in this manual are:

- G - General: available to users of IBM systems, products, and services without charge, in quantities to meet their normal requirements; can also be purchased by anyone through IBM branch offices.
- S - Sell: can be purchased by anyone through IBM branch offices.

Contents

Figures

Part 1: Concepts of PL/I

Chapter 1: Basic Characteristics of PL/I

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

This chapter contains brief discussions of most of the basic features to provide an overall description of the language. Each is treated in more detail in subsequent chapters.

Machine Independence

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

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

Program Structure

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

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

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

Data Types and Data Description

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

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

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

Default Assumptions

An important feature of PL/I is its default philosophy. If all the attributes

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

The default attributes assumed by the compiler are the standard default attributes of the PL/I language and the implementation precision defaults. However, the programmer can override these by use of the DEFAULT statement.

The compiler optionally produces an attribute listing which contains the identifiers used in a PL/I source program and a complete list of the attributes specified either by explicit, contextual, or implicit declarations, or by application of default rules. The programmer can use this listing to check that these attributes are consistent with his intentions.

Storage Allocation

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

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

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

Expressions

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

```
A + B * C
```

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

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

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

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

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

Data Collections

PL/I offers the programmer many ways of describing and operating on collections of data, or data aggregates. Arrays are collections of data elements, all of the same type, collected into lists or tables of one or more dimensions. Structures are hierarchical collections of data, not necessarily all of the same type. Each level of the hierarchy may contain other structures of deeper levels. An item that does not contain another structure must represent an elementary data item or array.

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

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

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

Input and Output

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

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

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

In a system employing the Time Sharing Option, data may be fed into, and output may be obtained from, a PL/I program using a terminal remote from the machine.

Record-oriented input/output is machine dependent. It deals with collections of data, called records, and transmits these one record at a time without any data conversion; the external representation is generally an exact copy of the internal representation. Because the aggregate is treated as a whole, and because no conversion is performed, this form of input/output is more efficient than stream-oriented input/output.

Teleprocessing facilities are provided by PL/I as part of the basic record-oriented transmission facilities.

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, but generally requires a greater programming effort.

Input and output operations for data banks involving a number of interrelated data sets is simplified by the use of file variables. All input/output statements can use file variables with file values established and modified during execution of the program.

Multitasking

The operating system has facilities for multiprogramming, that is, it allows a number of programs to be active concurrently. In the same way, PL/I has facilities to allow a number of procedures within a PL/I program to be active concurrently.

Any PL/I procedure may invoke another, in other words initiate the execution of another procedure. The programmer may specify that the procedures are to be tasks, which means that they may both be active concurrently. The invoked procedure is known as a subtask of the other, and is said to have been attached by it.

The advantage of multitasking is that CPU operations may be carried out in one task while an input/output operation (or other CPU operations, in the case of multiprocessing machines) is carried out concurrently in another. As soon as the CPU or the input/output operations in one task are completed, a search is made amongst all the active tasks for another one that requires the same resource. If more than one such task is found, the resource is assigned to the one having highest priority. The PL/I programmer may allow the system to allocate relative priorities or he may assign priorities to his tasks when they are attached.

A number of tasks may be dependent on each other at various points during their execution. For example, one task may require results obtained in another before it can be completed. In PL/I, the programmer may synchronize tasks at various points in their execution. An operation in one task may be made to await the completion of an operation in another task.

The optimizing and checkout compilers differ in their implementations of multitasking. Each task in a PL/I program compiled by the optimizing compiler forms a system task to be scheduled by the operating system. The checkout compiler constitutes a single task, and the compiler itself schedules the tasks created within a PL/I program.

Facilities of Two Compilers

The optimizing and checkout compilers are complementary program products. The main function of the optimizing compiler is to generate highly efficient object code, while that of the checkout compiler is to minimize the time a programmer needs to spend in debugging.

Both compilers may be used for batch processing, that is, processing in which a program must be compiled, and possibly executed, in full before the programmer obtains any result. The checkout compiler has the facility for conversational processing. In this mode, the program's execution is monitored from a keyboard terminal and temporary amendments may be made during execution as a result of information so obtained; new PL/I code may be temporarily included in the program, for instance. The best use is made of PL/I facilities when both compilers are employed. The program is compiled by the checkout compiler during the debugging stages, to allow the programmer to use his time most efficiently; the debugged program is then compiled by the optimizing compiler, to obtain object code that makes the most efficient use of the machine.

The language implemented by the two compilers is, in general, the same. There are a few exceptions concerned with the different primary function of each compiler. Certain optimizing features are not implemented by the checkout compiler and certain program checkout features are not implemented by the optimizing compiler. For instance, a number of statements instruct the checkout compiler to provide the programmer with information about the flow of control through his program during execution. Since the optimizing compiler does not have these facilities, it merely checks the statements' syntax and otherwise ignores them. Similarly, there are statement options concerned with generating the most efficient object code possible that are used by the optimizing compiler but which are syntax-checked and then ignored by the checkout compiler.

Compile-time Operations

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

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

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

Execution-time Facilities

PL/I includes statements and options that provide powerful facilities for debugging. Other features allow program amendment during execution; these require the use of the Time Sharing Option of the operating system, and of the checkout compiler. They allow the programmer to learn quickly about the behaviour of his program while it is being executed and also, in the appropriate processing environment, to correct it. Also, under the Time Sharing Option, stream I/O can be performed from and to a terminal, on programs compiled by either the checkout or the optimizing compiler.

The debugging facilities cause information to be written on the SYSPRINT file (and, if desired, at the terminal when the terminal is not defined as the SYSPRINT file) throughout execution or at designated points during execution. The programmer can, throughout execution, cause information to be written every time a reference to a selected variable occurs in a pre-defined situation or when a transfer of control takes place. Similarly, at designated points in the program being executed, the information to be written can include the values of selected variables, the names of the procedures currently

active, or the numbers of the statements involved in the latest transfers of control.

The time at which this output is available depends on the processing mode. In batch processing, information written on the SYSPRINT file is only available when the SYSPRINT file is printed, which is normally after execution has terminated. In conversational processing, information written on the SYSPRINT file can be immediately printed at the terminal; therefore the output provided by the debugging facilities can be made available immediately it is produced.

Program amendment during execution is possible only with conversational processing under the checkout compiler. The programmer can enter instructions at the terminal that cause program execution to be suspended and control passed to the terminal. He can then enter statements that are executed during the current suspension of execution or during a further suspension; this future suspension will be at a point specified by the programmer. These statements can, for instance, initiate the debugging facilities described above, change the value of a variable or insert extra statements in the program. The amendments made apply to the current conversational sessions only; they are not made part of the original program. Once normal execution has been resumed, they cannot be retrieved by the programmer, although their effect may last to the end of program execution.

Interrupt Activities

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

PL/I provides facilities for detecting a variety of exceptional conditions. It allows the programmer to specify, by means of a condition prefix, that an interrupt will occur if the condition should arise. By use of an ON statement, he can specify the action to be taken when an interrupt does occur. In conversational processing, the programmer can deal with any error condition immediately it occurs.

Operating System Facilities

A number of facilitites provided by the operating system can be called upon by the PL/I programmer. The most prominent ones, namely interlanguage communication, sort/merge, and checkpoint/restart are outlined below.

It is possible for a PL/I program to communicate with COBOL and FORTRAN routines at execution time, if the latter were compiled by a compiler developed by IBM for OS. A PL/I procedure may invoke a COBOI or FORTRAN routine, and may be invoked by a COBOL or FORTRAN main program or routine. In addition, a PL/I program may be used to create or access a COBOL or FORTRAN data set. All these facilities are provided by the PL/I language. Further communication is possible between PL/I and other languages if an assembler language interface is provided. Such interfaces are described in the following OS publications:

OS PL/I Optimizing Compiler: Programmer's Guide

and

OS PL/I Checkout Compiler: Programmer's Guide.

Provided the operating system has been generated with the appropriate sort/merge program, the sort/merge facilities may be utilized by the PL/I programmer. They may be used on records on PL/I-created data sets, on data passed by a PL/I program, and on data being passed to a PL/I program.

When a PL/I batch processing program compiled by the optimizing compiler is to run for an extended period, the operating system checkpoint/restart facility can be employed to minimize the losses caused by a machine or system failure. The programmer selects checkpoints in his program at which processing is to be recommenced following a failure. Only the processing carried out between the checkpoint and the failure may be lost. Results obtained up to the checkpoint are preserved on external storage, together with data (including a copy of the program and its associated storage) necessary for continuation of the run.

Sort/merge and checkpoint/restart facilities are described in the Programmer's Guides.

Chapter 2: Program Elements

There are no restrictions in the format of PL/I statements, apart from those imposed by the physical form of the source program. Consequently, programs can be written without consideration of special coding forms or checking to see that each statement begins in a specific column. Each statement may begin in the next column or position after the previous statement, or any number of blanks may intervene.

Character Sets

One of two character sets may be used to write a source program; either a 60-character set or a 48-character set. For a given external procedure, the choice between the two sets is optional. In practice, this choice will depend upon the available equipment.

60-CHARACTER SET

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

There are 29 alphabetic characters beginning with the currency symbol ($), the number sign (#), and the commercial "at" sign (@). These characters precede the 26 letters of the English alphabet in Extended Binary-Coded-Decimal Interchange Code (EBCDIC). For use with languages other than English, other characters may be substituted for $, #, and @.

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

There are 21 special characters. They are as follows:

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

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

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

48-CHARACTER SET

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

The restrictions and changes for this character set are described in section B, "Character Sets with EBCDIC and Card-Punch Codes".

---

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

USING THE CHARACTER SET

All the elements that make up a PL/I program are constructed from the PL/I character sets. There are two exceptions: character-string constants and comments may contain any character in the EBCDIC 8-bit code.

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

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

The arithmetic operators are:

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

The comparison operators are:

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

The bit-string operators are:

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

The string operator is:

||	denoting concatenation

Figure 2.1 shows some of the functions of other special characters.

Identifiers

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

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

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

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

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

```
A

FILE2

LOOP_3

RATE_OF_PAY

#32
```

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

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

Figure 2.1. Some functions of special characters

Some identifiers, as discussed in later chapters, cannot exceed seven characters in length and must not contain the break character. This limitation is placed upon certain names, called external names, that may be referred to by the operating system or by more than one separately compiled procedure. If an external name of a PL/I procedure contains more than seven characters, it is truncated by the compiler, which concatenates the first four characters with the last three characters. The entry name of a COBOL or FORTAN routine may have up to eight characters. If more than eight characters are specified, the leftmost eight are taken.

Use of Blanks

Blanks may be used freely throughout a PL/I program. They may surround operators and most other delimiters. In general, any number of blanks may appear wherever one blank is allowed, such as between words in a statement.

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

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

AB+BC is equivalent to AB + BC

TABLE(10) is equivalent to TABLE (10)

FIRST,SECOND is equivalent to FIRST, SECOND

ATOB is <u>not</u> equivalent to A TO B

Comments

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

The general format of a comment is:

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

The character pair /* indicates the beginning of a comment. The same character pair reversed, */, indicates its end. No blanks or other characters can separate the two characters of either composite pair; the slash and the asterisk must be immediately adjacent. The comment itself may contain any characters except the */ combination, which would be interpreted as terminating the comment. The initial /* must never be in columns 1 and 2 of a line.

Example:

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

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

Basic Program Structure

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

SIMPLE AND COMPOUND STATEMENTS

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

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

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

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

Examples of simple statements are:

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

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

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

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

The following is an example of the ON statement:

```
ON OVERFLOW GO TO OVFIX;
```

Statement Prefixes

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

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

A condition prefix specifies whether or not interrupts are to result from the occurrence of the named conditions. Condition names are language keywords, each of which represents an exceptional condition that might arise during execution of a program. Examples are OVERFLOW and SIZE. The OVERFLOW condition arises when the exponent of a floating-point number

exceeds the maximum allowed (representing a maximum value of about 10^{75}). The SIZE condition arises when a value is assigned to a variable with loss of high-order digits or bits.

When the programmer does not expect the condition to arise, he may disable it by preceding the condition name in a prefix by the word NO. If NO is used, there can be no intervening blank between the NO and the condition name.

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

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

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

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

```
(NOCONVERSION):

(SIZE,NOOVERFLOW):

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

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

Condition prefixes are discussed in chapter 14, "Exceptional Condition Handling and Program Checkout".

GROUPS AND BLOCKS

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

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

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

A procedure block may be invoked as a task, in which case it is executed concurrently with the invoking procedure. Tasks are discussed in chapter 17, "Multitasking".

Chapter 3: Data Elements

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

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

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

A constant (which can be a symbolic name) has a value that cannot change.

The following statement has both variables and constants:

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

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

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 last statement, only the digit 2 is a constant.

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.

A constant represented by a symbolic name has a value which is determined by the compiler and which the programmer does not need to know. Normally, such constants are associated with the control of the program; they represent addresses in internal storage rather than computational values. For instance, PL/I statements can be given labels. The identifier used for such a label is a symbolic name which represents a constant, namely the address of the code generated by that statement.

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

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

Data Types

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

Problem Data

The types of problem data are arithmetic and string.

ARITHMETIC DATA

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

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

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

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

Whenever a data item is assigned to a fixed-point variable, the declared precision is maintained. The assigned item is aligned on the decimal or binary point. Leading zeros are inserted if the assigned item contains fewer integer digits than declared; trailing zeros are inserted if it contains fewer fractional digits. A SIZE error may occur if the assigned item contains too many integer digits; truncation on the right may occur, without rounding, if it contains too many fractional digits.

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

Base, scale, and mode of arithmetic variables are specified by keywords; precision is specified by parenthesized decimal integer constants. The precision of arithmetic variables and constants is discussed in greater detail below.

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

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

Decimal Fixed-Point Data

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

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

```
3.1416

455.3

732

003

-5280

.0012
```

For expression evaluation, decimal fixed-point constants have an apparent precision (p,q), where p is the total number of digits in the constant and q is the number of digits specified to the right of the decimal point. For example:

```
3.14 has the precision (3,2)
```

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

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

DECLARE B FIXED (6,0) DECIMAL;

DECLARE C FIXED (7,-2) DECIMAL;

DECLARE D DECIMAL FIXED REAL(3,2)
```

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

The maximum number of decimal digits allowed is 15. Default precision, assumed when no specification is made, is (5,0). The internal coded arithmetic form of decimal fixed-point data is packed decimal. Packed decimal is stored two digits to the byte, with a sign indication in the rightmost four bits of the rightmost byte. Consequently, a decimal fixed-point data item is always stored as an odd number of digits, even though the declaration of the variable may specify the number of digits (p) as an even number. When the declaration specifies an even number of digits, the extra digit place is in the high-order position, and it participates in any operations performed upon the data item, such as in a comparison operation. Any arithmetic overflow or assignment into an extra high-order digit place can be detected only if the SIZE condition is enabled.

Binary Fixed-Point Data

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

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

```
10110B

11111B

101B

-111.01B

1011.111B
```

For expression evaluation, binary fixed-point constants have an apparent precision (p,q), where p is the total number of binary digits in the constant, and q is the number of binary digits specified to the right of the binary point. For example:

```
0000001B  has the precision (7,0)
```

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

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

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

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

The maximum number of binary digits allowed is 31. Default precision is (15,0). The internal coded arithmetic form of binary fixed-point data can be either a fixed-point binary halfword or fullword. A halfword is 15 bits plus a sign bit, and a fullword is 31 bits plus a sign bit. Any binary fixed-point data item with a precision of (15,0) or less is stored as a halfword, and with a precision greater than (15,0), up to the maximum precision, is stored as a fullword. The declared number of digits are considered to be in the low-order positions, but the extra high-order digits participate in any operations performed upon the data item. Any arithmetic overflow into such extra high-order digit positions can be detected only if the SIZE condition is enabled.

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

Decimal Floating-point Data

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

```
15E-23

15E23

4E-3

-48333E65

438E0

3141593E-6

.003141593E3
```

The last two examples represent the same value.

For expression evaluation, decimal floating-point constants have an apparent precision (p) where p is the number of digits of the constant to the left of the E (the mantissa). For example:

```
0.012E5  has the precision (4)
```

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

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

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

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

The maximum precision allowed for decimal floating-point data items is (33); the default precision is (6). The exponent cannot exceed two digits. A value range of approximately 10^{-78} to 10^{75} can be expressed by a decimal floating-point data item. The internal coded arithmetic form of decimal floating-point data is normalized hexadecimal floating-point, with the point assumed to the left of the first hexadecimal digit. If the declared precision is less than or equal to (6), short floating-point form is used; if the declared precision is greater than (6) and less than or equal to (16), long floating-point form is used; if the declared precision is greater than (16), extended floating-point form is used.

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

Binary Floating-point Data

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

```
101101E5B

101.101E2B

11101E-28B

-10.01E99B
```

For expression evaluation, binary floating-point constants have an apparent precision (p) where p is the number of binary digits to the left of the E (the mantissa). For example:

```
0.0101E33B has the precision (5)
```

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

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

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

The maximum precision allowed for binary floating-point data items is (109); the default precision is (21). The exponent cannot exceed three decimal digits. A value range of approximately 2^{-260} to 2^{252} can be expressed by a binary floating-point data item. The internal coded arithmetic form of binary floating-point data is normalized hexadecimal floating-point. If the declared precision is less than or equal to (21), short floating-point form is used; if the declared precision is greater than (21) and less than or equal to (53), long floating-point form is used; if the declared precision is greater than (53), extended floating-point form is used.

Complex Arithmetic Data

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

An imaginary constant is written as a real constant of any type immediately followed by the letter I.

Examples of imaginary constants as written in a program are:

```
27I

3.968E10I

11011.01BI
```

Each of these is considered to have a real part of zero. A complex value with a non-zero real part is represented in the following form:

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

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

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

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

Numeric Character Data

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

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

```
'999V99'
```

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

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

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

The format for declaring a numeric character variable is:

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

For example:

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

This specifies that any value assigned to PRICE is to be maintained as a character string of five decimal digits, with an assumed decimal point preceding the

rightmost two digits. Data assigned to PRICE will be aligned on the assumed point in the same way that point alignment is maintained for fixed-point decimal data.

The numeric picture specification specifies arithmetic attributes of data in much the same way that they are specified by the appearance of a constant. Only decimal data can be represented by picture characters. Complex data can be declared by specifying the COMPLEX attribute along with a single picture specification that describes either a fixed-point or a floating-point data item.

The maximum number of decimal digits allowed in a numeric character item is 15.

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

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

Consider the following example:

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

PRICE = 12.28;

COST = '$12.28';
```

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

```
VALUE = PRICE;

COST  = PRICE;

VALUE = COST;

PRICE = COST;
```

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

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

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

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

STRING DATA

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

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

Character-string Data

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

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

Examples of character-string constants are:

```
'LOGARITHM TABLE'

'PAGE 5'

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

'AC438-19'

(2)'WALLA '
```

The third example actually indicates SHAKESPEARE'S ''HAMLET'' with a length of 24. In the last example, the parenthesized number is a repetition factor, which indicates repetition of the characters that follow. This example specifies the constant 'WALLA WALLA ' (the blank is included as one of the characters to be repeated). The repetition factor must be an unsigned decimal integer constant, enclosed in parentheses. It has a maximum permissible value of 32767.

A null character-string constant is written as two quotation marks with no intervening blank.

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

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

This DECLARE statement specifies that the identifier NAME is to represent character-string data items, 15 characters in length. If a character string shorter than 15 characters were to be assigned to NAME, it would be left adjusted and padded on the right with blanks to a length of 15. If a longer string were assigned, it would be truncated on the right. (Note: If such truncation occurs it can be detected by use of the STRINGSIZE condition).

When no length is specified, the standard default assumption is a length of one.

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

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

This DECLARE statement specifies that the identifier NAME is to be used to represent varying-length character-string data items with a maximum length of 15. The actual length attribute for NAME at any particular time is the length of the data item assigned to it at that time. The programmer need not keep track of the length of a varying-length character string; this is done automatically. The length at any given time can be determined by the programmer, however, by use of the LENGTH built-in function, as discussed in chapter 13, "Editing and String Handling".

Character-string data is maintained internally in character format, that is, each character occupies one byte of storage. The maximum length allowed for variables declared with the CHARACTER attribute is 32,767. The maximum length allowed for a character-string constant before application of repetition factors varies according to the amount of storage available to the compiler, but it will never be less than 512. The minimum length for a character string is zero. The storage allocated for varying-length strings is two bytes longer than the declared maximum length. The initial two

bytes hold the string's current length, in bytes.

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

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

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

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

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

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

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

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

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

Bit-string Data

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

A null bit-string constant is written as two quotation marks with no intervening blank, followed immediately by the letter B.

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

```
'1'B

'11111010110001'B

(64)'0'B

''B
```

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

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

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

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

If no length is specified, a length of one is assumed.

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

Bit strings are stored eight bits to a byte. The maximum length allowed for a bit-string variable is 32,767. The maximum length allowed for a bit-string constant before application of repetition factors depends upon the amount of storage available to the compiler, but it will never be less than 4096 (512 bytes). The minimum length for a bit string is zero. The storage allocated for varying-length strings is two bytes longer than that required by the declared maximum length. The initial two bytes hold the string's current length, in bits.

UNITIALIZED VARIABLES

When the programmer makes a reference to an arithmetic or string variable such that the variable should contain a valid value - assigns the value to another variable for instance - errors can occur if this is the first reference to the variable. The programmer must ensure that a variable has been assigned a value before trying to access it. The checkout compiler checks whether this has been done.

To facilitate this checking, the compiler assigns a special value to each variable as soon as storage is allocated to it. An attempt to use a variable having this value will result in interruption of execution. The special value is one which the variable would not normally have. For instance, with a varying-length character string, the compiler assigns the variable a length of -1. Certain of these special values, however, might occassionally be used by the programmer. These are as follows.

Fixed length character strings:

X'FE' in the first byte

Picture data:

X'FE' in the first byte

Fixed-point binary data: values can be set by the user at sysgen time. Default values are:

halfword	X'8000', i.e. $-2^8 + 1$
fullword	X'80000000', i.e. $-2^{16} + 1$

If it is essential that one of the above values is used in a program to be run under the checkout compiler, the compiler options should specify that no checking for unitialized variables is carried out. The optimizing compiler does not check for unitialized variables.

Program Control Data

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

FILE DATA

A file data item represents information about a PL/I file. It may be a file constant, or the value of a file variable. A file constant can be assigned to a file variable: a reference to the file variable is a reference to the assigned file constant.

LABEL DATA

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

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

```
ABCDE:  MILES = SPEED*HOURS;
```

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

As used above, ABCDE can be classified further as a statement-label constant. A statement-label variable is an identifier that refers to statement-label constants. Consider the following example:

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

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

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

```
DECLARE LBL_X LABEL;
```

ENTRY DATA

Entry data is used only in connection with entry names, and has values which permit references to be made to entry points of procedures. Entry data may be an entry constant or the value of an entry variable.

An entry constant is an identifier that appears in the program as an entry name written as a prefix to a PROCEDURE or ENTRY statement. It permits references to be made to an entry point of a procedure.

Example:

```
P:  PROCEDURE;
CALL P1;
     .
     .
     .
CALL P1A;
     .
     .
     .
P1:  PROCEDURE;
     .
     .
     .
P1A:  ENTRY;
```

P1 and P1A are declared as entry constants. Control is transferred to the procedure entry points designated by P1 or P1A when a reference is made to either entry constant.

An entry variable is an identifier that refers to an entry constant. Consider the following example:

```
DECLARE EV ENTRY VARIABLE,
  (E1,E2) ENTRY;
    .
    .
    .
EV = E1;
CALL EV;
EV = E2;
CALL EV;
    .
    .
    .
```

EV is declared an entry variable by means of the VARIABLE attribute. The first CALL statement invokes an entry point represented by the entry constant E1. The second CALL invokes the entry point E2.

EVENT DATA

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

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

```
DECLARE ENDEVT EVENT;
```

For detailed information, see chapter 17, "Multitasking," chapter 12, "Record-Oriented Transmission", or "DISPLAY" in section J, "Statements".

TASK DATA

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

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

```
DECLARE ADTASK TASK;
```

For detailed information, see chapter 17, "Multitasking."

LOCATOR DATA

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

The value of a pointer variable is effectively an address of a location in storage, and so it can be used to qualify a reference to a variable that may have been allocated storage in several different locations.

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

Locator variables can be declared as in the following example:

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

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

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

For detailed information, see chapter 8, "Storage Control".

AREA DATA

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

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

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

The number of bytes of storage to be reserved can be stated explicitly, as it has been for AREA1 in the example; otherwise a default size is assumed. The default size is 1000 bytes; the theoretical maximum size is 16,777,200 bytes but in practice the maximum depends on the amount of storage available to the program.

For detailed information, see chapter 8, "Storage Control".

Data Organization

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

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

ARRAYS

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

Consider the following two declarations:

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

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

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

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

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

If the lower bound of a dimension is not 1, both the upper bound and the lower bound must be stated explicitly, with the two

numbers connected with a colon. For example:

```
DECLARE LIST_A (4:11);

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

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

The bounds of an array determine the way elements of the array can be referred to. For example, assume that the following data items are assigned to the array LIST, as declared above:

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

The different elements would be referred to as follows:

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

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

The same data could be assigned to LIST_A and LIST_B, as declared above (though not by direct assignment from LIST). In this case it would be would be referred to as follows:

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

Assume that the same data were assigned to TABLE, which is declared as a two-dimensional array (though note again that assignment could not be direct from LIST to TABLE). TABLE can be illustrated as a matrix of four rows and two columns, as follows:

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

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

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

Arrays are not limited to two dimensions; up to 15 dimensions can be declared for an array. In a reference to an element of any array, a subscripted name must contain as many subscripts as there are dimensions in the array.

Examples of arrays in this chapter have shown arrays of arithmetic data. All data types may be collected into arrays. String

arrays, either character or bit, are valid, as are arrays of label, entry, event, file, area, task, or locator data.

Expressions as Subscripts

The subscripts of a subscripted name need not be constants. Any expression that yields a valid arithmetic value can be used. If the evaluation of such an expression yields a value that is not a fixed-point binary integer, it is converted to FIXED BINARY(15,0), since subscripts are maintained internally as binary integers.

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

Cross-sections of Arrays

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

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

A reference to a cross-section of an array may be a reference to two or more elements of that array which may not be adjacent in storage, the elements specified by such a reference being separated by other elements which are not part of the cross-section. The storage represented by such a cross-section is known as non-connected storage. Certain restrictions apply to the use of non-connected storage; for example, a record variable (that is, a variable to or from which data is transmitted by a record-oriented transmission statement) must represent data in connected storage (that is, data items which are adjacent in storage).

STRUCTURES

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

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

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

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

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

```
                PAYROLL

LAST_NAME    REGULAR_HOURS    REGULAR_RATE

FIRST_NAME   OVERTIME_HOURS   OVERTIME_RATE
```

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

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

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

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

PAYROLL

NAME	HOURS	RATE
FIRST	REGULAR	REGULAR
LAST	OVERTIME	OVERTIME

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

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

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

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

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

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

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

This declaration would result in exactly the same structuring as the previous declaration. The maximum permissible number of levels is 15, and the highest permissible level number is 255.

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

Level numbers are specified with structure names only in DECLARE statements and, in the case of controlled structures, ALLOCATE statements. In references to the structure or its elements, no level numbers are used.

Qualified Names

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

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

The different names of a qualified name are connected by periods. Blanks may appear surrounding the period. Qualification is in the order of levels; that is, the name at the highest level must appear first, with the name at the deepest level appearing last.

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

ARRAYS OF STRUCTURES

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

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

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

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

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

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

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

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

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

Cross-Sections of Arrays of Structures

A reference to a cross-section of an array of structures is not permitted, that is, the asterisk notation cannot be used in a reference.

Other Attributes

Keyword attributes for data variables such as BINARY and DECIMAL are discussed briefly in the preceding sections of this chapter. Other attributes that are not peculiar to one data type may also be applicable. A complete discussion of these attributes is contained in section I, "Attributes". Some

that are especially applicable to a discussion of data type and data organization are DEFINED, LIKE, ALIGNED, UNALIGNED, and INITIAL.

DEFINED Attribute

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

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

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

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

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

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

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

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

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

Non-connected storage: The use of the DEFINED attribute to overlay arrays with arrays creates the possibility that array expressions can refer to array elements in non-connected storage (that is , array elements which are not adjacent in storage). It is possible for an array expression involving consecutive elements to refer to non-connected storage in the two following cases:

1. Where an array is declared with iSUB defining. An array expression which refers to adjacent elements in an array declared with iSUB defining can be a reference to non-connected storage (that is, a reference to elements of an overlayed array which are not adjacent in storage).

2. Where a string array is defined on a string array which has elements of greater length. Consecutive elements in the defined array are separated by the difference between the lengths of the elements of the base and defined arrays, and are considered to be held in non-connected storage.

LIKE Attribute

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

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

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

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

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

of COST_OF_LIVING, the declaration would have to be DECLARE 1 COST_OF_LIVING(12) LIKE BUDGET.

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

ALIGNED and UNALIGNED Attributes

In System/360 and System/370, information is held in units of eight bits, or a multiple of eight bits. Each eight-bit unit of information is called a byte. When PL/I data is stored in character form, each character occupies one byte.

Bytes may be handled separately or grouped together in fields. A halfword is a group of two consecutive bytes. A word is a group of four consecutive bytes. A double word is a field consisting of two words. Byte locations in storage are consecutively numbered starting with 0; each number is considered the address of the corresponding byte. A group of bytes in storage is addressed by the leftmost byte of the group.

Fixed-length fields, such as halfwords and double words, must be located in main storage on an integral boundary for that unit of information. A boundary is called integral for a unit of information when its address is a multiple of the length of the unit in bytes. For example, a word (four bytes) must be located in storage so that its address is a multiple of the number 4. A halfword (two bytes) must have an address that is a multiple of the number 2, and a doubleword (eight bytes) must have an address that is a multiple of the number 8 (see figure 3.1).

Halfwords, words, and doublewords may be accessed more readily than a field of the same length that is not aligned on an integral boundary. For this reason, it is a system requirement that data to be used in certain operations is aligned on one of the three integral boundaries.

It is possible in PL/I to align data on boundaries that will give the fastest possible execution. This is not always desirable, however, since there may be unused bytes between successive data elements, which increases use of storage. This is likely to be particularly important when the data items are members of aggregates that are to be used to create a data set; the unused bytes can greatly increase the amount of external storage required. The ALIGNED and UNALIGNED attributes allow the programmer to choose whether or not data is to be stored on the appropriate integral boundary.

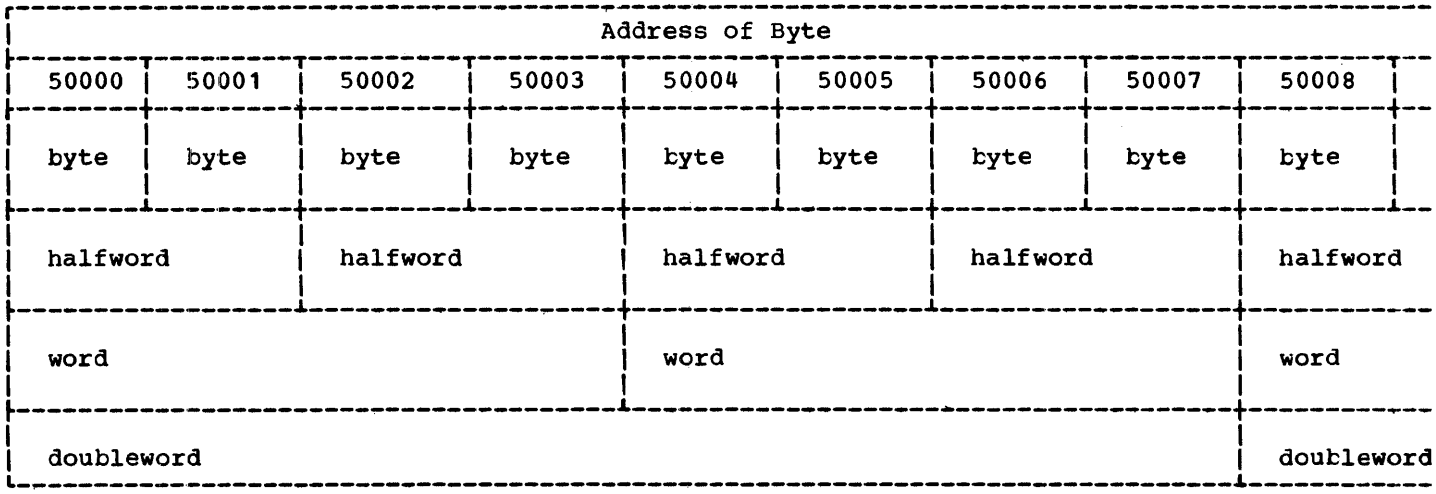


Figure 3.1. Section of main store showing alignment of fixed length fields

ALIGNED specifies that the data element is to be aligned on the storage boundary corresponding to its data type requirement. These requirements are specified in section K, "Data Mapping".

UNALIGNED specifies that each data element, with one exception, is mapped on the next byte boundary. The exception is for fixed-length bit strings, which are mapped on the next bit.

When the UNALIGNED attribute is specified, the compiler generates code that moves the data to an appropriate integral boundary before an operation is performed, if the operation requires data alignment. Consequently, although the UNALIGNED attribute may reduce storage requirements, it may increase execution time.

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

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

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

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

INITIAL Attribute

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

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

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

DECLARE TABLE (100,100) INITIAL CALL
   SUBR;

DECLARE A INIT((B*C));

DECLARE X INIT(SQRT(Z));
```

When storage is allocated for NAME, the character string 'JOHN DOE' (padded on the right to 10 characters) will be assigned to it. When PI is allocated, it will be initialized to the value 3.1416. Either value may be retained throughout the program, or it may be changed during execution.

The third example illustrates the CALL option. It indicates that the procedure SUBR is to be invoked to perform the initialization. The required values must be assigned to TABLE during the execution of SUBR.

The fourth example shows an INITIAL attribute which contains an expression. It specifies that A is to be initialized with the value of the product of B and C.

The fifth example illustrates the use of a function reference to initialize a data item.

For a variable that is allocated when the program is loaded, that is, a static variable, which remains allocated throughout execution of the program, any value specified in an INITIAL attribute is assigned only once. For automatic variables, which are allocated at each activation of the declaring block, any specified initial value is assigned with each allocation. For based and controlled variables, which are allocated at the execution of ALLOCATE statements (also LOCATE statements for based variables), any specified initial value is assigned with each allocation. Note, however, that this initialization of controlled variables can be overridden in the ALLOCATE statement.

The INITIAL attribute cannot be given for entry constants, file constants, DEFINED data, entire structures, or parameters (except CONTROLLED parameters).

Note: The CALL option or an expression containing one or more variables cannot be used with the INITIAL attribute for static data.

An area variable is automatically initialized with the value of the EMPTY built-in function, on allocation, after which any specified INITIAL is applied. An area can be initialized by assignment of another area, using the INITIAL attribute with or without the CALL option.

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

An array or an array of structures can be partly initialized or fully initialized. Uninitialized elements are specified by either omitting to put a value in the INITIAL attribute or by using an asterisk. For example:

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

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

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

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

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

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

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

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

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

Chapter 4: Expressions and Data Conversion

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

```
27

LOSS

A+B

(SQTY-QTY)*SPRICE
```

Any expression can be classified as an element expression (also called a scalar expression), an array expression, or a structure expression. Element variables, array variables, and structure variables can appear in the same expression.

An element expression is one that represents an element value. This definition includes an elementary name within a structure or a subscripted name that specifies a single element of an array.

An array expression is one that represents an array of values. This definition includes a structure, or part of a structure (a minor structure or element) that is given the dimension attribute.

A structure expression is one that represents a structured set of values. None of its operands are arrays, but an operand can be subscripted.

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

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

Examples of element expressions are:

```
C * D

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

RATE.PRIMARY - COST.PRIMARY(1)

A(4,4) * C

RATE.SECONDARY / 4

A(4,6) * COST.SECONDARY(2)
```

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

Examples of array expressions are:

```
A + B

A * C - D

B / 10B

RATE + COST
```

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

Examples of structure expressions are:

```
RATE * COST(2)

RATE / 2
```

Both of these expressions are structure expressions because at least one operand of each is a structure variable and no operand is an array; hence, each expression represents a structure value.

Use of Expressions

Expressions that are single constants or single variables may appear freely throughout a program. However, the syntax of many PL/I statements allows the appearance of operational expressions,

provided the result of the expression conforms with the syntax rules.

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

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

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

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

Data Conversion

OPERATIONAL EXPRESSIONS

An operational expression consists of one or more single operations. A single operation is either a prefix operation (an operator preceding a single operand) or an infix operation (an operator between two operands). The two operands of any infix operation, when the operation is performed, usually must be of the same data type.

The operands of an operation in a PL/I expression are automatically converted, if necessary, to a common representation before the operation is performed. General principles concerning these conversions are given in "Attributes of Targets" later in this chapter. Detailed rules for specific cases, including rules for computing the precision or length of a converted item, can be found in section F, "Data Conversion and Expression Evaluation."

Data conversion is mainly confined to problem data. The only conversion possible with program control data is between offset and pointer types (except that conversion to character strings takes place under the checkout compiler during stream output).

There are very few restrictions on the use of more than one representation in an expression. It must be realized, however, that such mixtures imply conversions. If conversions take place at execution time, they will slow down execution. Also, unless care is taken, conversion can result in loss of precision and can produce unexpected results. Mixed-representation expressions should, therefore, be avoided as far as possible, and when they are used the relevant conversion rules should be thoroughly understood by the programmer.

ASSIGNMENT

In addition to conversion performed in the evaluation of an expression, conversion will also occur when a data item (or the result of an expression evaluation) is assigned to a variable whose attributes differ from the attributes of the item assigned. The rules for such conversions are, with a few exceptions, the same as those for conversion in the evaluation of operational expressions.

Conversion also takes place during stream-oriented input/output (see chapter 11), and there are a number of other circumstances that cause conversion; a complete list is given in Section F.

PROBLEM DATA CONVERSION

Two classes of conversion can be performed on problem data: type conversion and arithmetic conversion.

Type conversions are those that take place between the four different types of problem data, namely:

character-string - data with the CHARACTER attribute

bit-string - data with the BIT attribute

numeric character - data with a PICTURE attribute that contains neither of the picture characters A and X.

coded arithmetic - data with FIXED or FLOAT, DECIMAL or BINARY, REAL or COMPLEX, and precision attributes.

(Strictly, numeric character data is merely a particular case of arithmetic data, but for the purpose of presenting the conversion rules, it is regarded as a separate type of representation.)

Arithmetic conversions are those that occur within the coded arithmetic form - conversions between fixed-point and floating-point scales, decimal and binary bases, and real and complex modes, and conversions of precision.

An example of type conversion is a bit string being converted to coded arithmetic representation during the evaluation of an arithmetic expression. The bit string is interpreted as an unsigned binary integer, as if it had the attributes FIXED BINARY(31,0) REAL, with a value equal to the positive binary value represented by the bit pattern in the string. If the current length of the string is greater than 31, excess bits on the left-hand end of the string are ignored.

An example of arithmetic conversion is an item being converted from fixed-point decimal representation to floating-point binary representation, both in real mode, during the evaluation of an arithmetic expression. The item retains the same value but the scale on which it is represented is changed from decimal to binary and its base is changed from fixed-point to floating-point. Also, the value of the precision attribute is increased by a factor of 3.32, because 3.32 times as many binary integers are required to represent a given value as decimal integers. The precision is rounded up to an integer after being multiplied by 3.32.

LOCATOR DATA CONVERSION

The only type of program control data that may be converted during evaluation of expressions, and execution of assignment statements, is locator data, that is, data with the OFFSET or POINTER attributes. During the evaluation of an expression (locator data may be included in comparison operations using the = and ¬= comparison operators), only offset to pointer conversion may occur. During an assignment, conversion from offset to pointer and from pointer to offset may occur.

USE OF BUILT-IN FUNCTIONS

As well as allowing conversions to take place during expression evaluation and on assignment, the programmer may initiate conversions when he requires them by means of PL/I built-in functions. (The concept of a built-in function is explained in chapter 9, "Subroutines and Functions," and detailed descriptions of the functions are given in section G, "Built-in Functions and Pseudovariables.")

The functions are:

CHAR
BIT
FIXED
FLOAT
DECIMAL
BINARY

Each function converts data to the attribute implied by its name. It will perform any type and arithmetic conversions that may be required. In addition to these functions, there are the COMPLEX built-in function, which converts two real arguments to a single complex value, and the function REAL, which extracts the real part of a complex value.

In the case of BIT and CHAR built-in functions, the programmer may specify the length attribute of the resultant string, and in the case of FIXED, FLOAT, DECIMAL, and BINARY, he may specify the precision of the result.

The precision of a data item may be controlled by means of the PRECISION built-in function.

Conversion between pointer and offset types may be initiated by the programmer using the OFFSET and POINTER built-in functions.

Most of the conversions performed by these built-in functions could equally readily be achieved by assignment to a PL/I variable having the required attributes (with the exception of the conversions performed by the COMPLEX built-in function). The programmer may, however, find the use of a built-in function more convenient than the creation of a variable solely for the purpose of carrying out a conversion.

Expression Operations

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

ARITHMETIC OPERATIONS

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

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

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

An expression of greater complexity can be composed of a set of such arithmetic operations. Note that prefix operators can precede and be associated with any of the operands of an infix operation. For example, in the expression A*-B, the minus sign preceding the variable B indicates that the value of A is to be multiplied by -1 times the value of B.

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

Results of Arithmetic Operations

After any necessary conversion of the operands in an expression has been carried out, the arithmetic operation is performed and a result is obtained. This result may be the value of the expression or it may be an intermediate result upon which further operations are to be performed.

Consider the expression

```
A * B + C
```

The operation A * B is performed first, to give an intermediate result. Then the value of the expression is obtained by performing the operation (intermediate result) + C.

The intermediate result is held in a temporary location designated by the compiler. It has attributes in the same way as any variable in a PL/I program. What attributes the result has depends on the attributes of the two operands (or the single operand in the case of a prefix operation) and on the operator involved. This dependence is further explained under "Attributes of Targets" later in this chapter.

An intermediate result may undergo conversion if a further operation is to be performed, and the value of an expression may be converted if it is assigned. These conversions follow exactly the same rules as the conversion of programmer-defined data.

Operations using Built-in Functions

There are three built-in functions in PL/I that allow the programmer to override the implementation precision rules for addition, subtraction, multiplication, and division operations. (The concept of a built-in function is explained in chapter 9, "Subroutines and Functions," and the functions are described in detail in section G, "Built-in functions and Pseudovariables.")

The functions are ADD, MULTIPLY, and DIVIDE. ADD may be used for subtraction simply by prefixing the operand to be subtracted with a minus sign. In using these functions, two operands are specified, together with the precision of the result. The base, scale, and mode of the result are as defined by the rules for conversion in the evaluation of expressions.

BIT-STRING OPERATIONS

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

```
¬  &  |
```

The first operator, the "not" symbol, can be used as a prefix operator only. The second and third operators, the "and" symbol and the "or" symbol, can be used as infix operators only. (The operators have the same function as in Boolean algebra.)

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

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

Bit-string operations are performed on a bit-by-bit basis. The effect of the "not"

operation is bit reversal; that is, the result of ¬1 is 0; the result of ¬0 is 1. The result of an "and" operation is 1 only if both corresponding bits are 1; otherwise, the result is 0. The result of an 'or' operation is 1 unless both operands are zero, in which case it is 0. The following table illustrates the result for each bit position for each of the operators:

A	B	¬A	¬B	A&B	A|B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

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

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

```
          ¬ A yields '101000'B

          ¬ C yields '001'B

        C & B yields '110000'B

        A | B yields '111111'B

        C | B yields '111111'B

    A | (¬C) yields '011111'B

¬((¬C)|(¬B)) yields '110111'B
```

Boolean Built-in Function

In addition to the "not", "and" and "or" operations using the operators ¬,& and |, Boolean operations may be performed using the BOOL built-in function. The concept of a built-in function is described in chapter 9, "Subroutines and Functions," and the function is described in detail in section G, "Built-in Functions and Pseudovariables."

COMPARISON OPERATIONS

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

```
< ¬< <= = ¬= >= > ¬>
```

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

There are four types of comparisons:

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

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

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

4. Program control data, which involves comparison of the internal coded forms of the operands. Only the comparison operators = and ¬= are permitted; area variables cannot be compared. The only conversion that can take place is offset to pointer; all other type differences between operands for program control data comparisons are in error.

If the operands of a problem data comparison are not immediately compatible (that is, if their data types are appropriate to different types of comparison), the operand of the lower precedence is converted to conform to the comparison type of the other. The precedence of comparison types is (1) algebraic (highest), (2) character, (3) bit. Thus, for example, if a bit string were to be compared with a fixed decimal value, the bit string would be converted to fixed binary for algebraic comparison with the decimal value (which would also be converted to fixed binary). In the comparison of strings of unequal lengths, the shorter string is padded on the right with blanks (in a character comparison) or '0'B (in a bit comparison).

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

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

```
IF A = B

    THEN action-if-true

    ELSE action-if-false
```

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

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

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

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

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

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

CONCATENATION OPERATIONS

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

```
||
```

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

The concatenation operator can cause conversion to string type since concatenation can be performed only upon strings, either character strings or bit strings. If either operand is character or decimal, any necessary conversions are performed to produce a character-string result. Otherwise if the operands are bit and binary, or both binary, conversions are performed to produce a bit-string result.

The results of concatenation operations are as follows:

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

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

If an operand requires conversion for the concatenation operation, the result is dependent upon the length of the character string to which the operand is converted. For example, if A has the attributes and value of the constant '010111'B, B of the constant '101'B, C of the constant 'XY,Z', and D of the constant 'AA/BB', then

```
   A||B yields '010111101'B

A||A||B yields '010111010111101'B

   C||D yields 'XY,ZAA/BB'

   D||C yields 'AA/BBXY,Z'

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

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

COMBINATIONS OF OPERATIONS

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

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

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

Assume that the variables given above are declared as follows:

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

- The decimal value of A would be converted to binary base.

- The binary addition would be performed, adding A and B.

- The binary result would be compared with the converted binary value of C.

- The bit-string result of the comparison would be extended to the length of the bit string D, and the "and" operation would be performed.

- The result of the "and" operation, a bit string of length 4, would be assigned to RESULT without conversion, but with truncation on the right.

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

Priority of Operators

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

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

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

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

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

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

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

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

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

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

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

The value of A would be converted to fixed-point binary, and the addition would be performed, yielding a fixed-point binary result (result_1). The value of C would be converted to a bit string (if valid for such conversion) and the "and" operation would be performed.

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

```
result_1 < result_2
```

result_2 would be converted to binary, and the algebraic comparison would be performed, yielding the bit-string result of the entire expression.

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

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

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

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

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

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

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

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

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

The evaluation will yield a character-string result.

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

```
A + (B ** 3)

(C * D) - E
```

Function Reference Operands

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

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

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

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

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

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

The code represented by the name in the function reference is called a function. The function SQRT is one of the PL/I built-in functions. Built-in functions, which provide a number of different operations, are a part of the PL/I language. A complete discussion of each appears in section G, "Built-in Functions and Pseudovariables." In addition, a programmer may write functions for other purposes (as described in chapter 9, "Subroutines and Functions"), and the names of those functions can be used in function references.

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

There are, however, several built-in functions that can be used as pseudovariables. A pseudovariable is a built-in function name that is used in a receiving field. Consider the following example:

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

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

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

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

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

All the built-in functions that can be used as pseudovariables are discussed in section G, "Built-in Functions and Pseudovariables." No programmer-written function can be used as a pseudovariable.

Attributes of Targets

The target of a conversion or expression operation is the receiving field to which the result of the conversion or operation is assigned. This section deals with the principles of determining attributes of such targets. Detailed rules are given in section F, "Data Conversion and Expression Evaluation."

In the case of a direct assignment, such as the statement

```
A = B;
```

in which conversion must take place, then the target is the variable on the left of the assignment symbol (in this case A). However, during the evaluation of an expression, targets are frequently temporary storage locations created by the compiler.

Consider the following example:

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

A = B + C;
```

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

1. The compiler-created temporary to which the converted binary equivalent of B is assigned.
2. The compiler-created temporary to which the binary result of the addition is assigned.
3. The compiler-created temporary to which the converted decimal fixed-point equivalent of the binary result is assigned.
4. A, the final destination of the result, to which the converted character-string equivalent of the decimal fixed-point representation of the value is assigned.

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

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

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

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

It should be realized that constants also have attributes; the constant 1.0 is different from the constants 1, '1'B, '1', 1B, or 1E0. Under the optimizing compiler, constants may be converted at compile time as well as at execution time, but in all cases, the rules are the same.

Array Expressions

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

Evaluation of an array expression yields an array result. All operations performed on arrays are performed on an

element-by-element basis, in row-major order. Therefore, all arrays referred to in an array expression must have the same number of dimensions, and each dimension must be of identical bounds.

Although comparison operators are valid for use with array operands, an array operand cannot appear in the IF clause of an IF statement. Only an element expression is valid in the IF clause, since the IF statement tests a single true or false result. However, the equality of two arrays of string data can be tested by using the STRING built-in function and pseudovariable to produce two element values. For example:

```
DECLARE (A,B) (10) CHAR(5);
     .
     .
     .
IF STRING(A) = STRING(B) THEN ...
```

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

PREFIX OPERATORS AND ARRAYS

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

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

INFIX OPERATORS AND ARRAYS

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

Array-and-element Operations

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

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

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

Consider the following assignment statement:

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

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

```
50   100  800

1200 1100 300
```

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

Array-and-array Operations

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

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

Examples of array infix expressions are:

If A is the array	2	4	3
	6	1	7
	4	8	2
and if B is the array	1	5	7
	8	3	4
	6	3	1
then A+B is the array	3	9	10
	14	4	11
	10	11	3
and A*B is the array	2	20	21
	48	3	28
	24	24	2

Array-and-structure Operations

The result of an operation in which an array and structure are connected by an infix operator is an array of structures with bounds identical to the array and structuring identical to the structure.

For example, given the following declaration:

```
DECLARE 1 A, 2 B, 2 C,
        X(2),
        Y(2) LIKE A;
```

the assignment statement:

```
Y = X + A;
```

is valid. This is equivalent to:

```
Y.B(1) = X(1) + A.B;
Y.C(1) = X(1) + A.C;
Y.B(2) = X(2) + A.B;
Y.C(2) = X(2) + A.C;
```

If the structure has a dimension attribute on the level 1 name, the operation becomes an array-and-array operation. If the array elements are structures, the rules about identical structuring given under "Structure Expressions" apply to the array elements and the structure.

Data Conversion in Array Expressions

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

Structure Expressions

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

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

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

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

PREFIX OPERATORS AND STRUCTURES

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

Note: Since structures may contain elements of many different data types, a prefix operation in a structure expression

would be meaningless unless the operation can be validly performed upon every element represented by the structure variable, which is either a major structure name or a minor structure name.

INFIX OPERATORS AND STRUCTURES

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

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

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

Structure-and-element Operations

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

Consider the following structure:

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

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

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

Structure-and-structure Operations

When an operation has two structure operands, it is the same as a series of element operations, one for each corresponding pair of elements. For example, if A is the structure shown in the previous example and if M is the following structure:

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

then A || M is equivalent to:

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

Structure Assignment BY NAME

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

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

Consider the following structures and assignment statements:

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

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

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

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

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

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

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

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

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

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

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

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

Exceptional Conditions

Three PL/I exceptional conditions may be raised during conversion of data: SIZE, CONVERSION, and STRINGSIZE. (The concept of a condition is explained in chapter 14, "Exceptional Condition Handling and Program Checkout," and the conditions are described in detail in section H, "On-Conditions.")

The SIZE condition is raised when significant digits are lost from the left-hand side of an arithmetic value. This can occur during conversion within an expression, or upon assigning the result of an expression. It is not raised in conversion to character string or bit string even if the value is truncated. It is raised on conversion to E or F format in edit-directed output if the field width specified will not hold the converted value of the list item. The SIZE condition is normally disabled, so an interrupt will occur only if the condition is raised within the scope of a SIZE prefix (except that, under the checkout compiler, standard system action takes place whether or not the condition is enabled).

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

The STRINGSIZE condition is raised when a character or bit string is assigned to a target that is too small to accommodate it. Characters or bits are truncated from the right-hand end of the string so as to match the length of the target. The STRINGSIZE condition is normally disabled, so that an interrupt will occur only within the scope of a STRINGSIZE condition prefix.

These three conditions may be raised also during the evaluation of an expression. In addition, four other conditions may be raised: FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, and ZERODIVIDE. Note that FIXEDOVERFLOW and OVERFLOW are raised when the implementation-defined maximum precisions are exceeded, not when the declared precision of a target is exceeded.

Chapter 5: Statement Classification

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

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

Classes of Statements

Statements can be grouped into the following classes:

Descriptive

Input/Output

Data Movement and Computational

Program Organization

Storage Control

Control

Exception Control

Preprocessor

Diagnostic

The names of the classes have been chosen for descriptive purposes only; apart from preprocessor statements they have no fundamental significance in the language. A statement may be included in more than one class, since it can have more than one function.

DESCRIPTIVE STATEMENTS

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

DECLARE and DEFAULT Statements

The DECLARE statement is the principal means of specifying the attributes of a name. A name used in a program need not always appear in a DECLARE statement; its attributes often can be determined by context. If the attributes are not explicitly declared and cannot be determined by context, default rules are applied. Default rules are either the standard default rules defined for the compilers or those defined by the programmer for a particular program using the DEFAULT statement. The combination of default rules and context determination can make it unnecessary, in some cases, to use a DECLARE statement.

The DEFAULT statement gives the programmer control over attributes which are applied by default, for the following:

explicitly declared identifiers

contextually declared identifiers

implicitly declared identifiers

descriptors in the ENTRY attribute

values returned by internal procedures

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

Other Descriptive Statements

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

INPUT/OUTPUT STATEMENTS

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

Each of the input/output statements is used with an associated FILE option to identify a file. The file option specifies a file expression which can be either a file constant, a file variable, or a function reference which returns a file value.

In the following list, the statements used when transferring data are grouped into two subclasses, RECORD I/O and STREAM I/O:

RECORD I/O Statements

- READ
- WRITE
- REWRITE
- LOCATE
- DELETE

STREAM I/O Statements

- GET
- PUT

I/O Control Statements

- OPEN
- CLOSE
- UNLOCK

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

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

As a result of these differences, record transmission is particularly applicable for processing large files that are written in an internal representation, such as in binary or decimal. Stream transmission may be used for processing keypunched data and for producing readable output, where editing is required.

Record Transmission Statements

The READ statement transmits records directly into internal storage and makes them available for processing. The WRITE statement causes records to be transmitted to the output device. The LOCATE statement allocates storage for a variable within an output buffer, setting a pointer to indicate the location in the buffer, having previously caused any record already located in a buffer for this file to be written out.

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

STREAM Transmission Statements

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

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

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

Note: The GET and PUT statements can also be used for internal data movement, by specifying a string name in the STRING option instead of specifying the FILE option. Although the facility may be used for moving data to and from a buffer, it is not actually a part of the input/output operation.

Input/Output Control Statements

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

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

The OPEN statement may be used to specify any file attribute except the ENVIRONMENT attribute. For a PRINT file, the length of each printed line and the number of lines per page can be specified only in an OPEN statement by the PAGESIZE and LINESIZE options. The LINESIZE option can be specified for a non-PRINT OUTPUT file to determine the length of the physical blocks transmitted to a device. The OPEN statement can also be used to specify a name (in the TITLE option) other than a file name, as a link between the data set and the file.

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

The UNLOCK statement releases, for use by other tasks, a record which has restricted access because it is associated with an EXCLUSIVE file.

DISPLAY Statement

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

DATA MOVEMENT AND COMPUTATIONAL STATEMENTS

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

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

Assignment Statement

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

```
NTOT=TOT;

AV=(AV*NUM+TAV*TNUM)/(NUM+TNUM);
```

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

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

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

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

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

Such a statement is executed in exactly the same way as a single assignment, except

that the value of B + C is assigned to both A and X. In general, it has the same effect as if the following two statements had been written:

```
A = B + C;

X = B + C;
```

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

PROGRAM ORGANIZATION STATEMENTS

The program organization statements are those statements used to delimit sections of a program into blocks and to manipulate these blocks. These statements are the PROCEDURE statement, the END statement, the ENTRY statement, the BEGIN statement, the FETCH statement, and the RELEASE statement.

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

PROCEDURE Statement

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

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

The sequence of statements defined by a procedure can be executed at any point at which the procedure name is known. This execution can be either synchronous (that is, the execution of the invoking procedure is suspended until control is returned to it) or asynchronous (that is, execution of the invoking procedure proceeds concurrently with that of the invoked procedure); for details of asynchronous operation, see chapter 17, "Multitasking." A procedure is invoked either by a CALL statement or by the appearance of its name in an expression, in which case the procedure is called a function reference. A function reference causes a value to be calculated and returned to the function reference for use in the evaluation of the expression. A function procedure cannot be executed asynchronously with the invoking procedure.

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

ENTRY Statement

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

Note: It is important to distinguish between the ENTRY statement, which specifies an entry to the procedure in which it occurs, and the ENTRY attribute. The ENTRY attribute is considered in chapter 9, in "Subroutines and Functions."

BEGIN Statement

Local definitions of names can also be made within begin blocks, which are delimited by a BEGIN statement and an associated END statement. The BEGIN and END statements specify that the statements contained between them are to be considered as an entity for the purpose of flow of control. Begin blocks are executed in the normal

flow of a program. One of the most common uses of a begin block is as the on-unit of an ON statement, in which case it is not executed through normal flow of control, but only upon occurrence of the specified condition. It is also useful for delimiting a section of a program in which some automatic storage is to be allocated.

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

END Statement

The END statement is used to signify the end of a block or group. Every block or group must have an END statement. However, the END statement may be explicit or implicit; a single END statement can be applied to a number of nested blocks and groups by the inclusion of the label of the containing block or group after the keyword END. The other END statements are then implied by the one containing the label, and need not be given explicitly. If no label follows END, the statement applies to only one group or block.

Execution of an END statement for a block terminates the block. However, it is not the only means of terminating a block, even though each block must have an END statement. For example, a procedure can be terminated by execution of a RETURN statement (see "Control Statements").

The effect of execution of an END statement for a group depends on whether or not the group is iterative (see "Control Statements"). If the group is iterative, execution of the END statement causes control to return to the beginning of the group until all iterations are complete, unless control is passed out of the group before then. If the group is noniterative, the END statement merely delimits the group (to enable the group to be treated as a single unit in the logic of the program), and control passes to the next statement.

FETCH and RELEASE Statements

The FETCH statement copies a procedure from auxiliary storage into main storage so that it may be invoked, for instance by a CALL statement later in the program. The RELEASE statement frees main storage thus allocated. If a procedure's entry name appears in a FETCH statement, then, even if this FETCH statement is never executed, the invoking statement will load the procedure before attempting to initiate its execution. Also, if the procedure's name appears in a RELEASE statement, but there is no FETCH statement in the invoking procedure, invocation will cause the loading of the invoked procedure.

STORAGE CONTROL STATEMENTS

As with many other conventions in PL/I, the conventions concerning storage allocation may be overridden by the programmer. Storage for variables is generally given the storage class AUTOMATIC by default, which means that the storage remains allocated from the time the procedure is activated until it is terminated. Alternatives to the AUTOMATIC attribute that may be chosen by the programmer are STATIC, in which case storage is allocated for the duration of the entire program, and CONTROLLED or BASED, in which case the storage can be allocated to the variable and freed under the control of the programmer, using the ALLOCATE and FREE statements.

ALLOCATE and FREE Statements

The ALLOCATE statement is used to assign storage to controlled and based data, independent of procedure block boundaries. The bounds of controlled arrays, the lengths of controlled strings, and the size of controlled areas, as well as their initial values, may also be specified at the time the ALLOCATE statement is executed. The FREE statement is used to free previously-allocated controlled and based storage when it is no longer required.

CONTROL STATEMENTS

Statements in a PL/I program, in general, are executed sequentially unless the flow of control is modified by the occurrence of

an interrupt or the execution of one of the following control statements:

```
GO TO

IF

DO

CALL

RETURN

END

STOP

EXIT

HALT
```

GO TO Statement

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

If the label variable is subscripted, the switch may be controlled by varying the subscript. The destination of a GO TO statement can also be specified by a function reference that returns a label value. By using label variables or function references, quite subtle switching can be effected. It is usually true, however, that simple control statements are the most efficient.

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

IF Statement

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

```
IF A = B

   THEN action-if-true

   ELSE action-if-false
```

A THEN or an ELSE clause consists of either a single or compound statement, a do-group (see "DO Statement" below), or a begin block. If the comparison is true, the THEN clause is executed. After execution of the THEN clause, the ELSE clause is not executed, and execution continues with the next statement. Note that the THEN clause can contain a GO TO statement or some other control statement that would result in a different transfer of control.

If the comparison is false, only 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 the ELSE clause is not executed. If A is not equal to B, the THEN clause is not executed, and the value of E is assigned to C.

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

```
IF A = B

   THEN GO TO LABEL_1;

   next-statement
```

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

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

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

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

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

Example 1:

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

Example 2:

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

Example 3:

```
IF A ¬= B THEN GO TO S;

IF C ¬= D THEN GO TO S;
GO TO R;

S:  B = B + 1;
```

DO Statement

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

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

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

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

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

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

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

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

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

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

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

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

```
DO I = J,K;

DO I = 1 TO 10, 13 TO 15;

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

The first statement specifies that the DO-group is executed once only with the value of I set equal to the value of J, and once only with the value of I set equal to the value of K.

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

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

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

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

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

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

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

Noniterative DO Statements

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

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

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

CALL, RETURN, and END Statements

A subroutine may be invoked by a CALL statement that names an entry point of the subroutine. When the multitasking facilities are not in use, control is returned to the activating, or invoking, procedure when a RETURN statement is executed in the subroutine or when execution of the END statement terminates the subroutine. If the CALL statement contains one of the multitasking options, TASK, EVENT, or PRIORITY, the subroutine is executed as a subtask with its own separate flow of control; in this case, the RETURN or END statement merely terminates the separate flow of control established for the subtask. (See chapter 17, "Multitasking.")

The RETURN statement with a parenthesized expression is used in a function procedure to return a value to a function reference.

Normal termination of a program occurs as the result of normal execution of the final END statement of the main procedure or of a RETURN statement in the main procedure, either of which returns control to the calling program, which may be the operating system. Termination of a program by any other method is abnormal.

STOP and EXIT Statements

The STOP and EXIT statements are both used to cause abnormal termination. The STOP statement terminates execution of the entire program, including all concurrent tasks. The EXIT statement terminates only the task that executes it, together with any attached tasks. (See chapter 17, "Multitasking.")

HALT Statement

The HALT statement is effective only in conversational processing; in batch processing it is a null operation. When included in a source program, it causes program execution to be suspended and control passed to the terminal.

EXCEPTION CONTROL STATEMENTS

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

In general, an exceptional condition is the occurrence of an unexpected action, such as an overflow error, or of an expected action, such as an end of file, that occurs at an unpredictable time. A detailed discussion of the handling of these conditions appears in chapter 14, "Exceptional Condition Handling and Program Checkout."

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

ON Statement

The ON statement is used to specify action to be taken when any subsequent occurrence of a specified condition causes a program

interrupt. ON statements may specify particular action for any of a number of different conditions. For all of these conditions, a standard system action exists as a part of PL/I, and if no ON statement is in force at the time an interrupt occurs, the standard system action will take place. For most conditions, the standard system action is to print a message and take action which usually leads to termination of execution.

The ON statement takes the form:

```
ON condition[SNAP]
    {SYSTEM;|on-unit}
```

The "condition" is one of those listed in section H, "On-Conditions." The "on-unit" is a single statement or a begin block that specifies action to be taken when that condition arises and an interrupt occurs. For example:

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

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

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

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

REVERT Statement

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

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

SIGNAL Statement

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

```
SIGNAL OVERFLOW;
```

This statement would simulate the occurrence of an overflow interrupt and would cause execution of the on-unit established for the OVERFLOW condition. If an on-unit has not been established, standard system action for the condition is performed. In most cases, the standard system action is the same as for when the on-unit is entered following an interrupt.

PREPROCESSOR STATEMENTS

PL/I allows a degree of control over the contents of the source program during the compilation. The programmer can specify, for example, that any identifier appearing in the source program will be changed; he can select parts of the program to be compiled without the rest; he can include text from an external source. These operations are performed by the preprocessor stage of the compiler, and are specified by preprocessor statements that appear among the other statements within the source program itself.

In general, preprocessor statements are identified by a leading percent symbol before the keyword; several of them have the same keywords as standard PL/I statements, and these have a similar effect at compile time to that of their counterparts at execution time.

The complete list of preprocessor statements is as follows:

% ACTIVATE

% assignment

% DEACTIVATE

% DECLARE

% DO

% END

% GO TO

% IF

% INCLUDE

% null

% PROCEDURE

Preprocessor RETURN

These statements are discussed in chapter 16, "Compile-Time Facilities" and in section J, "Statements."

LISTING CONTROL STATEMENTS

There are three statements that allow the programmer to control the format of the listing of his program. The statements are:

%PAGE

%SKIP

%CONTROL

They are described in Chapter 16, "Compile-time Facilities."

Although they have the initial % sign, these statements do not require the use of the preprocessor.

DIAGNOSTIC STATEMENTS

A program processed by the PL/I checkout compiler can include statements that provide a considerable amount of diagnostic information during execution. These statements:

1. Control a continuing output of diagnostic information throughout execution:

   CHECK|NOCHECK statement
   FLOW|NOFLOW statement

2. Produce diagnostic information at specific points during execution:

   PUT statement with one of the options:

   LIST
   DATA
   EDIT
   SNAP
   FLOW
   ALL

With the exception of a PUT statement with the LIST, DATA, or EDIT option, none of these statements provide diagnostic information when processed by the PL/I optimizing compiler. This compiler checks these statements for syntax and then ignores them; there is no output. In addition, the implementation of a PUT statement with the LIST or DATA option by the optimizing compiler is different from that of the checkout compiler. The checkout compiler implements such a statement by producing information about problem and program-control variables; the optimizing compiler produces information about problem variables only.

CHECK and NOCHECK Statements

When a CHECK statement is executed, information about the variables specified or assumed is put out whenever these variables occur in pre-defined situations. This continues to the end of program execution or until the CHECK statement is overridden by a NOCHECK statement.

The execution of a CHECK statement that specifies or assumes a particular identifier has the same result as if the CHECK condition has been enabled for every block in which the identifier is known. This applies to all such blocks in the current compilation and to all separately compiled blocks in which the identifier is known and which are active at the same time as the current block.

Information is put out for label and entry constants and for all variables. It comprises:

1. Problem variables:

   Name and value

2. Constants and program-control variables:

   Name, and, under the checkout compiler, details of the current situation of the constant or variable. For example, the details for a file variable include whether the file is open or closed.

The NOCHECK statement prevents output of CHECK information for the specified or assumed variables.

FLOW and NOFLOW Statements

Execution of a FLOW statement results in information being put out at every transfer of control within the current task during execution. This continues to the end of program execution or until a NOFLOW statement is executed.

At each transfer of control, the information put out comprises the statement number of the statement that caused the transfer of control, and the statement number of the statement that received control at that transfer.

The NOFLOW statement prevents the output of FLOW information at a transfer of control.

PUT Statements

When a PUT statement is executed, the output comprises:

LIST, DATA or EDIT

The name of the variable appears if DATA is used. The remaining output is:

Problem variables: Value

Program-control variables (LIST and DATA only): Current situation of the variable

SNAP

The current statement number and a list of the procedures currently active.

FLOW

The same information as for the FLOW statement, for the last n transfers of control. The value of n is specified in a compiler option.

ALL

Information about all the variables in the program, together with the information provided by the SNAP and FLOW options, and the values of the CN built-in functions. Options may be specified to limit the output.

Chapter 6: Program Organization

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

Blocks

A block is a delimited sequence of statements that constitutes a section of a program. It localizes names declared within the block and limits the allocation of variables. There are two kinds of blocks: procedure blocks and begin blocks. The optimizing compiler will accept a maximum of 255 blocks in one compilation. There is no limit for the checkout compiler.

PROCEDURE BLOCKS

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

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

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

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

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

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

BEGIN BLOCKS

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

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

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

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

Unlike procedures, begin blocks generally are not given control through special references to them. The normal sequence of control governing ordinary statement execution also governs the execution of begin blocks. Control passes into a begin block sequentially, following execution of the preceding statement. The only exception is a begin block used as the on-unit in an ON statement. In this case, the block is executed only upon occurrence of the specified condition.

Begin blocks are not essential to the construction of a PL/I program. However, there are times when it is advantageous to use begin blocks to delimit certain areas of a program. These advantages are discussed in this chapter and in chapter 7, "Recognition of Names."

INTERNAL AND EXTERNAL BLOCKS

Any block can contain one or more blocks. That is, a procedure, as well as a begin block, can contain other procedures and begin blocks. However, there can be no overlapping of blocks; a block that contains another block must totally encompass that block.

A procedure block that is contained within another block is called an internal procedure. A procedure block that is not contained within another block is called an external procedure. There must always be at least one external procedure in a PL/I program. (Note; Each external procedure is compiled separately. Entry names of external procedures cannot exceed seven characters.)

Begin blocks are always internal; they must always be contained within another block.

Internal procedure and begin blocks can also be referred to as nested blocks. Nested blocks, in turn, may have blocks nested within them, and so on. The outermost block must always be a procedure. Consider the following example:

```
A:  PROCEDURE;
    statement-a1
    statement-a2
    statement-a3
    B: BEGIN;
       statement-b1
       statement-b2
       statement-b3
       END B;
    statement-a4
    statement-a5
    C: PROCEDURE;
       statement-c1
       statement-c2
       D: BEGIN;
          statement-d1
          statement-d2
          statement-d3
          E: PROCEDURE;
             statement-e1
             statement-e2
             END E;
          statement-d4
          END D;
       END C;
    statement-a6
    statement-a7
    END A;
```

In the above example, procedure block A is an external procedure because it is not contained in any other block. Block B is a begin block that is contained in A; it contains no other blocks. Block C is an internal procedure; it contains begin block D, which, in turn, contains internal procedure E. This example contains three levels of nesting relative to A; B and C are at the first level, D is at the second level (but the first level relative to C) and E is at the third level (the second level relative to C, and the first level relative to D).

Under the optimizing compiler, the maximum permissible depth of nesting is 50. There is no limit under the checkout compiler.

Use of the END Statement with Nested Blocks and DO-Groups (Multiple Closure)

The use of the END statement with a procedure, begin block, or DO-group is governed by the following rules:

1. If a label is not used after END, the END statement closes (i.e., ends) that unclosed block headed by the BEGIN or PROCEDURE statement, or that unclosed DO-group headed by the DO statement, that physically precedes, and appears closest to, the END statement.

2. If the optional label is used after END, the END statement closes that unclosed block or DO-group headed by the BEGIN, PROCEDURE, or DO statement that has a matching label, and that physically precedes, and appears closest to, the END statement. Any unclosed blocks or DO-groups nested within such a block or DO-group are automatically closed by this END statement; this is known as multiple closure.

Multiple closure is a shorthand method of specifying a number of consecutive END statements. In effect, the compiler inserts the required number of END statements immediately preceding the END statement specifying multiple closure. For example, assume that the following external procedure has been defined:

```
FRST:  PROCEDURE;
       statement-f1
       statement-f2
       ABLK:  BEGIN;
              statement-a1
              statement-a2
              SCND:  PROCEDURE;
                     statement-s1
                     statement-s2
                     BBLK:  BEGIN;
                       statement-b1
                       statement-b2
                       END;
                     END;
              statement-a3
              END ABLK;
       END FRST;
```

In this example, begin block BBLK and internal procedure SCND effectively end in the same place; that is, there are no statements between the END statements for each. This is also true for begin block ABLK and external procedure FRST. In such cases, it is not necessary to use an END statement for each block, as shown; rather, one END statement can be used to end BBLK and SCND, and another END can be used to end ABLK and FRST. In the first case, the statement would be END SCND, because one END statement with no following label would close only the begin block BBLK (see the first rule above). In the second case, only the statement END FRST is required; the statement END ABLK is superfluous. Thus, the example could be specified as follows:

```
FRST: PROCEDURE;
      statement-f1
      statement-f2
      ABLK: BEGIN;
            statement-a1
            statement-a2
            SCND: PROCEDURE;
                  statement-s1
                  statement-s2
                  BBLK: BEGIN;
                        statement-b1
                        statement-b2
                  END SCND;
            statement-a3
      END FRST;
```

Note that a label prefix attached to an END statement specifying multiple closure is assumed to apply to the last END statement. Therefore all intervening groups and blocks will be terminated if control passes to such a statement. For example:

```
CBLK:       PROCEDURE;
            statement-c1
            statement-c2
DGP:        DO I = 1 TO 10;
            statement-d1
            GO TO LBL;
            statement-d2
LBL:        END CBLK;
```

In this example, the END CBLK statement closes the block CBLK and the iterative DO-group DGP. The effect is as if an unlabeled END statement for DGP appeared immediately after statement-d2, so that the transfer to LBL would prevent all but the first iteration of DGP from taking place, and statement-d2 would not be executed.

Activation of Blocks

Although the begin block and the procedure have a physical resemblance and play the same role in the allocation and freeing of storage, as well as in delimiting the scope of names, they differ in the way they are activated and executed. A begin block, like a single statement, is activated and executed in the course of normal sequential program flow (except when specified as an on-unit) and, in general, can appear wherever a single statement can appear. For a procedure, however, normal sequential program flow passes around the procedure, from the statement before the PROCEDURE statement to the statement after the END statement of that procedure. The only way in which a procedure can be activated is by a procedure reference.

A procedure reference is the appearance of an entry expression in one of the following contexts:

1. After the keyword CALL in a CALL statement.

2. After the keyword CALL in the CALL option of the INITIAL attribute.

3. As a function reference.

This chapter uses examples of the first of these; the material, however, is relevant to the other two forms as well. For further information, refer to the discussion of the INITIAL attribute in section I, "Attributes," and to chapter 9, "Subroutines and Functions."

The simplest form of the CALL statement is

```
CALL entry-constant;
```

If the entry constant is a label of a PROCEDURE statement it represents the primary entry point to the procedure; if it is a label of an ENTRY statement it represents a secondary entry point. The following is an example of a procedure containing secondary entry points.

```
     A: PROCEDURE;
        statement-1
        statement-2
  ERRT: ENTRY;
        statement-3
        statement-4
        statement-5
  NEXT: RETR: ENTRY;
        statement-6
        statement-7
        statement-8
        END A;
```

In this example, A is the primary entry point to the procedure, and ERRT, NEXT, and RETR specify secondary entry points. Actually, since they are both names for the same ENTRY statement, NEXT and RETR specify the same secondary entry point. The procedure may be activated by one of the following statements:

```
CALL A;
CALL ERRT;
CALL NEXT;
CALL RETR;
```

Alternatively, the appropriate entry name value could be assigned to an entry variable, and this entry variable could be used in the procedure reference. In the following example, the two CALL statements have the same effect.

```
DECLARE ENT1 ENTRY VARIABLE;
    .
    .
    .
ENT1 = ERRT;
    .
    .
    .
CALL ENT1;
    .
    .
    .
CALL ERRT;
```

When a procedure reference is executed, the procedure containing the specified entry point is activated and is said to be invoked; control is transferred to the specified entry point.[1] The point at which the procedure reference appears is called the point of invocation and the block in which the reference is made is called the invoking block. An invoking block remains active even though control is transferred from it to the block it invokes.

Whenever a procedure is invoked at its primary entry point, execution begins with the first executable statement in the invoked procedure. However, when a procedure is invoked at a secondary entry point, execution begins with the first executable statement following the ENTRY statement that defines that secondary entry point. Therefore, if all of the numbered statements in the last example are executable, the statement CALL A would invoke procedure A at its primary entry point, and execution would begin with statement-1; the statement CALL ERRT would invoke procedure A at the secondary entry point ERRT, and execution would begin with statement-3; either of the statements CALL NEXT or CALL RETR would invoke procedure A at its other secondary entry point, and execution would begin with statement-6. Note that any ENTRY statements encountered during sequential flow are never executed; control flows around the ENTRY statement as though the statement were a comment.

Any procedure, whether external or internal, can always invoke an external procedure, but it cannot always invoke an internal procedure that is contained in some other procedure. Those internal procedures that are at the first level of

---

[1] This statement does not apply when the CALL statement specifies one of the multitasking options. See "Multitasking."

nesting relative to a containing procedure can always be invoked by that containing procedure, or by each other. For example:

```
PRMAIN: PROCEDURE;
        statement-1
        statement-2
        statement-3
        A:  PROCEDURE;
            statement-a1
            statement-a2
            B:  PROCEDURE;
                statement-b1
                statement-b2
            END A;
        statement-4
        statement-5
        C:  PROCEDURE;
            statement-c1
            statement-c2
            END C;
        statement-6
        statement-7
        END PRMAIN;
```

In this example, PRMAIN can invoke procedures A and C, but not B; procedure A can invoke procedures B and C; procedure B can invoke procedure C; and procedure C can invoke procedure A but not B.

The foregoing discussion about the activation of blocks presupposes that a program has already been activated. A PL/I program becomes active when a calling program invokes the initial procedure. This calling program usually is the operating system, although it could be another program. The initial procedure, called the main procedure, must be an external procedure whose PROCEDURE statement has the OPTIONS(MAIN) specification, as shown in the following example:

```
CONTRL:  PROCEDURE OPTIONS(MAIN);
         CALL A;
         CALL B;
         CALL C;
         END CONTRL;
```

In this example, CONTRL is the initial procedure and it invokes other procedures in the program.

The following is a summary of what has been stated or implied about the activation of blocks:

- A program becomes active when the initial procedure is activated by the operating system.
- Except for the initial procedure, external and internal procedures contained in a program are activated only when they are invoked by a procedure reference.
- Begin blocks are activated through normal sequential flow or as on-units.
- The initial procedure remains active for the duration of the program.
- All activated blocks remain active until they are terminated (see below).

Termination of Blocks

In general, a procedure block is terminated when, by some means other than a procedure reference, control passes back to the invoking block or to some other active block. Similarly, a begin block is terminated when, by some means other than a procedure reference, control passes to another active block. There are a number of ways by which such transfers of control can be accomplished, and their interpretations differ according to the type of block being terminated.

Note that when a block is terminated, any task attached by that block is terminated (see chapter 17, "Multitasking").

BEGIN BLOCK TERMINATION

A begin block is terminated when any of the following occurs:

1. Control reaches the END statement for the block. When this occurs, control moves to the statement physically following the END, except when the block is an on-unit.
2. The execution of a GO TO statement within the begin block (or any block activated from within that begin block) transfers control to a point not contained within the block.
3. A STOP or EXIT statement is executed (thereby terminating execution of the current task and all its subtasks).
4. Control reaches a RETURN statement that transfers control out of the begin block and out of its containing procedure as well.
5. A procedure within which the begin block is contained has been attached as a task, and the attaching block terminates.

A GO TO statement of the type described in item 2 can also cause the termination of other blocks as follows:

If the transfer point is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are terminated.

For example, if begin block B is contained in begin block A, then a GO TO statement in B that transfers control to a point contained in neither A nor B effectively terminates both A and B. This case is illustrated below:

```
FRST: PROCEDURE OPTIONS(MAIN);
      statement-1
      statement-2
      statement-3
      A: BEGIN;
         statement-a1
         statement-a2
         B: BEGIN;
            statement-b1
            statement-b2
            GO TO LAB;
            statement-b3
            END B;
         statement-a3
         END A;
      statement-4
      statement-5
LAB:  statement-6
      statement-7
      END FRST;
```

After FRST is invoked, the first three statements are executed and then begin block A is activated. The first two statements in A are executed and then begin block B is activated (A remaining active). When the GO TO statement in B is executed, control passes to statement-6 in FRST. Since statement-6 is contained in neither A nor B, both A and B are terminated. Thus, the transfer of control out of begin block B results in the termination of intervening block A as well as termination of block B.

PROCEDURE TERMINATION

A procedure is terminated when one of the following occurs:

1. Control reaches a RETURN statement within the procedure. The execution of a RETURN statement causes control to be returned to the point of invocation in the invoking procedure. If the point of invocation is a CALL statement, execution in the invoking procedure resumes with the statement following the CALL. If the point of invocation is one of the other forms of procedure references (that is, a CALL option or a function reference), execution of the statement containing the reference will be resumed.

2. Control reaches the END statement of the procedure. Effectively, this is equivalent to the execution of a RETURN statement.

3. The execution of a GO TO statement within the procedure (or any block activated from within that procedure) transfers control to a point not contained within the procedure.

4. A STOP or EXIT statement is executed (thereby terminating execution of the current task and all its subtasks).

5. The procedure or a containing procedure has been attached as a task and the attaching block is terminated.

Items 1 and 2 are normal procedure terminations; items 3, 4, and 5 are abnormal procedure terminations.

As with a begin block, the type of termination described in item 3 can sometimes result in the termination of several procedures and/or begin blocks. Specifically, if the transfer point specified by the GO TO statement is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are terminated. Consider the following example:

```
   A: PROCEDURE OPTIONS(MAIN);
      statement-1
      statement-2
      B: BEGIN;
         statement-b1
         statement-b2
         CALL C;
         statement-b3
         END B;
      statement-3
      statement-4
      C: PROCEDURE;
         statement-c1
         statement-c2
         statement-c3
         D: BEGIN;
            statement-d1
            statement-d2
            GO TO LAB;
            statement-d3
            END D;
         statement-c4
         END C;
      statement-5
 LAB: statement-6
      statement-7
      END A;
```

In the above example, A activates B, which activates C, which activates D. In D, the statement GO TO LAB transfers control to statement-6 in A. Since this statement is not contained in D, C, or B, all three

blocks are terminated; A remains active. Thus, the transfer of control out of D results in the termination of intervening blocks B and C as well as the termination of block D.

PROGRAM TERMINATION

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

1. Control for the program reaches an EXIT statement in the major task. This is abnormal termination.

2. Control for the program reaches a STOP statement.[1] This is abnormal termination.

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

4. The ERROR condition is raised in the major task and there is no established on-unit for ERROR and FINISH, or, if one or both of the conditions has an established on-unit, on-unit exit is by normal return, rather than by GO TO branching. This is abnormal termination. The program is not terminated if ERROR was raised by a SIGNAL ERROR statement inserted by the checkout compiler in place of a statement in which an error had been detected. (Note that in conversational processing, the ERROR and FINISH conditions cause control to be passed to the terminal, and this is regarded as equivalent to an on-unit being entered; any statements then entered in immediate mode are processed as if in an ERROR or FINISH on-unit.

On termination of a program, whether normal or abnormal, control is returned to the calling program (this is usually the operating system control program).

Dynamic Loading of an External Procedure

A procedure invoked by a CALL statement or a CALL option of an INITIAL attribute, as described in "Activation of Blocks" in this chapter, or by a function reference, as described in chapter 9, "Subroutines and Functions", is generally resident in main storage throughout the execution of the entire program. If required, however, a procedure may be brought into main storage for only as long as it is required: the invoked procedure is dynamically loaded into, and dynamically deleted from, main storage during execution of the calling procedure.

Dynamic loading and deletion of procedures is particularly useful when a called procedure is not necessarily invoked every time the calling procedure is executed, and when conservation of main storage is more important than a short execution time.

The PL/I statements that initiate the loading and deletion of a procedure are FETCH and RELEASE. The appearance of an entry name in a FETCH or RELEASE statement indicates to the compiler that the procedure containing an entry point with that name will need to be fetched into main storage before it can be executed. When a FETCH statement is executed, the procedure is copied from auxiliary storage into main storage, unless a copy already exists in main storage. In addition, when a CALL statement or option or a function reference is executed, the procedure is copied into main storage, unless a copy exists already. Thus, a procedure may be loaded from auxiliary storage by:

1. execution of a FETCH statement;

   or

2. execution of a CALL statement or option or a function reference, provided that the name of the entry point of the procedure appears, somewhere in the calling procedure, in a FETCH or RELEASE statement.

In neither case is it an error if the procedure has already been fetched into main storage. In case 2, it is not necessary that control should pass through the FETCH or RELEASE statement, either before or after execution of the CALL or function reference.

Whichever statement caused the loading of the fetched procedure, execution of the CALL statement or option or the function reference invokes the procedure in the normal way.

The fetched procedure may be allowed to remain in main storage until execution of the whole program is completed. Alternatively, the storage it occupies may

---

[1] When multitasking is in operation, the program (i.e., the major task) is terminated when any task reaches a STOP statement. See chapter 17, "Multitasking."

be freed for other purposes at any time by means of the RELEASE statement.

Consider the following example, in which PROGA and PROGB are entry names of procedures resident on auxiliary storage.

```
PROG:  PROCEDURE;
           .
           .
           .
       FETCH PROGA;
           .
       CALL PROGA;
           .
       RELEASE PROGA;
           .
           .
           .
       CALL PROGB;
       GO TO FIN;
       FETCH PROGB;
           .
           .
           .
FIN:   END PROG;
```

PROGA will be loaded into main storage by the first FETCH statement, and will be executed when the first CALL statement is reached; its storage is released when the RELEASE statement is executed. PROGB will be fetched when the second CALL statement is reached, even though the FETCH statement referring to this procedure is never executed, and the same CALL statement will initiate execution of PROGB. Note that the same results would be achieved if the statement FETCH PROGA; were omitted; the appearance of PROGA in a RELEASE statement will cause the statement CALL PROGA; to fetch the procedure, as well as invoke it.

The fetched procedure is compiled and link-edited separately from the calling procedure. The programmer must ensure that the entry name specified in FETCH, RELEASE, and CALL statements and options, and in function references, is known in auxiliary storage. The job control statements necessary to achieve this are discussed in OS PL/I Checkout Compiler: Programmer's Guide and OS PL/I Optimizing Compiler: Programmer's Guide

Rules concerning the use of dynamically-loaded procedures are:

1. Only external procedures may be fetched.
2. Identifiers with the EXTERNAL attribute are not permitted in a fetched procedure.
3. Identifiers with the CONTROLLED or FILE attributes are not permitted in a fetched procedure unless they are parameters. Note that this means any file used in the fetched procedure, including either of the standard stream-oriented I/O default files SYSIN or SYSPRINT, must be passed from the calling procedure.
4. Storage for STATIC variables in the fetched procedure is allocated when the FETCH statement is executed, and is freed when a corresponding RELEASE statement is executed. Each time a procedure is fetched into main storage, a STATIC variable either is given the value specified in an INITIAL attribute, or, if there is no INITIAL attribute, is uninitialized.
5. The FETCH, RELEASE, and CALL statements must specify entry constants. Entry variables are not permitted. Note that an entry constant may have no more than seven characters.
6. Fetched procedures may not fetch further procedures.

Storage Allocation

Storage allocation is the process of associating an area of storage with a variable so that the data item(s) to be represented by the variable may be recorded internally. When storage has been associated with a variable, the variable is said to be allocated. Allocation for a given variable may take place statically, that is, before the execution of the program, or dynamically, during execution. A variable that is allocated statically remains allocated for the duration of the program. A variable that is allocated dynamically will relinquish its storage either upon the termination of the block containing that variable or at the request of the programmer, depending upon its storage class.

The manner in which storage is allocated for a variable is determined by the storage class of that variable. There are four storage classes: static, automatic, controlled, and based. Each storage class is specified by its corresponding storage class attribute: STATIC, AUTOMATIC, CONTROLLED, and BASED, respectively. The last three define dynamic storage allocation.

Storage class attributes may be declared explicitly for element, array, and major structure variables. If a variable is an array or a major structure variable, the storage class declared for that variable

applies to all of the elements in the array or structure.

All variables that have not been explicitly declared with a storage class attribute are given the AUTOMATIC attribute, with one exception: any variable that has the EXTERNAL attribute is given the STATIC attribute.

Chapter 8, "Storage Control" discusses how the various storage classes may be used.

Reactivation of an Active Procedure (Recursion)

An active procedure that can be reactivated from within itself or from within another active procedure is said to be a recursive procedure; such reactivation is called recursion.

A procedure can be invoked recursively only if the RECURSIVE option has been specified in its PROCEDURE statement. This option also applies to the names of any secondary entry points that the procedure might have.

The environment (that is, values of automatic variables, etc.) of every invocation of a recursive procedure is preserved in a manner analogous to the stacking of allocations of a controlled variable (see chapter 8, "Storage Allocation"). An environment can thus be thought of as being "pushed down" at a recursive invocation, and "popped up" at the termination of that invocation. Note that a label constant in the current block always contains information identifying the current invocation of the block that contains the label. Consider the following example:

```
RECURS: PROCEDURE RECURSIVE;
  DECLARE X STATIC EXTERNAL INITIAL (0);
    .
    .
    .
  X=X+1;
  PUT DATA (X);
  IF X =5 THEN GO TO LAB;
  CALL AGN;
  X =X-1;
  PUT DATA (X);
    .
    .
    .
  LAB:  END RECURS;

AGN: PROCEDURE RECURSIVE;
  DECLARE X STATIC EXTERNAL INITIAL (0);
    .
    .
    .
  X=X+1;
  PUT DATA(X);
    .
    .
    .
  CALL RECURS;
  X=X-1;
  PUT DATA (X);
  END AGN;
```

In the above example, RECURS and AGN are both recursive procedures. Since X is static and has the INITIAL attribute, it is allocated and initialized before execution of the program begins.

The first time that RECURS is invoked, X is incremented by 1 and X=1 is transmitted by the PUT statement. Since X is less than 5, AGN is invoked. In AGN, X is incremented by 1 and X=2 is transmitted (also by a PUT statement). AGN then reinvokes RECURS.

This second invocation of RECURS is a recursive invocation, because RECURS is still active. X is incremented as before, and then X=3 is transmitted. X is still less than 5, so AGN is invoked again. Since AGN is active when invoked, this invocation of AGN is also recursive. X is incremented once again, X=4 is transmitted, and RECURS is invoked for the third time.

The third invocation of RECURS results in the transmission of X=5. But, since X is no longer less than 5, GO TO LAB is executed, and then RECURS is terminated. However, only the third invocation of RECURS is terminated, with the result that control returns to the procedure that invoked RECURS for the third time; that is, control returns to the statement following CALL RECURS in the second invocation of AGN. At this point X is decremented by 1 and X=4 is transmitted. Then the second invocation of AGN is terminated, and control returns to the procedure that invoked AGN for the second time; that is, control returns to the statement following CALL AGN in the second invocation of RECURS. Here X is decremented again and X=3 is transmitted, after which the second invocation of RECURS is terminated and control returns to the first invocation of AGN. X is decremented again, X=2 is transmitted, the first invocation of AGN is terminated, and control returns to the first invocation of RECURS. X is decremented, X=1 is transmitted, and the first invocation of RECURS is terminated. Control then returns to the procedure that invoked RECURS in the first place.

Note that if a label constant is assigned to a label variable in a particular invocation, a GO TO statement naming that variable in another invocation would restore the environment that existed when the assignment was performed.

Note also that the environment of a procedure invoked from within a recursive procedure by means of an entry variable is the one that was current when the entry constant was assigned to the variable. Consider the following example:

```
   I=1;
   CALL A;        /*FIRST INVOCATION OF A*/
 A:PROC RECURSIVE;
   DECLARE EV ENTRY VARIABLE STATIC;
   IF I=1 THEN DO;
      I=2;
      EV=B;
      CALL A;     /*SECOND INVOCATION OF A*/
   END;
   ELSE CALL EV; /*INVOKES B WITH
                   ENVIRONMENT OF FIRST
                   INVOCATION OF A*/
   B:PROC;
     GO TO OUT;
   END;
 OUT:END A;
```

The GO TO statement in the procedure B will transfer control to the END A; statement in the first invocation of A, and will thus terminate B and both invocations of A.

Prologues and Epilogues

Each time a block is activated, certain activities must be performed before control can reach the first executable statement in the block. This set of activities is called a prologue. Similarly, when a block is terminated, certain activities must be performed before control can be transferred out of the block; this set of activities is called an epilogue.

Prologues and epilogues are the responsibility of the compiler and not of the programmer. They are discussed here because knowledge of them may assist the programmer in improving the performance of his program.

Prologues

A prologue is code that is executed as the first step in the activation of a block. In general, activities performed by a prologue are as follows:

- Computing dimension bounds and string lengths for automatic and DEFINED variables.
- Allocating storage for automatic variables and initialization, if specified.
- Determining which currently active blocks are known to the procedure, so that the correct generations of automatic storage are accessible, and the correct on-units may be entered.
- Allocating storage for dummy arguments that may be passed from this block.

The prologue may need to evaluate expressions for initial values (including iteration factors), and for array bounds, string lengths, and area sizes.

For each block in the program, the optimizing compiler assigns these values in the following order:

1. Values that are independent of other declarations in the block. (Values may be inherited from an outer block.)
2. Values that are dependent on other declarations in the block. If a value depends on more than one other declaration in the block, correct initialization is not guaranteed. For example:

   ```
   DCL I INIT(10), J INIT(I), K INIT(J);
   ```

   Correct initialization of K is not guaranteed.

The checkout compiler has no restriction on the number of dependencies; it evaluates the expressions in the order required by the dependencies (provided the dependencies can be determined from inspection of the DECLARE statement alone.)

Note that declarations of data items must not be mutually interdependent. For example, the following declaration is invalid:

```
DCL A(B(1)), B(A(1));
```

Note that interdependency can occur with more than two data items. For example, the following declaration is also invalid:

```
DCL A(B(1)), B(C(1)), C(A(1));
```

Epilogues

An epilogue is code that is executed as the final step in the termination of a block. In general, the activities performed by an epilogue are as follows:

- Re-establishing the on-unit environment existing before the block was activated.
- Releasing storage for all automatic variables allocated in the block.

Chapter 7: Recognition of Names

A PL/I program consists of a collection of identifiers, constants, and special characters used as operators or delimiters. Identifiers themselves may be either keywords or names with a meaning specified by the programmer. The PL/I language is constructed so that the compiler can determine from context whether or not an identifier is a keyword, so there is no list of reserved words that must not be used for programmer-defined names[1]. Any identifier may be used as a name; the only restriction is that at any point in a program a name can have one and only one meaning. For example, the same name cannot be used for both a file and a floating-point variable.

Note: The above is true so long as the 60-character set is used. Certain identifiers of the 48-character set cannot be used as programmer-defined identifiers in a program written using the 48-character set; these identifiers are: GT, GE, NE, LT, NG, LE, NL, CAT, OR, AND, NOT, and PT.

It is not necessary, however, for a name to have the same meaning throughout a program. A name declared within a block has a meaning only within that block. Outside the block it is unknown unless the same name has also been declared in the outer block. In this case, the name in the outer block refers to a different data item. This enables programmers to specify local definitions and, hence, to write procedures or begin blocks without knowing all the names being used by other programmers writing other parts of the program.

Since it is possible for a name to have more than one meaning, it is important to define which part of the program a particular meaning applies to. In PL/I a name is given attributes and a meaning by a declaration (not necessarily explicit). The part of the program for which the meaning applies is called the scope of the declaration of that name. In most cases, the scope of a name is determined entirely by the position at which the name is declared within the program (or assumed to be declared if the declaration is not explicit). There are cases in which more than one generation of data may exist with the same name (such as in recursion); such cases are considered separately.

In order to understand the rules for the scope of a name, it is necessary to understand the terms "contained in" and "internal to."

Contained In:

> All of the text of a block, from the PROCEDURE or BEGIN statement through the corresponding END statement, is said to be contained in that block. Note, however, that the labels of the BEGIN or PROCEDURE statement heading the block, as well as the labels of any ENTRY statements that apply to the block, are not contained in that block. Nested blocks are contained in the block in which they appear.

Internal To:

> Text that is contained in a block, but not contained in any other block nested within it, is said to be internal to that block. Note that entry names of a procedure (and labels of a BEGIN statement) are not contained in that block. Consequently, they are internal to the containing block. Entry names of an external procedure are treated as if they were external to the external procedure.

In addition to these terms, the different types of declaration are important. The three different types -- explicit declaration, contextual declaration, and implicit declaration -- are discussed in the following sections.

Explicit Declaration

A name is explicitly declared if it appears:

1. In a DECLARE statement
2. In a parameter list
3. As a statement label
4. As a label of a PROCEDURE or ENTRY statement.

---

[1] Though the uses of the 48-character set composite symbols, and, under the checkout compiler, of the file SYSPRINT, are restricted.

The appearance of a name in a parameter list is the same as if a DECLARE statement for that name appeared immediately following the PROCEDURE or ENTRY statement in which the parameter list occurs (though the same name may also appear in a DECLARE statement internal to the same block).

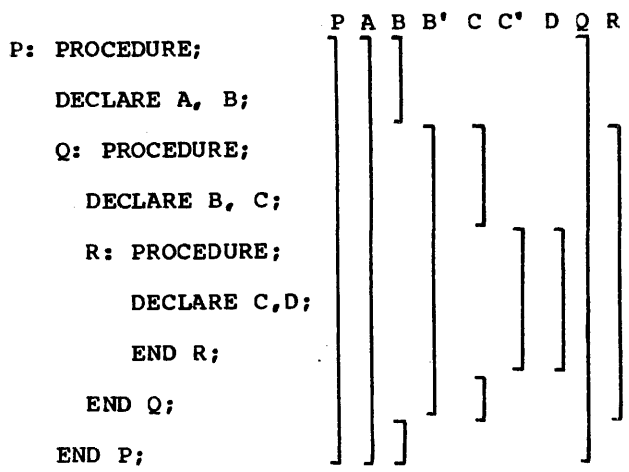
The appearance of a name as the label of either a PROCEDURE or ENTRY statement constitutes a declaration within the procedure containing the one to which it refers.

The appearance of a label prefix on a statement constitutes explicit declaration of the label.

SCOPE OF AN EXPLICIT DECLARATION

The scope of an explicit declaration of a name is that block to which the declaration is internal, including all contained blocks except those blocks (and any blocks contained within them) to which another explicit declaration of the same identifier is internal.

For example:



The lines to the right indicate the scope of the names. B and B' indicate the two distinct uses of the name B; C and C' indicate the two uses of the name C.

Contextual Declaration

When a name appears in certain contexts, some of its attributes can be determined without explicit declaration. In such a case, if the appearance of a name does not lie within the scope of an explicit declaration for the same name, the name is said to be contextually declared.

A name that has not been declared explicitly will be recognized and declared contextually in the following cases:

1. A name that appears in a CALL statement, in a CALL option, or followed by an argument list is given the BUILTIN and INTERNAL attributes. Built-in functions and pseudovariables without arguments, such as ONCHAR, ONSOURCE, DATE and DATAFIELD, should be declared explicitly with the BUILTIN attribute, contextually using a null argument list, for example, ONCHAR(), or implicitly by using a DEFAULT statement, for example,

   ```
   DEFAULT RANGE (ON, DAT) BUILTIN;
   ```

2. A name that appears in a FILE or COPY option, or a name that appears in an ON, SIGNAL, or REVERT statement for a condition that requires a file name, is given the FILE attribute.

3. A name that appears in an ON CONDITION, SIGNAL CONDITION, or REVERT CONDITION statement is recognized as a programmer-defined condition name.

4. A name that appears in an EVENT option or in a WAIT statement is given the EVENT attribute.

5. A name that appears in a TASK option is given the TASK attribute.

6. A name that appears in the BASED attribute, in a SET option, or on the left-hand side of a pointer qualification symbol is given the POINTER attribute.

7. A name that appears in an IN option, or in the OFFSET attribute, is given the AREA attribute.

Examples of contextual declaration are:

```
READ FILE (PREQ) INTO (Q);

ALLOCATE X IN (S);
```

In these statements, PREQ is given the FILE attribute, and S is given the AREA attribute.

SCOPE OF A CONTEXTUAL DECLARATION

The scope of a contextual declaration is determined as if the declaration were made in a DECLARE statement immediately

following the PROCEDURE statement of the external procedure in which the name appears.

Note that contextual declaration has the same effect as if the name were declared in the external procedure, even when the statement that causes the contextual declarations is internal to a block (called B, for example) that is contained in the external procedure. Consequently, the name is known throughout the entire external procedure, except for any blocks in which the name is explicitly declared. It is as if block B has inherited the declaration from the containing external procedure.

Since a contextual declaration cannot exist within the scope of an explicit declaration, it is impossible for the context of a name to add to the attributes established for that name in an explicit declaration. For example, the following procedure is invalid:

```
P:  PROC (F);
     .
     .
     .
    READ FILE(F) INTO(X);
     .
     .
     .
    END P;
```

The identifier F is in a parameter list and is, therefore, explicitly declared. The standard default attributes REAL DECIMAL FLOAT conflict with the attributes that would normally be given to F by its appearance in the FILE option. Such use of the identifier is in error.

Implicit Declaration

If a name appears in a program and is not explicitly or contextually declared, it is said to be implicitly declared. The scope of an implicit declaration is determined as if the name were declared in a DECLARE statement immediately following the PROCEDURE statement of the external procedure in which the name is used. A name used only in a contained procedure will be known in the containing procedure.

Unless the DEFAULT statement causes programmer-defined defaults to override the standard defaults, an implicit declaration causes standard default attributes to be applied, depending upon the first letter of the name. If the name begins with any of the letters I through N it is given the attributes REAL FIXED BINARY (15,0). If the name begins with any other letter including one of the alphabetic extenders $, #, or @, it is given the attributes REAL FLOAT DECIMAL (6).

Examples of Declarations

Scopes of data declarations are illustrated in figure 7.1. The brackets to the left indicate the block structure; the brackets to the right show the scope of each declaration of a name. In the diagram, the scopes of the two declarations of Q and R are shown as Q and Q' and R and R'.

P is declared in the block A and known throughout A since it is not redeclared.

Q is declared in A, and redeclared in B. The scope of the first declaration is all of A except B; the scope of the second declaration is block B only.

R is declared in block C, but a reference to R is also made in block B. The reference to R in block B results in an implicit declaration of R in A, the external procedure. Two separate names with different scopes exist, therefore. The scope of the explicitly declared R is C; the scope of the implicitly declared R is all of A except block C.

I is referred to in block C. This results in an implicit declaration in the external procedure A. As a result, this declaration applies to all of A, including the contained procedures B, C, and D.

S is explicitly declared in procedure D and is known only within D.

Scopes of entry constant and statement label declarations are illustrated in figure 7.2. The example shows two external procedures. The names of these procedures, A and E, are assumed to be explicitly declared with the EXTERNAL attribute within the procedures to which they apply. In addition, E is explicitly declared in A as an external entry constant. The explicit declaration of E applies throughout block A. It is not linked to the explicit declaration of E that applies throughout block E. The scope of the name E is all of block A and all of block E. The scope of the name A is only all of the block A, and not E.

However, it could appear in an external entry declaration in E, which would then result in the scope of A being all of A and all of E.

The label L1 appears with statements internal to A and to C. Two separate

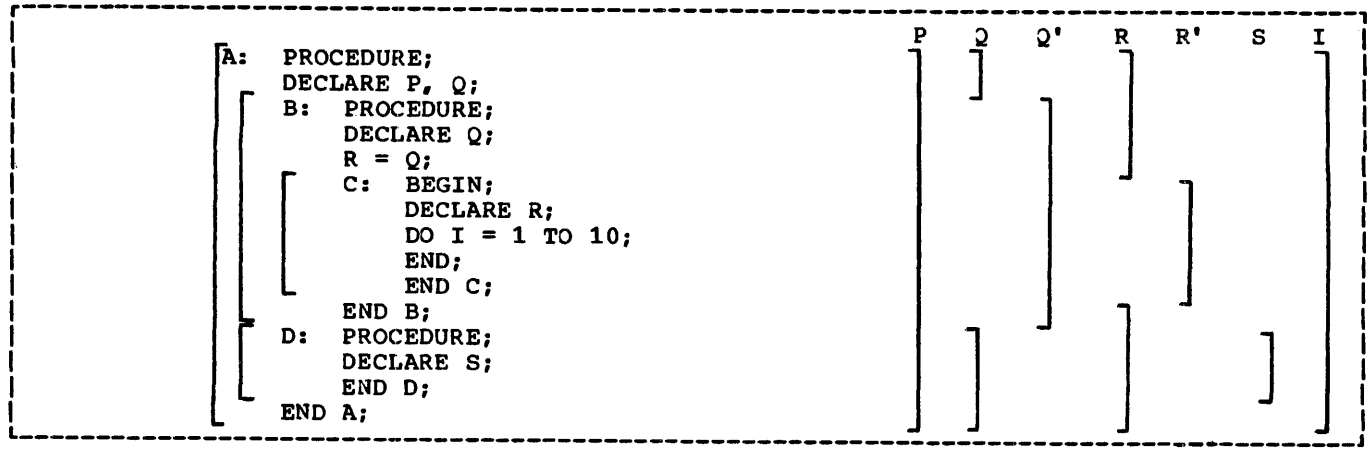


Figure 7.1. Scopes of data declarations

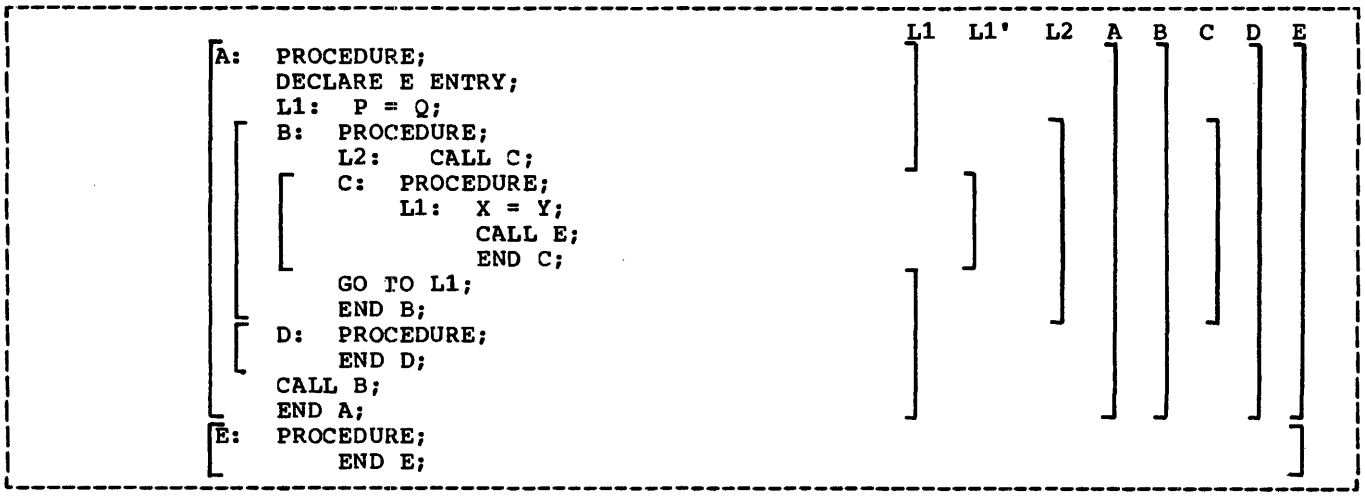


Figure 7.2. Scopes of entry and label declarations

declarations are therefore established; the first applies to all of block A except block C, the second applies to block C only. Therefore, when the GO TO statement in block B is executed, control is transferred to L1 in block A, and block B is terminated.

D and B are explicitly declared in block A and can be referred to anywhere within A; but since they are INTERNAL, they cannot be referred to in block E (unless passed as an argument to E).

C is explicitly declared in B and can be referred to from within B, but not from outside B.

L2 is declared in B and can be referred to in block B, including C, which is contained in B, but not from outside B.

Internal and External Attributes

The scope of a name with the INTERNAL attribute is the same as the scope of its declaration. Any other explicit declaration of that name refers to a new object with a different, non-overlapping scope.

A name with the EXTERNAL attribute may be declared more than once in the same program, either in different external procedures or within blocks contained in external procedures. Each declaration of the name establishes a scope. These declarations are linked together and, within a program, all declarations of the same identifier with the EXTERNAL attribute refer to the same name. The scope of the name is the sum of the scopes of all the

declarations of that name within the program.

Note: External names of PL/I data cannot be more than seven characters long and must not contain the _ (break) character.

Since these declarations all refer to the same thing, they must all result in the same set of attributes. It may be impossible for the compiler to check all declarations, particularly if the names are declared in different procedures, so care should be taken to ensure that different declarations of the same name with the EXTERNAL attribute do have matching attributes. The attribute listing, which is available as optional output from these compilers, helps to check the use of names. The following example illustrates the above points in a program:

```
A: PROCEDURE;
   DECLARE S CHARACTER (20);
   DCL SET ENTRY(FIXED DECIMAL(1)),
   OUT ENTRY(LABEL);
   CALL SET (3);
E: GET LIST (S,M,N);
   B:  BEGIN;
       DECLARE X(M,N), Y(M);
       GET LIST (X,Y);
       CALL C(X,Y);
       C:  PROCEDURE (P,Q);
           DECLARE P(*,*), Q(*),
             S BINARY FIXED EXTERNAL;
           S = 0;
           DO I = 1 TO M;
           IF SUM (P(I,*)) = Q(I)
              THEN GO TO B;
           S = S+1;
           IF S = 3 THEN CALL OUT (E);
           CALL D(I);
       B:  END;
           END C;
       D:  PROCEDURE (N);
           PUT LIST ('ERROR IN ROW ',
             N, 'TABLE NAME ', S);
           END D;
       END B;
   GO TO E;
   END A;

OUT:  PROCEDURE (R);
      DECLARE R LABEL,
        (M,L) STATIC INTERNAL
          INITIAL (0),
        S BINARY FIXED EXTERNAL,
        Z FIXED DECIMAL(1);
      M = M+1; S=0;
      IF M<L THEN STOP; ELSE GO TO R;
SET:  ENTRY (Z);
      L=Z;
      RETURN;
      END OUT;
```

A is an external procedure name; its scope is all of block A, plus any other blocks where A is declared as external.

S is explicitly declared in block A and block C. The character string declaration applies to all of block A except block C; the fixed binary declaration applies only within block C. Notice that although D is called from within block C, the reference to S in the PUT statement in D is to the character string S, and not to the S declared in block C.

N appears as a parameter in block D, but is also used outside the block. Its apearance as a parameter establishes an explicit declaration of N within D since there is no other declaration of N within D; the references outside D cause an implicit declaration of N in block A. These two declarations of the name N refer to different objects, although in this case, the objects have the same data attributes, which are, by standard default , FIXED (15,0), BINARY, and INTERNAL.

X and Y are known throughout B and could be referred to in block C or D within B, but not in that part of A outside B.

P and Q are parameters, and therefore if there were no other declaration of these names within the block, their appearance in the parameter list would be sufficient to constitute an explicit declaration. However, a separate DECLARE statement is required in order to specify that P and Q are arrays and it is this that is the explicit declaration. Note that although the arguments X and Y are declared as arrays and are known in block C, it is still necessary to declare P and Q in a DECLARE statement to establish that they, too, are arrays. (The asterisk notation indicates that the bounds of the parameters are the same as the bounds of the arguments.)

I and M are not explicitly declared in the external procedure A; they are therefore implicitly declared and are known throughout A, even though I appears only within block C.

The second external procedure in the example has two entry names, SET and OUT. These are considered to be explicitly declared with the ENTRY and EXTERNAL attributes. They must also be declared explicitly with the ENTRY attribute in procedure A. Since ENTRY implies EXTERNAL, the two entry constants SET and OUT are known throughout the two external procedures.

The label B appears twice in the program, once as the label of a begin block, which is an explicit declaration, as a label in A. It is redeclared as a label within block C by its appearance as a prefix to the END statement. The reference

to B in the GO TO statement within block C therefore refers to the label of the END statement within block C. Outside block C, any reference to B would be to the label of the begin block.

Note that C and D can be called from any point within B but not from that part of A outside B, nor from another external procedure. Similarly, since E is known throughout the external procedure A, a transfer to E may be made from any point within A. The label B within block C, however, can only be referred to from within C. Transfers out of a block by a GO TO statement can be made; but such transfers into a nested block generally cannot. An exception is shown in the external procedure OUT, where the label E from block A is passed as an argument to the label parameter R.

The statement GO TO R causes control to pass to the label E, even though E is declared within A, and not known within OUT.

The variables M and L are declared within the block OUT to be STATIC; their values are preserved between calls to OUT.

In order to identify the S in the procedure OUT as the same S in the procedure C, both have been declared with the attribute EXTERNAL.

Scope of Member Names of External Structures

When a major structure name is declared with the EXTERNAL attribute in more than one block, the attributes of the corresponding structure members must be the same in each case, although the corresponding member names need not be identical. Names of members of structures always have the INTERNAL attribute, and cannot be declared with any scope attribute. However, a reference to a member of an external structure, using the member name known to the block containing the reference, is effectively a reference to that member in all blocks in which the external name is known, regardless of whether the corresponding member names are identical. For example:

```
PROCA: PROCEDURE;
       DECLARE 1 A EXTERNAL,
                 2 B,
                 2 C;
       .
       .
       .
END PROCA;
```

```
PROCB: PROCEDURE;
       DECLARE 1 A EXTERNAL,
                 2 B,
                 2 D;
       .
       .
       .
END PROCB;
```

In this example, if A.B is changed in PROCA, it is also changed for PROCB, and vice versa; if A.C is changed in PROCA, A.D is changed for PROCB, and vice versa.

Multiple Declarations and Ambiguous References

Two or more declarations of the same identifier internal to the same block constitute a multiple declaration, unless at least one of the identifiers is declared within a structure in such a way that name qualification can be used to make the names unique.

Two or more declarations anywhere in a program of the same identifier as EXTERNAL names with different attributes constitute a multiple declaration.

Multiple declarations are in error.

A name need have only enough qualification to make the name unique. Reference to a name is always taken to apply to the identifier declared in the innermost block containing the reference. An ambiguous reference is a name with insufficient qualification to make the name unique.

The following examples illustrate both multiple declarations and ambiguous references:

```
DECLARE 1 A, 2 C, 2 D, 3 E;
   BEGIN;
   DECLARE 1 A, 2 B, 3 C, 3 E;
   A.C = D.E;
```

In this example, A.C refers to C in the inner block; D.E refers to E in the outer block.

```
DECLARE 1 A, 2 B, 2 B, 2 C, 3 D, 2 D;
```

In this example, B has been multiply declared. A.D refers to the second D, since A.D is a complete qualification of only the second D; the first D would have to be referred to as A.C.D.

```
DECLARE 1 A, 2 B, 3 C, 2 D, 3 C;
```

In this example, A.C is ambiguous because neither C is completely qualified by this reference.

```
DECLARE 1 A, 2 A, 3 A;
```

In this example, A refers to the first A, A.A refers to the second A, and A.A.A refers to the third A.

```
DECLARE X;

DECLARE 1 Y, 2 X, 3 Z, 3 A,
            2 Y, 3 Z, 3 A;
```

In this example, X refers to the first DECLARE statement. A reference to Y.Z is ambiguous; Y.Y.Z refers to the second Z; and Y.X.Z refers to the first Z.

Application of Default Attributes

Every identifier in a PL/I source program requires a complete set of attributes. However, the attributes specified in a DECLARE statement need rarely be the complete set of attributes for the identifier. Moreover, contextual declaration can result in only a partial declaration of an identifier. For each partially declared identifier the set of attributes is completed implicitly by the compiler by application of default rules.

Default rules which are determined for the implementations are termed standard default rules; alternative default rules can be defined by the programmer who wishes either to modify the standard default rules, or develop a completely new set of default rules. The DEFAULT statement is used for this purpose. Its use is described in a later section of this chapter.

PROCESSES IN THE APPLICATION OF ATTRIBUTES

Attribute processing by the compiler takes place in the following order:

1. Defactoring of attributes.
2. Application of the LIKE attribute.
3. Application of ALIGNED or UNALIGNED attributes to structure members.
4. Establishment and application of explicit declarations.
5. Establishment and application of contextual declarations.
6. Establishment of implicit declarations.
7. Application of attributes specified in the DEFAULT statements (if present), for explicitly, contextually, and implicitly declared identifiers; then application of standard default attributes.
8. Resolution of identical identifiers, including identifiers used in attributes, or declared in different blocks of a procedure.

From this it should be seen that attributes applied by default cannot override attributes of the same class applied to an identifier by explicit or contextual declaration. Further, any attributes applied by default are largely dependent on attributes already applied. This is fundamental to understanding the use of the DEFAULT statement.

APPLICATION OF STANDARD DEFAULTS

Standard default rules are applied for a class of attributes when an attribute of a particular class, such as scope, scale, base, or mode, etc., has not been applied either by explicit or contextual declaration. A summary of the standard defaults for file attributes appears in chapter 10, "Input and Output." A summary of standard default assumptions for both problem and program control data are given below. A complete description of standard default assumptions is given in section I, "Attributes."

Problem Data

If the problem data is not known to be either of character or of arithmetic type, arithmetic type is assumed.

Arithmetic Data: The standard defaults vary according to the information specified for the data:

1. If an arithmetic data item is partially specified in an explicit

declaration, the attributes assumed by default are:

Explicit declarations	Default attributes assumed
BINARY	REAL, FLOAT
DECIMAL	REAL, FLOAT
FIXED	REAL, DECIMAL
FLOAT	REAL, DECIMAL
REAL	FLOAT, DECIMAL
FIXED BINARY	REAL
FIXED DECIMAL	REAL
FLOAT BINARY	REAL
FLOAT DECIMAL	REAL
REAL FIXED	DECIMAL
REAL FLOAT	DECIMAL
REAL BINARY	FLOAT
REAL DECIMAL	FLOAT

Note that if COMPLEX is declared instead of REAL, the attributes are the same as for REAL, and are applied to each of the two parts.

2. If a base but not a scale is specified, the scale assumed depends on the presence of a scale factor in the precision attribute. If there is a scale factor, FIXED is assumed, if there is not, FLOAT is assumed.

   For example:

```
DCL A BINARY(5),
    B BINARY(5,2);
```

   The assumed attributes for A are REAL FLOAT: for B, they are REAL FIXED.

3. If mode, scale, and base are not specified by a DECLARE or DEFAULT statement, the attributes assumed depend on the initial letter of the identifier.

Initial letter	Default attributes assumed
$,#,&,A - H	REAL FLOAT DECIMAL
I - N	REAL FIXED BINARY
O - Z	REAL FLOAT DECIMAL

A value returned from a function reference can have default rules applied to determine its base, scale, and mode. Default attributes for a returned value are obtained by applying default rules to the function name as if it were an arithmetic identifier.

Precision of arithmetic data: Standard default precisions for arithmetic data are:

Attributes	Precision
FIXED BINARY	(15,0)
FIXED DECIMAL	(5,0)
FLOAT BINARY	(21)
FLOAT DECIMAL	(6)

Other attributes of arithmetic data: The assumed attributes are ALIGNED, and AUTOMATIC if INTERNAL, or STATIC if EXTERNAL.

String data: If the length of a character or bit string is undefined, a length of 1 is assumed. The attributes UNALIGNED, and AUTOMATIC if INTERNAL, or STATIC if EXTERNAL, are assumed.

Structures and structure members: Level-one structures are assumed AUTOMATIC if INTERNAL, and STATIC if EXTERNAL. Minor structures and structure members cannot be declared to have storage or scope attributes.

Arrays and data elements: UNALIGNED is assumed for data elements of string or picture type. ALIGNED is assumed for all other data types. Scope and storage depend on the data type.

Program Control Data Types

ENTRY: An entry constant declared in a DECLARE statement, or as a statement prefix on a PROCEDURE or ENTRY statement, is assumed EXTERNAL. An entry variable is assumed INTERNAL.

LABEL, POINTER, OFFSET, AREA, EVENT, TASK: Identifiers declared with any one of these attributes are assumed ALIGNED, and AUTOMATIC if INTERNAL, STATIC if EXTERNAL. If the size is not specified for an area variable, the default size of 1000 bytes is applied.

DEFAULT Statement

The function of the DEFAULT statement is to give the programmer control over the default attributes assigned to identifiers. The DEFAULT statement cannot be used to override the attributes assigned to identifiers by explicit or contextual declarations.

The DEFAULT statement can be used to modify the standard default rules or to specify a complete set of programmer-defined default rules. It can specify attributes for identifiers whose attribute sets are not complete after explicit, implicit, or contextual declaration, for the descriptors in entry declarations, and for the attributes in the RETURNS option of PROCEDURE and ENTRY statements. Standard default rules can be restored after programmer-defined default rules have been established in a program.

A simplified general form of the DEFAULT statement is as follows:

```
DEFAULT
{RANGE({identifier}|{letter:letter}|{*})}
{DESCRIPTORS                            }
 [attribute-specification];
```

RANGE Option: The RANGE option specifies the identifiers to which the associated default rules are to be applied. The range can be specified as either two letters separated by a colon, or as a single identifier. For example, the option:

```
RANGE (A:J)...
```

applies to all identifiers with initial letters in the range A through J. The option:

```
RANGE(ABC)...
```

applies to all identifiers with the initial three letters 'ABC' such as ABC, ABCD, and ABCDE.

The RANGE option can also be specified as:

```
RANGE (*)
```

whereby all possible initial alphabetic characters, from A through Z, and the characters $, @, and # are specified.

DESCRIPTORS Option: The DESCRIPTORS option specifies that the associated default rules are to be applied to non-null parameter descriptors.

Attribute Specification: The attribute specification is a list of attributes from which selected attributes are applied to identifiers in the specified range. Attributes in the list may appear in any order and must be separated by blanks.

Only those attributes that are necessary to complete the declaration of a data item are taken from the list of attributes. If the list does not supply all the required attributes, then standard default attributes are applied. Therefore, specification of any attribute that is a standard default is unnecessary. For example:

```
DEFAULT RANGE(T) POINTER;
```

This means that any identifier that begins with the letter T is a pointer. The complete list of attributes that apply to these identifiers is POINTER, AUTOMATIC, INTERNAL, and ALIGNED.

Attributes that conflict when applied to a data item do not necessarily conflict when they appear in an attribute specification. For example:

```
DEFAULT RANGE(S) BINARY VARYING;
```

This means that any identifier that begins with the letter S and is declared explicitly with the BIT or CHARACTER attribute will receive the VARYING attribute; all others (that are not declared explicitly or contextually as other than arithmetic data) will receive the BINARY attribute.

The VALUE option is used within the attribute specification to specify attributes that are represented by a decimal integer constant or an expression. These are the attributes length, size, and precision. For example:

```
DEFAULT RANGE(*) VALUE(AREA(2000));
```

This statement gives a default size of 2000 to all area variables. The dimension attribute can be specified directly in an attribute specification provided it appears first in the list.

Example 1:

Assume that the following ranges of initial letters are to correspond to the attributes given:

Initial letters	Attributes required
A - D	REAL FLOAT DECIMAL
E - H	REAL FLOAT BINARY
I - N	REAL FIXED BINARY
O - Z	REAL FIXED DECIMAL

The precisions to be assumed are the default precisions for these implementations. A DEFAULT statement to establish these additional default rules is:

```
DEFAULT RANGE(E:H) BINARY,
RANGE(O:Z)FIXED;
```

In this statement additional default rules for two ranges of initial letters are specified. The standard default rules for identifiers with initial letters outside the ranges E - H and O - Z are unchanged.

Example 2:

A DEFAULT statement can specify that all implicitly-declared data has the same attribute.

```
DEFAULT RANGE (*) PICTURE '99999';
```

This statement causes all implicitly-declared identifiers to be assumed numeric character type with the attributes REAL PICTURE '99999'.

If values other than the standard defaults are required, the argument of the VALUE option should always contain an attribute to qualify the precision, string length, or area size for a particular default attribute. For example:

a.
```
DEFAULT RANGE (S:T) CHARACTER
VALUE (CHARACTER (10));
```

b.
```
DEFAULT RANGE (*) VALUE (FIXED
BINARY(31),FLOAT DECIMAL(33),
FLOAT BINARY(109), FIXED
DECIMAL(15));
```

The first example specifies that all implicitly-declared identifiers with the initial letters S and T are to receive the default attribute CHARACTER and a default string length of ten characters. The second example specifies that all identifiers of arithmetic type with undefined precisions will have the precisions as defined in the argument to the keyword VALUE. (In this instance the precisions specified are the maximum precisions permitted.)

Note that the only attributes which the VALUE option can influence are precision, string length, and area size. Other attributes in the option, such as CHARACTER and FIXED BINARY in the above examples, merely indicate which attributes the value is to be associated with. Consider the following example.

```
DEFAULT RANGE(I) VALUE(FIXED
DECIMAL(8,3));

I = 1;
```

If it is not declared explicitly, I will be given the standard default attributes FIXED BINARY(15,0). It will not be influenced by the default statement, because this statement specifies only that the default precision for FIXED DECIMAL identifiers is to be (8,3).

Restoring Standard Defaults

The following statement:

```
DEFAULT RANGE(*), DESCRIPTORS;
```

overrides, for all identifiers, any programmer-defined default rules established in a containing block. It can be used to restore standard defaults for contained blocks.

To restore standard defaults to a particular identifier, the keyword SYSTEM can be specified in its DECLARE statement.

Scope of the DEFAULT Statement

The scope of a DEFAULT statement is the block in which it is specified, and any blocks contained in that block, except that if a DEFAULT statement in a contained block specifies all or part of the range specified in a DEFAULT statement in a containing block, the statement in the contained block overrides the other for the range that they have in common. For example:

```
A:  PROC;
   DEFAULT RANGE(A:I) FIXED BINARY;
     .
     .
     .
B:  PROC;
   DEFAULT RANGE(I) DECIMAL;
     .
     .
     .
   END A;
```

In procedure B, DECIMAL overrides BINARY for identifiers beginning with I, and FIXED is not inherited. Standard defaults will be applied for alignment, scope, storage class, mode, and precision.

A DEFAULT statement in an internal block affects only explicitly declared identifiers. This is because the scope of contextually and implicitly declared identifiers is determined as if their declaration were made in a DECLARE statement immediately following the PROCEDURE statement of the external procedure in which the name appears.

Factored Default Specification

A default specification can be factored. For example, the following statement:

```
DEFAULT (RANGE(A:C) FIXED, RANGE(D:F)
FLOAT) DECIMAL;
```

specifies that arithmetic identifiers with the initial letters A to C receive the attributes FIXED DECIMAL, and those with the initial letters D to F receive the attributes FLOAT DECIMAL.

Programmer-defined Defaults for Parameter Descriptors

The DEFAULT statement can be used to specify attributes for parameter descriptors. The keyword DESCRIPTORS designates the list of attributes which follows it as an attribute specification for parameter descriptors. For example:

```
DEFAULT DESCRIPTORS BINARY;
DCL X ENTRY (FIXED, FLOAT);
```

the attribute BINARY is added to each parameter descriptor in the list, producing the equivalent list:

```
(FIXED BINARY, FLOAT BINARY)
```

The DESCRIPTORS default attributes are not applied to parameters having null descriptors, that is, parameters for which no attributes are specified in the parameter descriptor, and whose attributes must therefore match those of the corresponding arguments.

Programmer-defined Default for the RETURNS Option

The default attributes of implicitly declared values returned from function procedures are dependent on the entry name used to invoke the procedure. The DEFAULT statement can be used to specify such attributes when the entry name, or the initial letter of the entry name, is specified in the DEFAULT statement.

For example, the following statements:

```
DEFAULT RANGE (X) FIXED BINARY;
X : PROC(Y);
```

would be interpreted as:

```
X : PROC(Y) RETURNS (FIXED BINARY);
```

Restrictions of the Use of the DEFAULT Statement

The DEFAULT statement must not specify the attributes ENTRY, ENVIRONMENT, RETURNS, LIKE, VARIABLE, or any file attributes other than FILE. It cannot be used to specify structuring, although structure elements can have defaults applied according to a RANGE specification.

Although the DEFAULT statement may specify the dimension attribute for identifiers that have not been declared explicitly, a subscripted identifier would be contextually declared with the attribute BUILTIN. Therefore the dimension attribute can be applied by default only to explicitly declared identifiers. For example:

```
DEFAULT RANGE (ARRAY) (10,10) FIXED
BINARY;
DCL ARRAY1, ARRAY2;
```

Both ARRAY1 and ARRAY2 are explicitly declared two-dimensional arrays of 100 elements, each with the attributes FIXED and BINARY.

Chapter 8: Storage Control

The purpose of this chapter is to describe how the PL/I programmer can control the allocation of storage. Allocation is the process of obtaining storage for a variable. A generation of a variable refers to a particular allocation of it. The four storage classes STATIC, AUTOMATIC, CONTROLLED, and BASED allow the programmer to exercise as much control as he requires for a particular program.

All variables require storage; this applies both to problem data, such as string and arithmetic, and to program control data such as label variables, entry variables, and file variables. The declaration of a variable must include a storage class attribute even if only by default. The name of a variable is effectively the address of the variable, and the attributes specified for a variable describe the amount of storage required and how it is to be interpreted. For example:

```
DECLARE X FIXED BINARY (31,0) AUTOMATIC;
```

The name X addresses a fullword, i.e., four bytes, that contains a value to be interpreted as a fixed-point binary integer. For static and automatic variables, this concept is not very important, but when considering controlled and, particularly, based variables it is relevant.

It should be understood that at no point in a PL/I program does the programmer have access to the absolute address of a variable within main storage, because the allocation of storage for variables is managed by the compiler. The programmer does not specify where in main storage the allocation is to be made. He can, however, specify where it is to be allocated relative to storage already allocated for instance by allocating based variables in an area variable.

The degree of storage control that can be exercised depends on the class of storage used.

Static Storage

Variables declared with the STATIC attribute are allocated prior to the execution of a program and remain allocated until the program terminates. The program has no control on the allocation of static variables during execution. Programs often need data that is used whenever the program is executed. For example, all arithmetic constants specified in a program are stored in a manner similar to variables declared STATIC. The difference is that constants cannot be changed during program execution whereas the values of static variables can. Although static variables can be declared at any point in a program, they are all allocated prior to execution. But it is important to note that static variables follow normal scope rules for the validity of references to them. For example:

```
A:PROC OPTIONS(MAIN);
    .
    .
    .
  B:PROC;

    DECLARE X STATIC INTERNAL;

    .
    .
    .
    END B;
    .

  END A;
```

Although the variable X is allocated throughout the program, it can be referenced only within procedure B or any block contained in B.

If static variables are initialized using the INITIAL attribute, the initial values must be specified as constants with the exception of pointer variables as noted below. And any specification of extents, for instance array bounds, must also be constants. Thus if static storage is used, it must be borne in mind that whatever allocation has been specified when the program was written will be retained throughout the execution of the program. Static storage should be used for all data that may be referred to by the programmer at any point in a program. A STATIC pointer or offset variable may be initialized only by using the NULL built-in function.

All other forms of storage allocation are dynamic, that is, the storage is obtained during the execution of the program. Because of this, the programmer can exert more control.

Automatic Storage

Automatic variables are allocated on entry to the block in which they have been declared. They can be reallocated many times during the execution of a program. The programmer controls their allocation by his design of the block structure of his program. For example:

```
A:PROC;
  .
  .
  .
  CALL B;

  B:PROC;
    DECLARE X,Y AUTO;
    .
    .
    .
    END B;

  .
  .
  .
  CALL B;
```

Each time procedure B is invoked, the variables X and Y are allocated storage, and when B terminates the storage is released; consequently, the values they contained are lost. The storage that has been freed is available for reallocation to other variables. Thus, whenever a block (procedure or begin) is active, storage is allocated for all variables declared automatic within that block, and whenever a block is inactive no storage is allocated for the automatic variables in that block. Only one allocation of a particular automatic variable can exist, except for those procedures that are called recursively or by more than one task.

Array bounds, string lengths, and area sizes for automatic variables can be specified as expressions. This means not only that storage can be allocated when it is required but also that the required amount of storage can be allocated. For example:

```
A:PROC;
  DECLARE N FIXED BIN;
  .
  .
  .
  B:PROC;
    DECLARE STR CHAR(N);
```

The character string STR will have a length defined by the value of the variable N that existed when procedure B was invoked. However, storage is conserved at the possible expense of speed of execution because of the extra operations required to evaluate such expressions.

EFFECT OF RECURSION ON AUTOMATIC VARIABLES

A procedure that can be invoked when it is already active in the same task is said to be recursive. The values of variables allocated in one activation of such a procedure must be protected from change by other activations. This is arranged by stacking the variables. A stack operates on a last-in first-out basis; the most recent generation of an automatic variable is the only one that can be referenced. Note that static variables are not affected by recursion. Thus they are useful for communication across recursive invocations. This also applies to automatic variables that are declared in a procedure that contains a recursive procedure and to controlled and based variables. For example:

```
A:PROC;
  DCL X;
  .
  .
  .
    B:PROC RECURSIVE;
    DCL Z,
        Y STATIC;

    CALL B;
    .
    .
    .
    END B;
END A;
```

A single generation of the variable X exists throughout invocations of procedure B. The variable Z will have a different generation for each invocation of procedure B. The variable Y can be referred to only in procedure B and will not be reallocated at each invocation. (The concept of stacking of variables is also of importance in the discussion of controlled variables.)

Controlled Storage

Variables declared as CONTROLLED are allocated only when they are specified in an ALLOCATE statement. The programmer has individual control over each controlled variable. Effectively, they are

independent of the program block structure, but not completely. The scope of a controlled variable, when declared internal, is the block in which it is declared and any contained blocks. The declaration of a controlled variable describes only how much storage will be required when the variable is allocated and how it is to be interpreted. For example:

```
A:PROC;
  DCL X CONTROLLED;
  .
  .
  .
  CALL B;
  .
  .
  .
    B:PROC;
 ALLOCATE X;
    .
    .
    .
    END B;
END A;
```

The variable X can be validly referred to within procedure B and that part of procedure A that follows the CALL statement. Any reference to the value of the variable before execution of the CALL statement is in error. Once a controlled variable has been allocated, it remains allocated either until a FREE statement that names the variable is encountered or until the end of the program. Note that the scope of a controlled variable may not be the whole program; this creates a situation analogous to that for the STATIC INTERNAL variable described under "Static Storage" earlier, i.e., it exists but cannot be referenced.

The FREE statement frees the storage allocated for a controlled variable. The storage can then be re-used for other allocations.

Generally, controlled variables are useful when large data aggregates with adjustable extents are required in a program. For example:

```
DCL A(M,N) CTL;
  .
  .
  .
GET LIST(M,N);
ALLOCATE A;
GET LIST(A);
  .
  .
  .
FREE A;
  .
  .
  .
```

This program sequence allocates the exact storage required depending on the input data and discards the data (and frees its storage) when no longer required. This method can be more efficient than the alternative of setting up a begin block, because no prologue or epilogue is required.

ALLOCATE STATEMENT FOR CONTROLLED VARIABLES

A controlled variable can be allocated only by an ALLOCATE statement. The general form of the ALLOCATE statement for controlled variables is:

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

The "identifier" is any variable that has the CONTROLLED attribute. It can be an element, array, or structure, but cannot be subscripted or qualified. Permitted attributes are those that specify dimensions, the length of strings, and the size of areas. (Areas are discussed later in this chapter but in this context they are simply variables whose storage is adjustable.) This enables the programmer to alter the amount of storage for a particular generation of a variable. These attributes are:

```
dimension
CHARACTER(length)
BIT(length)
AREA(size)
```

The dimension attribute can appear with any of the others. For example:

```
DCL X(20) CHAR(5) CONTROLLED;
  .
  .
  .
ALLOCATE X(25) CHAR(6);
  .
  .
  .
```

The attribute values specified in an ALLOCATE statement always override those given in the DECLARE statement for the same variable. However, the attributes themselves must agree. Thus the dimension attribute must specify the same number of dimensions. As in a DECLARE statement, element expressions can be used to specify bounds, lengths, and sizes.

The INITIAL attribute can also be specified in an ALLOCATE statement.

Initial values given in an ALLOCATE statement override those, if any, given in a DECLARE statement.

FREE STATEMENT FOR CONTROLLED VARIABLES

Storage for a controlled variable is freed, and therefore its value is lost, when a FREE statement is executed that names the variable. The form of the FREE statement is:

```
FREE identifier[,identifier]...;
```

The "identifier" has the same restrictions as in the ALLOCATE statement.

If the FREE statement names a variable that has not been allocated, no action is taken.

Implicit Freeing

If a controlled variable is to remain allocated until the end of a task, it need not be explicitly freed by a FREE statement. All controlled storage is automatically freed at the termination of the task in which it was allocated.

MULTIPLE GENERATIONS OF CONTROLLED VARIABLES

If storage for a controlled variable is reallocated before being freed the first generation is preserved, i.e., stacked. The second generation becomes the current generation; the first generation cannot be directly accessed until the current generation has been freed. This is similar to the process described for automatic variables in a recursive procedure. For controlled variables, however, stacking and unstacking of variables occur at ALLOCATE and FREE statements rather than at block boundaries and are independent of invocation of procedures within a task.

Although values of successive generations of a controlled variable are stacked, values can be obtained from the most recent generation to help create a new generation. If, in an ALLOCATE or DECLARE statement, a bound, length, or size is specified by an expression that contains references to the variable, the value is taken from the most recent previous generation. For example:

```
DCL X(20)FIXED BIN CTL;
  .
  .
  .
ALLOCATE X;
  .
  .
  .
ALLOCATE X(X(1));
```

In the first allocation of X the upper bound is specified by the DECLARE statement, i.e., 20. In the second allocation the upper bound is specified by the value of the first element of the first generation of X.

Asterisk Notation

If, in an ALLOCATE statement, dimensions, lengths, or sizes are indicated by asterisks, values are inherited from the most recent previous generation. For arrays, the asterisk must be used for every dimension of the array, not just one of them. For example:

```
DCL X(10,20) CHAR(5) CTL;
  .
  .
  .
ALLOCATE X;
  .
  .
  .
ALLOCATE X(10,10);
  .
  .
  .
ALLOCATE X(*,*);
```

In this example, the first generation of X has bounds (10,20); the second and third generations have bounds (10,10). The elements of each generation of X are all character strings of length five.

The asterisk notation can also be used in a DECLARE statement, but has a different meaning. For example:

```
DCL Y CHAR(*) CTL,
      N FIXED BIN;
      N=20;
        .
        .
        .
      ALLOCATE Y;
        .
        .
        .
      ALLOCATE Y CHAR(N);
```

This simply means that the length of the character string Y is to be taken from the

previous generation unless it is specified in an ALLOCATE statement, in which case Y is given the specified length. This allows the programmer to defer the specification of the string length until the actual allocation of storage.

CONTROLLED STRUCTURES

When a structure is controlled, any arrays, strings, or areas it contains can be adjustable. For this reason, it is permissible to describe the relative structuring in an ALLOCATE statement. For example:

```
DCL 1 A CTL,
      2 B(-10:10),
      2 C CHAR(*) VARYING;
      .
      .
      .
ALLOCATE 1 A,
           2 B,
           2 C CHAR(5);
      .
      .
      .
FREE A;
```

When the structure is allocated, A.B has the extent -10 to +10 and A.C is a VARYING character string with maximum length 5 and the value null. When the structure is freed, only the major structure name is given. All of a controlled structure must be freed or allocated; it is an error to attempt to obtain storage for part of a structure.

ALLOCATION BUILT-IN FUNCTION

Where the allocation and freeing of a variable depend on flow of control, it is useful to be able to determine if the variable has been allocated. The ALLOCATION built-in function returns a binary integer value indicating the number of generations that can be accessed in the current task for a given controlled variable. If the variable is not allocated, the value zero is returned. The function reference has the form:

ALLOCATION(a)

where a must be a controlled variable.

Besides the ALLOCATION built-in function, other built-in functions that may be useful are the array-handling functions DIM, which determines the extent of a specified dimension of an array, and LBOUND and HBOUND, which determine the lower and upper bound respectively of a specified dimension of a given array. Similarly for strings, the built-in function LENGTH, returns the current length of the string.

Based Storage

A based variable is fundamentally different from all other storage classes in that the name of a based variable does not identify the location of a generation in main storage; a declaration of a based variable is only a description of the generation, i.e., the amount of storage required and how that storage is to be interpreted. The location of the generation is identified by a separate variable called a locator variable. A locator variable is either a pointer variable or an offset variable. Offset variables are discussed later in this chapter in conjunction with area variables.

Although a declaration for a controlled variable is also only a description of the storage, once an ALLOCATE statement has been executed for the variable, its name also identifies the location of the variable. For this reason, it is impossible to refer to more than one generation of a controlled variable at a particular point in a program. In fact, the ALLOCATE statement can also be used for a based variable, but because the location of any generation is identified by an independent locator variable, it is possible to refer at any point in a program to any generation of a based variable by using an appropriate locator value.

BASED VARIABLES

A declaration of a based variable has the keyword BASED and, optionally, the name of a locator variable that can be assumed to be associated with the based variable. For example:

```
DCL X FIXED BIN BASED(P);
```

For this declaration the value of the variable P will identify the location of the variable X, except when the reference is otherwise explicitly qualified, as described below.

The association of a pointer variable in this way is not a special relationship. P can be used to identify locations of other

based variables and other locators can be used to identify other generations of the X.

LOCATOR QUALIFICATION

Because a reference to the value of a based variable consists of two parts, it is a qualified reference and to distinguish this from a reference to a member of a structure, it is called a locator-qualified reference. The composite symbol -> (a minus sign immediately followed by a greater than sign) represents 'qualified by' or 'points to'. For example:

```
P -> X
```

X must be a based variable and P must be a locator expression. The reference means: that generation of X identified by the value of the locator P. X is said to be explicitly locator-qualified.

When a based variable is associated with a locator variable in a declaration, the programmer need specify only the name of the based variable in a reference. For example:

```
DCL X FIXED BIN BASED(P);
  .
  .
  .
ALLOCATE X;
  .
  .
  .
X = X + 1;
```

The ALLOCATE statement sets a value in the pointer variable P so that the reference X applies to allocated storage. The references to X in the assignment statement are implicitly locator-qualified by P. References are explicitly locator-qualified as follows:

```
Q->X = Q->X + 1;
```

This assignment statement has the same effect as that of the previous example. A based variable can be declared without naming a pointer variable; in this case any reference to the based variable must always be explicitly locator-qualified.

(Note that PL/I allows a more general form of locator qualification than is described here; see "Multiple Locator Qualification" at the end of this chapter. However, the general form is not essential to an understanding of the remainder of this chapter.)

POINTER VARIABLES

A pointer variable is declared contextually if it appears in the declaration of a based variable, if it appears as a locator qualifier, or if it appears in the SET option of an ALLOCATE, LOCATE, or READ statement. It can also be declared explicitly as in the following example:

```
DCL Q POINTER;
```

Because Q is a variable it must have a storage class; in this case, AUTOMATIC is applied by default. Note that a pointer variable is a program control variable and therefore cannot be manipulated in the same way as arithmetic values. Pointer variables can be collected in arrays and structures.

Pointer Expression

A pointer expression is either a pointer variable, which can be qualified or subscripted, or a function reference that returns a pointer value.

A pointer expression can be used in the following ways:

1. As a locator qualifier, in association with a declaration of a based variable.
2. In a comparison operation, for example in a IF statement (pointer values can be compared whether equal or not equal).
3. As an argument in a procedure reference.

Setting Pointer Variables

Before a reference is made to a pointer-qualified variable, the pointer must have a value. A pointer value is obtained from any of the following:

1. The NULL built-in function.
2. The ADDR built-in function.
3. A READ or LOCATE statement.
4. An ALLOCATE statement.

All pointer values are originally derived from one of these three methods. Such values can then be manipulated by

assignment that copies a pointer value to a pointer variable; by locator conversion that converts an offset value to a pointer value, or vice versa; by passing the pointer value as an argument in a procedure reference; and by returning a pointer value from a function procedure.

ADDR BUILT-IN FUNCTION

The ADDR built-in function returns a pointer value that identifies the first byte of a variable. The variable can have any data type or organization and any storage class. For example:

```
P = ADDR(X);
```

where P is a pointer variable and X is any connected variable. The argument to the built-in function can be a subscripted qualified reference. For example:

```
DCL A(3,2) CHARACTER(5) BASED(P),
    C(3,2) CHARACTER(5);
    .
    .
    .
P = ADDR(C);
Q = ADDR(A(2,1));
```

In this example, the arrays A and C refer to the same storage. The elements B and C(2,1) also refer to the same storage.

Notice that when a based variable is overlaid in this way no new storage is allocated - the based variable uses the same storage as the variable on which it is overlaid (A(3,2) in the example).

This overlay technique can be achieved by use of the DEFINED attribute, but an important difference is that for DEFINED the overlay is permanent. When based variables are overlaid, the association can be changed at any time in the program by assigning a new value to the pointer variable. Note that although PL/I does not permit the overlay of variables with different data types, e.g., overlaying an integer with a bit string, it is possible in this implementation. However, it should be understood that incompatibilities between the attributes of the based variable and the attributes of the variable being overlaid will be detected only when running under the checkout compiler with the NOCOMPATIBLE option.

The ADDR built-in function does not supply any information on the organization of a variable. Therefore, if the variable is an aggregate, it should be in connected storage if it is to be referenced as an entity. For example, if the variable is a cross-section of an array, the elements must not be interleaved. Furthermore, in this implementation, if the variable is a varying-length string or an area, control information is an integral part of the variable. A varying-length string is prefixed by a two-byte length field, and an area is prefixed by 16 bytes of control information. Thus if the ADDR function is performed on these types of variable, the pointer value identifies the start of the control information.

Other rules that apply to the use of the ADDR function are given in section G, "Built-in Functions".

BASED VARIABLES AND INPUT/OUTPUT

Based variables can be transmitted using either stream-oriented or record-oriented transmission.

In the list-directed form of stream-oriented transmission, provided the based variables are locator-qualified (implicitly or explicitly), they are treated in the same way as other types of variable. For example:

```
GET LIST (P->X);
```

For data-directed transmission, however, only a based variable that has been associated with a locator expression in a declaration can be transmitted. For example:

```
DCL Y BASED(Q), Z BASED;
.
.
.
PUT DATA(Y);
```

The variable Z cannot be transmitted in a PUT DATA or GET DATA (that is, data-directed I/O) statement. Chapter 11 discusses the techniques and facilities of stream-oriented transmission.

Record-oriented transmission provides two processing modes: move mode, which moves data into or out of an allocated generation of a variable either directly or indirectly via a buffer; or, locate mode, which only moves the data into or out of a buffer and identifies the storage allocated within the buffer. Although based variables can be transmitted using either mode, they are designed to be used with locate mode. Based variables are used in locate mode to describe the contents of a buffer, and therefore allow data to be processed while it is in the buffer. Note

that locate mode only applies to SEQUENTIAL BUFFERED files. Chapter 12, "Record-Oriented Transmission," discusses the two modes more fully.

READ with SET Statement

In locate mode, the READ statement has the form:

```
READ FILE(file-expression)
SET(element-pointer-variable);
```

This statement places a record in a buffer and identifies its location by setting the specified pointer variable. Any based variable qualified by this pointer variable describes the contents of the buffer. For example:

```
DCL X CHAR(20) BASED(P),
    Y(20) CHAR(1) BASED(P);
          .
          .
          .
READ FILE(IN) SET(P);
          .
          .
          .
```

In this program segment, a record is read into a buffer and the pointer variable P identifies its location. The record in the buffer is treated simultaneously by the based variable X as a fixed-length character string and by the based variable Y as an array of single characters. Note that P is declared contextually as a pointer variable and that a reference to X or Y is implicitly qualified by P.

The next I/O operation on the file (including closing the file) frees the buffer.

LOCATE Statement

The LOCATE statement complements the READ with SET statement and is used for output from a buffer. The form is:

```
LOCATE based-variable
FILE(file-expression)
 [SET (element-pointer-variable)];
```

This statement allocates storage in a buffer for a specified based variable. The SET option need only be specified if the based variable has not been associated with a pointer variable in a declaration.

The LOCATE statement operates differently from all other transmission statements. Because the statement sets a pointer to a storage address, there is nothing to transmit until values have been assigned to that storage. The LOCATE statement transmits the previous record (i.e., the contents of storage obtained by a previous LOCATE statement), frees the storage for that record, and allocates storage for the next record. The current record is also transmitted if a WRITE or CLOSE statement is executed for the same file. The following example shows the use of the LOCATE statement:

```
    DCL 1 STR BASED(P),
          2 NAME CHAR(20),
          2 RATE FIXED(5,2);
          .
          .
          .
OUTPUT:LOCATE STR FILE(OUT);
          .
          .
          .
    /*ASSIGN VALUES TO STR*/
          .
          .
          .
    GO TO OUTPUT;
```

By using locate mode the programmer can specify that a number of different forms of record be held in the same file. For example:

```
    DCL 1 STR1 BASED(P),
          2 CODE CHAR(1),
          2 X CHAR(30),
        1 STR2 BASED(Q),
          2 CODE CHAR(1),
          2 X(8) FIXED BIN;
          .
          .
          .
    READ FILE(IN) SET(P);
    IF STR1.CODE= '2' THEN DO;
       Q=P;
       I=Q->X(1);
    END;
```

In this program segment each based structure has an element CODE that identifies the structure. A record is read and its location is set in P. Depending on the value of CODE, the value of P is assigned to Q so that the record can be interpreted as STR2.

If an element varying-length string is transmitted using locate mode, the SCALARVARYING option of the ENVIRONMENT attribute must be specified for the file (see chapter 12, "Record-Oriented Transmission"). The records will include a two-byte length prefix.

SELF-DEFINING DATA(REFER OPTION)

A self-defining record is one which contains information about its own fields, such as the length of a string. A based structure can be declared so that such data can be manipulated. String lengths, array bounds, and area sizes can all be defined by variables declared within the structure. When the structure is allocated (by either an ALLOCATE statement or a LOCATE statement), the value of an expression is assigned to a variable that defines a length, bound, or size. For any other reference to the structure, the value of the defining variable is used.

The REFER option is used in the declaration of a based structure to specify that, on allocation of the structure, the value of an expression is to be assigned to a variable in the structure and is to represent the length, bound, or size of another variable in the structure. The REFER option has the following general format:

```
element-expression REFER
 (element-variable)
```

The value of the element-expression must be capable of being converted to an integer. Any variables used as operands in the expression must not belong to the structure containing the REFER option.

The element-variable, known as the object of the REFER option, must be the name of a member of the structure being declared. It must not be locator-qualified or subscripted and it must precede the member it defines. For example:

```
DECLARE 1 STR BASED(P),
          2 X FIXED BINARY,
          2 Y (L REFER (X)),
        L FIXED BINARY INITIAL(1000);
```

This declaration specifies that the based structure STR will consist of an array Y and an element X. When STR is allocated, the upper bound is set to the current value of L which is assigned to X. For any other reference to Y, such as a READ statement that sets P, the bound value is taken from X.

Any number of REFER options may be used in the declaration of a structure provided that at least one of the following restrictions is satisfied:

1. All objects of REFER options are declared at logical level two, that is, not declared within a minor structure. For example:

```
DECLARE 1 STR BASED,
          2 (M,N),
          2 ARR(I REFER (M),
            J REFER(N)),
          2 X;
```

When this structure is allocated, the values assigned to I and J will set the bounds of the two-dimensional array ARR.

2. The structure is declared so that no padding between members of the structure can occur. Section K, "Data Mapping," describes the rules by which structures are mapped. For example:

```
DECLARE 1 STR UNALIGNED BASED (P),
          2 B FIXED BINARY,
          2 C,
            3 D FLOAT DECIMAL,
            3 E (I REFER (D))
              CHAR(J REFER (B)),
          2 G FIXED DECIMAL;
```

Because this structure has the UNALIGNED attribute, all items require only byte alignment. Therefore regardless of the values of B and D (the REFER objects) no padding will occur. Note that D is declared within a minor structure.

3. If the REFER option is used only once in a structure declaration, restrictions 1 and 2 can be ignored provided that:

   a. For a string length or area size, the option is applied to the last element of the structure.

   b. For an array bound, the option is applied either to the last element of the structure or to a minor structure that contains the last element. The array bound must be the upper bound of the leading dimension. For example:

```
DCL 1 STR BASED (P),
      2 X FIXED BINARY,
      2 Y,
        3 Z FLOAT DECIMAL,
        3 M FIXED DECIMAL,
      2 D (L REFER (M)),
        3 E (50),
        3 F (20);
```

Note that the leading dimension of an array can be inherited from a higher level. For example, if we had declared STR(4) in the above example, the leading dimension would have been inherited from STR(4) and so it would not have been possible to use the REFER option in D.

This declaration does not satisfy restrictions 1 or 2; the REFER object M is declared within a minor structure and padding will occur. However, restriction 3 is satisfied as the REFER option is applied to a minor structure that contains the last element.

If the value of the object of a REFER option varies during the program then:

1. The structure must not be freed until the object is restored to the value it had when allocated.

2. The structure must not be written out while the object has a value greater than the value with which it was allocated.

3. The structure may be written out when the object has a value equal to or less than the value it has when allocated. The number of elements, the string length, or area size actually written will be that indicated by the current value of the object. For example:

```
DCL 1 REC BASED (P),
      2 N,
      2 A (M REFER(N)),
    M INITIAL (100);
        .
        .
        .
ALLOCATE REC;

N = 86;

WRITE FILE (X) FROM (REC);
```

In this example, 86 elements of REC are written. It would be an error to attempt to free REC at this point since N must be restored to the value it has when allocated (i.e., 100). If N was assigned a value greater than 100, an error would occur when the WRITE statement was encountered.

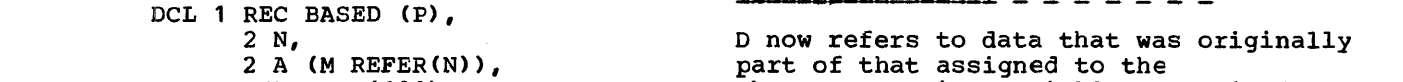
When the value of a refer object has been changed, the next reference to the structure causes remapping. For example:

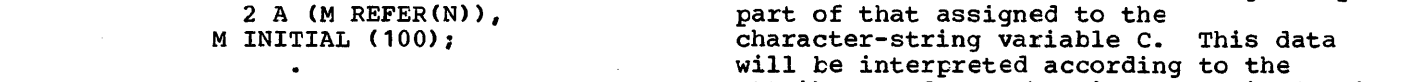
```
DCL 1 A BASED(P),
      2 B,
      2 C (I REFER(B)),
      2 D,
I INIT(10);
ALLOCATE A;
   .
   .
   .
B = 5;
```

The next reference to A after the assignment to B will cause the structure to be remapped to reduce the upper bound of C from 10 to 5, and to allocate to D storage immediately following the new last element of C. Although the structure is remapped, no data is reassigned - the contents of the part of storage originally occupied by the structure A are unchanged. If the programmer does not take account of remapping, errors can occur. Consider the following example, in which there are two REFER options in the one structure:

```
DCL 1 A BASED (P),
      2 B FIXED BINARY (15,0),
      2 C CHAR (I1 REFER (B)),
      2 D FIXED BINARY (15,0),
      2 E CHAR (I2 REFER (D)),
(I1,I2) INIT (10);
ALLOCATE A;
   .
   .
   .
B = 5;
```

The mapping of A with the original and new values of B is as follows:





D now refers to data that was originally part of that assigned to the character-string variable C. This data will be interpreted according to the attributes of D - that is, as a fixed-point decimal number - and the value obtained will be taken to be the length of E. Hence, the length of E is unpredictable.

LIST PROCESSING

List processing is the name for a number of techniques to help manipulate collections of data. Although arrays and structures in PL/I are also used for manipulating collections of data, list processing techniques are more flexible in that they allow collections of data to be indefinitely reordered and extended during program execution. It is not the purpose here to illustrate these techniques but simply to show how based variables and locator variables serve as a basis for this type of processing.

A list that has at least one pointer within each member that identifies the location of another member in the list is called a chained or threaded list. The primary application of the ALLOCATE and FREE statements is to build these lists.

ALLOCATE STATEMENT FOR BASED VARIABLES

The form of the ALLOCATE statement is:

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

The based variable can be any data type or organization. The SET option is needed if the based variable was declared without an associated pointer variable or if it is required to leave the pointer that was declared with the based variable unchanged, and to set a different pointer to the generation of the based variable that is being allocated.

Both based and controlled variables can be allocated in the same statement.

FREE STATEMENT FOR BASED VARIABLES

The form of the FREE statement is:

```
FREE[locator-qualifier->]
based-variable [IN(area-variable)]
[,[locator-qualifier->]
based-variable [IN(area-variable)]]...;
```

A particular generation of a based variable is freed by specifying a pointer qualifier in the statement. If a qualifier is omitted, the pointer variable associated with the based variable in its declaration is used; it is an error in this case if a pointer variable has not been associated with the based variable.

A FREE statement cannot be used to free a locate-mode I/O buffer.

Both based and controlled variables can be freed in the same statement.

MULTIPLE GENERATIONS OF BASED VARIABLES

All current generations of a based variable can be referred to by specifying appropriate pointer variables. In list processing, a number of based variables with many generations can be included in a list. Members of the list are chained together by a pointer in one member identifying the location of another member. Note that the allocation of a based variable cannot specify where in main storage the variable is to be allocated. In practice a chain of items may be scattered throughout main storage. But by accessing each pointer the next member is found. A member of a list is usually a structure that includes a pointer variable. For example:

```
    DCL 1 STR BASED(H),
          2 P POINTER,
          2 DATA,
        T  POINTER;
          .
          .
          .
          ALLOCATE STR;
          T=H;
          .
          .
          .
NEXT:ALLOCATE STR SET(T->P);
          T=T->P;
          .
          .
          .
          GO TO NEXT;
```

In this program segment, a list of structures is created. The structures are generations of STR and are linked by the pointer variable P in each generation. The independent pointer variable T identifies the previous generation during the creation of the list. The first ALLOCATE statement sets the pointer H to identify it. Ultimately the pointer H identifies the start, or head, of the list. The second ALLOCATE statement sets the pointer P in the previous generation to identify the location of this new generation. The assignment statement T=T->P; updates pointer T to identify the location of the new generation.

Figure 8.1 shows a diagrammatic representation of a one-directional chain.

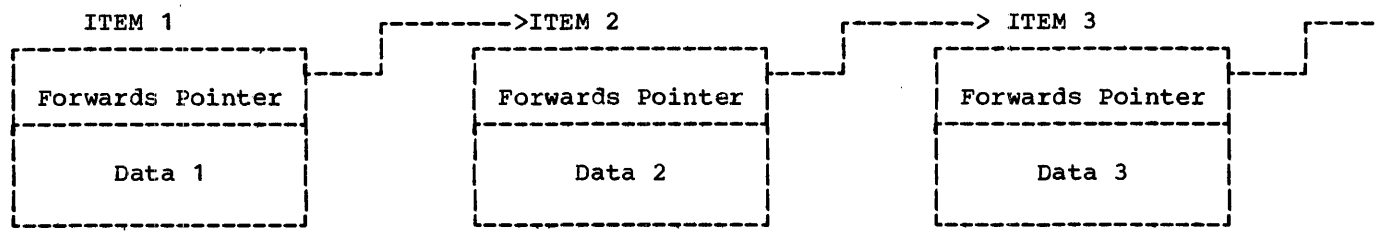


Figure 8.1. Example of one-directional chain

Note that, unless the value of P in each generation is assigned to a separate pointer variable for each generation, the generations of STR can be accessed only in the order in which the list was created. For the above example, the following statements can be used to access each generation in turn:

```
      T=H;
NXT:T->DATA=X;
      .
      .
      .
      T=T->P;
      GO TO NXT;
```

NULL BUILT-IN FUNCTION

When a list is created in the way described, it is necessary to indicate the end of the list. The NULL built-in function returns a pointer value that cannot identify a location in storage. Thus by setting the pointer in the last generation in a list to the value of NULL a positive indication of the end of the list is given. For example:

```
    T=H;
NXT:  IF T->P¬=NULL THEN
                DO;
                T->DATA=X;
                .
                .
                .
                T=T->P;
                GO TO NXT;
                END;
```

This program segment can be used instead of the previous example to scan the list; it is assumed that the pointer P in the final generation of STR has been set to the value of NULL.

In general, the value of a NULL built-in function is used whenever a pointer (or offset) variable should not identify a location in storage. Note that the only way a pointer can acquire the null value is by assignment of the NULL built-in function (apart from one special case, namely the assignment of the value returned by the ADDR built-in function when passed an unallocated controlled variable). The value of a pointer variable that no longer identifies a generation of a based variable, for example, when a based variable has been freed, is undefined.

TYPES OF LIST

The foregoing examples showed a simple list processing technique, the creation of a unidirectional list. More complex lists can be formed by adding other pointer variables into the structure. If a second pointer were added, it could be made to point to the previous generation. The list would then be bidirectional; from any item in the list, the previous and next items could be accessed by using the appropriate pointer value. Instead of the last pointer value being set to the value of NULL, it can be set to point to the first item in the list, thus creating a ring or circular list.

A list need not consist only of generations of a single based variable. Generations of different based structures can be included in a list by setting the appropriate pointer values. Items can be added and deleted from a list by manipulating the values of pointers. A list can be restructured by manipulating the pointers, so that the processing of data in the list may be simplified.

By reducing the amount of movement of data within main storage, the programmer can generally achieve a considerable saving on processing time. Note, however, that each pointer requires four bytes of storage and any allocated based variable requires at least eight bytes of storage, even if it is a bit string of length one.

AREAS

When a based variable is allocated, the storage is obtained from wherever it is available. Consequently, a list of allocated based variables could be scattered widely throughout main storage. For internal operations on the list, this is not significant, because items are readily accessed using the pointers. However, if the list is to be transmitted to a data set, the items would have to be collected together. Items allocated within an area variable are already collected and can be transmitted or assigned as a unit while still retaining their separate identities.

It is desirable to identify the locations of based variables within an area variable relative to the start of the area variable. Offset variables are defined for this purpose. If pointer variables were used they would be unlikely to be valid when the area variable were transmitted back to main storage.

Area Variables

The AREA attribute defines an area of storage that is to be reserved for the allocation of based variables. The declaration of an area variable has the form:

```
DCL identifier AREA [(size)];
```

The amount of storage to be reserved is given in bytes; i.e. the integral value of "size". If size is not given, a default of 1000 bytes is assumed.

The size of an area is adjustable in the same way as a string length or an array bound and therefore it can be specified by an expression or an asterisk (for a controlled area or parameter) or by a REFER option (for a based area). The maximum size of an area is limited only by the amount of main storage available to the program.

In addition to the declared size, an extra 16 bytes of control information, which contains such details as the amount of storage in use, precedes the reserved size of an area.

The amount of reserved storage that is actually in use is known as the extent of the area. The maximum extent is represented by the area size. Based variables can be allocated and freed within an area at any time during execution. This means that the extent of an area varies as storage is used. Because any based variable can be allocated within an area, they could require different amounts of storage. When a based variable is freed, the storage it occupied is marked as available for other allocations. In fact the implementation maintains a chain of available storage within an area; the head of the chain is held within the 16 bytes of control information. Inevitably, as based variables with different storage requirements are allocated and freed, gaps will occur in the area when allocations do not fit available spaces. Thus the extent of an area may contain allocations that have been freed but are still significant. A significant allocation is one that has not been freed or that has been freed but has at least one unfreed allocation following it. When an area has no significant allocations, the extent is zero.

Note that based variables are always allocated in multiples of eight bytes.

No operators, not even comparison, can be applied to area variables.

Offset Variables

Offset variables are a special form of pointer used exclusively with area variables. The value of an offset variable indicates the location of a based variable within an area variable relative to the start of the area. Because the based variables are identified relatively, if the area variable is assigned to a different part of main storage, the offset values are not invalidated. Note that offset variables do not preclude the use of pointer variables within an area. An offset variable is declared as follows:

```
    DCL identifier
OFFSET[(element-area-variable)];
```

The association of an area variable with an offset variable is not a special relationship; an offset variable can be associated with any area variable by means of the POINTER built-in function (see "Locator Conversion" below). The advantage of making such an association in declaration is that a reference to the offset variable implies reference to the associated area variable.

Note that the appearance of an area variable in the declaration of an offset is a contextual declaration of the area variable.

Locator Conversion

When an offset variable is used in a reference, it is implicitly converted to a pointer value; the address value of an associated area variable is added to the offset value. Explicit conversion of an offset to a pointer value is accomplished using the POINTER built-in function. For example:

```
DCL P POINTER, O OFFSET(A),B AREA;
          .
          .
          .
P = POINTER(O,B);
```

This statement assigns a pointer value to P, giving the location of a based variable, identified by offset O in area B. Because the area variable is different from that associated with the offset variable, the programmer must ensure that the offset value is valid for the different area. It would be valid, for example, if area A had been assigned to area B prior to the invocation of the function.

The OFFSET built-in function complements the POINTER built-in function and returns an offset value derived from a given pointer and area. The given pointer value must identify the location of a based variable in the given area.

In practice, these functions need rarely be used as most conversions are carried out implicitly.

Offset Expressions

Because an offset is implicitly converted to a pointer value, offset expressions can be used interchangeably with pointer expressions. An offset expression can be used as a locator qualifier, in association with a declaration of a based variable, in a comparison operation, or as an argument in a procedure reference. Note, however, that an offset variable cannot be specified in the SET option of a READ or LOCATE statement.

ALLOCATE Statement with the IN Option

An offset value is originally obtained either by conversion of a pointer value or by the SET option of the ALLOCATE statement. This form of the ALLOCATE statement is as follows:

```
ALLOCATE based-variable
       [IN(element-area-variable)]
       [SET(locator-variable)];
```

This statement allocates storage for a based variable within the specified area.

The variable has an offset relative to the start of the area, and this offset value is assigned to the locator variable specified in the SET option. Conversion takes place if the locator variable is of pointer type. Either or both of the options IN and SET can be implied. For example:

```
DCL X BASED(O),
    Y BASED(P),
    A AREA,
    O OFFSET(A);
    .
    .
    .
    ALLOCATE X;
    ALLOCATE Y IN(A);
```

The storage class of area A and offset O is AUTOMATIC by default. The first ALLOCATE statement is equivalent to:

```
ALLOCATE X IN(A) SET(O);
```

The second ALLOCATE statement is equivalent to:

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

The programmer must ensure that all implications can be resolved. If, for example, the offset O had not been associated with the based variable X, the SET option would be required.

When the IN and SET options are specified rather than implied, it is permissible to use an offset variable that has been declared with no associated area. The area in the SET option may also be

different from the one in the DECLARE statement, provided it is contained within that area. For example:

```
DCL 01 OFFSET(A1),
    02 OFFSET,
    A2 AREA BASED(P);
ALLOCATE A2 IN(A1) SET(P);
    .
    .
    .
ALLOCATE X IN(A2) SET(01);
ALLOCATE Y IN(A2) SET(02);
```

The offset variables 01 and 02 have the values of the offsets of the variables X and Y, in, respectively, the areas A1 and A2.

The following example shows how a list can be built in an area variable using offset variables. This example is a rewrite of the example given in "Multiple Generations of Based Variables" earlier in this chapter.

```
DCL A AREA,
    (T,H) OFFSET(A),
    1 STR BASED(H),
      2 P OFFSET(A),
      2 DATA;
    .
    .
    .
    ALLOCATE STR IN(A);
    T=H;
    .
    .
    .
NEXT:ALLOCATE STR SET(T->P);
    T=T->P;
    .
    .
    .
    GO TO NEXT;
```

FREE Statement with the IN Option

A based variable allocated within an area variable can be freed by specifying the area variable by the IN option:

```
FREE based-variable
    [IN(element-area-variable)];
```

Multiple freeing of both based and controlled variables can be made by the same FREE statement. When all the current allocations of variables within an area variable are to be freed, the EMPTY built-in function is the most convenient method.

EMPTY Built-in Function

When an area variable is allocated, it automatically has the empty state, i.e., the area extent is zero. The value of the EMPTY built-in function can be assigned to an area variable to free all allocations in the variable. The function reference does not require arguments but must be given a null argument list if the name has not been declared BUILTIN. For example:

```
DECLARE A AREA,
        I BASED (P),
        J BASED (Q);
          .
          .
          .
ALLOCATE I IN(A), J IN (A);
          .
          .
          .
A = EMPTY();
/*EQUIVALENT TO:
  FREE I IN (A), J IN (A); */
```

Note that the area variable itself is not freed, its storage is retained for further allocations of based variables.

AREA ASSIGNMENT

The value of an area expression can be assigned to one or more area variables by an assignment statement. Area-to-area assignment has the effect of freeing all allocations in the target area and then assigning the extent of the source area to the target area, in such a way that all offsets for the source area are valid for the target area. For example:

```
DECLARE X BASED (O(1)),
        O(2) OFFSET (A),
        (A,B) AREA;
          .
          .
          .
        ALLOCATE X IN (A);
        X = 1;
        ALLOCATE X IN (A) SET (O(2));
        O(2) -> X = 2;
        B = A;
```

Given this program segment and using the POINTER built-in function, the references POINTER (O(2),B)->X and O(2)->X will represent the same value allocated in areas B and A respectively.

If a source area containing no allocations is assigned to a target area, the effect is merely to free all allocations in the target area.

A possible use for area assignment is to allow for expansion of a list of based variables beyond the bounds of its original area. When an attempt is made to allocate a based variable within an area that contains insufficient free storage to accommodate it, the AREA condition is raised (see below). The on-unit for this condition could be to change the value of a pointer qualifying the reference to the inadequate area, so that it pointed to a different area; on return from the on-unit, the allocation would be attempted again, within the new area. Alternatively, the on-unit could write out the area and reset it to EMPTY.

AREA ON-condition

The AREA condition is raised in any of the following circumstances:

1. When an attempt is made to allocate a based variable within an area that contains insufficient free storage for the allocation to be made.

2. When an attempt is made to perform an area assignment, and the target area is too small to accommodate the extent of the source area.

3. When a SIGNAL AREA statement is executed.

The ONCODE built-in function can be used to determine whether the condition was raised by an allocation, an assignment, or a SIGNAL statement. On normal return from the on-unit, the action is as follows:

1. If the condition was raised by an allocation, the allocation is re-attempted. If the on-unit has changed the value of a pointer qualifying the reference to the inadequate area so that it points to another area, the allocation is re-attempted within the new area. Note that if the on-unit does not effectively correct the fault, a loop may result.

2. If the condition was raised by an area assignment, or by a SIGNAL statement, execution continues at the point of interrupt.

If no on-unit is specified, the system will comment and raise the ERROR condition.

INPUT/OUTPUT OF AREAS

The area facility is designed to allow easy input and output of complete lists of based variables as one unit, to and from RECORD files. On output, only the area extent, together with the 16 bytes of control information, is transmitted (although the extent does include freed allocations which are still significant). Thus the unused part of an area does not take up space on the data set. Because the extents of areas may vary, V-format or U-format records should be used. The maximum record length required is governed by the area length (i.e., area size + 16).

MULTIPLE LOCATOR QUALIFICATION

Locator qualification is the association of one or more locator values with a based variable to identify a particular generation of the based variable. Reference to a based variable can be explicitly qualified as follows:

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

A number of general rules can be stated concerning the use of locator qualification:

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

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

3. When more than one locator qualifier is used in a reference, only the first, or leftmost, can be a function reference; all other locator qualifiers must themselves be based variables. Note, however, that an entry variable can be based and can represent a function that returns a locator value.

4. When more than one locator qualifier is used, they are evaluated from left to right.

Reference to a based variable can also be implicitly qualified. The locator value used to determine the generation of a based variable that is implicitly qualified is the one declared with the based variable.

Because the locator declared with a based variable can also be based, a chain of locator qualifiers can be implied. For example:

```
DECLARE (P(10),Q) POINTER,
        R POINTER BASED (Q),
        V BASED (P(3)),
        W BASED (R),
        Y BASED;
ALLOCATE R,V,W;
```

Given this declaration and allocation, the following are valid references:

1. P(3) -> V
2. V
3. Q -> R -> W
4. R -> W
5. W

References 1 and 2 are equivalent as are references 3, 4 and 5. Note that any reference to Y must include a qualifying locator variable.

Levels of Locator Qualification

A pointer that qualifies a based variable represents one level of locator qualification; an offset represents two levels because it is implicitly qualified within an area. The number of levels is not affected by a locator being subscripted and/or an element of a structure. Under the optimizing compiler, the maximum number of levels of locator qualification allowed in a reference depends on the available storage, but it will never be less than ten; there is no limit under the checkout compiler. For example:

```
DECLARE X BASED (P),
        P POINTER BASED (Q),
        Q OFFSET (A);
```

Given this declaration the references: X, P -> X, and Q -> P -> X all represent three levels of locator qualification.

Chapter 9: Subroutines and Functions

Introduction

The block structure of PL/I permits the use of subroutines and programmer-defined functions. Subroutines and functions are groups of statements that can:

1. be invoked from different points in a program to perform the same frequently-used process.
2. process data passed from different points of invocation.
3. return control, and in the case of functions, return a value derived from the execution of the function, to a point immediately following the point of invocation.

Subroutines and functions may be either internal or external to the invoking block. Built-in functions are always external procedures which are permanently maintained in a PL/I system environment, and are an integral part of the PL/I language.

The rules given in this chapter for the use of subroutines and functions depend on whether the subroutine or function is an external or internal procedure: this is because the compiler can determine the relationship between two procedures from the procedures themselves when the invoked procedure is internal to the invoking procedure. When the invoked procedure is external the relationship must be given explicitly in the invoking procedure. Consequently it is necessary to supply more information about an external subroutine or procedure in the invoking procedure to enable the compiler to produce the required object program.

A subroutine is a procedure invoked by a CALL statement or CALL option of an INITIAL attribute.

A function, either programmer-defined or built-in, is invoked by the presence of a 'function reference' in an expression. A function reference is an entry expression which represents an entry name of a function. (An entry name is an identifier which represents a particular entry point of a procedure.)

The definitive difference between a subroutine and a function in PL/I is that a subroutine does not return data values to the point of invocation, whereas a function procedure returns a value to replace the function reference in the evaluation of the expression in which the function reference appears.

Both subroutines and functions can make use of data known in the invoking block. There are two methods by which data can be made available:

1. Data represented by names which are known in both the invoking block and the invoked procedure. For information about the rules for deciding where a name is known see chapter 7, "Recognition of Names".
2. Arguments and Parameters: values from the invoking block can be passed to the invoked procedure by writing arguments in an argument list associated with a CALL statement or option, or function reference; these values are made available by parameters in the invoked procedure.

   Parameters are identifiers which appear in the parameter list of an invoked entry point. The number of arguments and parameters must be the same; the maximum number permitted for a particular entry point is 64.

   A parameter has no storage associated with it: it is simply a means of allowing the invoked procedure to access storage allocated in the invoking procedure. A reference to a parameter in a procedure is effectively a reference to the corresponding argument. Any change to the value of the parameter is made to the value of the argument. However in certain circumstances a dummy argument is created and the value of the original argument is not changed. These are:

   a. When the attribute of an argument differ from those of the corresponding parameter. The value of the original argument is converted and assigned to a dummy.
   b. When only a value is passed as an argument. For example, when an argument is a constant.
   c. When the argument is an iSUB-defined array.

In these cases, a reference to the parameter is effectively a reference to the dummy. The dummy and the parameter initially have the same value as the original argument, but subsequent changes to the parameter do not affect the original argument's value. Storage for dummy arguments is within that belonging to the invoking procedure.

Both internal and external subroutines and functions are normally link-edited, and loaded into main storage at the same time as the calling procedure. An external subroutine or function may, however, be compiled, link-edited, and loaded separately from the calling procedure. By the use of FETCH and RELEASE statements in the calling procedure, the subroutine or function is allowed to remain on auxiliary storage until required in the calling procedure, at which time it is fetched into main storage; and it may be deleted from main storage when it is no longer required. This dynamic loading of external procedures is described in chapter 6, "Program Organization".

Entry points of Subroutines and Functions

A subroutine or function procedure may have one or more entry points.

PROCEDURE Statement: The primary entry point to a procedure is established by the PROCEDURE statement.

ENTRY Statement: Secondary entry points to a procedure are established by the ENTRY statement.

Each PROCEDURE and subsidiary ENTRY statement can specify its own parameters and, in the case of function procedures, returned value attributes. However, the environment established on entry to a block at a PROCEDURE statement is identical to the environment established when the same block is invoked at a secondary entry point. Each entry point has an associated entry name. The length of the name for an external entry-point to a PL/I procedure is limited to seven characters.

Entry names are explicitly declared in the invoking block as entry constants for internal procedures by their presence as prefixes to PROCEDURE or ENTRY statements; it is an error to declare an internal entry name in a DECLARE statement. External entry names must be declared explicitly as entry constants with the ENTRY attribute. Entry variables are identifiers with the attributes ENTRY and VARIABLE which represent entry constants assigned to them. A reference to an entry variable is a reference to its latest assigned entry constant value.

Use of the ENTRY Attribute

The general form of the ENTRY attribute is:

```
identifier ENTRY
           [(parameter descriptor list)]
           [VARIABLE]
           [RETURNS (attribute list)]
           [OPTIONS (options list)]
```

The parameter descriptor list is used to specify the attributes of the parameters associated with the entry point represented by the identifier. The parameter descriptor must provide accurate information about the attributes of the parameters so that the compiler can create the correct dummy arguments. If the parameter descriptor list is omitted from an external entry declaration, the compiler must assume that the attributes of any arguments match those of the corresponding parameters. No conversions are performed. Further information is given under the heading "Parameter Descriptor List" in this chapter.

The RETURNS attribute may be given to specify the attributes of the value returned by the function procedure.

The OPTIONS attribute is required if the entry point is in an external function or subroutine that has been compiled by a COBOL or FORTRAN compiler. Further information is given in chapter 19, "Interlanguage Communications".

Exit-Points of Subroutines and Functions

The RETURN statement is used to return control to the point immediately following the point of invocation; the GOTO statement is used to transfer control to some other point; and the END statement can also be used to return control from a subroutine procedure in the same way as a RETURN statement. For a function procedure, the RETURN statement must specify an element expression whose value is given to the function reference in the expression in which it appears.

RETURNS Attribute and RETURNS Option

The RETURNS attribute specifies for the invoking block the attributes of the value to be received from the function procedure. The RETURNS option specifies for the function procedure the attributes that a value to be returned should have. If the value does not have these attributes, the appropriate conversion is performed before the function relinquishes control and returns the value.

If the RETURNS option is not specified, the attributes of the returned value are assumed by default according to the initial letters of the entry-point name. The standard default assumptions are: REAL FIXED BINARY (15,0) for initial letters in the range (I:N) and REAL FLOAT DECIMAL (6) for the ranges (A:H) and (O:Z) and the characters $, #, @.

The RETURNS attribute must not be specified for an internal entry name because the compiler can determine the attributes of the returned value from the function procedure itself. If it is not specified for an external entry name or an entry variable, the compiler assumes default attributes (determined from the name of the entry point) for the value returned from the function. Consequently the RETURNS attribute and the RETURNS option must both be given in the situation when an external function procedure must return a value with attributes which cannot be determined correctly by default. The attributes in both the RETURNS attribute and the RETURNS option should agree, since the value returned by the function will have the attributes specified in the option, whereas the invoking procedure always assumes that the value will have the attributes specified in the RETURNS attribute.

Subroutines

The PL/I statements associated with the use of subroutine procedures are discussed below.

A subroutine is a procedure that usually requires arguments to be passed to it in an invoking CALL statement. It can be either an external or an internal procedure. A reference to such a procedure is known as a subroutine reference. The general format of a subroutine reference in a CALL statement or CALL option of an INITIAL attribute is as follows:

```
CALL entry-expression
                [(argument[,argument]...)];
```

Whenever a subroutine is invoked, the arguments of the invoking statement are associated with the parameters of the entry point, and control is then passed to that entry point. The subroutine is thus activated, and execution of the subroutine procedure can begin.

Upon termination of a subroutine, control is usually returned to the invoking block. A subroutine can be terminated by any of the following statements.

END Statement: Control reaches the final END statement of the subroutine. Execution of this statement causes control to be returned to the CALL statement from which the subroutine was invoked (unless control passes to another task).

RETURN Statement: Control reaches a RETURN statement in the subroutine. This causes the same normal return caused by the END statement.

GO TO Statement: Control reaches a GO TO statement that transfers control out of the subroutine. (This is not permitted if the subroutine is invoked by the CALL option of the INITIAL attribute.) The GO TO statement may specify a label in a containing block (the label must be known within the subroutine), or it may specify a parameter that has been associated with a label argument passed to the subroutine. Although this is a valid termination of the subroutine, it is not normal return of control, as effected by an END or RETURN statement.

EXIT Statement: The EXIT statement encountered in a subroutine abnormally terminates execution of that subroutine and of the task associated with the procedure that invoked it.

STOP Statement: The STOP statement encountered in a subroutine abnormally terminates execution of that subroutine and of the entire program associated with the procedure that invoked it.

Use of Subroutines: The following examples illustrate how a subroutine interacts with the procedure that invokes it.

```
  PRMAIN:  PROCEDURE;
DECLARE NAME CHARACTER (20),
  ITEM BIT(5), OUTSUB ENTRY;
                .
                .
                .
          CALL OUTSUB (NAME, ITEM);
                .
                .
                .
          END PRMAIN;


  OUTSUB:  PROCEDURE (A,B);
DECLARE A CHARACTER (20),
  B BIT(5);
                .
                .
                .
          PUT LIST (A,B);
                .
                .
                .
          END OUTSUB;
```

In procedure PRMAIN, NAME is declared as a character string, and ITEM as a bit string. The CALL statement in PRMAIN invokes the procedure called OUTSUB, and the parenthesized list included in this procedure reference contains the two arguments being passed to OUTSUB. The PROCEDURE statement defining OUTSUB declares two parameters, A and B. When OUTSUB is invoked, NAME is associated with A and ITEM is associated with B. Each reference to A in OUTSUB is treated as a reference to NAME and each reference to B is treated as a reference to ITEM. Therefore, the PUT LIST (A,B) statement causes the values of NAME and ITEM to be written into the standard system output file, SYSPRINT. Note that in the declaration of OUTSUB within PRMAIN, no parameter descriptor need be associated with the ENTRY attribute, since the attributes of NAME and ITEM match those of, respectively, A and B.

A name is explicitly declared to be a parameter by its appearance in the parameter list of a PROCEDURE or ENTRY statement. However, its attributes, unless defaults apply, must be explicitly stated within that procedure in a DECLARE statement.

It can be seen that the use of arguments and parameters provides the means for generalizing procedures so that data whose names may not be known within such procedures can, nevertheless, be operated upon.

```
A:  PROCEDURE;
    DECLARE RATE FLOAT (10), TIME FLOAT(5),
         DISTANCE FLOAT(15), MASTER FILE;
                .
                .
                .
    CALL READCM (RATE, TIME, DISTANCE,
         MASTER);
                .
                .
                .

READCM: PROCEDURE (W,X,Y,Z);
        DECLARE W FLOAT (10), X FLOAT(5),
                Y FLOAT(15), Z FILE;
                .
                .
                .
        GET FILE (Z) LIST (W,X,Y);
        Y = W*X;
        IF Y > 0 THEN RETURN;
            ELSE PUT LIST('ERROR READCM');
        END READCM;
        END A;
```

The arguments RATE, TIME, DISTANCE, and MASTER are passed to the parameters W, X, Y, and Z. Consequently, in the subroutine, a reference to W is the same as a reference to RATE, X the same as TIME, Y the same as DISTANCE, and Z the same as MASTER.

Functions

Unlike a subroutine, which is invoked by a CALL statement or a CALL option, a function is invoked by the appearance of the function name (and associated arguments) in an expression. Such an appearance is called a function reference. Like a subroutine, a function can operate upon the arguments passed to it and upon other known data. But unlike a subroutine, a function is written to compute a single value which is returned, with control, to the point of invocation. This single value can be of any data type except entry. An example of a function reference is contained in the following procedure:

```
     MAINP: PROCEDURE;
                .
                .
                .
            GET LIST (A, B, C, Y);
                .
                .
                .
            X = Y**3+SPROD(A,B,C);
```

In the above procedure, the assignment statement

```
X = Y**3+SPROD(A,B,C);
```

contains a reference to a function called SPROD. The parenthesized list following the function name contains the arguments that are being passed to SPROD. Assume that SPROD has been defined as follows:

```
SPROD: PROCEDURE (U,V,W);
         .
         .
         .
       IF U > V + W
              THEN RETURN (0);
              ELSE RETURN (U*V*W);
         .
         .
         .
       END SPROD;
```

When SPROD is invoked by MAINP, the arguments A, B, and C are associated with the parameters U, V, and W, respectively. Since attributes have not been explicitly declared for the arguments and parameters, default attributes of FLOAT DECIMAL (6) are applied to each argument and parameter.

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.

RETURN Statement: The RETURN statement is the usual way by which a function is terminated and control is returned to the invoking procedure. Its use in a function differs somewhat from its use in a subroutine; in a function, not only does it return control but it also returns a value to the point of invocation. The general form of the RETURN statement, when it is used in a function, is as follows:

```
RETURN (element-expression);
```

The value of the element expression 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 is taken as the value of the function reference, and evaluation of the invoking expression continues.

GO TO Statement: A function can also be terminated by execution of a GO TO statement. If this method is used, evaluation of the expression that invoked the function will not be completed, and control will go to the designated statement. As in a subroutine, the transfer point specified in a GO TO statement may be a parameter that has been associated with a label argument. For example, assume that MAINP and SPROD have been defined as follows:

```
MAINP:  PROCEDURE;
          .
          .
          .
        GET LIST (A,B,C,Y);
        X = Y**3+SPROD(A,B,C,LAB1);
          .
          .
          .
LAB1:   CALL ERRT;
          .
          .
          .
        END MAINP;

SPROD:  PROCEDURE (U,V,W,Z);
        DECLARE Z LABEL;
          .
          .
          .
        IF U > V + W
              THEN GO TO Z;
              ELSE RETURN (U*V*W);
          .
          .
          .
      END SPROD;
```

In MAINP, LAB1 is explicitly declared to be a statement label constant by its appearance as a label for the CALL ERRT statement. When SPROD is invoked, LAB1 is associated with parameter Z. Since the attributes of Z must agree with those of LAB1, Z is declared to have the LABEL attribute. When the IF statement in SPROD is executed, a test is made. If U is greater than V + W, the THEN clause is executed, control returns to MAINP at the statement labeled LAB1, and evaluation of the expression that invoked SPROD is discontinued. If U is not greater than V + W, the ELSE clause is executed and a return to MAINP is made in the normal fashion. Additional information about the use of label arguments and label parameters is contained in the section "Relationship of Arguments and Parameters" in this chapter.

Note: In some instances, a function may be so defined that it does not require an argument list. In such cases, the appearance of an external function name within an expression will be recognized as a function reference only if the function name has been explicitly declared to be an entry name. See "ENTRY Attribute" in this chapter for additional information.

ATTRIBUTES OF RETURNED VALUES

RETURNS Attribute: The RETURNS attribute is specified in a DECLARE statement for an external entry name. It specifies for the invoking block the attributes of the value returned by that function. It further specifies, by implication, the ENTRY attribute for the name. Unless attributes for the returned value can be determined correctly by default, any invocation of an external function must appear within the scope of a declaration with the RETURNS attribute for the entry name.

The general format of the RETURNS attribute is:

```
RETURNS (attribute-list)
```

A RETURNS attribute specifies that within the invoking procedure the value returned from an external function procedure is to be treated as though it had the attributes given in the attribute list. The word treated is used because no conversion is performed in an invoking block upon any value returned to it. The attributes given in a RETURNS attribute must agree with the data attributes given in the corresponding RETURNS option, since the value returned will have attributes determined from the RETURNS option.

The RETURNS attribute cannot be given for an internal procedure. The attributes of the returned value are determined from the RETURNS option at the entry point, if given; otherwise according to default rules as applied to the identifier of the entry constant.

RETURNS Option: The RETURNS option is specified in a PROCEDURE or ENTRY statement of a function procedure. It specifies the attributes to which the value returned by the function will be converted before return.

Generic Entry Names and References

A generic entry name represents a family of procedure entry points, each member of which can be invoked by a generic reference, that is, a procedure reference using the generic name in place of the actual entry name. The member invoked is determined according to the number and attributes of the arguments specified in the generic reference; the member that is invoked is the first one whose generic descriptor list matches the arguments both in number and attributes.

A generic name must be declared with the GENERIC attribute. The general format of this attribute is as follows:

```
generic name GENERIC (entry-expression)
    WHEN (generic-descriptor-list)
      [,entry-expression
    WHEN(generic-descriptor-list)]...);

    where generic-descriptor-list is:

    ([descriptor[,descriptor]...])
```

Each entry-expression corresponds to one procedure entry point in the family. The entry expression can be an entry name or an expression which represents an entry name. Each descriptor in the generic-descriptor list corresponds to a single argument, and may specify attributes that the corresponding argument must have in order that the associated entry name can be selected. Where no descriptor is required, it may be either omitted or indicated by an asterisk. The asterisk form is essential if the missing descriptor is the only descriptor. For example, whereas (,) represents two descriptors (*) represents one. The generic descriptor list which is to represent the absence of any argument takes the form:

```
....ENTRY1 WHEN( )...
```

An entry expression is chosen from those specified in a generic declaration by a process known as generic selection. Generic selection is performed by comparing arguments specified in a function reference or CALL statement with the contents of the generic descriptor list supplied with each entry expression in the GENERIC declaration. Firstly, each generic descriptor list is checked, in order of appearance in the declaration to determine whether it contains the same number of descriptors as there are arguments in the reference to the generic name.

When a generic descriptor list with the same number of descriptors as arguments is found, each descriptor is tested with the corresponding argument to determine whether attributes given in the descriptor are attributes of the argument. For example, if a generic descriptor list contains:

```
.....(FLOAT,FIXED)
```

and the corresponding two arguments have attributes such as DECIMAL FLOAT(6) and BINARY FIXED(15,0) either explicitly, contextually, implicitly, or by default, then each attribute in the generic-descriptor list is an attribute of the corresponding argument and the selection is successful. However, if either argument did not have the attributes

in the corresponding descriptor, the selection process would consider the next generic member with just two descriptors. For example consider the following statement:

```
DECLARE CALC GENERIC
        (FXDCAL WHEN (FIXED,FIXED),
        FLOCAL WHEN (FLOAT,FLOAT),
        MIXED WHEN (FLOAT,FIXED));
```

This statement defines CALC as a generic name having three members, FXDCAL, FLOCAL, and MIXED. One of these three function procedures will be invoked by a generic reference to CALC, depending on the characteristics of the two arguments in that reference. For example, consider the following statement:

```
Z=X+CALC(X,Y);
```

If X and Y are floating-point and fixed-point, respectively, MIXED will be invoked.

If all the descriptors are omitted or consist of an asterisk, the first entry name with the correct number of descriptors is selected.

The program is in error if no generic descriptor list is found to match the attributes of the arguments to a particular generic function reference.

Built-in Functions

Besides function references to procedures written by the programmer, a function reference may invoke one of a comprehensive set of pre-defined functions called built-in functions.

Built-in functions are an intrinsic part of PL/I. They include not only the commonly used arithmetic functions but also other necessary or useful functions related to language facilities, such as functions for manipulating strings and arrays.

Built-in functions are invoked in the same way that programmer-defined functions are invoked. However, many built-in functions can return an array of values, whereas a programmer-defined function can return only an element value.

Note: Some built-in functions will actually be compiled as in-line code rather than as procedure invocations.

The use of a built-in function with a list, such as SUBSTR (X,Y,Z) or INDEX(A,'B'), etc., is recognized without further identification being necessary to establish the identifier as a built-in function. However, any built-in function or pseudovariable which does not have a parenthesized argument list, such as ONCHAR, ONSOURCE, TIME, etc., must be either declared explicitly with the attribute BUILTIN, or specified with a null argument list (for example TIME()) in the block in which the identifier is used as a built-in function.

Built-in function names can be used as programmer-defined names. Consequently, ambiguity may occur if a built-in function reference is used in a block that is contained in another block in which the same identifier is declared for some other purpose. To avoid this ambiguity, the BUILTIN attribute can be declared for a built-in function name in any block that has inherited, from a containing block, some other declaration of the identifier. Consider the following example.

```
A: PROCEDURE;
     .
     .
     .
   B:  BEGIN;
       DECLARE SQRT FLOAT BINARY;
         .
         .
         .
       C:  BEGIN;
           DECLARE SQRT BUILTIN;
             .
             .
             .
           END C;
         .
         .
         .
       END B;
     .
     .
     .
   END A;
```

Assume that in external procedure A, SQRT is contextually declared with the attribute BUILTIN. Consequently, any reference to SQRT would refer to the built-in function of that name. In B, however, SQRT is declared to be a floating-point binary variable, and it cannot be used in any other way. Finally, in C, SQRT is declared with the BUILTIN attribute so that any reference to SQRT will be recognized as a reference to the built-in function and not to the floating-point binary variable declared in B.

Note that a variable having the same identifier as a built-in function can be implicitly declared as an arithmetic variable by, for instance, its appearance on the left-hand side of an assignment

symbol (in an assignment statement, a DO statement, or a repetitive specification) or in the data list of a GET statement, provided that it is neither enclosed within nor immediately followed by an argument list. (This also applies to the names ONCHAR, ONSOURCE, and PRIORITY which are pseudovariables that do not require arguments.) For example, if the statement SQRT = 1 had appeared in begin block B instead of the DECLARE statement, SQRT would have been implicitly declared as a floating-point decimal variable.

A programmer can even use a built-in function name as the entry name of a programmer-defined function and, in the same program, use both the built-in function and the programmer-defined function. This can be accomplished by use of the BUILTIN attribute when the programmer-defined function is an internal procedure, and by use of the BUILTIN and ENTRY attributes when the programmer-defined function is an external procedure.

The following example illustrates use of the BUILTIN attribute in conjunction with an internal function procedure.

```
A:  PROCEDURE;

SQRT:  PROCEDURE (PARAM)
                  RETURNS (FIXED (6,2));
       DECLARE PARAM FIXED (12);
         .
         .
         .
       END SQRT;
         .
         .
         .
   X = SQRT(Y);
         .
         .
         .
   B:  BEGIN;
       DECLARE SQRT BUILTIN;
         .
         .
         .
       Z = SQRT (P);
         .
         .
         .
       END B;
         .
         .
         .
   END A;
```

The use of SQRT as the label of the second PROCEDURE statement is an explicit declaration of the identifier as an entry name. The function reference in the assignment statement in A thus refers to the programmer-written SQRT function. In the begin block B, the identifier SQRT is declared with the BUILTIN attribute. Consequently, the function reference in the assignment statement in B refers to the built-in SQRT function.

For a programmer-written internal function using the name of a built-in function any reference to the identifier in the containing block would be a reference to the programmer-written function. In the above example the attributes of the returned value are specified in the RETURNS option of the procedure statement for SQRT. Since the function procedure is internal, these attributes are known to the calling procedure.

In the case of a programmer-written external function procedure using as an entry name the name of a built-in function, any procedure containing a reference to that function procedure name must also contain an entry declaration of that name; otherwise a reference to the identifier would be a reference to the built-in function. In the above example, if the begin block B were not contained in A, there would be no need to specify the BUILTIN attribute; unless the identifier SQRT is given attributes other than BUILTIN (by explicit or contextual declaration), it refers to the built-in function. If the procedure SQRT were an external procedure, procedure A would need the following statement to declare explicitly SQRT AS AN ENTRY NAME, and to specify the attributes of the values passed to and returned from the programmer written function procedure.

```
DCL SQRT ENTRY (FIXED (12)) RETURNS
(FIXED(6,2));
```

FORTRAN Library Functions

Library functions, analagous to PL/I built-in functions, are associated with FORTRAN compilers. These functions may be invoked from a PL/I program by means of PL/I interlanguage communication facilities. The facilities are described in Chapter 19.

Built-in Subroutines

A PL/I programmer can avail himself of certain operating system facilities by using built-in subroutines. These have entry names that are defined by the implementation and are invoked by means of the CALL statement. The operating system

facilities and the corresponding entry names are as follows.

Checkpoint/restart (implemented by the optimizing compiler only): PLICKPT, PLIREST, PLICANC

A CALL statement specifying PLICKPT, PLIREST, or PLICANC is treated as a null statement by the checkout compiler.

Sort/merge: PLISRTA, PLISRTB, PLISRTC, PLISRTD

In addition, there is a subroutine, PLIDUMP, that provides an edited dump of main storage, and another, PLIRETC, that allows the user to set the return code of his program.

The entry names are known as built-in names, and can be explicitly or contextually declared to have the BUILTIN attribute. They are not reserved words.

The use of these subroutines is described in the following publications: OS PL/I Optimizing Compiler: Programmer's Guide and OS PL/I Checkout Compiler: Programmer's Guide.

Relationship of Arguments and Parameters

When a function or subroutine is invoked, a relationship is established between the arguments of the invoking statement or expression and the parameters of the invoked entry point. This relationship is dependent upon whether or not dummy arguments are created.

DUMMY ARGUMENTS

In the preceding discussions of arguments and parameters, it is pointed out that the name of an argument, not its value, is passed to a subroutine or function. However, this is not always possible. A constant, for example, has no name; nor does an operational expression. Therefore, the compiler provides storage for such values and associates the name of the corresponding parameter with each. These storage locations are called dummy arguments. The PL/I programmer 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 when the original argument is any of the following:

1. A constant.
2. An expression involving operators.
3. An expression in parentheses.
4. A variable whose data attributes are different from the data attributes declared for the parameter. This does not apply when an expression other than a decimal integer constant is used to define the bounds, length or size of a controlled parameter: the compiler assumes that the argument and parameter bounds, length or size match.*
5. A function reference with an argument list.
6. A controlled string or area, or a string or area with an adjustable length or size, associated with a non-controlled parameter whose length or size is a constant.
7. An iSUB-defined array.

The attributes of a dummy argument created for an argument to be passed to an internal procedure are derived as follows:

1. From the attributes declared for the associated parameter in the internal procedure.
2. For the bounds of an array, the length of a string or the size of an area, if specified by asterisk notation in the parameter declaration, from the bound, length or size of the argument itself.

In all other cases, a reference to the argument is passed directly (in effect, the storage address of the argument is passed). The parameter becomes identical with the passed argument; thus, changes to the value of a parameter will be reflected in the value of the original argument only if a dummy argument is not passed.

---

*In the case of arguments and parameters with the PICTURE attribute, a dummy argument will be created unless the picture specifications match exactly, after any repetition factors have been applied. The only exception is that an argument or parameter with a + sign in a scaling factor matches a parameter or argument without the + sign.

ENTRY ATTRIBUTE

The ENTRY attribute is used to identify the entry name of an external procedure. The use of the ENTRY attribute to identify the entry constant of an internal procedure is invalid; its use to identify each entry point of an external procedure is mandatory. The general form of the ENTRY attribute is described in "Use of the ENTRY Attribute", earlier in this chapter.

Note that the format allows the keyword ENTRY to be specified without an accompanying parameter descriptor list when used to identify a function entry name that does not require arguments, or when the arguments and parameters match. The parameter descriptor list must be specified with an ENTRY attribute that identifies the entry name of an external procedure if arguments do not match parameters. The use of the attribute VARIABLE in an entry declaration establishes the identifier as an entry variable. An entry variable represents an entry constant after assignment of the entry constant to the entry variable. If an entry variable is used in a function reference or CALL statement to invoke an entry point to which arguments are to be passed, the entry variable should be declared with a parameter descriptor list which specifies the attributes of the parameters of the entry point, otherwise erroneous arguments may be passed.

Parameter Descriptor Lists

Each set of attributes, or descriptor, in the parameter descriptor list in the ENTRY attribute specification corresponds to one parameter of the subroutine or function invoked, and if given, specifies the attributes of that parameter. The attributes of an individual parameter are separated by blanks to form a parameter descriptor for each parameter; parameter descriptors in a parameter descriptor list are separated by commas. In general, if the attributes of an argument do not agree with those of its corresponding parameter (as specified in a parameter descriptor list), a dummy argument is constructed for that argument if conversion is possible. The dummy argument contains the value of the original argument converted to conform with the attributes of the corresponding parameter. Thus, when the subroutine or function is invoked, it is the dummy argument that is passed to it.

When a descriptor list is given with the ENTRY attribute, each parameter of the subroutine or function must be accounted for. When the attributes of the argument and parameter match, the descriptor may be either omitted or indicated by an asterisk, but commas delimiting the descriptors must not be omitted. For example, the statement:

```
DECLARE SUBR ENTRY (FIXED,,FLOAT);
```

specifies that SUBR is an entry point that has three parameters: the first and third have the attributes FIXED and FLOAT, respectively, while the attributes of the second are assumed to be the same as those of the argument being passed. Since the attributes of the second parameter are not stated, no assumptions are made.

As mentioned earlier, the ENTRY attribute may be specified without a parameter descriptor list. It is used in this way to indicate that the associated identifier is an entry name. Such an indication is necessary if an identifier is not otherwise recognizable as an entry name, that is, if it is not explicitly declared to be an entry name by its appearance as a label of a PROCEDURE or ENTRY statement.

Therefore, if a reference is made to an entry name in a block in which it does not appear in this way, the identifier must be given the ENTRY attribute explicitly. For example, assume that the following has been specified:

```
A:  PROCEDURE;
      .
      .
      .
    PUT LIST (RANDOM);
      .
      .
      .
    END A;
```

Assume also that A is an external procedure and RANDOM is an external function that requires no arguments and returns a random number. As the procedure is shown above, RANDOM is not recognizable within A as an entry name, and the result of the PUT statement therefore is undefined. In order for RANDOM to be recognized within A as an entry name, it must be declared to have the ENTRY attribute. For example:

```
A: PROCEDURE;
    DECLARE RANDOM ENTRY;
      .
      .
      .
    PUT LIST (RANDOM);
      .
      .
      .
    END A;
```

Now, RANDOM is recognized as an entry name, and the appearance of RANDOM in the PUT statement cannot be interpreted as anything but a function reference. Therefore, the PUT statement results in the output transmission of the random number returned by RANDOM.

Note: The ENTRY attribute is implied -- and therefore need not be stated explicitly -- for an identifier that is declared in a DECLARE statement to have one of the entry name attributes RETURNS, OPTIONS, REDUCIBLE, or IRREDUCIBLE.

Entry Expressions as Arguments

When an entry name is specified as an argument of a function or subroutine reference, one of the following applies:

1. If the entry expression argument, call it M, is specified with an argument list of its own, it is recognized as a function reference; M is invoked, and the value returned by M effectively replaces M and its argument list in the containing argument list. For example:

   ```
   CALL A (M(B));
   ```

   This passes the value returned by the function procedure M.

   If the entry expression argument appears with a null argument list, it is taken to be a function reference with no arguments. For example:

   ```
   CALL A(B());
   ```

   This passes, as the argument to procedure A, the value returned by the function procedure B.

2. If the entry expression argument has no argument list and appears within parenthesis, a dummy entry variable is created. For example:

   ```
   CALL A((B));
   ```

   This passes, as the argument to procedure A, the value of the entry name B.

3. When a built-in function name or an entry expression is used without an argument list as an argument to a built-in function, the function specified by the argument is not invoked provided that the built-in function will accept an argument of type ENTRY. If the built-in function will not accept an entry argument, the argument is assumed to be a reference to the value of the function. For example:

   ```
   DCL DATE BUILTIN, Z CHAR(2);

   Z = SUBSTR (DATE,5,2);
   ```

   The days field is extracted from the value returned by the DATE built-in function

4. If the entry expression argument to a user-defined function appears without an argument list and neither within an operational expression nor within parentheses, the entry expression itself is passed to the function or subroutine being invoked. In such cases, the entry expression is not taken to be a function reference, even if it is the name of a function that does not require arguments. For example:

   ```
   CALL A(B);
   ```

   This passes the entry expression B as an argument to procedure A. If the corresponding parameter in A has been declared with the attribute ENTRY, it will be given the attribute VARIABLE by default. If B is an entry variable, it will be passed to the parameter in the same way as for any argument whose attributes match those of the parameter. If B is an entry constant a dummy is created and passed, as for any constant argument.

   If an identifier is known as an entry name and appears as an argument and if the parameter descriptor for that argument specifies an attribute other than ENTRY, the entry name will be invoked and its returned value passed. If the value returned has different attributes from those specified in the parameter descriptor, conversion is performed. For example:

```
A:  PROCEDURE;
   DECLARE B ENTRY,
           C ENTRY(FLOAT);
      .
      .
      .
   X = C(B);
      .
      .
      .
   END A;
```

In this case, B is invoked and its returned value is passed to C.

Consider the following example:

```
CALLP:  PROCEDURE;
       DECLARE RREAD ENTRY,
               SUBR ENTRY (ENTRY, FLOAT,
                  FIXED BINARY, LABEL);
         .
         .
         .
       GET LIST (R,S);
         .
         .
         .
       CALL SUBR (RREAD, SQRT(R), S,
          LAB1);
         .
         .
         .
 LAB1:  CALL ERRT(S);
         .
         .
         .
       END CALLP;

 SUBR:  PROCEDURE(NAME, X, J, TRANPT);
       DECLARE NAME ENTRY, TRANPT LABEL;
         .
         .
         .
       IF X > J THEN CALL NAME(J);
                ELSE GO TO TRANPT;
         .
         .
         .
       END SUBR;
```

In this example, assume that CALLP, SUBR, and RREAD are external. In CALLP, both RREAD and SUBR are explicitly declared to have the ENTRY attribute. The explicit declaration for SUBR is used to provide information about the characteristics of the parameters of SUBR. Four arguments are specified in the CALL SUBR statement. These arguments are interpreted as follows:

1. The first argument, RREAD, is recognized as an entry name (because of the ENTRY attribute declaration). This argument is not in conflict with the first parameter descriptor specified in the ENTRY attribute declaration for SUBR in CALLP. Therefore, since RREAD is recognized as an entry name and not as a function reference, the entry name is passed at invocation. Since NAME is an entry parameter, it is given the attribute VARIABLE by default. Since RREAD is a constant, a dummy entry argument is created, and this is passed to NAME.

2. The second argument, SQRT(R), is recognized as a built-in function reference because of the argument list accompanying the entry name. SQRT is invoked, and the value returned by SQRT is assigned to a dummy argument, which will be passed to the subroutine SUBR. The attributes of the dummy argument agree with those of the second parameter, as specified in the parameter attribute list declaration. When SUBR is invoked, the dummy argument is passed to it.

3. The third argument, S, is simply a decimal floating-point element variable. However, since its attributes do not agree with those of the third parameter, a dummy argument is created containing the value of S converted to the attributes of the third parameter. When SUBR is invoked, the dummy argument is passed.

4. The fourth argument, LAB1, is a statement-label constant. Its attributes agree with those of the fourth parameter. But since it is a constant, a dummy argument is created for it. When SUBR is invoked, the dummy argument is passed.

In SUBR, four parameters are explicitly declared in the PROCEDURE statement. If no further explicit declarations were given for these parameters, arithmetic default attributes would be supplied for each. Therefore, since NAME must represent an entry name, it is explicitly declared with the ENTRY attribute, and since TRANPT must represent a statement label, it is explicitly declared with the LABEL attribute. X and J are arithmetic, so the defaults are allowed to apply.

Note that the appearance of NAME in the CALL statement does not constitute a contextual declaration of NAME as a built-in procedure. Such a contextual declaration is made if no explicit declaration applies. However the appearance of NAME in the PROCEDURE statement of SUBR constitutes an explicit declaration of NAME as a parameter. If the attributes of a parameter are not explicitly declared in a complementary DECLARE statement, arithmetic defaults apply. Consequently, NAME must be explicitly declared to have the ENTRY

attribute; otherwise, it would be assumed to be a binary fixed-point variable, and its use in the CALL statement would result in an error.

ALLOCATION OF PARAMETERS

Since a parameter has no associated storage within the invoked procedure, it cannot be declared to have any of the storage class attributes STATIC, AUTOMATIC, or BASED. It can, however, be declared to have the CONTROLLED attribute. Thus, there are two classes of parameters, as far as storage allocation is concerned: those that have no storage class, i.e., simple parameters, and those that have the CONTROLLED attribute, i.e., controlled parameters.

A simple parameter may be associated with an argument of any storage class. However, if more than one generation of the argument exists, the parameter is associated only with that generation existing at the time of invocation.

A controlled parameter must always have a corresponding controlled argument. Such an argument cannot be subscripted, cannot be an element of a structure, and cannot cause a dummy to be created. If more than one generation of the argument exists at the time of invocation, the parameter corresponds to the entire stack of these generations. Thus, at the time of invocation, a controlled parameter represents the current generation of the corresponding argument. A controlled parameter may be allocated and freed in the invoked procedure, thus allowing the manipulation of the allocation stack of the associated argument. A simple parameter cannot be specified in an ALLOCATE or FREE statement.

When no parameter descriptor is given, the entire stack is passed. In this case, the parameter may be simple or controlled and be correspondingly associated with either the latest generation or the entire stack.

Parameter Attributes

Parameters cannot be declared with the attributes DEFINED or BASED. A parameter may be used as a base identifier for overlay defining and it may be used for record-oriented transmission only provided it has the CONNECTED attribute. A parameter always has the attribute INTERNAL. It must be a level-one identifier.

Parameter Bounds, Lengths, and Sizes

If an argument is an array, a string, or an area, the bounds of the array, the length of the string, or the size of the area must be declared for the corresponding parameter. The number of dimensions and the bounds of an array parameter, or the length and size of an area or string parameter, must be the same as the current generation of the corresponding argument. Usually, this can be assured simply by specifying actual numbers for the bounds, length, or size of the parameter.

If the bounds, length, or size are not known at the time the subroutine or function is written, they may be specified by asterisks, for simple parameters, or asterisks or expressions for controlled parameters.

Simple Parameter Bounds, Lengths, and Sizes

When the actual length, bounds, or size of a simple parameter may be different for different invocations, they can be specified in a DECLARE statement by asterisks. When an asterisk is used, the length, bounds, or size are taken from the current generation of the corresponding argument.

An asterisk is not allowed as the length specification of a character or bit string that is an element of an aggregate, if the corresponding argument is such that a dummy is created. The string length must be specified as a decimal integer constant.

Controlled Parameter Bounds, Lengths, and Sizes

The bounds, length, or size of a controlled parameter can be represented in a DECLARE statement either by asterisks or by element expressions.

Asterisk Notation: When asterisks are used, length, bounds, or size of the controlled parameter are taken from the current generation of the corresponding argument. Any subsequent allocation of the controlled parameter uses these same bounds, length, or size, unless they are overridden by a different length , bounds, or size specification in the ALLOCATE statement. If no current generation of the argument exists, the asterisks only determine the dimensionality of the parameter, and an ALLOCATE statement in the

invoked procedure must specify bounds, length, or size for the controlled parameter before other references to the parameter can be made.

Expression Notation: The bounds, length, or size of a controlled parameter can also be specified by element expressions. These expressions are evaluated at the time of allocation. Each time the parameter is allocated, the expressions are re-evaluated to give current bounds , length, or size for the new allocation. However, such expressions in a DECLARE statement can be overridden by a bounds , length, or size specification in the ALLOCATE statement itself. For example:

```
MAIN:  PROCEDURE OPTIONS(MAIN);
       DECLARE (A(20), B(30), C(100),
                D(100))CONTROLLED,
                NAME CHARACTER (20),
                I FIXED(3,0);
          .
          .
          .
       ALLOCATE A,B;
       CALL SUB1(A,B);
          .
          .
       FREE A,B;
          .
          .
          .
       FREE A,B;
       GET LIST (NAME,I);
       CALL SUB2 (C,D,NAME,I);
          .
          .
          .
       FREE C,D;
          .
          .
          .
       END MAIN;

SUB1:  PROCEDURE (U,V);
       DECLARE (U(*), V(*)) CONTROLLED;
          .
          .
          .
       ALLOCATE U(30), V(40);
          .
          .
          .
       RETURN;
       END SUB1;

SUB2:  PROCEDURE (X,Y,NAMEA,N);
       DECLARE (X(N),Y(N))CONTROLLED,
                NAMEA CHARACTER (*),
                N FIXED(3,0);
          .
          .
          .
       ALLOCATE X,Y;
          .
          .
          .
       RETURN;
       END SUB2;
```

In the procedure MAIN, the arrays A, B, C, and D are declared with the CONTROLLED storage class attribute; NAME and I are AUTOMATIC by default.

When SUB1 is invoked, A and B, which have been allocated as declared, are passed. SUB1 declares its parameters with the asterisk notation. The ALLOCATE statement, however, specifies bounds for the arrays; consequently, the allocated arrays, which are actually a second generation of A and B, have bounds different from the first generation. If no bounds were specified in the ALLOCATE statement, the bounds of the first and the new generation would be identical.

On return to MAIN, the first FREE statement frees the second generation of A and B (allocated in SUB1 as parameters), and the second FREE statement frees the first generation (allocated in MAIN).

When SUB2 is invoked, C and D are passed to X and Y, NAME is passed to NAMEA, and I is passed to N. In SUB2, X and Y are declared with bounds that depend upon the value of I (passed to N). When X and Y are allocated, this value determines the bounds of the allocated array.

Although NAME (corresponding to NAMEA) is not controlled, the asterisk notation for the length of NAMEA indicates that the length is to be picked up from the argument (NAME).

ARGUMENT AND PARAMETER TYPES

In general, an argument and its corresponding parameter may be of any data organization and type. However, not all parameter/argument relationships are so clear-cut. Some need further definition and clarification; these are given below.

If a parameter is an element, i.e., a variable that is neither a structure nor an array, the argument must be an element expression. If the argument is a

subscripted variable, the subscripts are evaluated before the subroutine or function is invoked and the name of the specified element is passed. If the argument passed to an external procedure is a constant, the attributes of the corresponding parameter must agree with the attributes indicated by the constant, unless there is a corresponding parameter descriptor in the entry declaration.

If a parameter is an array, the argument may be an array expression or an element expression. If the argument is an element expression, the corresponding parameter descriptor or declaration must specify the bounds of the array parameter. The bounds must be specified as decimal integer constants. This causes the construction of a dummy array argument, whose bounds are those of the array parameter. The value of the element expression is then assigned to the value of each element of the dummy array argument.

If a parameter is a structure, the argument must be a structure expression or an element expression. If the argument is an element expression, the corresponding parameter descriptor for an external entry point must specify the structure description of the structure parameter (only level numbers need be used -- see the discussion of the ENTRY attribute in section I, "Attributes", for details). This causes the construction of a dummy structure argument, whose description matches that of the structure parameter. The value of the element expression then becomes the value of each element of the dummy structure argument. The relative structuring of the argument and the parameter must be the same; the level numbers need not be identical. The element value must be one that can be converted to conform with the attributes of all the elementary names of the structure.

If the parameter is an array of structures, the argument can be the expression representing an element, an array, a structure or an array of structures.

If a parameter is a label, the argument must be either a label variable or a label constant. If the argument is a label constant, a dummy argument is constructed.

If the parameter is an entry, the argument must be an entry name or a generic name. If the argument is a generic name the parameter descriptor (or parameter declaration, if the invoked procedure is internal) must give parameter descriptions to enable generic selection to be made before passing an entry. Under the optimizing compiler, entry variables passed as arguments are assumed to be aligned, so that no dummy argument is created when only the alignments of argument and parameter differ. Note that the name of a mathematical built-in function can be passed as an argument but no other built-in function name can be passed.

If a parameter is a file, the argument must be a file variable or file constant.

For example:

```
E:  PROCEDURE;
DECLARE F1 FILE;
CALL E1(F1);
   .
   .
   .
E1:  PROCEDURE(F2);
   DECLARE F2 FILE;
   CALL E2(F2);
     .
     .
     .
E2:  PROCEDURE (F3);
   DECLARE F3 FILE;
     .
     .
     .
   END E;
```

The file parameters F1, F2, and F3 all refer to the same file. Input/output on-units for file parameters are discussed in chapter 14, "Execution Condition Handling and Program Checkout".

If the parameter is a fixed length string, and if a dummy argument is not to be created, then the argument must also be a fixed length string. Similarly, if a dummy is not to be created when the parameter is a varying length string, the argument must be a varying length string. Whenever a varying-length element string argument is passed to a non-varying element string parameter whose length is undefined (i.e. specified by an asterisk), the current length of the argument is passed to the invoked procedure. When the argument is a varying-length string array passed to

a non-varying undefined-length parameter, only one length is passed, namely the maximum length.

If a parameter is a locator of either pointer or offset type, the argument must be a locator expression of either type. If the types differ, a dummy argument is created. The parameter descriptor of an offset parameter must not specify an associated area.

If the parameter is an area , the argument must be an area expression. If the sizes differ, a dummy argument is created.

Passing an Argument to the Main Procedure

A single argument can be passed using the PARM field in the statement for the step executing the PL/I program. See OS PL/I Optimizing Compiler: Programmer's Guide and OS PL/I Checkout Compiler: Programmer's Guide. If this facility is used, the parameter must be declared as a VARYING character string; the maximum length is 100, and the current length is set equal to the argument length at object time. For example:

```
TOM:  PROC (PARAM) OPTIONS (MAIN);
      DCL PARAM CHAR(100) VARYING;
```

The value in the PARM field of the EXEC statement for the execution job step will be passed to TOM.

Storage is allocated only for the current length of the argument; the source program will overwrite adjacent information if a value greater than the current length is assigned to the parameter.

Chapter 10: Input and Output

Introduction

PL/I includes input and output statements that enable data to be transmitted between the internal and external storage devices of a computer. A collection of data external to a program is called a data set. Transmission of data from a data set to a program is termed input, and transmission of data from a program to a data set is called output.

PL/I input and output statements are concerned with the logical organization of a data set and not with its physical characteristics; a program can be designed without specific knowledge of the input/output devices that will be used when the program is executed. To allow a source program to deal primarily with the logical aspects of data rather than with its physical organization in a data set, PL/I employs a symbolic representation of a data set called a file. A file can be associated with different data sets at different times during the execution of a program.

Two types of data transmission can be used by a PL/I program. In stream-oriented transmission, the organization of the data in the data set is ignored within the program, and the data is treated as though it actually were a continuous stream of individual data items in character form; data is converted from character form to internal form on input, and from internal form to character form on output. In record-oriented transmission, the data set is considered to be a collection of discrete records. No data conversion takes place during record transmission; on input the data is transmitted exactly as it is recorded in the data set, and on output it is transmitted exactly as it is recorded internally.[1] It is possible for the same data set to be processed at different times by either stream transmission or record transmission; however, all items in the data set would have to be in character form.

Stream-oriented transmission is ideal for simple jobs, particularly those that use punched card input and have limited output; a minimum of coding is required of the programmer, especially for punched card input and printed output. Stream-oriented transmission also allows communication with the program at execution time from a terminal, if the program is being run under the Time Sharing Option. However, compared with record-oriented transmission, stream-oriented transmission is less efficient in terms of execution time because of the data conversion it involves, and more space is required on external storage devices because all data is in character form.

Record-oriented transmission is more versatile than stream-oriented transmission, with regard to both the manner in which data can be processed and the types of data set that it can process. Since data is recorded in a data set exactly as it appears in main storage, any data format is acceptable; no conversion problems will arise, but the programmer must have a greater awareness of the structure of his data.

This chapter discusses those aspects of PL/I input and output that are common to stream-oriented and record-oriented transmission, including files and their attributes, and the relationship of files to data sets. The next two chapters describe the input and output statements that can be used in a PL/I program, and the various data set organizations that are recognized in PL/I.

Data Sets

Data sets are stored on a variety of auxiliary storage media, such as punched cards, reels of magnetic tape, magnetic disks, and magnetic drums. Despite their variety, these media have many common characteristics that permit standard methods of collecting, storing, and transmitting data. For convenience, the general term volume is used to refer to a unit of auxiliary storage, such as a reel of magnetic tape or a disk pack, without regard to its specific physical composition.

The data items within a data set are arranged in distinct physical groupings

---

[1] This is not strictly true for ASCII data sets - see "Information Interchange Codes" in this chapter.

called blocks.[2] These blocks allow the data set to be transmitted and processed in portions rather than being transmitted in its entirety before any processing is carried out. For processing purposes, each block may consist of logical subdivisions called records, each of which contains one or more data items. A block can comprise part of a record, a single record, or several records. (Sometimes a block is called a physical record, because it is the unit of data that is physically transmitted to and from a volume, and its logical subdivisions are called logical records.)

When a block contains two or more records, the records are said to be blocked. Blocked records permit more compact and efficient use of auxiliary storage. The use of blocked records can also improve the throughput of a program where a large number of short records are to be processed, by reducing the number of physical input/output operations.

Most data processing applications are concerned with logical records rather than blocks. Therefore, the input and output statements of PL/I generally refer to logical records; this allows the programmer to concentrate on the data to be processed, without being directly concerned about its physical organization in external storage.

INFORMATION INTERCHANGE CODES

In System/360 and System/370, the standard code used to represent data, both in main storage and on auxiliary storage, is EBCDIC (extended binary-coded-decimal interchange code). In general, PL/I programs compiled by the optimizing or checkout compiler use EBCDIC to record all character data. The operating system does, however, support the use of an alternative code, namely ASCII (American Standard Code for Information Interchange), to represent data on auxiliary storage, and such data sets may be read or created using PL/I. The support is limited to data sets held on magnetic tape.

Translation between the two codes is performed by the operating system. Apart from the options specified in the ENVIRONMENT attribute, the same PL/I program may be used to handle an ASCII data set as would be used for a standard EBCDIC data set. On output, translation from EBCDIC to ASCII is performed immediately before data is written from a buffer to external storage. On input, translation is performed from ASCII to EBCDIC as soon as a buffer is filled with data.

In PL/I, only CHARACTER data may be written onto an ASCII data set. Each character in the ASCII code is represented by a seven-bit pattern and there are 128 such patterns. In EBCDIC, each character has an eight-bit pattern, and there are 256 possibilities. The ASCII set includes a substitute character (the SUB control character) that is used to represent EBCDIC characters having no valid ASCII code. (In the American National Standards Institute table, this is the character having the column 1, row 10 position.) Upon reading this data, the character would be translated to the EBCDIC SUB character, which has the bit pattern 00111111.

Files

To allow a source program to deal primarily with the logical aspects of data rather than with its physical organization in a data set, PL/I employs a symbolic representation of a data set called a file. This symbolic representation determines how input and output statements access and process the associated data set. Unlike a data set, however, a file has significance only within the source program and does not exist as a physical entity external to the program.

PL/I requires that an identifier which represents a file be declared with the FILE attribute. Such an identifier may either be a file constant or a file variable. A file variable is a data item to which a file constant can be assigned. After assignment, a reference to the file variable has the same significance as a reference to the assigned file constant. Each data set processed by a PL/I program must be associated with a file constant identifier.

File Constants: The individual characteristics of each file are described with keywords called file description attributes. The following lists show the attributes that apply to each type of data transmission:

---

[2] This discussion has to be slightly modified for teleprocessing applications, where the data set is in fact a queue of messages and the term "block" is not strictly applicable. However, a message is similar to a block in that it may consist of one or more records. Teleprocessing is discussed in chapter 12, "Record-Oriented Transmission."

Stream-oriented Transmission

FILE
STREAM
INPUT
OUTPUT
PRINT
ENVIRONMENT

Record-oriented Transmission

FILE
RECORD
INPUT
OUTPUT
UPDATE
SEQUENTIAL
DIRECT
TRANSIENT
BUFFERED
UNBUFFERED
BACKWARDS
KEYED
EXCLUSIVE
ENVIRONMENT

File variables: A file variable is an identifier that has the attributes FILE and VARIABLE; it cannot have any of the file description attributes (except FILE). File variables can be collected into arrays or structures. Note that the VARIABLE attribute can be implied by, for example, the dimension attribute.

File expressions: A file expression can be a reference to a file constant, a file variable, or a function reference which returns a value with the FILE attribute.

A detailed description of each of these attributes appears in section I, "Attributes." The discussions below give a brief description of each of the file description attributes and show how these attributes are declared for a file.

FILE ATTRIBUTE

The FILE attribute indicates that the associated identifier is a file constant or variable. For example, the identifier MASTER is declared to be a file constant in the following statement:

```
DECLARE MASTER FILE;
```

In the following statement, the identifier ACCOUNT is declared to be a file variable, and ACCT1, ACCT2, ... are declared to be file constants; the file constants may subsequently be assigned to the file variable.

```
DECLARE ACCOUNT FILE VARIABLE,
        ACCT1 FILE,
        ACCT2 FILE,
            .
            .
            .
```

The following example shows how the VARIABLE attribute may be implied.

```
DECLARE PAYREC(10) FILE;
```

PAYREC(I), where I has a value from 1 to 10, has the attribute FILE by explicit declaration and the attribute VARIABLE by implication of the dimension attribute (10) in the DECLARE statement.

The attributes associated with a file constant fall into two categories: alternative attributes and additive attributes. An alternative attribute is one that is chosen from a group of attributes. If no explicit or implicit declaration is given for one of the alternative attributes in a group and if one of the alternatives is required, a default attribute is assumed.

An additive attribute is one that must be stated explicitly or is implied by another explicitly stated attribute. The additive attribute KEYED is implied by the DIRECT attribute. The additive attribute PRINT can be implied by the standard output file name SYSPRINT. An additive attribute can never be implied by default.

Note: With the exception of the INTERNAL and EXTERNAL scope attributes, all the alternative and additive attributes imply the FILE attribute. Therefore, the FILE attribute need not be specified for a file that has at least one of the alternative or additive attributes already specified explicitly.

ALTERNATIVE ATTRIBUTES

PL/I provides five groups of alternative file attributes. Each group (except scope, which is discussed in section I, "Attributes") is discussed individually. Following is a list of the groups.

Group Type	Alternative Attributes	Default Attribute
Usage	STREAM|RECORD	STREAM
Function	INPUT|OUTPUT|UPDATE	INPUT
Access	SEQUENTIAL|DIRECT|TRANSIENT	SEQUENTIAL
Buffering	BUFFERED|UNBUFFERED	BUFFERED
Scope	EXTERNAL|INTERNAL	EXTERNAL

The scope attributes are discussed in detail in section I, "Attributes."

STREAM and RECORD Attributes

The STREAM and RECORD attributes describe the type of data transmission (stream-oriented or record-oriented) to be used in input and output operations for the file.

The STREAM attribute causes a file to be treated as a continuous stream of data items recorded only in character form.

The RECORD attribute causes a file to be treated as a sequence of records, each record consisting of one or more data items recorded in any internal form.

```
DECLARE MASTER FILE RECORD,
        DETAIL FILE STREAM;
```

INPUT, OUTPUT, and UPDATE Attributes

The function attributes determine the direction of data transmission permitted for a file. The INPUT attribute applies to files that are to be read only. The OUTPUT attribute applies to files that are to create or, in some cases, extend data sets. The UPDATE attribute (which applies only to RECORD files) describes a file that is to be used for both input and output; it allows records to be inserted into an existing data set and other records already in that data set to be altered.

SEQUENTIAL, DIRECT and TRANSIENT Attributes

The access attributes apply only to a file with the RECORD attribute, and describe how the records in the file are to be accessed.

The SEQUENTIAL attribute specifies that records in the data set are to be accessed in physical sequence or in key sequence order.

The DIRECT attribute specifies that records in a data set may be accessed in any order. The location of the record in the data set is determined by a character-string "key"; therefore, the DIRECT attribute implies the KEYED attribute. The associated data set must be in a direct-access volume.

The TRANSIENT attribute applies to files used for teleprocessing applications. A TRANSIENT file is associated with a data set which consists of a queue of messages. The message queue data set contains messages originating from and destined for remote terminals while in transit between a message control program and the PI/I message processing program. The action of reading a record removes that record from the data set. Access is sequential, but the file must have the KEYED attribute since a key is used to identify the terminal concerned; a buffer is always used, and so the file must also have the BUFFERED attribute. Teleprocessing is discussed in chapter 12, "Record-Oriented Transmission."

BUFFERED and UNBUFFERED Attributes

The buffering attributes apply only to a file that has either the SEQUENTIAL or TRANSIENT, and RECORD attributes. The BUFFERED attribute indicates that records transmitted to and from a file must pass through an intermediate internal-storage area. If BUFFERED is specified, data transmission is, in most cases, overlapped automatically with processing.

The UNBUFFERED attribute indicates that a record in a data set need not pass through a buffer but may be transmitted directly to and from the main storage associated with a variable. When UNBUFFERED is specified, data transmission is not overlapped automatically with processing; the programmer must use the EVENT option to achieve such overlapping.

Note: Specification of UNBUFFERED does not preclude the use of buffers. In nearly all cases, "hidden buffers" are required. These cases are listed in the discussion of the BUFFERED and UNBUFFERED attributes in section I, "Attributes."

ADDITIVE ATTRIBUTES

The additive attributes are:

PRINT

BACKWARDS

KEYED

EXCLUSIVE

ENVIRONMENT (option-list)

PRINT Attribute

The PRINT attribute applies only to files with the STREAM and OUTPUT attributes. It indicates that the file is eventually to be printed, that is, the data associated with the file is to appear on printed pages, although it may first be written on some other medium. The PRINT attribute causes the initial byte of each record of the associated data set to be reserved for a printer control character.

BACKWARDS Attribute

The BACKWARDS attribute applies only to SEQUENTIAL RECORD INPUT files and only to data sets on magnetic tape. It indicates that a file is to be accessed in reverse order, beginning with the last record and proceeding through the file until the first record is accessed.

KEYED Attribute

The KEYED attribute applies only to files with the RECORD attribute. It indicates that records in the file can be accessed using one of the key options (KEY, KEYTO, or KEYFROM) of data transmission statements or of the DELETE statement. Note that the KEYED attribute does not necessarily indicate that the actual keys exist on, or are to be written in, or are to be read from the data set; consequently, it need not be specified unless one of the key options is to be used. The nature and use of keys is discussed in detail in chapter 12, "Record-Oriented Transmission."

EXCLUSIVE Attribute

When access to a record is restricted to one task, the record is said to be locked by that task. The EXCLUSIVE attribute, which can be specified for DIRECT UPDATE or INPUT files only, provides a temporary locking mechanism to prevent one task from interfering with an operation by another task. It can be suppressed by the NOLOCK option on the READ statement. Figure 10.1 shows the effects of various operations on an EXCLUSIVE file.

The EXCLUSIVE attribute will also lock a record on a data set that is shared between two PL/I jobs in a multi-programming environment. The effect is as for sharing between two tasks.

ENVIRONMENT Attribute

The ENVIRONMENT attribute provides information that allows the compiler to determine the method of accessing the data associated with a file. It specifies the physical organization of the data set that will be associated with the file, and indicates how the data set is to be handled.

The general format of the ENVIRONMENT attribute is

ENVIRONMENT (option-list)

The ENVIRONMENT attribute can be given in a file declaration or as an option of the CLOSE statement. When ENVIRONMENT is specified in a CLOSE statement, the only option allowed is LEAVE or REREAD.

The options appropriate to the two types of data transmission are described in chapter 11, "Stream-Oriented Transmission," and chapter 12, "Record-Oriented Transmission," both in Part I.

Attempted Operation	Current State of Addressed Record		
	Unlocked	Locked by this task	Locked by another task
READ NOLOCK	Proceed	Proceed	Wait for unlock
READ	1. Lock record 2. Proceed	Proceed	Wait for unlock
DELETE/REWRITE	1. Lock record 2. Proceed 3. Unlock[1] record	1. Proceed 2. Unlock[1] record	Wait for unlock
UNLOCK	No effect	Unlock record	No effect
CLOSE FILE	Raise ERROR if there are records locked by another task. Otherwise, unlock all records locked in this task, and proceed with closing.		
Terminate Task	Unlock all records locked by task. Close file, if opened in this task		

[1]The unlocking occurs at the end of the operation, on completion of any on-units entered because of the operation (that is, at the corresponding WAIT statement when the EVENT option has been specified). If the EVENT option has been specified with a READ statement, the operation is not completed until the corresponding WAIT statement is reached; in the meantime, no attempt to delete or rewrite the record should be made.

Figure 10.1. Effect of operations on EXCLUSIVE files

Opening and Closing Files

Before the data associated with a file can be transmitted by input or output statements, certain file preparation activities must occur, such as checking for the availability of external storage media, positioning the media, and allocating appropriate operating system support. Such activity is known as opening a file. Also, when processing is completed, the file must be closed. Closing a file involves releasing the facilities that were established during the opening of the file.

PL/I provides two statements, OPEN and CLOSE, to perform these functions. These statements, however, are optional. If an OPEN statement is not executed for a file, the file is opened automatically before the first data transmission statement for that file is executed; in this case, the automatic file preparation consists of standard system procedures that use information about the file as specified in a DECLARE statement (or assumed from a contextual declaration derived from the transmission statement). Similarly, if a file has not been closed before completion of the task in which the file was opened, the file is closed automatically upon completion of the task.

When a file for stream input, sequential input, or sequential update is opened, the associated data set is positioned at the first record. When a BACKWARDS file is opened, the associated data set is positioned at the last record.

OPEN Statement

Execution of an OPEN statement causes one or more files to be opened explicitly. The OPEN statement has the following basic format:

```
OPEN FILE(file-expression) [option group]
   [,FILE(file-expression) [option
      group]]...;
```

The option list of the OPEN statement can specify any of the alternative and additive attributes, except ENVIRONMENT, INTERNAL, and EXTERNAL. Attributes included as options in the OPEN statement are merged with those stated in a DECLARE statement. The same attributes need not be listed in both an OPEN statement and a DECLARE statement for the same file, and, of course, there must be no conflict. Other options that can only appear in the OPEN statement are the TITLE option, used to associate the file with the data set, and the PAGESIZE and LINESIZE options, used to

specify the layout of a data set. The TITLE option is discussed below under "Associating Data Sets with Files," and the PAGESIZE and LINESIZE options, which apply only to STREAM files, in chapter 11, "Stream-oriented Transmission." The option list may precede the FILE (file expression) specification.

The OPEN statement is executed by library routines that are loaded dynamically at the time the OPEN statement is executed. Consequently, execution time can be reduced if more than one file is specified in the same OPEN statement, since the routines need be loaded only once, regardless of the number of files being opened. Note, however, that such multiple opening may require temporarily more internal storage than might otherwise be needed.

For a file to be opened explicitly, the OPEN statement must be executed before any of the input and output statements listed below in "Implicit Opening" are executed for the file.

Implicit Opening

An implicit opening of a file occurs when one of the statements listed below is executed for a file for which an OPEN statement has not already been executed. The type of statement determines which unspecified alternatives are applied to the file when it is opened.

The following list contains the statement identifiers and the attributes deduced from each:

Statement Identifier	Attributes Deduced
GET	STREAM, INPUT
PUT	STREAM, OUTPUT
READ	RECORD, INPUT
WRITE	RECORD, OUTPUT
LOCATE	RECORD, OUTPUT, SEQUENTIAL, BUFFERED
REWRITE	RECORD, UPDATE
DELETE	RECORD, UPDATE
UNLOCK	RECORD, DIRECT, UPDATE, EXCLUSIVE

Notes:

1. INPUT and OUTPUT are deduced from READ and WRITE only if UPDATE has not been explicitly declared.
2. If a GET statement contains a COPY option, execution of the GET statement causes implicit opening of either the specified file as a STREAM OUTPUT file or the standard output file SYSPRINT.

An implicit opening caused by one of the above statements is equivalent to preceding the statement with an OPEN statement that specifies the deduced attributes.

Merging of Attributes

There must be no conflict between the attributes specified in a file declaration and the attributes merged as the result of opening the file. For example, the attributes INPUT and UPDATE are in conflict, as are the attributes UPDATE and STREAM.

After the attributes are merged, the attribute implications listed below are applied prior to the application of the default attributes discussed earlier. Implied attributes can also cause a conflict. If a conflict in attributes exists after the application of default attributes, the UNDEFINEDFILE condition is raised.

Following is a list of merged attributes and attributes that each implies after merging:

Merged Attributes	Implied Attributes
UPDATE	RECORD
SEQUENTIAL	RECORD
DIRECT	RECORD, KEYED
BUFFERED	RECORD, SEQUENTIAL
UNBUFFERED	RECORD, SEQUENTIAL
PRINT	OUTPUT, STREAM
BACKWARDS	RECORD, SEQUENTIAL, INPUT
KEYED	RECORD
EXCLUSIVE	RECORD

The following two examples illustrate attribute merging for an explicit opening using a file constant and a file variable.

File constant:

```
DECLARE LISTING FILE STREAM;
   .
   .
   .
OPEN FILE(LISTING) PRINT;
```

Attributes after merge due to execution of the OPEN statement are STREAM and PRINT. Attributes after implication are STREAM, PRINT, and OUTPUT. Attributes after default application are STREAM, PRINT, OUTPUT, and EXTERNAL.

File variable:

```
DECLARE ACCOUNT FILE VARIABLE,
        (ACCT1,ACCT2,...)  FILE
              OUTPUT;
            .
            .
            .
ACCOUNT = ACCT1;
OPEN FILE(ACCOUNT) PRINT;
            .
            .
            .
ACCOUNT = ACCT2;
OPEN FILE(ACCOUNT) RECORD UNBUFFERED;
```

The file ACCT1 has been opened with attributes (by explicit and implicit declaration) STREAM, EXTERNAL, PRINT, and OUTPUT. The file ACCT2 has been opened with attributes RECORD, EXTERNAL, OUTPUT, SEQUENTIAL, and UNBUFFERED.

The following example illustrates implicit opening.

```
DECLARE MASTER FILE KEYED INTERNAL
        ENVIRONMENT (INDEXED F
        RECSIZE(120) KEYLEN(8));
        .
        .
        .
READ FILE (MASTER) INTO
   (MASTER_RECORD) KEYTO(MASTER_KEY);
```

Attributes after merge due to the opening caused by execution of the READ statement are KEYED, INTERNAL, RECORD, and INPUT. Attributes after implication are KEYED, INTERNAL, RECORD, and INPUT (no additional attributes are implied). Attributes after default application are KEYED, INTERNAL, RECORD, INPUT, SEQUENTIAL, and BUFFERED.

Associating Data Sets with Files

With batch processing under the OS, the association of a file with a specific data set is accomplished using job control language, outside the PL/I program. At the time a file is opened, the PL/I file name is associated with the name (ddname) of a data definition statement (DD statement), which is, in turn, associated with the name of a specific data set (dsname). Note that the direct association is with the name of a DD statement, not with the name of the data set itself.

A ddname can be associated with a PL/I file either through the file name or through the character-string value of the expression in the TITLE option of the associated OPEN statement.

If a file is opened implicitly, or if no TITLE option is included in the OPEN statement that causes explicit opening of the file, the ddname is assumed to be the same as the file name. If the file name is longer than eight characters, the ddname is assumed to be composed of the first eight characters of the file name.

Note: Since external names are limited to seven characters, an external file name of more than seven characters is shortened into a concatenation of the first four and the last three characters of the file name. Such a shortened name is not, however, the name used as the ddname in the associated DD statement.

Consider the following statements:

```
1.  OPEN FILE(MASTER);

2.  OPEN FILE(OLDMASTER);

3.  READ FILE(DETAIL)...;
```

When statement number 1 is executed, the file name MASTER is taken to be the same as the ddname of a DD statement in the current job step. When statement number 2 is executed, the name OLDMASTE is taken to be the same as the ddname of a DD statement in the current job step. (The first eight characters of a file name form the ddname. Note, that if OLDMASTER is an external name, it will be shortened by the compiler to OLDMTER for use within the program.) If statement number 3 causes implicit opening of the file DETAIL, the name DETAII is taken to be the same as the ddname of a DD statement in the current job step.

In each of the above cases, a corresponding DD statement must appear in the job stream; otherwise, the UNDEFINEDFILE condition would be raised. The three DD statements would appear, in part, as follows:

```
1.  //MASTER   DD DSNAME=...

2.  //OLDMASTE DD DSNAME=...
```

3. //DETAIL DD DSNAME=...

If a file is opened explicitly by an OPEN statement that includes a TITLE option, the ddname is taken from the TITLE option, and the file constant is not used outside the program. The TITLE option appears in an OPEN statement in the following format:

```
OPEN FILE(file-expr) TITLE(expression);
```

The expression in the TITLE option is evaluated and, if necessary, converted to a character string, which is assumed to be the ddname identifying the appropriate data set. If the character string is longer than eight characters, only the first eight characters are used. The following OPEN statement illustrates how the TITLE option might be used:

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

If this statement were executed, there must be a DD statement in the current job step with DETAIL1 as its ddname. It might appear, in part, as follows:

```
//DETAIL1  DD  DSNAME=DETAILA,...
```

Thus, the data set DETAILA is associated with the file DETAIL through the ddname DETAIL1.

Although a data set name represents a specific collection of data, the file name can, at different times, represent entirely different data sets. In the above example of the OPEN statement, the file DETAIL1 is associated with the data set named in the DSNAME parameter of the DD statement DETAIL1. If the file were closed and reopened, a TITLE option specifying a different ddname could be used, and then the file could be associated with a different data set.

If the file expression in the statement which explicitly or implicitly opens the file is not a file constant, then the DD statement name must be the same as the value of the file expression. The following example illustrates how a DD statement should be associated with the value of a file variable.

```
PRICES = RPRICE;

OPEN FILE(PRICES);
```

The DD card should associate the data set with the file constant RPRICE, which is the value of the file variable PRICES, thus:

```
//RPRICE DD DSNAME=...
```

Use of the TITLE option allows a programmer to choose dynamically, at open time, one among several data sets to be associated with a particular file name. Consider the following example:

```
DECLARE 1 INREC, 2 FIELD_1...,
          2 FILE_IDENT CHARACTER(8),
        DETAIL FILE INPUT...,
        MASTER FILE INPUT...;

OPEN FILE(DETAIL);

READ FILE (DETAIL) INTO (INREC);

OPEN FILE (MASTER) TITLE(FILE_IDENT);
```

Assume that the program containing these statements is used to process several different detail data sets, each of which has a different corresponding master data set. Assume, further, that the first record of each detail data set contains, as its last data item, a character string that identifies the appropriate master data set. The following DD statements might appear in the current job step:

```
//DETAIL    DD DSNAME=...

//MASTER1A  DD DSNAME=MASTER1A...

//MASTER1B  DD DSNAME=MASTER1B...

//MASTER1C  DD DSNAME=MASTER1C...
```

In this case, MASTER1A, MASTER1B, and MASTER1C represent three different master files. The first record of DETAIL would contain as its last item, either 'MASTER1A', 'MASTER1B', or 'MASTER1C', which is assigned to the character-string variable FILE_IDENT. When the OPEN statement is executed to open the file MASTER, the current value of FILE_IDENT would be taken to be the ddname, and the appropriate data set identified by that ddname would be associated with the file name MASTER.

Another similar use of the TITLE option is illustrated in the following statements:

```
DCL IDENT(3) CHAR(1)
    INIT('A', 'B', 'C');
DO I = 1 TO 3;
    OPEN FILE(MASTER)
         TITLE('MASTER1'||IDENT(I));
      .
      .
      .
    CLOSE FILE(MASTER);
END;
```

In this example, IDENT is declared as a character-string array with three elements having as values the specific character strings 'A', 'B', and 'C'. When MASTER is

opened during the first iteration of the DO-group, the character constant 'MASTER1' is concatenated with the value of the first element of IDENT, and the associated ddname is taken to be MASTER1A. After processing, the file is closed, dissociating the file name and the ddname. During the second iteration of the group, MASTER is opened again. This time, however, the value of the second element of IDENT is taken, and MASTER is associated with the ddname MASTER1B. Similarly, during the final iteration of the group, MASTER is associated with the ddname MASTER1C.

Note: The character set of the job control language does not contain the break character (_). Consequently, this character cannot appear in ddnames. Care should thus be taken to avoid using break characters among the first eight characters of file names, unless the file is to be opened with a TITLE option with a valid ddname as its expression. The alphabetic extender characters $, @, and #, however, are valid for ddnames, but the first character must be one of the letters A through Z.

Use of a file variable also allows a number of files to be manipulated at various times by a single statement. For example:

```
DECLARE F FILE VARIABLE,
        A FILE,
        B FILE,
        C FILE;
          .
          .
        F=A;
LAB:    READ FILE (F) .....;
          .
          .
        F=B;
        GO TO LAB;
          .
          .
        F=C;
        GO TO LAB;
```

The statement labeled LAB is used to read the three files A, B, and C, each of which may be associated with a different data set. Note that the files A, B, and C remain open after the READ statement has been executed in each instance. When a number of data sets is to be accessed by a single statement, use of a file variable rather than the TITLE option may improve program efficiency by allowing a file for each data set to remain open for as long as it is required by the program. Using the TITLE option could necessitate closing and reopening a file whenever it is to be associated with a new data set.

CLOSE Statement

The basic form of the CLOSE statement is:

```
CLOSE FILE (file-expr) [ENVIRONMENT
  ({LEAVE|REREAD})]
    [,FILE (file-expr) [ENVIRONMENT
      ({LEAVE|REREAD})]]...;
```

Executing a CLOSE statement dissociates the specified file from the data set with which it became associated when the file was opened. The CLOSE statement also dissociates from the file all attributes established for it by the implicit or explicit opening process. If desired, new attributes may be specified for the file constant in a subsequent OPEN statement. However, all attributes explicitly given to the file constant in a DECLARE statement remain in effect.

As with the OPEN statement, closing more than one file with a single CLOSE statement can save execution time, but it may require the temporary use of more internal storage than would otherwise be needed.

The LEAVE and REREAD options are used to control the disposition of magnetic tapes.

Note: Closing an already closed file or opening an already opened file has no effect apart from increasing the execution time of the program.

STANDARD FILES

Two standard files are provided that can be used by any PL/I program. One is the standard input file SYSIN, and the other is the standard output file SYSPRINT. These files need not be declared or opened explicitly; a standard set of attributes is applied automatically. For SYSIN, the attributes are STREAM INPUT, and for SYSPRINT they are STREAM OUTPUT PRINT. Both file names, SYSIN and SYSPRINT, are assumed to have the EXTERNAL attribute, even though SYSPRINT contains more than seven characters.

The FILE option need not be specified in GET and PUT statements when these files are to be used. GET and PUT statements that do not name a file are equivalent to:

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

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

Any other references to SYSIN and SYSPRINT (such as in ON statements or in record-oriented statements) must be stated explicitly.

Under the optimizing compiler, the identifiers SYSIN and SYSPRINT are not reserved for the specific purposes described above. They can be used for other purposes besides identifying standard files. Other attributes can be applied to them, either explicitly or contextually, but the PRINT attribute is applied automatically to SYSPRINT when it is declared or opened as a STREAM OUTPUT file unless the INTERNAL attribute is declared for it.

Under the checkout compiler, the file SYSPRINT is used for diagnostic messages, and the file SYSIN may be used to hold the source program. When the compiler uses one of the files, the file is opened with certain attributes that may not be altered; the programmer consequently needs to exercise care if he declares SYSPRINT or SYSIN explicitly. Full details of the restrictions are given in the OS PL/I Checkout Compiler: Programmer's Guide.

Even under the optimizing compiler, care must be taken when SYSIN or SYSPRINT is declared as anything other than a STREAM file. The compiler causes, in effect, the identifier SYSIN to be inserted into each GET statement in which no file constant is explicitly stated and the identifier SYSPRINT to be inserted into each PUT statement in which no file constant is explicitly stated. Consequently, the following would be in error:

```
DECLARE (SYSIN,SYSPRINT) FIXED
     DECIMAL (4,2);
          .
          .
          .
GET LIST (A,B,C);
PUT LIST (D,E,F);
```

The identifier SYSIN would be inserted into the GET statement, and SYSPRINT in the PUT statement. In this case, however, they would not refer to the standard files, but to the fixed-point variables declared in the block.

Chapter 11: Stream-Oriented Transmission

Introduction

This chapter describes the input and output statements used in stream-oriented transmission. Those features that apply equally to stream-oriented and record-oriented transmission, including files, file attributes, and opening and closing files, are described in chapter 10, "Input and Output".

In stream-oriented transmission, a data set is treated as a continuous stream of data items in character form; within a program, block and record boundaries are ignored. However, a data set is considered to consist of a series of lines of data, and each data set that is created or accessed by stream-oriented transmission has a line size associated with it. In general, a line is equivalent to a record in the data set; however, the line size does not necessarily equal the record size.

There are three modes of stream-oriented transmission: list-directed, data-directed, and edit-directed. The transmission statements used in all three modes require the following information:

1. The name of the file associated with the data set from which data is to be obtained or to which data is to be assigned.
2. A list of program variables to which data items are to be assigned during input or from which data items are to be obtained during output. This list is called a data list. On output in list- and edit-directed modes, the data list can also include expressions.
3. For edit-directed mode, the format of each data item in the stream.

Under certain conditions some of this required information can be implied.

LIST-DIRECTED TRANSMISSION

List-directed transmission permits the user to read and write out data without having to specify the format of the items in the stream.

Input: In general, the data items in the stream are character strings in the form of optionally signed valid constants or in the form of expressions that represent complex constants. The variables to which the data items are to be assigned are specified by a data list. Items are separated by a comma and/or one or more blanks.

Output: The data values to be transmitted are specified by a variable, a constant, or an expression that represents a data item. Each data item placed in the stream is a character-string representation that reflects the attributes of the variable. Items are separated by one or more blanks. Leading zeros of arithmetic data are suppressed. Binary items are expressed in decimal representation.

For PRINT files, data items are automatically aligned on implementation-defined preset tab positions. These positions are 1, 25, 49, 73, 97, and 121, but provision is made for the programmer to alter these values.

DATA-DIRECTED TRANSMISSION

Data-directed transmission permits the user to transmit self-identifying data.

Input: Each data item in the stream is in the form of an assignment that specifies both the value and the variable to which it is to be assigned. In general, values are in the form of constants. Items are separated by a comma and/or one or more blanks. A semicolon must end each group of items to be accessed by a single GET statement. A data list in the GET statement is optional, since the semicolon determines the number of items to be obtained from the stream.[1]

Output: The data values to be transmitted may be specified by an optional data list. Each data item placed in the stream has the form of an assignment statement without a semicolon. Items are separated by one or

---

[1] These rules are slightly amended when the program is initiated and data entered from a terminal under TSO. Details are given in the following OS publications: Time Sharing Option: PL/I Optimizing Compiler and Time Sharing Option: PL/I Checkout Compiler.

more blanks. The last item transmitted by each PUT statement is followed by a semicolon. Leading zeros of arithmetic data are suppressed. The character representation of each value reflects the attributes of the variable, except for binary items, which appear as values expressed in decimal notation.

If the data list is omitted, it is assumed to specify all variables that are known within the block containing the statement and are permitted in data-directed output.

For PRINT files, data items are automatically aligned on the implementation-defined preset tab positions referred to under "List-Directed Transmission".

EDIT-DIRECTED TRANSMISSION

Edit-directed transmission permits the user to specify the variables to which data is to be assigned or to specify data to be transmitted, and to specify the format for each item on the external medium.

Input: Data in the stream is a continuous string of characters; different data items are not separated. The variables to which the data is to be assigned are specified by a data list. Format items in a format list specify the number of characters that contain the value to be assigned to each variable and describe characteristics of the data (for example, the assumed location of a decimal point).[2]

Output: The data values to be transmitted are defined by a data list. The format that the data is to have in the stream is defined by a format list.

Data Transmission Statements

Stream-oriented trahsmission uses only one input statement, GET, and one output statement, PUT. A GET statement gets the next series of data items from the stream, and a PUT statement puts a specified set of data items into the stream. The variables to which data items are assigned, and the variables or expressions from which they are transmitted, are generally specified in a data list with each GET or PUT statement. The statements may also include options that specify the origin or destination of the data or indicate where it appears in the stream relative to the preceding data.

The following is a summary of the stream-oriented data transmission statements and their options:

STREAM INPUT:

```
GET [{FILE (file-expression)} |{STRING
     (character-string-variable)}]
    [data-specification]
    [COPY[(file-expression)]]
    [SKIP [(expression)]];
```

Note that neither the COPY option nor SKIP option can be used with the STRING option in a GET statement.

STREAM OUTPUT:

```
PUT [{FILE (file-expression)} |{STRING
     (character-string-variable)}]
    [data-specification]
    [SKIP [(expression)]];
```

Note that the SKIP option cannot be used with the STRING option in a PUT statement.

STREAM OUTPUT PRINT:

```
PUT [FILE (file-expression) ]
    [data-specification]
      [PAGE[LINE(expression)]]
      [SKIP[(expression)]    ]
      [LINE (expression)     ];
```

The options may appear in any order. The data specification can have one of the following forms:

```
[LIST] (data-list)

DATA [(data-list)]

EDIT {(data-list) (format-list)}...

SNAP

FLOW

ALL [(character-string-expression)]
```

If a GET or PUT statement includes a data list that is not preceded by one of the keywords LIST,DATA, or EDIT, then LIST is assumed. In such a statement, the data list must immediately follow the GET or PUT keyword; any options required must be specified after the data list.

---

[2]These rules are slightly amended when the program is initiated and data entered from a terminal under TSO. Details are given in the following OS publications: OS Time Sharing Option: PL/I Optimizing Compiler and OS Time Sharing Option: PL/I Checkout Compiler.

The SNAP, FLOW and ALL options in the data specification cause information about the program to be put into the stream. These options can only be used in a PUT statement. The information is provided only if the PUT statement is processed by the PL/I checkout compiler; if such a PUT statement is included in a program that is processed by the PL/I optimizing compiler, these options are checked for syntax errors and then ignored. The use of the options is described in chapter 15, "Execution-Time Facilities of the PL/I Checkout Compiler".

The data specification can be omitted only if one of the control options (PAGE, SKIP, or LINE) appears. Format lists may use any of the following format items:

A,B,C,E,F, P,R,X	which may be used with any STREAM or STRING option
SKIP[(w)] COLUMN (w)	which may be used with any STREAM file
PAGE LINE (w)	which may be used with STREAM OUTPUT PRINT files

The statements are discussed individually in detail in section J, "Statements".

Options of Transmission Statements

FILE and STRING Options

The FILE option specifies the file upon which the operation is to take place. The STRING option allows GET and PUT statements to be used to transmit data between internal storage locations rather than between internal and external storage. If neither the FILE option nor the STRING option appears in a GET statement, the standard input file SYSIN is assumed; if neither option appears in a PUT statement, the standard output file SYSPRINT is assumed.

Examples of the use of the FILE option are given in some of the statements below. Chapter 13, "Editing and String Handling", illustrates the use of the STRING option.

COPY Option

The COPY option may appear only in a GET FILE statement. It specifies that the stream is to be written, exactly as read, onto the file named in the COPY specification. If no file name is given, the default is the standard output file SYSPRINT. For example, the statement:

```
GET FILE(SYSIN) DATA(A,B,C) COPY(DPL);
```

not only transmits the values assigned to A, B, and C in the input stream to the variables with these names, but also causes them to be written, exactly as they appear in the input stream, on the file DPL. If they were written, by default, on the SYSPRINT file, they would appear in data-directed format. Data items that are skipped on input, and not transmitted to internal variables, are copied intact into the output stream.

SKIP Option

The SKIP option specifies a new current line (or record) within the data set. The parenthesized expression is converted to an integer w. The data set is positioned to the start of the wth line (record) relative to the current line (record).

For non-PRINT files, if the expression is omitted or if w is not greater than zero, a value of 1 is assumed. For PRINT files, if w is less than or equal to zero, the effect is that of a carriage return with the same current line; if the expression is omitted, 1 is assumed.

The SKIP option takes effect before the transmission of any values defined by the data specification, even if it appears after the data specification. Thus, the statement:

```
PUT LIST(X,Y,Z) SKIP(3);
```

causes the values of the variables X, Y, and Z to be printed on the standard output file SYSPRINT commencing on the third line after the current line.

When printing at a terminal in conversational mode, SKIP(w) with w greater than 3 is equivalent to SKIP(3). In other words, no more than three lines may be skipped.

PAGE Option

The PAGE option can be specified only for PRINT files. It causes a new current page to be defined within the data set. The PAGE option takes effect before the transmission of any values defined by the data specification (if any), even if it appears after the data specification.

When printing at a terminal in conversational mode, the PAGE option causes three lines to be skipped.

LINE Option

The LINE option can be specified only for PRINT files. It causes blank lines to be inserted so that the next line will be the wth line of the current page, where w is the value of the parenthesized expression when converted to an integer. The LINE option takes effect before the transmission of any values defined by the data specification (if any), even if it follows the data specification. If both the PAGE option and the LINE option appear in the same statement, the PAGE option is applied first. For example, the statement

```
PUT FILE(LIST) DATA(P,Q,R) LINE(34) PAGE;
```

causes the values of the variables P, Q, and R to printed in data-directed format on a new page, commencing at line 34.

When printing at a terminal in conversational mode, the LINE option always causes three lines to be skipped.

Data Specifications

Data specifications are given in GET and PUT statements to identify the data to be transmitted.

DATA LISTS

List-directed and edit-directed data specifications require a data list to specify the data items to be transmitted. A data list is optional for a data-directed data specification.

General format:

```
(data-list)
```

where "data list" is defined as:

```
element [,element]...
```

Syntax rules:

The nature of the elements depends upon whether the data list is used for input or for output. The rules are as follows:

1. On input, a data-list element for edit-directed and list-directed transmission can be one of the following: an element, array, or structure variable, a pseudovariable other than STRING, or a repetitive specification (similar to a repetitive specification of a DO group) involving any of these elements. For a data-directed data specification, a data-list element can be an element, array, or structure variable. None of the names in a data-directed data list can be subscripted, locator-qualified, or iSUB-defined, but qualified (that is, structure-member), simple-defined, or string-overlay-defined names are allowed.

2. On output, a data-list element for edit-directed and list-directed data specifications can be one of the following: an element expression, an array expression, a structure expression, or a repetitive specification involving any of these elements. For a data-directed data specification, a data-list element can be an element, array, or structure variable, or a repetitive specification involving any of these elements. It must not be locator-qualified or iSUB-defined, but may be qualified (that is, a member of a structure), or simple- or string-overlay-defined. Subscripts are allowed for data-directed output.

3. The elements of a data list can be:

Input:	Problem data:	Arithmetic String
Output:	Problem data:	Arithmetic String
	Program control data:	Area Entry Event File Label Offset Pointer Task

Entry and label constants may not be specified.

A data list that specifies program-control data can only be used in PUT DATA or PUT LIST statements that are to be processed by the checkout compiler or PUT DATA statements that are to be processed under the optimizing compiler. In the latter case, the name of the variable is transmitted, but not its value.

4. A data list must always be enclosed in parentheses.

Repetitive Specification

The general format of a repetitive specification is shown in figure 11.1.

Syntax rules:

1. An element in the element list of the repetitive specification can be any of those allowed as data-list elements as listed above.

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

   a. Each expression in the specification is an element expression.

   b. In the specification, expression-1 represents the starting value of the control variable or pseudovariable. Expression-3 represents the increment to be added to the control variable after each repetition of data-list elements in the repetitive specification. Expression-2 represents the terminating value of the control variable. Expression-4 represents a second condition to control the number of repetitions. The exact meaning of the specification is identical to that of a DO statement with the same specification. When the last specification is completed, control passes to the next element in the data list.

3. Each repetitive specification must be enclosed in parentheses, as shown in the general format. Note that if a repetitive specification is the only element in a data list, two sets of outer parentheses are required, since the data list must have one set of parentheses and the repetitive specification must have a separate set.

4. As figure 11.1 shows, the "specification" portion of a repetitive specification can be repeated a number of times, as in the following form:

```
DO I = 1 TO 4, 6 TO 10;
```

   Repetitive specifications can be nested; that is, an element of a repetitive specification can itself be a repetitive specification. Each DO portion must be delimited on the right with a right parenthesis (with its matching left parenthesis added to the beginning of the entire repetitive specification).

   When DO portions are nested, the rightmost DO is at the outer level of nesting. For example, consider the following statement:

```
GET LIST (((A(I,J) DO I = 1 TO 2)
         DO J = 3 TO 4));
```

   Note the three sets of parentheses, in addition to the set used to delimit the subscript. The outermost set is the set required by the data list; the next is that required by the outer repetitive specification. The third set of parentheses is that required by the inner repetitive specification. This statement is equivalent to the following nested DO-groups:

```
DO J = 3 TO 4;
   DO I = 1 TO 2;
   GET LIST (A (I,J));
   END;
END;
```

   It gives values to the elements of the array A in the following order:

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

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

A "specification" has the following format:

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

Figure 11.1. General format for repetitive specifications

Under the optimizing compiler, the maximum permissible level of nesting is 50. There is no such limit under the checkout compiler.

Note: Although the DO keyword is used in the repetitive specification, a corresponding END statement is not allowed.

Transmission of Data-list Elements

If a data-list element is of complex mode, the real part is transmitted before the imaginary part.

If a data-list element is an array variable, the elements of the array are transmitted in row-major order, that is, with the rightmost subscript of the array varying most frequently.

If a data-list element is a structure variable, the elements of the structure are transmitted in the order specified in the structure declaration.

For example, if a declaration is:

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

then the statement:

```
PUT FILE(X) LIST(A);
```

would result in the output being ordered as follows:

```
A.B(1) A.C(1) A.B(2) A.C(2) A.B(3)
   A.C(3)...etc.
```

If, however, the declaration had been:

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

then the same PUT statement would result in the output being ordered as follows:

```
A.B(1) A.B(2) A.B(3)...A.B(10)
A.C(1) A.C(2) A.C(3)...A.C(10)
```

If, within a data list used in an input statement for list-directed or edit-directed transmission, a variable is assigned a value, this new value is used if the variable appears in a later reference in the data list. For example:

```
GET LIST (N,(X(I) DO I=1 TO N), J, K,
   SUBSTR (NAME, J,K));
```

When this statement is executed, data is transmitted and assigned in the following order:

1. A new value is assigned to N.
2. Elements are assigned to the array X as specified in the repetitive specification in the order X(1),X(2),...X(N), with the new value of N used to specify the number of items to be assigned.
3. A new value is assigned to J.
4. A new value is assigned to K.
5. A substring of length K is assigned to the string variable NAME, beginning at the Jth character.

List-directed Data Specification

General format for a list-directed data specification, either input or output is as follows:

```
[LIST] (data-list)
```

The data list is described under "Data Lists", above. The keyword LIST specifies the list-directed mode of transmission.

Examples of list-directed data specifications:

```
LIST (CARD, RATE, DYNAMIC_FLOW)

LIST ((THICKNESS(DISTANCE)
   DO DISTANCE = 1 TO 1000))

LIST (P, Z, M, R)

LIST (A*B/C, (X+Y)**2)
```

The specification in the last example can be used only for output, since it contains values specified by expressions. Such expressions are evaluated when the statement is executed, and the result is placed in the stream.

List-directed Data in the Stream

Problem data in the stream, either input or output, is of character data type and has one of the following general forms:

```
[+|-] arithmetic-constant

      character-string-constant

      bit-string-constant

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

A string constant must be one of the two permitted forms listed above; iteration and string repetition factors are not allowed. A blank must not follow a sign preceding a real constant, and must not precede or follow the central + or - in complex expressions.

The format of program control data is described in chapter 15, "Execution-time Facilities of the Checkout Compiler".

List-directed Input Format

When the data named is an array, the data consists of constants, the first of which is assigned to the first element of the array, the second constant to the second element, etc., in row-major order.

A structure name in the data list represents a list of the contained element variables and arrays in the order specified in the structure description.

On input, data items in the stream must be separated either by a blank or by a comma. This separator may be surrounded by an arbitary number of blanks. A null field in the stream is indicated either by the first non-blank character in the data stream being a comma, or by two commas separated by an arbitrary number of blanks. A null field specifies that the value of the associated item in the data list is to remain unchanged.

The transmission of the list of constants on input is terminated by expiration of the list or at the end of the file. In the former case, the file is positioned in the stream ready for the next GET statement. More than one blank can separate two data items, and a comma separator may be preceded or followed by one or more blanks. If the items are separated by a comma, then the first character to be scanned when the next GET statement is executed will be the one immediately following the comma. If the items are separated by blanks only, the first item scanned will be the next non-blank character. In the following example,s X represents a non-blank character, b represents a blank, and ↑ indicates the position of the file at the start of the next GET statement.

```
Xbb,bbbXXX
    ↑

XbbbbXXXX
     ↑

Xb,]←end of record
   ↑
```

Note that if a record terminates with a semi-colon, the succeeding record is not read in until the next GET statement requires it.

If the data is a character-string constant, the surrounding quotation marks are removed, and the enclosed characters are interpreted as a character string.

If the data is a bit-string constant, enclosing quotation marks and the trailing character B are removed, and the enclosed characters are interpreted as a bit string.

If the data is an arithmetic constant or complex expression, it is interpreted as coded arithmetic data with the base, scale, mode, and precision implied by the constant.

List-directed Output Format

The values of the element variables and expressions in the data list are converted to character representations and transmitted to the data stream. The conversions follow the normal rules for arithmetic to character conversions, except that floating-point items are not rounded.

A blank separates successive data items transmitted. (For PRINT files, items are separated according to program tab settings.)

The length of the data field placed in the stream is a function of the attributes of the data item, including precision and length. Detailed discussions of the conversion rules and their effect upon precision are listed in the descriptions of conversion to character type in section F, "Data Conversion and Expression Evaluation".

Binary data items are converted to decimal notation before being placed in the stream.

For numeric character values, the character-string value is transmitted.

Bit strings are converted to character representation of bit-string constants, consisting of the characters 0 and 1,

enclosed in quotation marks, and followed by the letter B.

Character strings are written out as follows. If the file does not have the attribute PRINT, enclosing quotation marks are supplied, and contained single quotation marks or apostrophes are replaced by two quotation marks. The field width is the current length of the string plus the number of added quotation marks. If the file has the attribute PRINT, enclosing quotation marks are not supplied, and contained single quotation marks or apostrophes are unmodified. The field width is the current length of the string.

Data-directed Data Specification

General format for a data-directed data specification, either for input or output, is as follows:

```
DATA[(data-list)]
```

General rules:

1. The data list is described in "Data Lists" in this chapter. For input, the data list cannot contain subscripted names. Names of structure elements in the data list need only have enough qualification to resolve any ambiguity; full qualification is not required. On input, if the stream contains an unrecognisable element-variable or a name that does not have a counterpart in the data list, the NAME condition is raised.

2. Omission of the data list implies that a data list is assumed. This assumed data list contains all the names that are known to the block and to any containing blocks, and that are valid for data-directed transmission.

   On input, if the stream contains an unrecognisable element-variable or an unknown name, the NAME condition is raised. If the assumed data list contains a name that is not included in the stream, the value of the associated variable remains unchanged.

   On output, all items in the assumed data list are transmitted. Where two or more blocks containing the PUT statement each have declarations of items which have the same name, all the items will be transmitted, the known item appearing first.

3. Recognition of a semicolon or an end-of-file in an input stream causes transmission to cease, whether or not a data list is specified. On output, a semicolon is written into the stream after the last data item transmitted by each PUT statement.

Data-directed Data in the Stream

The data in the stream associated with data-directed transmission is in the form of a list of element assignments. For problem data, they have the following general format (the optionally signed constants, like the variable names and the equal signs, are in character form):

```
element-variable = data value
   [{b|,}element-variable = data
   value]...;
```

General rules for problem data:

1. The element variable may be a subscripted name. Subscripts must be optionally signed decimal integer constants.

2. On input, the element assignments may be separated by either a blank (b in the above format) or a comma. Redundant blanks are ignored. On output, the assignments are separated by a blank. (For PRINT files, items are separated according to program tab settings.)

3. Each data value in the stream has one of the forms described for list-directed transmission.

4. On input a semi-colon following an element assignment terminates the list of element assignments to be transmitted by the execution of a single GET DATA statement, and thereby determines the number of element assignments that are actually transmitted by a particular statement. On output a semi-colon is transmitted on completion of a PUT DATA statement.

5. Locator qualifiers cannot appear in the stream. The locator qualifier declared with the based variable is used to establish the generation. Based variables that have not been declared with a locator qualifier cannot be transmitted.

   Under the optimizing compiler, the following restrictions apply to based variables in the data list:

   a. The variable must not be based on an OFFSET variable.

b. The variable must not be a member of a structure declared with the REFER option.

c. The pointer on which the variable is based must not be based, defined, or a parameter, and it must not be a member of an array or structure.

6. Under the optimizing compiler, defined variables in the data list must not have been defined:

   a. On a controlled variable.

   b. On an array with one or more adjustable bounds.

   c. With a POSITION attribute that specifies other than a constant.

Data-directed Data Specification for Input

General rules for data-directed input:

1. If the data specification does not include a data list, the names in the stream may be any names known at the point of transmission. Qualified names in the input stream must be fully qualified. The name must not contain more than 256 characters.

2. If a data list is used, each element of the data list must be an element, array, or structure variable. Names cannot be subscripted, but qualified names are allowed in the data list. All names in the stream should appear in the data list; however, the order of the names need not be the same, and the data list may include names that do not appear in the stream. For example, consider the following data list, where A, B, C, and D are names of element variables:

```
DATA (B, A, C, D)
```

   This data list may be associated with the following input data stream:

```
A= 2.5, B= .0047, D= 125, Z= 'ABC';
```

   Note: C appears in the data list but not in the stream; its value remains unaltered. Z, which is not in the data list, raises the NAME condition.

3. If the data list includes the name of an array, subscripted references to that array may appear in the stream although subscripted names cannot appear in the data list. The entire array need not appear in the stream; only those elements that actually appear in the stream will be assigned. If a subscript is out of range, or is missing, the NAME condition is raised.

   Let X be the name of a two-dimensional array declared as follows:

```
DECLARE X (2,3);
```

   Consider the following data list and input data stream:

Data Specification	Input Data Stream
DATA (X)	X(1,1)= 7.95, X(1,2)= 8085, X(1,3)= 73;

   Although the data list has only the name of the array, the associated input stream may contain values for individual elements of the array. In this case, only three elements are assigned; the remainder of the array is unchanged.

4. If the data list includes the names of structure elements, then fully qualified names must appear in the stream, although full qualification is not required in the data list. Consider the following structures:

```
DECLARE 1 CARDIN, 2 PARTNO, 2 DESCRP,
        2 PRICE, 3 RETAIL, 3 WHSL;
```

   If it is desired to read a value for CARDIN.PRICE.RETAIL, the data specification and input data stream could have the following forms:

Data Specification	Input Data Stream
DATA (CARDIN.RETAIL)	CARDIN.PRICE. RETAIL = 4.28;

5. Interleaved subscripts cannot appear in qualified names in the stream. All subscripts must be moved all the way to the right, following the last name of the qualified name. For example, assume that Y is declared as follows:

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

   An element name would have to appear in the stream as follows:

```
Y.A.B(2,3,8)= 8.72
```

   The name in the data list could not contain the subscript.

Data-directed Data Specification for Output

General rules for data-directed output:

1. An element of the data list may be an element, array, or structure variable, or a repetitive specification involving any of these elements or further repetitive specifications. Subscripted names can appear. For problem data, the names appearing in the data list, together with their values, are transmitted in the form of a list of element assignments separated by blanks and terminated by a semicolon. (For PRINT files, items are separated according to program tab settings.)

   The rules applying to program control data are given in chapter 15, "Execution-time Facilities of the Checkout Compiler."

2. Array variables in the data list are treated as a list of the contained subscripted elements in row-major order.

   Consider an array declared as follows:

   ```
   DECLARE X (2,4) FIXED;
   ```

   If X appears in a data list as follows:

   ```
   DATA (X)
   ```

   on output, the output data stream would have the form:

   ```
   X(1,1)= 1 X(1,2)= 2 X(1,3)= 3
   X(1,4)= 4 X(2,1)= 5 X(2,2)= 6
   X(2,3)= 7 X(2,4)= 8;
   ```

   Note: In actual output, more than one blank would follow the equal sign. In conversion from coded arithmetic to character, leading zeros are converted to blanks, and up to three additional blanks may appear at the beginning of the field.

3. Subscript expressions that appear in a data list are evaluated and replaced by their values.

4. Items that are part of a structure appearing in the data list are transmitted with the full qualification, but subscripts follow the qualified names rather than being interleaved. For example, if a data list is specified for a structure element transmitted under data-directed output as follows:

   ```
   DATA (Y(1,-3).Q)
   ```

   the associated data field in the output stream is of the form:

   ```
   Y.Q(1,-3)= 3.756;
   ```

5. Structure names in the data list are interpreted as a list of the contained element or elements, and any contained arrays are treated as above.

   For example, consider the following structure:

   ```
   1 A, 2 B, 2 C, 3 D
   ```

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

   ```
   DATA (A)
   ```

   and the values of B and D are 2 and 17, respectively, the associated data fields in the output stream would be as follows:

   ```
   A.B= 2 A.C.D= 17;
   ```

6. In the following cases, data-directed output is not valid for subsequent data-directed input:

   a. When the character-string value of a numeric character variable does not represent a valid optionally signed arithmetic constant. For example, this is always true for complex numeric character variables.

   b. When a program control variable is transmitted such a variable must not be specified in an input data list.

Length of Data-directed Output Fields

The length of the data field on the external medium is a function of the attributes declared for the variable and, since the name is also included, the length of the fully qualified subscripted name. The field length for output items converted from coded arithmetic data, numeric character data, and bit-string data is the same as that for list-directed output data, and is governed by the rules for data conversion to character type as described in section F, "Data Conversion and Expression Evaluation".

For character-string data, the contents of the character string are written out

```
AB:  PROCEDURE;
                                    Input Stream
     DECLARE (A(6), B(7)) FIXED;
                                    B(1)=1, B(2)=2, B(3)=3,
     GET FILE (X) DATA (B);
                                    B(4)=1, B(5)=2, B(6)=3, B(7)=4;
     DO I  = 1 TO 6;

     A (I) = B (I+1) + B (I);
                                    Output Stream
     END;
                                    A(1)= 3 A(2)= 5 A(3)= 4 A(4)= 3
     PUT FILE (Y) DATA (A);
                                    A(5)= 5 A(6)= 7;
     END AB;
```

Figure 11.2. Example of data-directed transmission (both input and output)

enclosed in quotation marks. Each quotation mark contained within the character string is represented by two successive quotation marks.

Example

In the example shown in figure 11.2, A is declared as a one-dimensional array of six elements; B is a one-dimensional array of seven elements. The procedure calculates and writes out values for A(I) = B(I+1) + B(I).

Edit-directed Data Specification

General format for an edit-directed data specification, either for input or output, is as follows:

```
EDIT {(data-list) (format-list)}
     [(data-list)(format-list)]...
```

1. The data list, which must be enclosed in parentheses, is described in "Data Lists", above. The format list, which also must be enclosed in parentheses, contains one or more format items. There are three types of format items: data format items, which describe data in the stream; control format items, which describe page, line, and spacing operations; and remote format items, which specify the label of a separate statement that contains the format list to be used. Format lists and format items are discussed in more detail in "Format Lists", below.

   Note: Program-control variables cannot be specified in data lists for edit-directed transmission.

2. For input, data in the stream is considered to be a continuous string of characters not separated into individual data items. The number of characters for each data item is specified by a format item in the format list. The characters are treated according to the associated format item.

3. For output, the value of each item in the data list is converted to a format specified by the associated format item and placed in the stream in a field whose width also is specified by the format item.

4. For either input or output, the first data format item is associated with the first item in the data list, the second data format item with the second item in the data list, and so forth. If a format list contains fewer format items than there are items in the associated data list, the format list is re-used; if there are excessive format items, they are ignored. Suppose a format list contains five data format items and its associated data list specifies ten items to be transmitted. Then the sixth item in the data list will be associated with the first data format item, and so forth. Suppose a format list contains ten data format items and its associated data list specifies only five items. Then the sixth through the tenth format items will be ignored.

5. An array or structure variable in a data list is equivalent to n items in the data list, where n is the number of element items in the array or structure, each of which will be associated with a separate use of a data format item.

6. If a control format item is encountered, the control action is executed, and the data list item is paired with the next format item.

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

8. On output, data items are not automatically separated, but arithmetic data items generally include leading blanks because of data conversion rules and zero suppression.

Examples:

```
GET EDIT (NAME, DATA, SALARY)
    (A(N), X(2), A(6), F(6,2));

PUT EDIT ('INVENTORY='||INUM,INVCODE)
    (A,F(5));
```

The first example specifies that the first N characters in the stream are to be treated as a character string and assigned to NAME; the next two characters are to be skipped; the next six are to be assigned to DATA in character format; and the next six characters are to be considered as an optionally signed decimal fixed-point constant and assigned to SALARY.

The second example specifies that the character string 'INVENTORY=' is to be concatenated with the value of INUM and placed in the stream in a field whose width is the length of the resultant string. Then the value of INVCODE is to be converted to character to represent an optionally signed decimal fixed-point integer constant and is then to be placed in the stream right-adjusted in a field with a width of five characters (leading characters may be blanks). Note that values represented by expressions and constants can appear in output data lists only.

Format Lists

Each edit-directed data specification must be associated with a format list.

General format:

```
(format-list)
```

where "format list" is defined as:

```
{ item                 } [ , item                 ]
{ n item               } [ , n item               ] ...
{ n (format-list)      } [ , n (format-list)      ]
```

Syntax rules:

1. Each "item" represents a format item as described below.

2. The letter n represents an iteration factor, which is either an expression enclosed in parentheses or an unsigned decimal integer constant. If it is the latter, a blank must separate the constant and the following format item. The iteration factor specifies that the associated format item or format list is to be used n successive times. A zero iteration factor specifies that the associated format item or format list is to be skipped and not used (the data list item will be associated with the next data format item). If an expression is used to represent the iteration factor, it is evaluated and converted to an integer, which must be non-negative, once for each set of iterations. The associated format item or format list is that item or list of items immediately to the right of the iteration factor.

General rule:

There are three types of format items: data format items, control format items, and the remote format item. Data format items specify the external forms that data fields are to take. Control format items specify the page, line, column, and spacing operations. The remote format item allows format items to be specified in a separate FORMAT statement elsewhere in the block.

Detailed discussions of the various types of format items appear in section E, "Edit-Directed Format Items". The following discussions show how the format items are used in edit-directed data specifications.

Data Format Items

On input, each data format item specifies the number of characters to be associated with the data item and how to interpret the external data. The data item is assigned to the associated variable

named in the data list, with necessary conversion to conform to the attributes of the variable. On output, the value of the associated element in the data list is converted to the character representation specified by the format item and is inserted into the data stream.

There are six data format items: fixed-point (F), floating-point (E), complex (C), picture (P), character-string (A), and bit-string (B). They are, in general, specified as follows:

```
F (w[,d[,p]])

E (w,d[,s])

C (real-format-item [,real-format-item])

P 'picture-specification'

A [(w)]

B [(w)]
```

In this list, the letter w represents an expression that specifies the number of characters in the field. The letter d specifies the number of digits to the right of a decimal point; it may be omitted for fixed-point integers. The real format item of the complex format item represents the appearance of either an F, E or P format item. The picture specification of the P format item can be either a numeric character specification or a character-string specification. On output, data associated with E and F format items is rounded if the internal precision exceeds the external precision.

A third specification (p) is allowed in the F format item; it is a scaling factor. A third specification (s) is allowed in the E format item to specify the number of digits that must be maintained in the first subfield of the floating-point number. These specifications are discussed in detail in section E, "Edit-Directed Format Items".

Note: Fixed-point binary and floating-point binary data items must always be represented in the input stream with their values expressed in decimal digits. The F and E format items may then be used to access them, and the values will be converted to binary representation upon assignment. On output, binary items are converted to decimal values and the associated F or E format items must state the field width and point placement in terms of the converted decimal number.

The following examples illustrate the use of format items:

1. `GET FILE (INFILE) EDIT (ITEM) (A(20));`

   This statement causes the next 20 characters in the file called INFILE to be assigned to ITEM. The value is automatically transformed from its character representation specified by the format item A(20), to the representation specified by the attributes declared for ITEM.

   Note: If the data list and format list were used for output, the length of a string item need not be specified in the format item if the field width is to be the same as the length of the string, that is, if no blanks are to follow the string.

2. `PUT FILE (MASKFLE) EDIT (MASK) (B);`

   Assume MASK has the attribute BIT (25); then the above statement writes the value of MASK in the file called MASKFLE as a string of 25 characters consisting of 0's and 1's. A field width specification can be given in the B format item. It must be stated for input.

3. `PUT EDIT (TOTAL) (F(6,2));`

   Assume TOTAL has the attributes FIXED (4,2); then the above statement specifies that the value of TOTAL is to be converted to the character representation of a fixed-point number and written into the standard output file SYSPRINT. A decimal point is to be inserted before the last two numeric characters, and the number will be right-adjusted in a field of six characters. Leading zeros will be changed to blanks, and, if necessary, a minus sign will be placed to the left of the first numeric character.

   The conversion from internal decimal fixed-point type to character type is performed according to the normal rules for conversion. Extra characters may appear as blanks preceding the number in the converted string. And, since leading zeros are converted to blanks, additional blanks may precede the number. If a decimal point or a minus sign appears, either will cause one leading blank to be replaced.

   In edit-directed output, the field width specification in the format item (in this case, the 6 in the F(6,2) format item) can be used to truncate leading zeros. In this specification, one zero is truncated. TOTAL would be converted to a character string of

length seven. If all four digits of the converted number are greater than zero, the number, with its inserted decimal point, will require five digit positions; if the number is negative, another digit position will be required for the minus sign. Consequently, the F(6,2) specification will always allow all digits, the point, and a possible sign to appear, but will remove the extra blank by truncation.

4. GET FILE(A) EDIT (ESTIMATE) (E(10,6));

   This statement obtains the next ten characters from the file called A and interprets them as a floating-point decimal number. A decimal point is assumed before the rightmost six digits of the mantissa. An actual point within the data can override this assumption. The value of the number is converted to the attributes of ESTIMATE and assigned to this variable.

5. GET EDIT (NAME, TOTAL)
   (P'AAAAA',P'9999');

   When this statement is executed, the standard input file SYSIN is assumed. The first five characters must be alphabetic or blank and they are assigned to NAME. The next four characters must be nonblank numeric characters and they are assigned to TOTAL.

Control Format Items

The control format items are the spacing format item (X), and the COLUMN, LINE, PAGE, and SKIP format items. The spacing format item specifies relative spacing in the data stream. The PAGE and LINE format items can be used only with PRINT files and, consequently, can only appear in PUT statements. All but PAGE generally include expressions. LINE, PAGE, and SKIP can also appear separately as options in the PUT statement; SKIP can appear as an option in the GET statement. The following examples illustrate the use of the control format items:

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

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

2. GET FILE(IN) EDIT(MAN,OVERTIME)
   (SKIP(1), A(6), COLUMN(60), F(4,2));

   This statement positions the data set associated with file IN to a new line; the first six characters on the line are assigned to MAN, and the four characters beginning at character position 60 are assigned to OVERTIME.

3. PUT FILE(OUT) EDIT (PART, COUNT)
   (A(4), X(2), F(5));

   This statement places in the file named OUT four characters that represent the value of PART, then two blank characters, and finally five characters that represent the fixed-point value of COUNT.

4. The following examples show the use of the COLUMN, LINE, PAGE, and SKIP format items in combination with one another.

```
PUT EDIT ('QUARTERLY STATEMENT')
    (PAGE, LINE(2), A(19));
PUT EDIT
    (ACCT#, BOUGHT, SOLD,
     PAYMENT, BALANCE)
    (SKIP(3), A(6), COLUMN(14),
     F(7,2), COLUMN(30), F(7,2),
     COLUMN(45), F(7,2),
     COLUMN(60), F(7,2));
```

   The first PUT statement specifies that the heading QUARTERLY STATEMENT is to be written on line two of a new page in the standard output file SYSPRINT. The second statement specifies that two lines are to be skipped (that is, "skip to the third following line") and the value of ACCT# is to be written, beginning at the first character of the fifth line; the value of BOUGHT, beginning at character position 14; the value of SOLD, beginning at character position 30; the value of PAYMENT, beginning at character position 45; and the value of BALANCE at character position 60.

Note: Control format items are executed at the time they are encountered in the format list. Any control format list that appears after the data list is exhausted will have no effect.

Remote Format Item

The remote format item (R) specifies the label of a FORMAT statement (or a label expression whose value is the label of a FORMAT statement) located elsewhere; the FORMAT statement and the GET or PUT statement specifying the remote format item must be internal to the same block. The FORMAT statement contains the remotely

situated format items. This facility permits the choice of different format specifications at execution time, as illustrated by the following example:

```
DECLARE SWITCH LABEL;
GET FILE(IN) LIST(CODE);
IF CODE = 1
    THEN SWITCH = L1;
    ELSE SWITCH = L2;
GET FILE(IN) EDIT (W,X,Y,Z)
    (R(SWITCH));
L1:  FORMAT (4 F(8,3));
L2:  FORMAT (4 E(12,6));
```

SWITCH has been declared to be a label variable; the second GET statement can be made to operate with either of the two FORMAT statements.

Expressions in Format Items

The w, p, d, and s specifications in data format items, as well as the specifications in control format items, need not be decimal integer constants. Expressions are allowed. They may be variables or other expressions.

A value read into a variable can be used in a format item associated with another variable later in the data list.

```
PUT EDIT (NAME,NUMBER,CITY)
    (A(N),A(N-4),A(10));

GET EDIT (M,STRING_A,I,STRING_B)
    (F(2),A(M),X(M),F(2),A(I));
```

In the first example, the value of NAME is inserted in the stream as a character string left-adjusted in a field of N characters; NUMBER is left-adjusted in a field of N-4 characters; and CITY is left-adjusted in a field of 10 characters. In the second example, the first two characters are assigned to M. The value of M is then taken to specify the number of characters to be assigned to STRING_A and also to specify the number of characters to be ignored before two characters are assigned to I, whose value then is used to specify the number of characters to be assigned to STRING_B.

PRINT Files

The PRINT attribute can be applied only to a STREAM OUTPUT file. It indicates that the data in the file is ultimately intended to be printed (although it may first be written on a medium other than the printed page). The first data byte of each record of a PRINT file is reserved for an American National Standard (ANS) printer control character; the compiler causes the control characters to be inserted automatically when statements containing the control options and format items PAGE, SKIP, and LINE are executed.

The layout of a PRINT file can be controlled by the use of the options and format items listed in figure 11.3. (Note that LINESIZE, SKIP, and COLUMN can also be used for non-PRINT files.) LINESIZE and PAGESIZE establish the dimensions of the printed area of the page, excluding footings. The LINESIZE option specifies the maximum number of characters to be included in each printed line; if it is not specified for a PRINT file, a default value of 120 characters is assumed (but there is no default for a non-PRINT file). The PAGESIZE option specifies the maximum number of lines to appear in each printed page; if it is not specified, a default value of 60 lines is assumed. Consider the following example:

```
OPEN FILE(REPORT) OUTPUT STREAM PRINT
   PAGESIZE(55) LINESIZE(110);
```

This statement opens the file REPORT as a PRINT file. The specification PAGESIZE(55) indicates that each page should contain a maximum of 55 lines. An attempt to write on a page after 55 lines have already been written (or skipped) will raise the ENDPAGE condition. The standard system action for the ENDPAGE condition is to skip to a new page, but the programmer can establish his own action through use of the ON statement.

The ENDPAGE condition is raised only once per page. Consequently, printing can be continued beyond the specified PAGESIZE after the ENDPAGE condition has been raised the first time. This can be useful, for example, if a footing is to be written at the bottom of each page. For example:

```
ON ENDPAGE(REPORT) BEGIN;
   PUT FILE(REPORT) SKIP LIST
       (FOOTING);
   N = N + 1;
   PUT FILE(REPORT) PAGE LIST
       ('PAGE '||N);
   PUT FILE(REPORT) SKIP (3);
   END;
```

Option	Edit-directed format item	Statement in which option or format item appears	Effect
LINESIZE(w)[1]	-	OPEN	Establishes line width
PAGESIZE(w)	-	OPEN	Establishes page width
PAGE	PAGE	PUT	Skip to new page
LINE(w)	LINE(w)	PUT	Skip to specified line
SKIP[(x)][1]	SKIP[(x)][1]	PUT	Skip specified number of lines
-	COLUMN(w)[1]	PUT	Skip to specified character position in line

[1]Can also be used with non-PRINT files: see "Options of Transmission Statements" and "Control Format Items", above, and "Line Size and Record Format", below.

Figure 11.3. Options and format items for controlling layout of PRINT files

Assume that REPORT has been opened with PAGESIZE(55), as shown in the previous example. When an attempt is made to write on line 56 (or to skip beyond line 55), the ENDPAGE condition will arise, and the begin block shown here will be executed. The first PUT statement specifies that a line is to be skipped, and the value of FOOTING, presumably a character string, is to be printed on line 57 (when ENDPAGE arises, the current line is always PAGESIZE+1). The page number is incremented, the file REPORT is set to the next page, and the character string 'PAGE' is concatenated with the new page number and printed. The final PUT statement causes three lines to be skipped, so that the next printing will be on line 4. Control returns from the begin block to the PUT statement that caused the ENDPAGE condition, and the data is printed. Any SKIP option specified in that statement will have no further effect, however.

Note that SIGNAL ENDPAGE is ignored if there is no ENDPAGE on-unit.

The specification LINESIZE(110) indicates that each line on the page can contain a maximum of 110 characters. An attempt to write a line greater than 110 characters will cause the excess characters to be placed on the next line.

Standard File SYSPRINT

Unless the standard file SYSPRINT is declared explicitly, it is always given the attribute PRINT. Under the optimizing compiler, a new page is initiated automatically when the file is opened. If the first PUT statement that refers to the file has the PAGE option, or if the first PUT statement includes a format list with PAGE as the first item, a blank page will appear. Under the checkout compiler, no new page is started when an explicit or implicit OPEN is executed for SYSPRINT, because the file is used by the compiler to transmit diagnostic messages. SYSPRINT is always open under the checkout compiler.

ENVIRONMENT Attribute

The ENVIRONMENT attribute specifies information about the physical organization of the data set associated with a file. The information is contained in a parenthesized option list; the general format is:

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

The options applicable to stream-oriented transmission are:

```
F|FB|FS|FBS|V|VB|D|DB|U
RECSIZE(record-length)
BLKSIZE(block-size)

BUFFERS(n)

CONSECUTIVE

{LEAVE }
{REREAD}

ASCII
BUFOFF[(n)]
```

The options may appear in any order and are separated by blanks, The options themselves cannot contain blanks.

The options are discussed below.

RECORD FORMAT OPTIONS

Although record boundaries are ignored in stream-oriented transmission, record format is important when a data set is being created, not only because it affects the amount of storage space occupied by the data set and the efficiency of the program that processes the data, but also because the data set may later be processed by record-oriented transmission. Having specified the record format, the programmer need not concern himself with records and blocks as long as he uses only stream-oriented transmission; he can consider his data set as a series of characters arranged in lines, and can use the SKIP option or format item (and, for a PRINT file, the PAGE and LINE options and format items) to select a new line.

Records can have one of the following formats:

Fixed-length	F	unblocked
	FB	blocked
	FBS	blocked, standard
	FS	unblocked, standard
Variable-length	V	unblocked
	VB	blocked
	D	unblocked (see "ASCII Data Sets")
	DB	blocked (see "ASCII Data Sets")
Undefined-length	U	(cannot be blocked)

Blocking and deblocking of records is performed automatically.

All records, whatever the format, consist of data bytes and, optionally, control or prefix bytes. Variable-length records include control and prefix bytes to specify record and block lengths; the use of these bytes is described later in this section. In addition, any record (whatever the format) associated with a PRINT file has the first data byte interpreted as a printer control character. The compiler analyzes the relevant PUT statement and inserts the appropriate character (or a default character).

Fixed-length Records

All records in the data set are the same length.

F-format: The records are unblocked; each record constitutes a single block.

FB-format: The records are blocked, some of the blocks may be shorter blocks, that is they may be shorter than the specified block size.

FS-format: The records are unblocked; each record constitutes a single block. For direct-access storage, every track except the last one is filled to capacity.

FBS-format: The records are blocked. Only the last block can be a short block.

A sequential data set is said to contain FBS-format records if:

1. All records in the data set are FB-format.
2. For direct-access storage, every track except the last one is filled to capacity.
3. No blocks except the last one are truncated.

Data sets with FBS-format can be read more efficiently from direct-access storage than data sets with truncated blocks.

Variable-length Records

Each record can be a different length.

V-format: The records are unblocked; each record constitutes a single block. Each record consists of:

Four control bytes
Data bytes

The four control bytes contain the record length (that is, the length of the current record); this value is inserted automatically, and requires no action by the programmer.

In addition, four extra control bytes are placed at the beginning of the block (that

is, the record). These bytes contain the block size; the value is inserted in the same way as the record length.

VB-format: The records are blocked. Each record consists of:

Four control bytes
Data bytes

The four control bytes have the same purpose as in V-format records. The block has four extra control bytes for the block size in the same way as V-format records.

D- and DB-format: see "ASCII Data Sets".

Undefined-length Records

All processing is the responsibility of the programmer. If a length specification is required in the record, the programmer must provide one and also interpret it.

RECSIZE Option

The RECSIZE option specifies the record length. This is the sum of:

1. The length required for data. For variable-length and undefined records, this is the maximum length.
2. Any control bytes required. Variable-length records require four, for the record length; fixed-length and undefined-length records do not require any.

The record length can be specified as a decimal integer constant, or as a variable with the attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum: Fixed-length, and undefined-length (except ASCII data sets): 32,760 bytes. Variable-length (except ASCII data sets): 32,756 bytes ASCII data sets: 9999 bytes

Zero value: A search for a valid value is made in (in the following order):

DD statement for the data set associated with the file

Data set label

If neither of these can provide a value, default action is taken (see "Record Format Defaults", later in this section).

Negative value: The UNDEFINEDFILE condition is raised.

A value implied by the LINESIZE option overrides a value specified in the RECSIZE option.

BLKSIZE Option

The BLKSIZE option specifies the block size. This is the sum of:

1. The lengths of all the records in the block. For variable length records, the length of each record includes the four control bytes for the record length.
2. Any control bytes required. Variable-length blocked records require four for the blocksize; fixed-length and undefined-length records do not require any.

or

Any block prefix bytes (ASCII data sets)

The block size can be specified as a decimal integer constant, or as a variable with the attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum: 32,760 bytes (or 9999 for an ASCII data set for which BUFOFF is specified without a prefix-length value)

Zero value: A search for a valid value is made in (in the following order):

DD statement for the data set associated with the file

Data set label

If neither of these can provide a value, default action is taken (see "Record Format Defaults", later in this section).

Negative value: the UNDEFINEDFILE condition is raised.

The relationship of the block size to the record length depends on the record format:

FB-format or FBS-format: The block size must be a multiple of record length

VB-format: The block size must be equal to or greater than the sum of:

The lengths of all the records in the block
Four control bytes for the block size

DB-format: The blocksize must be equal to or greater than the sum of:

The lengths of all the records in the block
Length of the block prefix (if block is prefixed)

Note:

1. The BLKSIZE option can be used with unblocked (F-,V-, or D-format) records as follows:

   a. The BLKSIZE option, but not the RECSIZE option, is specified. The record length is set equal to the block size (minus any control or prefix bytes) and the record format is unchanged.

   b. Both the BLKSIZE and the RECSIZE options are specified, and the relationship of the two values is compatible with blocking for the record format used. The records are assumed to be blocked and the record format is set to FB, VB, or DB whichever is appropriate.

2. If, for FB-format or FBS-format records, the block size equals the record length, the records are assumed to be unblocked and the record format is set to F.

Record Format Defaults

If any of the record format options is not specified in the ENVIRONMENT attribute, or in the associated DD statement or data set label, the following action is taken:

INPUT files:

Record format: The UNDEFINEDFILE condition is raised.

Block size or record length: If one of these is specified, a search is made for the other in the associated DD statement or data set label. If the search provides a value, the UNDEFINEDFILE condition is raised if this value is imcompatible with the value in the specified option. If the search is unsuccessful, a value is derived from the value for the specified option (with the addition or subtraction of any control or prefix bytes).

If neither is specified, the UNDEFINEDFILE condition is raised.

OUTPUT files:

Record format: Set to VB-format, or if ASCII option specified, to DB-format

Record length: The specified or default LINESIZE value is used:
PRINT files:
F, FB, FBS, or U:LINESIZE + 1
V, VB, D, or DB: LINESIZE + 5
Non-PRINT files:
F, FB, FBS, or U:LINESIZE
V, VB, D, or DB: LINESIZE + 4

Block size: FB, or FBS: 5*(record length)
VB: 5*(record length) + 4
DB:5*(record length) + (block prefix) (see note 3)

BUFFER offset: F, FB, or U:0
D, or DB: 4

Note:

1. The standard default for LINESIZE is 120.

2. If the default block size as calculated above is greater than 32,760 the block size is set equal to (record length + 4), and the records are set to V-format, except when the ASCII option is specified. With ASCII data sets, if the default blocksize is greater than 32,760, or 9999 if BUFOFF is specified without a prefix-length value, then the block size is set equal to (record length + length of block prefix) and the record format is set to D.

3. With DB-format records on output files, the length of the block prefix (that is, the buffer offset) must always be either 0 or 4.

4. The optimizing and checkout compilers will also accept the form of record format specification used for the PL/I(F) compiler. In this form, the record length and block size are included in the format specification.

BUFFER ALLOCATION

A buffer is a main storage area that is used for the intermediate storage of data transmitted to and from a data set. The use of buffers allows transmission and computing time to be overlapped. Buffers are essential for the automatic blocking and deblocking of records.

BUFFERS Option

The option BUFFERS(n) in the ENVIRONMENT attribute specifies the number(n) of buffers to be allocated for a data set; this number must not exceed 255 (or such other maximum as was established at system generation). If the number of buffers is not specified or is specified as zero, two buffers are assumed.

The number of buffers can be specified in the BUFNO subparameter of a DD statement instead of in the ENVIRONMENT attribute.

DCB Subparameter

Some of the information that can be specified in the options of the ENVIRONMENT attribute can also be specified in the DCB subparameter of a DD statement. The table gives a list of equivalents.

ENV Option	DCB Subparameter
Record format	RECFM
RECSIZE	LRECL
BLKSIZE	BLKSIZE
BUFFERS	BUFNO
ASCII	ASCII
BUFOFF	BUFOFF

DATA SET ORGANIZATION

The organization of a data set determines how data is recorded in the data set, and how the data is subsequently retrieved so that it can be transmitted to the program. This implementation recognizes three data set organizations, CONSECUTIVE, INDEXED, and REGIONAL. A data set that is to be accessed by stream-oriented transmission must have CONSECUTIVE organization; since this is the default for data set organization, it need not be specified at all for a STREAM file.

CONSECUTIVE Data Sets

The records in a CONSECUTIVE data set are arranged sequentially in the order in which they were written; they can be retrieved only in the same order. After the data set has been created, the associated file can be opened for input (to read the data), or for output (to extend the data set by adding records at the end, or to replace the contents of the data set by new data: the effect of using an OUTPUT file to process an existing data set depends on the DISP parameter of the associated DD statement).

ENVIRONMENT Option	DISP Subparameter	Action
REREAD	-	Positions the current volume to reprocess the data set. Repositioning for a BACKWARDS file is at the physical end of the data set.
LEAVE	-	Positions the current volume at the logical end of the data set. Positioning for a BACKWARDS file is at the physical beginning of the data set.
Neither REREAD nor LEAVE	PASS	Positions the volume at the end of the data set
	DELETE	Rewinds the current volume
	KEEP, CATLG, UNCATLG	Rewinds and unloads the current volume

Figure 11.4. Effect of LEAVE and REREAD options

MAGNETIC TAPE HANDLING OPTIONS

LEAVE and REREAD Options

The volume disposition options allow the programmer to specify the action to be taken when the end of a magnetic tape volume is reached, or when a data set on a magnetic tape volume is closed. The LEAVE option prevents the tape from being rewound. The REREAD option rewinds the tape to permit reprocessing of the volume or data set. If neither of these is specified, the action at end of volume or on closing of a data set is controlled by the DISP parameter of the associated DD statement. The effects of the options are summarized in figure 11.4.

ASCII DATA SETS

Data sets on magnetic tape using ASCII may be created and accessed in PL/I. The implementation supports F, U, and D record formats. F and U formats are treated in the same way as with other data sets; D and DB formats, which correspond to V and VB formats with other data sets, are described below.

In addition to the record format, two other ENVIRONMENT options may be specified: ASCII, and the buffer offset option BUFOFF.

ASCII Option

This option specifies that the code used to represent data on the data set is ASCII.

BUFOFF Option and Block Prefix Fields

At the beginning of each block in an ASCII data set, there may be a field known as the block prefix field. It may be from one to 99 bytes long. The buffer offset option indicates the length of this field to data management, so that the accessing or creation of data is started at this offset from the beginning of each physical block. PL/I does not support access to this field, and in general it does not contain information which is used in OS implementations. There is one situation in which data management does use information in the block prefix: with unblocked or blocked variable length records (that is, D- or DB-format records), the block prefix field may be used to record the length of the block. In this case, it is four bytes long and contains a right-aligned, decimal character value that gives the length of the block in bytes, including the block prefix field itself. It is then exactly equivalent to a block length field.

The format of the buffer offset option is BUFOFF [(n)]. A numerical value equal to the length of the prefix can be specified for "n". It may be specified as either a decimal integer constant or as a variable with the attributes FIXED BINARY(31,0) STATIC. Its minimum value is

zero and its maximum is 99. The absence of a prefix length specification indicates that the block prefix is to be used as a block length field; it implies that the field is four bytes long. The length of the block is inserted in the prefix by data management.

On input, any ASCII data set may be accessed if it has a block prefix field of length one to 99 bytes, or no block prefix field at all; and it may be accessed whether or not the block prefix field is used as a block length field. On output, a data set using any one of the three valid record formats may be created without a block prefix, but the only situation in which the creation of a block prefix is supported by PL/I is when it is used as a block length field.

The BUFOFF option may be used with ASCII data sets only.

D-format and DB-format Records

Each record may be of a different length. The two different formats are:

D-format: The records are unblocked; each record constitutes a single block. Each record consists of:

Four control bytes
Data bytes

The four control bytes contain the length of the record; this value is inserted by data management and requires no action from the programmer. In addition, there may be, at the start of the block, a block prefix field, which may contain the length of the block.

DB-format: The records are blocked. All other information given for D-format applies to DB-format.

Default Rules

In addition to the rules given under "Record Format Defaults", the following rule applies:

If ASCII is not specified in either the ENVIRONMENT option or the DD statement, but one of BUFOFF, D, or DB is specified, then ASCII is assumed.

Chapter 12: Record-Oriented Transmission

Introduction

This chapter describes the input and output statements used in record-oriented transmission. Those features of PL/I that apply equally to record-oriented and stream-oriented transmission, including files, file attributes, and opening and closing files, are described in chapter 10, "Input and Output".

In record-oriented transmission, data in a data set is considered to be a collection of records recorded in any format acceptable to the operating system. No data conversion is performed during record-oriented transmission: on input, the READ statement either causes a single record to be transmitted to a program variable exactly as it is recorded in the data set, or else sets a pointer to the record in a buffer; on output, the WRITE, REWRITE, or LOCATE statement causes a single record to be transmitted from a program variable exactly as it is recorded internally. Although data is actually transmitted to and from a data set in blocks, the statements used in record-oriented transmission are concerned only with records; the records are blocked and deblocked automatically.

Data Transmitted

Most variables, including parameters and DEFINED variables, can be transmitted by record-oriented transmission statements, and in general, the information given in this chapter may be applied equally to all variables. There are certain special considerations for a few types of data, and these are given below.

Data Aggregates

There are some restrictions applied to data The following restrictions apply to data aggregates:

1. An aggregate must be in connected storage. (An aggregate parameter must have the CONNECTED attribute).

2. For the LOCATE statement, the variable must be a level 1 based variable.

Unaligned Bit Strings

The following may not be transmitted.

1. BASED, DEFINED, parameter, subscripted, or structure-base-element variables that are unaligned fixed-length bit strings.

2. Minor structures whose first or last base elements are unaligned fixed-length bit strings (except where they are also the first or last elements of the containing major structure).

3. Major structures that have the BASED or DEFINED attribute or are parameters, and that have unaligned fixed-length bit strings as their first or last elements.

Varying-length Strings and Area Variables

A locate mode output statement (see "LOCATE Statement", later in this Chapter) specifying a varying-length string causes the transmission of a field having a length equal to the maximum length of the string, plus a two-byte prefix denoting the current length of the string. The SCALARVARYING option must be specified for the file. A locate mode output statement specifying an area variable causes the transmission of a field as long as the declared size of the area, plus a 16-byte prefix containing control information.

A move mode output statement (see "WRITE Statement" and "REWRITE Statement" later in this chapter) specifying a varying-length string variable transmits only the current length of the string. A two-byte prefix is included only if the SCALARVARYING option is specified for the file. A move mode statement specifying an element area variable or a structure whose last element is an area variable transmits only the current extent of the area plus a 16-byte prefix.

DATA TRANSMISSION STATEMENTS

The following is a general description of the record-oriented data transmission

statements; they are described in detail in section J, "Statements".

There are four statements that actually cause transmission of records to or from auxiliary storage. They are READ, WRITE, LOCATE, and REWRITE. A fifth statement, the DELETE statement, is used to delete records from an UPDATE file. The attributes of the file determine which statements can be used.

READ Statement

The READ statement can be used with any INPUT or UPDATE file. It causes a record to be transmitted from the data set to the program, either directly to a variable or to a buffer. In the case of blocked records, a READ statement with the appropriate option causes a record to be transferred from a buffer to the variable or sets a pointer to the record in a buffer; consequently, not every READ statement causes transmission of data from an input device.

WRITE Statement

The WRITE statement can be used with any OUTPUT file or DIRECT UPDATE file, but not with a SEQUENTIAL UPDATE file. It causes a record to be transmitted from the program to the data set. For unblocked records, transmission may be directly from a variable or from a buffer. For blocked records, the WRITE statement causes a logical record to be placed into a buffer; only when the blocking of the records is complete is there actual transmission of data to an output device.

REWRITE Statement

The REWRITE statement causes a record to be replaced in an UPDATE file. For SEQUENTIAL UPDATE files, the REWRITE statement specifies that the last record read from the file is to be rewritten; consequently a record must be read before it can be rewritten. For DIRECT UPDATE files, any record can be rewritten whether or not it has first been read.

LOCATE Statement

The LOCATE statement can be used only with an OUTPUT SEQUENTIAL BUFFERED file. It allocates storage within an output buffer for a based variable, setting a pointer to the location in the buffer as it does so. This pointer can then be used to refer to the allocation so that data can be moved into the buffer. When a complete block of logical records is present in a buffer, the block is transmitted to an output device.

DELETE Statement

The DELETE statement specifies that a record in an UPDATE file be deleted.

UNLOCK Statement

The UNLOCK statement is used in association with a READ statement that refers to an EXCLUSIVE file. The UNLOCK statement makes the specified record available to other tasks in addition to that for which the READ statement was issued.

Options of Transmission Statements

Options that are allowed for record-oriented data transmission statements differ according to the attributes of the file and the characteristics of the associated data set. Lists of all of the allowed combinations for each type of file are given in figures 12.6, 12.7, and 12.9, later in this chapter.

Each option consists of a keyword followed by a value, which is a file expression, a variable, or an expression, enclosed in parentheses. In any statement, the options may appear in any order.

FILE Option

The FILE option must appear in every record-oriented statement. It specifies the file upon which the operation is to take place. It consists of the keyword FILE followed by a file expression enclosed in parentheses. An example of the FILE option is shown in each of the statements in this section.

INTO Option

The INTO option can be used in the READ statement for any INPUT or UPDATE file. The INTO option specifies a variable into which the logical record is to be read.

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

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

Note that the INTO option can name an element string variable of varying length. When using a READ statement with the FROM option, only the current length of a varying-length string is transmitted to a data set and a two-byte prefix specifying the length may or may not be attached; it will only be attached if the SCALARVARYING option of the ENVIRONMENT attribute is specified for the file.

If SCALARVARYING was not declared then, on input, the implementation calculates the string length from the record length and attaches it as a two-byte prefix. For varying-length bit strings, this calculation rounds up the length to a multiple of 8 and therefore the calculated length may be greater than the maximum declared length.

FROM Option

The FROM option must be used in the WRITE statement for any OUTPUT or DIRECT UPDATE file. It can also be used in the REWRITE statement for any UPDATE file. The FROM option specifies the variable from which the record is to be written.

Note that the FROM option can name an element string variable of varying length. (See "INTO Option" above).

Records are transmitted as an integral number of bytes. Therefore, if a bit string (or a structure that starts or ends with a bit string) that is not aligned on a byte boundary, is transmitted, the record will contain bits at the start or end that are not part of the string.

The FROM option can be omitted from a REWRITE statement for SEQUENTIAL BUFFERED UPDATE files. If the last record was read by a READ statement with the INTO option, REWRITE without FROM has no effect on the record in the data set; but if the last record was read by a READ statement with the SET option, the record will be updated, in the buffer, by whatever assignments were made and copied back onto the data set.

```
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.

SET Option

The SET option can be used with a READ statement or a LOCATE statement. It specifies that a named pointer variable is to be set to point to the location in the buffer into which data has been moved during the READ operation, or which has been allocated by the LOCATE statement.

```
READ FILE (X) SET (P);
```

This statement specifies that the value of the pointer variable P is to be set to the location in the buffer of the next sequential record. If the SET option is omitted, the pointer declared with the record variable will be set.

Note that if an element string variable of varying-length is transmitted, the SCALARVARYING option must be specified for the file.

IGNORE Option

The IGNORE option can be used in a READ statement for any SEQUENTIAL INPUT or SEQUENTIAL UPDATE file. It includes an expression whose integral value specifies a number of records to be skipped over and ignored. If the value of the expression is negative or zero, no records are skipped.

```
READ FILE (IN) IGNORE (3);
```

This statement specifies that the next three records in the file are to be skipped.

If a READ statement includes none of the options INTO, SET, and IGNORE, IGNORE(1) is assumed.

KEY Option

The KEY option applies only to KEYED files associated with data sets of INDEXED or REGIONAL organization. (The types of data set organization applicable to record-oriented transmission are discussed under "Data Set Organization", later in this chapter.) The option consists of the keyword KEY followed by a parenthesized expression, which may be a character-string constant, a variable, or any other element expression; if necessary, the expression is evaluated and converted to a character string. The rules governing the length of the character string and what it represents are discussed below under "INDEXED Organization" and "REGIONAL Organization" later in this chapter.

The KEY option identifies a particular record. It can be used in a READ statement for an INPUT or UPDATE file, or in a REWRITE statement for a DIRECT UPDATE file. (The KEY option can be used in a READ statement for a SEQUENTIAL file only if the associated data set has INDEXED organization.)

```
READ FILE (STOCK) INTO (ITEM)
  KEY (STKEY);
```

This statement specifies that the record identified by the character-string value of the variable STKEY is to be read into to the variable ITEM.

KEYFROM and KEYTO Options

The KEYFROM and KEYTO options apply only to KEYED files associated with data sets of INDEXED or REGIONAL organization, or to TRANSIENT files. Each option consists of the keyword KEYFROM or KEYTO followed by an expression in parentheses. For KEYFROM, the expression may be a character-string constant, or any other element expression; if necessary, the expression is evaluated and converted to a character string. For KEYTO, the expression must be a character-string variable or pseudovariable whose value is less than 256 bytes long. The rules governing the lengths of the character strings and what they represent are discussed below, under "INDEXED Organization" and "REGIONAL Organization" (except for TRANSIENT files, which are discussed under "Teleprocessing").

The KEYFROM option specifies a key that identifies the record on the data set, or (for TRANSIENT files) the terminal to which the message or record is to be transmitted. It can be used in a WRITE statement for a SEQUENTIAL OUTPUT or DIRECT UPDATE file or a DIRECT OUTPUT file that has REGIONAL organization, or in a LOCATE statement.

```
WRITE FILE (LOANS) FROM (LOANREC)
  KEYFROM (LOANNO);
```

This statement specifies that the value of LOANREC is to be written as a record in the file LOANS, and that the character string value of LOANNO is to be used as the key with which it can subsequently be retrieved.

The KEYTO option specifies the name of the variable into which the key (or terminal identifier, if the file is TRANSIENT) of a record is to be read. It can be used in a READ statement for a SEQUENTIAL INPUT, SEQUENTIAL UPDATE, or TRANSIENT INPUT file.

```
READ FILE (DETAIL) INTO (INVTRY)
  KEYTO (KEYFLD);
```

This statement specifies that the next record in the file DETAIL is to be read into the variable INVTRY, and that the key of the record is be read into the variable KEYFLD.

EVENT Option

The EVENT option consists of the keyword EVENT followed by the name of an event variable in parentheses. (The appearance of a name in the EVENT option constitutes a contextual declaration of an event variable.) The option can appear in any READ, WRITE, REWRITE, or DELETE statement for an UNBUFFERED file with CONSECUTIVE or REGIONAL organization or for any DIRECT file.

The EVENT option specifies that the input or output operation is to take place asynchronously (i.e., while other processing continues) and that no I/O conditions (except for UNDEFINEDFILE) are raised until a WAIT statement, specifying the same event variable, is executed by the same task. For example:

```
READ FILE (MASTER) INTO (REC_VAR)
     EVENT (RECORD_1);
        .
        .
        .
WAIT (RECORD_1);
```

When any expressions in the options of the READ statement have been evaluated, the input operation is started. As soon as this has happened, the statements following are executed. Any RECORD, TRANSMIT, KEY,

or ENDFILE condition will not be raised until control reaches the WAIT statement. If, when the WAIT statement is executed, the input operation is not complete, and if none of the four conditions is raised, execution of further statements is suspended until the operation is complete. When the operation is successfully completed, processing continues with the next statement following the WAIT statement. If any of the four conditions arise owing to execution of the READ statement, the condition(s) will be raised when the WAIT statement is executed. For this implementation, only the conditions TRANSMIT and RECORD can occur together; TRANSMIT is always processed first. Then, upon normal return from any on-units entered, processing continues with the next statement following the WAIT statement. Although the EVENT option specifies asynchronous processing, none of the four conditions can cause an interrupt until they are synchronized with processing by the WAIT statement.

Note that for consecutive and regional sequential files only one outstanding input/output operation is allowed for a file unless a higher number is specified in the NCP option of the environment attribute or DCB subparameter. The ERROR condition is raised if an attempt is made to initiate an input/output operation on a file in excess of the number allowed, while a previous input/output operation has not been waited for.

Once a statement containing an EVENT option has been executed, the event variable named in the option is considered to be active; while it is active, the same event variable cannot be specified again in an EVENT option. The event variable becomes inactive again only after execution of the corresponding WAIT statement or when the file is closed.

The EVENT option can also be used with the CALL statement to specify asynchronous execution of procedures (see chapter 17, "Multitasking"), and with the DISPLAY statement with the REPLY option.

NOLOCK Option

The NOLOCK option can be used in a READ statement that refers to an EXCLUSIVE file. It specifies that the record accessed by the READ statement will not be locked between completion of a READ statement and commencement of the corresponding REWRITE; the record will continue to be available to other tasks in addition to that which issued the READ statement.

Processing Modes

Record-oriented transmission offers the programmer two methods of handling his data. He can process data within a declared storage area; this is termed the move mode because the data is actually moved into or out of main storage either directly or via a buffer. Alternatively, the programmer can process his data while it remains in a buffer (that is, without moving it into a declared storage area); this is termed the locate mode, because the execution of a data transmission statement merely identifies the location of the storage allocated to a record in the buffer. The locate mode is applicable only to SEQUENTIAL BUFFERED files. Which mode is used is determined by the data transmission statements and options used by the programmer.

MOVE MODE

In the move mode, a READ statement causes a record to be transferred from external storage to the variable named in the INTO option (via an input buffer if a BUFFERED file is used); a WRITE or REWRITE statement causes a record to be transferred from the variable named in the FROM option to external storage (perhaps via an output buffer). The variables named in the INTO and FROM options can be of any storage class.

Consider the following example, which is illustrated in figure 12.1:

```
NEXT:  READ FILE(IN) INTO(DATA);
         .
         .
         .
       WRITE FILE (OUT) FROM (DATA);
       GO TO NEXT;
```

The first time the READ statement is executed, a block is transmitted from the data set associated with the file IN to an input buffer, and the first record in the block is assigned to the variable DATA; further executions of the READ statement assign successive records from the buffer to DATA. When all the records in the buffer have been processed, the next READ statement causes a new block to be transmitted from the data set although this

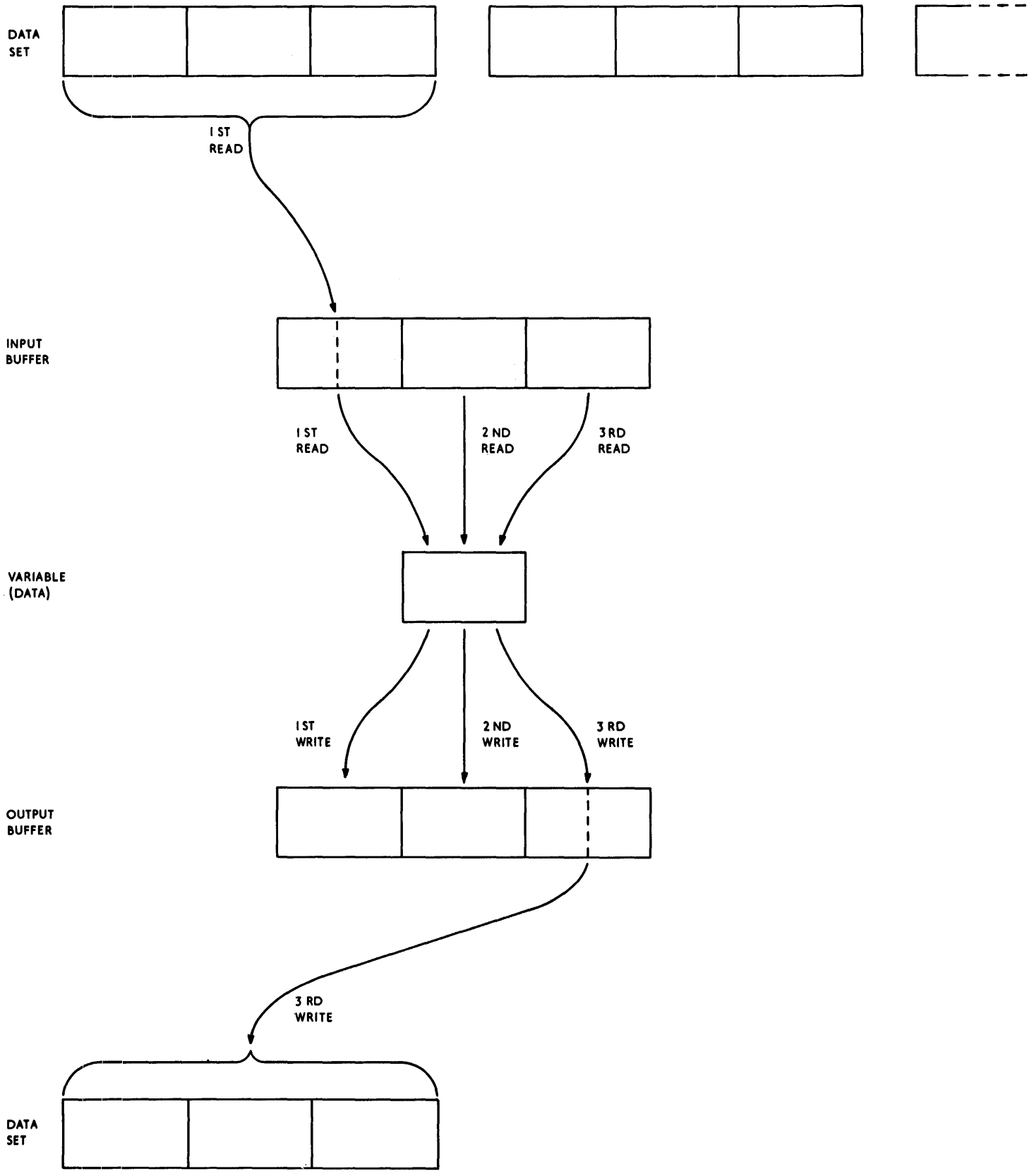


Figure 12.1. Input and output: move mode

READ statement will probably access a new record in an alternative buffer, thus permitting overlapped data transmission and processing. The WRITE statement is executed in a similar manner, building physical records in an output buffer and transmitting them to the data set associated with the file OUT each time the buffer is filled.

The move mode may be simpler to use than the locate mode since there are no buffer alignment problems. Furthermore, it can result in faster execution when there are numerous references to the contents of the same record, because of the overhead incurred by the indirect addressing technique used in locate mode.

It is possible to use the move mode access technique and avoid internal movement of data in the following cases:

1. SEQUENTIAL UNBUFFERED files with: CONSECUTIVE organization with either U-format records, or F-format records which are not larger than the variable specified in either the INTO or FROM option; and REGIONAL(1) organization with F-format records which are not larger than the variable specified in the FROM or INTO option.

2. DIRECT files with REGIONAL(1) or REGIONAL(2) organization and F-format records; and REGIONAL(3) organization with F-format or U-format records.

LOCATE MODE

Locate mode requires the use of based variables. A based variable is effectively overlaid on the data in the buffer, and different based variables can be used to access the same data by associating the same pointer with each one; thus the same data can be interpreted in different ways. Locate mode can also be used to read self-defining records, in which information in one part of the record is used to indicate the structure of the rest of the record; for example, this information could be a count of the number of repetitions of a subfield, or a code identifying which one of a class of structures should be used to interpret the record.

A READ statement causes a block of data to be transferred from the data set to an input buffer if necessary, and then sets a pointer variable named in the SET option to point to the location in the buffer of the next record; the data in the record can then be processed by reference to the based variable associated with the pointer variable. The record is available only until the execution of the next READ statement that refers to the same file.

A LOCATE statement causes storage for a based variable to be allocated in an output buffer, and sets a pointer variable to identify the allocated storage. The based variable can now have values assigned to it. The next LOCATE, WRITE, or CLOSE statement for the same file will, if necessary, transmit the data in the output buffer to the data set. After transmission the storage used for the buffer is freed; hence, only the latest one can be accessed.

Locate mode frequently provides faster execution than move mode since there is less movement of data, and less storage may be required. But it must be used carefully; in particular, the programmer must be aware of how his data will be aligned in the buffer and how structured data will be mapped; structure mapping and data alignment are discussed in section K, "Data Mapping".

Figure 12.2 illustrates the following example, which uses locate mode for input and move mode for output:

```
        DCL DATA BASED(P);
           .
           .
NEXT:   READ FILE(IN) SET(P);
           .
           .
           .
        WRITE FILE(OUT) FROM(DATA);
        GO TO NEXT;
```

The first time the READ statement is executed, a block is transmitted from the data set associated with the file IN to an input buffer, and the pointer variable P is set to point to the first record in the buffer; any reference to the variable DATA or to any other based variable qualified by the pointer P will then in effect be a reference to this first record. Further executions of the READ statement set the pointer variable P to point to succeeding records in the buffer. When all the records in the buffer have been processed, the next READ statement causes a new block to be transmitted from the data set.

It is doubtful whether the use of locate mode for both input and output in the above example would result in increased efficiency. An alternative would be to use move mode for input and locate mode for output, for example:

```
        DCL DATA BASED(P);
           .
           .
NEXT:   LOCATE DATA FILE(OUT);
        READ FILE(IN) INTO(DATA);
           .
           .
           .
        GO TO NEXT;
```

Each execution of the LOCATE statement reserves storage in an output buffer for a new allocation of the based variable DATA and sets the pointer variable P to point to this storage. The first execution of the READ statement causes a block to be transmitted from the data set associated with the file IN to an input buffer, and the first record in the block to be assigned to the first allocation of DATA; subsequent executions of the READ statement assign successive logical records to the current allocation of DATA. When all the records in the buffer have been processed, the next READ statement causes a new block to be transmitted from the data set. Each record is available for processing during the period between the execution of the READ statement and the next execution of the LOCATE statement. When no further space is available in the output buffer, the next execution of the LOCATE statement causes a block to be transmitted to the data set associated with the file OUT, and a new buffer to be allocated.

Note that if the READ statement raises the ENDFILE condition, the file OUT will have been allocated a buffer which will be transmitted when the file is closed.

ENVIRONMENT Attribute

The ENVIRONMENT attribute specifies information about the physical organization of the data set associated with a file. The information is contained in a parenthesized option list; the general format is:

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

The options applicable to record-oriented transmission are:

```
F|FB|FS|FBS|V|VB|VS|VBS|D|DB|U
RECSIZE(record-length)
BLKSIZE(block-size)

BUFFERS (n)

{CONSECUTIVE          }
{INDEXED              }
{REGIONAL({1|2|3})    }
{TP({M|R})            }

{LEAVE  }
{REREAD }

TOTAL

{CTLASA }
{CTL360 }

COBOL

INDEXAREA [(index-area-size)]
NOWRITE
ADDBUFF

GENKEY

NCP(n)

TRKOFL

SCALARVARYING

KEYLENGTH(n)

KEYLOC(n)

ASCII
BUFOFF[(n)]
```

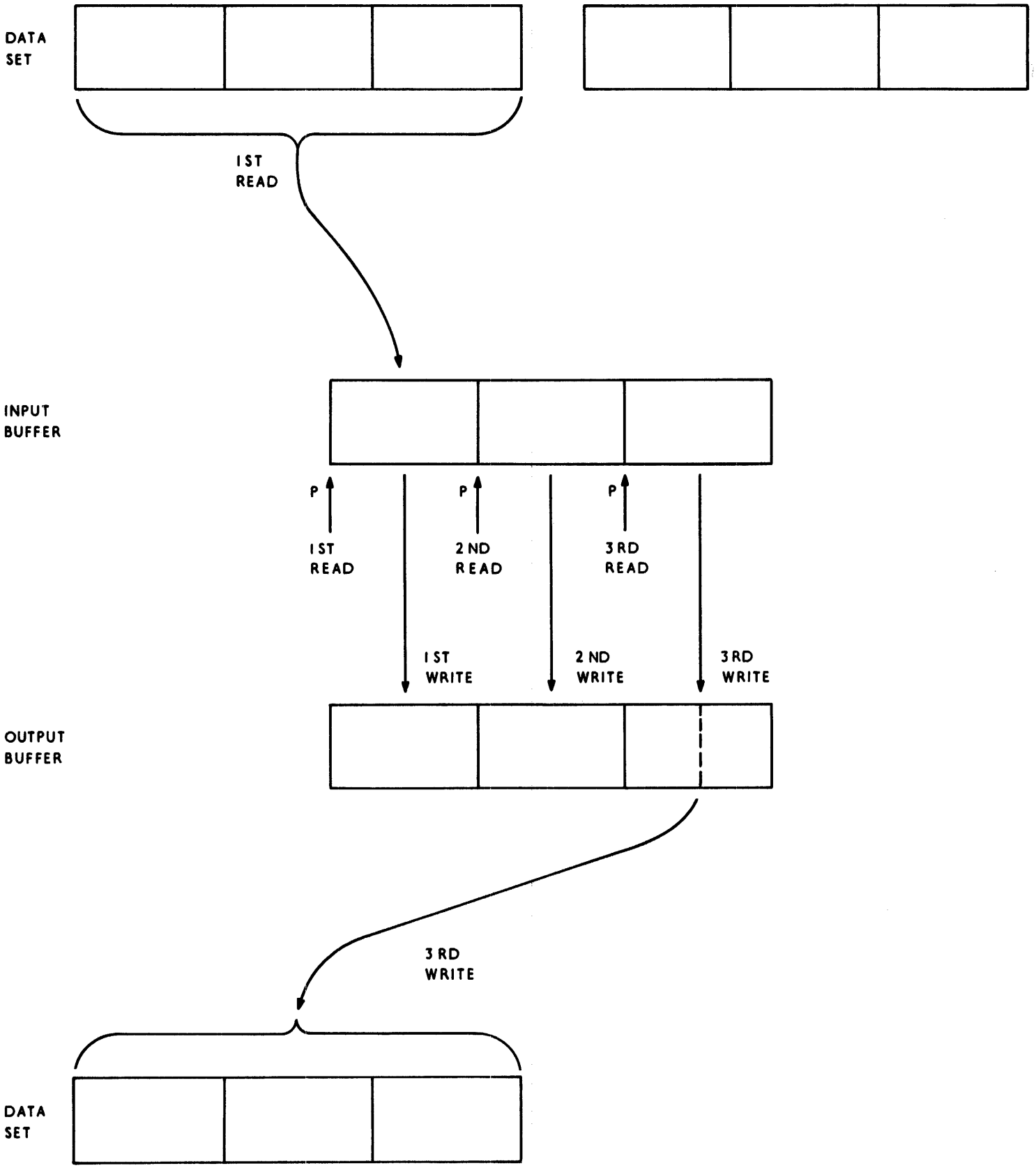


Figure 12.2. Locate mode input, move mode output

A constant or variable can be used with those ENVIRONMENT options that require decimal integer arguments, such as block sizes and record lengths. The variable must be unsubscripted and unqualified with the attributes FIXED BINARY(31,0) and STATIC.

The options may appear in any order, and are separated by blanks. The options themselves cannot contain blanks.

The options are discussed below.

RECORD FORMAT OPTIONS

Records can have one of the following formats:

Fixed-length	F	unblocked
	FB	blocked
	FS	unblocked, standard
	FBS	blocked, standard
Variable length	V	unblocked
	VB	blocked
	VS	spanned
	VBS	blocked, spanned
	D	unblocked (see "ASCII Data Sets")
	DB	blocked (see ASCII Data Sets")
Undefined-length	U	(cannot be blocked)

Blocking and deblocking of records is performed automatically.

All records, whatever the format, consist of data bytes and, optionally, control or prefix bytes. Variable-length records include control or prefix bytes to specify record and block lengths; the use of these bytes is described later in this section. In addition, any record (whatever the format) can have an optional printer or machine control character in the first data byte. The programmer must insert the character himself, and must indicate the presence of such a character by means of the CTLASA or CTL360 options of the ENVIRONMENT attribute, or by means of the equivalent field of the DCB subparameter in the associated DD statement.

The SCALARVARYING option (described later in this section) can be specified with records of any format. This option cannot be specified if the first data byte contains a printer or a machine control character, as this would lead to an ambiguous interpretation of this byte.

Fixed-length Records

All records in the data set are the same length.

F-format: The records are unblocked; each record constitutes a single block

FB-format: The records are blocked. Some of the blocks may be short blocks, that is, they may be shorter than the specified block size.

FS-format: The records are unblocked; each record constitutes a single block. For direct-access storage, every track except the last one is filled to capacity.

FBS-format: The records are blocked. Only the last block can be a short block.

A consecutive data set is said to contain FBS-format records if:

1. All records in the data set are FB-format
2. For direct-access storage, every track except the last one is filled to capacity.
3. No blocks except the last one are truncated.

Data sets with standard format (FS or FBS) records can be read from direct-access storage more efficiently than data sets with truncated blocks or embedded unfilled tracks.

Variable-length Records

Each record can be a different length.

V-format: The records are unblocked; each record constitutes a single block. Each record consists of:

Four control bytes

Data bytes

The four control bytes contain the record length (that is, the length of the current record including the four control bytes); this value is inserted automatically and requires no action by the programmer. In addition, four extra control bytes are placed at the beginning of the block. These bytes contain the block size including all control bytes; the value is inserted in the same way in the record length.

VB-format: The records are blocked. Each record consists of:

Four control bytes

Data bytes

The four control bytes have the same purpose as in V-format records. The block has four extra control bytes for the block size, in the same way as for V-format records.

VS-format: Each record constitutes at least one block. On CONSECUTIVE data sets, record length can be greater than block size; if it is, the record can 'span' several blocks. A spanned record is divided into segments, and each segment occupies a block. Therefore a block consists of:

Four block control bytes

Four record or segment control bytes

Data Bytes

The block control bytes contain the length of the block; the record (or segment) control bytes contain the length of the record (or segment). These values are inserted automatically and require no action by the programmer.

VS-format records can be specified for data sets with CONSECUTIVE or REGIONAL(3) organization only. The VS record format option must be specified as an option of ENVIRONMENT, not in the DCB subparameter of the DD card.

CONSECUTIVE: Record length can be equal to or greater than block size; each block contains one record or record segment.

REGIONAL(3): Record length cannot be greater than block size. A record can only be segmented across track boundaries, when a complete record will not fit into the space remaining on the current track. Each such segment constitutes a block.

VBS-format: Each record constitutes part of a block, a block or several blocks. Each block consists of:

Four block control bytes
One of the following:

One or more complete records
One or more complete records, and either one or two record segments.
Two record segments
A single record segment

Each complete record or each record segment consists of:

Four record or segment control bytes
Data bytes

The control bytes have the same purpose as in VB-format records.
VBS-format records can be specified for data sets with CONSECUTIVE organization only.

D- and DB-format: see "ASCII Data Sets"

Segmentation and reassembly of records, like blocking and deblocking, take place automatically, and require no action by the programmer.

Undefined-length Records

All processing is the responsibility of the programmer; if a length specification is required in the record, the programmer must provide it, and must interpret it.

RECSIZE Option

The RECSIZE Option specifies the record length. For files other than transient ones, this is the sum of:

1. The length required for data. For variable-length and undefined-length records, this is the maximum length.
2. Any control bytes required. Variable-length records require four, for the record length; fixed-length and undefined-length records do not require any.

For a transient file, it is the sum of:

1. The four V-format control bytes.
2. One flag byte.
3. Eight bytes for the key.
4. The maximum length required for the data.

The record length can be specified as a decimal integer constant or as a variable with the attributes FIXED BINARY (31,0) STATIC.

The value is subject to the following conventions:

Maximum: Fixed-length, and undefined (except ASCII data sets): 32,760 bytes
V-format, and VS- and VBS-format with UPDATE files: 32,756 bytes
VS-and VBS-format with INPUT and OUTPUT files: no limit
ASCII data sets: 9999

For VS- and VBS-format records longer than 32,756 bytes, the length must be specified in the RECSIZE option of ENVIRONMENT, and the DCB subparameter of the DD card must specify LRECL=X.

Zero value: A search for a valid value is made in (the following order):

DD statement for the data set associated with the file

Data set label

If neither of these can provide a value, default action is taken (see "Record Format Defaults", later in this section).

Negative value: The UNDEFINEDFILE condition is raised

BLKSIZE Option

The BLKSIZE option specifies the maximum block size on the data set. The length of a block is the sum of:

1. The total length(s) of one of the following:

   A single record

   A single record and either one or two record segments

   Several records

   Several records and either one or two record segments

   Two record segments

   A single record segment

   For variable length records, the length of each record or record segment includes the four control bytes for the record or segment length.

   The above list summarizes all the possible combinations of records and record segments options: fixed- or variable-length blocked or unblocked, spanned or non-spanned. When specifying a block size for spanned records, the programmer must be aware that each record and each record segment will require four control bytes for the record length, and that these quantitites are in addition to the four control bytes required for each block.

2. Any further control bytes required. Variable-length blocked records require four, for the block size; fixed-length and undefined-length records do not require any.

   or

   Any block prefix bytes required (ASCII data sets only).

The value can be specified as a decimal integer constant, or as a variable with the attributes FIXED BINARY (31,0) STATIC.

The value is subject to the following conventions:

Maximum: 32,760 bytes (or 9999 for an ASCII data set for which BUFOFF without a prefix-length value has been specified)

Zero value: A search for a valid value is made (in the following order):

DD statement for the data set associated with the file

Data set label

If neither of these can provide a value, default action is taken (see "Record Format Defaults")

Negative value: The UNDEFINEDFILE condition is raised

The relationship of the block size to the record length depends on the record format:

FB-format or FBS format: The block size must be a multiple of the record length

VB-format: The block size must be equal to or greater than the sum of:

The maximum length of any record

Four control bytes

VS-format or VBS-format: The block size can be less than, equal to, or greater than the record length.

DB-format: The blocksize must be equal to or greater than the sum of:

The maximum length of any record

The length of the block prefix (if block is prefixed)

Note:

1. The BLKSIZE option can be used with unblocked (F-, V-, or D-format) records, as follows:

   a. The BLKSIZE option, but not the RECSIZE option, is specified. The record length is set equal to the block size, (minus any control or prefix bytes) and the record format is unchanged.

   b. Both the BLKSIZE and the RECSIZE options are specified, and the relationship of the two values is compatible with blocking for the record format used. The records are assumed to be blocked and the record format is set to FB VB, or DB whichever is appropriate.

2. If, for FB-format or FBS-format records, the block size equals the record length, the records are assumed to be unblocked and the record format is set to F.

Record Format Defaults

If any of the record format options is not specified, the following action is taken.

Record format: The UNDEFINEDFILE condition is raised.

Block size or record length: If one of these is specified, a search is made for the other in the associated DD statement or data set label. If the search provides a value, the UNDEFINEDFILE condition is raised if this value is incompatible with the value in the specified option. If the search is unsuccessful, a value is derived from the specified option (with the addition or subtraction of any control or prefix bytes). If neither is specified, the UNDEFINEDFILE file condition is raised.

Note: The optimizing and checkout compilers will also accept the form of record format specification used for the PL/I(F) compiler. In this form, the record length and block size are included in the format specification.

BUFFER ALLOCATION

A buffer is an internal storage area that is used for the intermediate storage of data transmitted to and from a data set. The use of buffers can speed up processing of SEQUENTIAL files. Buffers are essential for the automatic blocking and deblocking of records and for locate-mode transmission.

BUFFERS Option

The option BUFFERS(n) in the ENVIRONMENT attribute specifies, for CONSECUTIVE and INDEXED data sets, the number (n) of buffers to be allocated for a data set; this number must not exceed 255 (or such other maximum as was established at system generation). If the number of buffers is not specified for a BUFFERED file or is specified as zero, two buffers are assumed. A REGIONAL data set is always allocated two buffers.

In teleprocessing, the BUFFERS option specifies the number of buffers available for a particular message queue, that is, for a particular TRANSIENT file. The buffer size is specified in the message control program for the installation. The number of buffers specified should, if possible, be sufficient to provide for the longest message to be transmitted.

DATA SET ORGANIZATION

The organization of a data set determines how data is recorded in a data set volume, and how the data is subsequently retrieved so that it can be transmitted to the program. Records are stored in and retrieved from a data set either sequentially on the basis of successive physical or logical positions, or directly by the use of keys specified in data transmission statements. These storage and retrieval methods provide PL/I with five general data set organizations: CONSECUTIVE, INDEXED, REGIONAL, TP(M), and TP(R). If the data set organization is not specified, a default is obtained thus:

1. If the merged attribute from the DECLARE and OPEN statements do not include TRANSIENT: the default is CONSECUTIVE.

2. If the attributes include TRANSIENT: the default is TP(M).

CONSECUTIVE, INDEXED, and REGIONAL Data Sets

In a data set with CONSECUTIVE organization, records are organized solely on the basis of their successive physical positions; records are retrieved only in sequential order, and keys are not used. The records of an INDEXED data set are arranged in logical sequence according to keys associated with each record; the records are arranged in ascending key sequence, and indexes are maintained in the data sets and are used for retrieval of records. A data set with REGIONAL organization is divided into regions, each of which is identified by a region number and contains one or more records; for retrieval, the key supplied gives the region number or track at which the search for the record is to commence.

CONSECUTIVE data sets are the simplest of the three types to create and use, and they have the advantage that less external storage is required. However, records in a CONSECUTIVE data set can be updated only in their existing sequence, and if records are to be inserted a new data set must be created. Updating is not supported for magnetic tape.

Although an INDEXED data set must be created sequentially, once it exists records can be retrieved, updated, added, or deleted at random. Sequential processing of an INDEXED data set is slower than that of a corresponding CONSECUTIVE data set, because the records it contains are not necessarily retrieved in physical sequence; furthermore, random access is less efficient for an INDEXED data set than for a REGIONAL data set, because the indexes must be searched to locate a record. Other disadvantages of an INDEXED data set are that it requires more external storage space than a CONSECUTIVE data set, and that all volumes of a multi-volume data set must be mounted even for sequential processing.

Direct access of REGIONAL data sets is quicker than that of INDEXED data sets, but they have the disadvantage that sequential processing may present records in random sequence; the order of sequential retrieval is not necessarily that in which the records were presented, nor need it be related to the relative key values. Blocked records are not permitted in a REGIONAL data set.

Optimization of Input/Output Operations

In general, I/O operations are performed by library subroutines called from compiled code. Under certain conditions, however, the optimizing compiler can provide in-line code to carry out these operations, thus saving the overheads of library calls. This gives considerably faster execution of the I/O statements.

For an I/O statement to be executed in-line, the data set being accessed or created must be CONSECUTIVE, and the file used must be a non-parameter file constant with the attributes SEQUENTIAL, BUFFERED, and either INPUT or OUTPUT. The ENVIRONMENT attribute must specify the following options: TOTAL, SCALARVARYING (if varying-length strings are to be transmitted), and either F, FB, FS, FBS, D, DB, V, or U record format. The file declaration would therefore be as follows:

```
DCL F FILE RECORD SEQUENTIAL BUFFERED
      INPUT|OUTPUT ENV(CONSECUTIVE
      F|FB|U TOTAL [SCALARVARYING]
      [RECSIZE(n)][BLKSIZE(n)]);
```

The standard default attributes and option are underlined. At least one of the underlined attributes must be specified, otherwise the file would be given the attribute STREAM by default.

The statement READ SET will always be implemented by in-line code if it specifies a file declared or indicated as above, except when a file with the attribute BACKWARDS is used to transmit U-format records. The other record I/O statements, namely READ INTO, WRITE FROM, and LOCATE, generate in-line code provided:

1. the record variable declaration does not include an expression as a string length, an array bound, or an area size;

   and

2. the ENVIRONMENT attribute specifies the record size for F-, FB-, FS-, FBS-, or V-format records, or the block size for U-format records.

I/O statements compiled by the checkout compiler always generate a library call.

When in-line code is employed to implement an I/O statement, the compiler gives an informatory message.

The speed of I/O operations when accessing an INDEXED data set can be improved by specifying the INDEXAREA, NOWRITE, and ADDBUFF options. Details are given under "Data Management Optimization" in this Chapter.

Teleprocessing Data Sets

A teleprocessing data set comprises a queue of messages that constitute the input to a PL/I message processing program. The messages are retrieved sequentially; keys are used to identify the terminal associated with the message.

The TP(M) option specifies that the file is a teleprocessing file and can only be associated with a teleprocessing data set. Each I/O operation in the PL/I program causes a complete message to be transmitted to or from the data set. The message can consist of one logical record, or several logical records, on the data set.

The TP(R) option is the same except that each I/O operation applies to one logical record only in the data set. This record can be a message or part of a complete message.

A teleprocessing file can be declared with the following attributes only:

```
FILE
RECORD
INPUT or OUTPUT
BUFFERED or UNBUFFERED
TRANSIENT
KEYED
ENVIRONMENT
```

For teleprocessing applications, the only environment options that can be specified are:

```
TP({M|R})
RECSIZE(record-length)
BUFFERS(n)
```

Record format must not be specified for teleprocessing programs.

MAGNETIC TAPE HANDLING OPTIONS

LEAVE and REREAD Options

The volume disposition options allow the programmer to specify the action to be taken when the end of a magnetic tape volume is reached, or when a data set on a magnetic tape volume is closed. The LEAVE option prevents the tape from being rewound. The REREAD option rewinds the tape to permit reprocessing of the data set. If neither of these is specified, the action at end of volume or on closing of a data set is controlled by the DISP parameter of the associated DD statement. The effects of the options are summarized in figure 12.3.

PRINTER/PUNCH CONTROL (CTL360/CTLASA)

The printer/punch control options CTLASA and CTL360 apply only to OUTPUT files associated with CONSECUTIVE data sets. They specify that the first character of a record is to be interpreted as a control character.

1. The CTLASA option specifies ANSI standard control characters.
2. The CTL360 option specifies IBM machine code control characters.

The codes that can be used with these options are listed with their actions in figures 12.4 and 12.5.

DATA INTERCHANGE (COBOL)

The COBOL option facilitates the interchange of data between programs written in PL/I and programs written in COBOL. It specifies that structures in the data set associated with the file will be mapped as they would be in American National Standard COBOL. The COBOI structures can be SYNCHRONIZED or unsynchronized; it is the programmer's responsibility to ensure that the associated PL/I structure has the equivalent alignment stringency, that is, it must be ALIGNED or UNALIGNED, respectively.

ENVIRONMENT Option	DISP Subparameter	Action
REREAD	-	Positions the current volume to reprocess the data set. Repositioning for a BACKWARDS file is at the physical end of the data set.
LEAVE	-	Positions the current volume at the logical end of the data set. Repositioning for a BACKWARDS file is at the physical beginning of the data set.
Neither REREAD nor LEAVE	PASS	Positions the volume at the end of the data set
	DELETE	Rewinds the current volume
	KEEP, CATLG, UNCATLG	Rewinds and unloads the current volume

Figure 12.3. Effect of LEAVE and REREAD options

CTLASA code	CTL360 code bytes		
action before printing	action after printing	action without printing	Action
+	00000001	-	Print only (no space)
b	00001001	00001011	Space 1 line
0	00010001	00010011	Space 2 lines
-	00011001	00011011	Space 3 lines
1	10001001	10001011	Skip to channel 1
2	10010001	10010011	Skip to channel 2
3	10011001	10011011	Skip to channel 3
4	10100001	10100011	Skip to channel 4
5	10101001	10101011	Skip to channel 5
6	10110001	10110011	Skip to channel 6
7	10111001	10111011	Skip to channel 7
8	11000001	11000011	Skip to channel 8
9	11001001	11001011	Skip to channel 9
A	11010001	11010011	Skip to channel 10
B	11011001	11011011	Skip to channel 11
C	11100001	11100011	Skip to channel 12

Figure 12.4. 1403 Printer control codes

CTLASA code	CTL360 code bytes	Action
V	00000001	Select stacker 1
W	01000001	Select stacker 2
-	10000001	Select stacker 3

Figure 12.5. 2540 Card Read Punch control codes

A file with the COBOL option can be used only for READ INTO, WRITE FROM, and REWRITE FROM statements. The file name cannot be passed as an argument or assigned to a file variable. The variable to be transmitted must not be subscripted.

If an ON-condition is raised during the execution of a READ statement, the variable named in the INTO option cannot be used in the on-unit. If the completed INTO variable is required, there must be a normal return from the on-unit.

The restrictions noted below apply to the handling of a file with the COBOL option. The PL/I equivalents of COBOL data types are given in chapter 19, "Interlanguage Communication Facilities".

The EVENT option can be used only if the compiler can determine that the PL/I and COBOL structure mappings are identical (i.e., all elementary items have identical boundaries). If the mappings are not identical, or if the compiler cannot tell whether they are identical, an intermediate variable is created to represent the level 1 item as mapped by the COBOL algorithm. The PL/I variable is assigned to the intermediate variable before a WRITE statement is executed, or assigned from it after a READ statement has been executed.

IN-LINE CODE OPTIMIZATION (TOTAL)

The purpose of this option is to aid the optimizing compiler in the production of efficient compiled code. In particular, it allows the compiler to use in-line code for certain I/O operations (see "Optimization of Input/Output Operations" in this chapter). It specifies that no attributes will be merged from the OPEN statement or the I/O statement. In other words, the complete set of attributes is to be built up at compile time from explicitly declared and default attributes.

The UNDEFINEDFILE condition is raised if any attribute that was not explicitly declared appears on the OPEN statement, or if the I/O statement implies a file attribute that conflicts with a declared or default attribute.

It is not an error to specify TOTAL when using the checkout compiler.

DATA MANAGEMENT OPTIMIZATION (INDEXAREA/NOWRITE/ADDBUFF)

The data management optimization options in the ENVIRONMENT attribute increase program efficiency, in certain circumstances, when DIRECT files are used to access INDEXED data sets.

The INDEXAREA option improves the input/output speed of a DIRECT INPUT or DIRECT UPDATE file with INDEXED data set organization, by having the highest level of index placed in main storage. The "index area size" enables the programmer to limit the amount of main storage he is prepared to allow for an index area. The size, when specified, must be a decimal integer constant or a variable with attributes FIXED BINARY (31,0) STATIC whose value lies within the range zero through 32,767. If the "index area size" is not specified, the highest level index is moved unconditionally into main storage. If an index area size is specified, the highest level index is held in main storage, provided that its size does not exceed that specified. If the specified size is less than zero or greater than 32,767, the compiler issues a warning message and ignores the option.

The NOWRITE option is used for DIRECT UPDATE files. It informs the compiler that no records are to be added to the data set and that data management modules concerned solely with adding records are not required; it thus allows the size of the object program to be reduced.

The ADDBUFF option can be specified for a DIRECT INPUT or DIRECT UPDATE file with INDEXED data set organization and F-format records to indicate that an area of internal storage is to be used as a workspace in which records on the data set can be rearranged when new records are added. The size of the workspace is assumed to be equivalent to one track of the direct device used. The option need not be specified for DIRECT INDEXED files with V-format records, as the workspace is automatically allocated for such files.

KEY CLASSIFICATION (GENKEY)

The GENKEY (generic key) option applies only to INDEXED data sets. It enables the programmer to classify keys recorded in a data set and to use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.

A generic key is a character string that identifies a class of keys: all keys that begin with the string are members of that class. For example, the recorded keys 'ABCD', 'ABCE', and 'ABDF' are all members of the classes identified by the generic keys 'A' and 'AB', and the first two are also members of the class 'ABC'; and the three recorded keys can be considered to be unique members of the classes 'ABCD', 'ABCE', and 'ABDF', respectively.

The GENKEY option allows the programmer to start sequential reading or updating of an INDEXED data set from the first non-dummy record that has a key in a particular class; the class is identified by the inclusion of its generic key in the KEY option of a READ statement.[1] Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

In the following example, a key length of more than three bytes is assumed.

```
DCL IND FILE RECORD SEQUENTIAL KEYED
   UPDATE ENV (INDEXED GENKEY);
          .
          .
          .

READ FILE(IND) INTO(INFIELD) KEY ('ABC');
          .
          .
          .
NEXT:  READ FILE (IND) INTO (INFIELD);
          .
          .
          .
      GO TO NEXT;
```

The first READ statement causes the first non-dummy record in the data set whose key begins with 'ABC' to be read into INFIELD; each time the second READ statement is executed, the non-dummy record with the next higher key will be retrieved. Repeated execution of the second READ statement could result in reading records from higher key classes since no indication is given when the end of a key class is reached. It is the responsibility of the programmer to check each key if he does not wish to read beyond the key class. Any subsequent execution of the first READ statement would reposition the file to the first record of the key class 'ABC'.

If the data set contains no records with keys in the specified class, or if all the records with keys in the specified class are dummy records, the KEY condition is raised. The data set is then positioned either at the next record that has a higher key or at the end of the file.

Note how the presence or absence of the GENKEY option affects the execution of a READ statement that supplies a source key that is shorter than the key length specified in the KEYLEN subparameter of the DD statement that defines the data set. GENKEY causes the key to be interpreted as a generic key, and the data set is positioned to the first non-dummy record in the data set whose key begins with the source key. If the GENKEY option is not specified, a short source key is padded on the right with blanks to the specified key length, and the data set is positioned to the record that has this padded key(if such a record exists).

The use of the GENKEY option does not affect the result of supplying a source key whose length is greater than or equal to the specified key length. The source key, truncated on the right if necessary, identifies a specific record (whose key can be considered to be the only member of its class).

---

[1]Note that, although the first record having a key in a particular class can be retrieved by READ KEY, the actual key cannot be obtained unless the records have embedded keys, since the KEYTO option cannot be used in the same statement as the KEY option.

NUMBER OF CHANNEL PROGRAMS (NCP)

The NCP option specifies the number of incomplete input/output operations with the EVENT option that can be handled for the file at any one time. For consecutive and regional sequential files, it is an error to allow more than the specified number of events to be outstanding.

For indexed files, any excess operations are queued, and no exceptional condition is raised. However, specification of the number of channel programs required may aid optimization of I/O with an indexed file. The NCP option has no effect with a regional direct file.

The decimal integer constant specified with NCP must have a value in the range 1 through 99; otherwise, 1 is assumed.

This option is equivalent to the NCP subparameter of the DCB parameter of the DD statement.

TRACK OVERFLOW (TRKOFL)

Track overflow is a feature of the operating system which can be incorporated at system generation time; it requires the record overflow feature on the direct access storage control unit. Track overflow allows a record to overflow from one track to another. It is useful in achieving a greater data packing efficiency, and allows the size of a record to exceed the capacity of a track.

Note: Track overflow is not available for REGIONAL(3) or INDEXED data sets.

VARYING-LENGTH STRING OPTION (SCALARVARYING)

The SCALARVARYING option is used in the input/output of varying-length strings. The transmission of element varying-length strings using locate mode is possible only when this option is specified. This is achieved by the inclusion or recognition of a two-byte length prefix to an element varying-length string when the string is transmitted.

When storage is allocated for a varying-length string, the compiler includes a two-byte prefix that specifies the current length of the string. For an element varying-length string, this prefix is included on output, or recognized on input, only if SCALARVARYING is specified for the file.

When locate mode statements (LOCATE and READ SET) are used to create and read a data set with element varying-length strings, SCALARVARYING must be specified to indicate that a length prefix is present, since the pointer that locates the buffer will always be assumed to point to the start of the length prefix.

Notes:

1. When SCALARVARYING is specified and element varying-length strings are transmitted, the programmer must allow two bytes in the record length to include the length prefix.
2. A data set created using SCALARVARYING should be accessed only by a file that also specifies SCALARVARYING.
3. SCALARVARYING and CTLASA/CTL360 must not be specified for the same file, as this causes the first data byte to be ambiguous.
4. SCALARVARYING must not be used with data sets created by the PL/I (F) compiler; this compiler neither creates nor recognizes a length prefix.

KEY LENGTH OPTION (KEYLENGTH)

The KEYLENGTH option specifies the length of the recorded key for KEYED files. KEYLENGTH must be specified for INDEXED or REGIONAL(3) files.

KEY LOCATION OPTION (KEYLOC)

The KEYLOC option can be used with INDEXED data sets, when the data set is created, to specify the start position of an embedded key in a record. The position given must be within the limits:

$$1 \leq n \leq \text{recordsize} - \text{keylength} + 1$$

That is, the key cannot be larger than the record, and must be contained completely within the record.

If KEYLOC is not specified, the value of the RKP subparameter of the DCB parameter of the DD statement is used. If this subparameter is not specified, then RKP=0 is assumed.

Note:

1. The RKP value for a particular byte always differs from the KEYLOC value. See "Embedded Key", in "INDEXED ORGANIZATION", later in this chapter.
2. When KEYLOC is specified, the key is always part of the variable. When RKP is specified, the key is part of the variable only when RKP≥1
3. If SCALARVARYING is specified, the emabedded key must not immediately precede or follow the first byte: hence, KEYLOC must specify greater than 2.

DCB Subparameters

Some of the information that can be specified in the options of the ENVIRONMENT attribute can also be specified in the subparameters of the DCB parameter of a DD statement. The table gives a list of equivalents:

ENV Option	DCB Subparameter
Record format	RECFM
RECSIZE	LRECL
BLKSIZE	BLKSIZE
BUFFERS	BUFNO
CTLASA/CTL360	RECFM
NCP	NCP
TRKOFL	RECFM
KEYLENGTH	KEYLEN
KEYLOC	RKP
ASCII	ASCII
BUFOFF	BUFOFF

ASCII DATA SETS

Data sets on magnetic tape using ASCII may be created and accessed in PL/I. The implementation supports F, FB, U, D, and DB record formats. F, FB, and U formats are treated in the same way as with other data sets; D and DB formats, which correspond to V and VB formats with other data sets, are described below.

In addition to the record format, two other ENVIRONMENT options may be specified: ASCII, and the buffer offset option BUFOFF.

Only character data may be written onto an ASCII data set: when the data set is created, transmission must be from a character string variable. This variable may have the attribute VARYING as well as CHARACTER, but the two length bytes of a varying-length character string can not be transmitted; in other words, varying-length character strings can not be transmitted to an ASCII data set using a SCALARVARYING file. Also, data aggregates containing varying-length strings may not be transmitted.

Since an ASCII data set must be on magnetic tape, it must be of CONSECUTIVE organization. The associated file must be BUFFERED.

ASCII Option

This option specifies that the code used to represent data on the data set is ASCII.

BUFOFF Option and Block Prefix Fields

At the beginning of each block in an ASCII data set, there may be a field known as the block prefix field. It may be from one to 99 bytes long. The buffer offset option indicates the length of this field to data management, so that the accessing or creation of data is started at this offset from the beginning of each physical block. PL/I does not support access to this field, and in general it does not contain information which is used in these implementations. There is one situation in which data management does use information in the block prefix: with variable length records (that is, D- or DB-format records), the block prefix field may be used to record the length of the block. In this case, it is four bytes long and contains a right-aligned, decimal character value that gives the length of the block in bytes, including the block prefix field itself. It is then exactly equivalent to a block length field.

The format of the buffer offset option is BUFOFF[(n)]. A numerical value equal to the length of the prefix may be specified for "n". It may be specified as either a decimal integer constant or as a variable with the attributes FIXED BINARY(31,0) STATIC. Its minimum value is zero and its maximum is 99. The absence of a prefix length specification indicates that the block prefix is to be used as a block length field; it implies that the field is four bytes long. The length of the block is inserted in the prefix by data management.

On input, any ASCII data set may be accessed if it has a block prefix field of length one to 99 bytes, or no block prefix field at all; and it may be accessed whether or not the block prefix field is used as a block length field. On output, a data set using any one of the valid record formats may be created without a block prefix, but the only situation in which the creation of a block prefix is supported by PL/I is when it is used as a block length field. The only permissible buffer offset specification on output is therefore BUFOFF, with no prefix length specification.

The BUFOFF option may be used with ASCII data sets only.

D-format and DB-format Records

Each record may be of a different length. The two formats are:

D-format: The records are unblocked; each record constitutes a single block. Each record consists of:

Four control bytes
Data bytes

The four control bytes contain the length of the record; this value is inserted by data management and requires no action from the programmer. In addition, there may be, at the start of the block, a block prefix field, which may contain the length of the block.

DB-format: The records are blocked. All other information given for D-format applies to DB-format.

Default Rules

In addition to the rules given under "Record Format Defaults", the following rule applies:

If ASCII is not specified in either the ENVIRONMENT option or the DD statement, but one of BUFOFF, D, or DB is specified, then ASCII is assumed.

Consecutive Organization

In a data set with CONSECUTIVE organization, the records have no keys. When the data set is created, records are written consecutively in the order in which they are presented. The records can be retrieved only in the order in which they were written or in the reverse order when using the BACKWARDS attribute; therefore the associated file must have the SEQUENTIAL attribute. A CONSECUTIVE data set can have F-format, FB-format, FBS-format, V-format, VB-format, VS-format, VBS-format, D-format, DB-format, or U-format records. The BACKWARDS attribute cannot be specified when a data set has V-, VB-, VS-, VBS-, D-, or DB-format records.

Note the difference between the CONSECUTIVE option of the ENVIRONMENT attribute and the SEQUENTIAL attribute. CONSECUTIVE specifies the physical organization of a data set; SEQUENTIAL specifies how a file is to be processed. A data set with CONSECUTIVE organization must be associated with a SEQUENTIAL file; but a data set with INDEXED or REGIONAL organization can be associated with either a SEQUENTIAL or DIRECT file.

A CONSECUTIVE data set on magnetic tape can be read forwards or backwards. If the data set is to be read backwards, the associated file must have the BACKWARDS attribute. If a data set is first read or written forwards and then read backwards in the same program, the LEAVE option should be specified in the ENVIRONMENT attribute to prevent normal rewind when the file is closed (or, with a multi-volume data set, when volume-switching occurs). Variable-length record data sets cannot be read backwards.

Once a CONSECUTIVE data set has been created, the file that accesses it can be opened for SEQUENTIAL INPUT, OUTPUT or, for direct access data sets, UPDATE. If it is opened for OUTPUT, DISP=MOD must be specified in the DD statement; records can then be added to the end of the data set. (If DISP=MOD is not specified, the data set will be overwritten.) Figure 12.6 lists the data transmission statements and options that can be used to create and access a CONSECUTIVE data set.

SEQUENTIAL UPDATE

When a CONSECUTIVE data set is accessed by a SEQUENTIAL UPDATE file, a record must be retrieved with a READ statement before it can be updated by a REWRITE statement; however, every record that is retrieved need not be rewritten. A REWRITE statement will always update the last record read.

Consider the following:

```
READ FILE(F) INTO(A);
     .
     .
     .
READ FILE(F) INTO(B);
     .
     .
     .
REWRITE FILE(F) FROM(A);
```

The REWRITE statement updates the record which was read by the second READ statement. The record that was read by the first statement cannot be rewritten after the second READ statement has been executed.

Indexed Organization

A data set with INDEXED organization must be on a direct-access device. Its records, which can be either F-format or V-format records, blocked or unblocked, are arranged in logical sequence according to keys that are associated with each record. A key is a character string that can identify each record uniquely. Logical records are arranged in the data set in ascending key sequence according to the System/360 and System/370 collating sequence. Indexes associated with the data set are used by the operating system data-management routines to locate a record when the key is supplied.

Unlike CONSECUTIVE organization, INDEXED organization does not require every record to be accessed in sequential fashion. An INDEXED data set must be created sequentially; but, once it has been created, the associated file may have the attribute SEQUENTIAL or DIRECT as well as INPUT or UPDATE. When the file has the DIRECT attribute, records may be retrieved, added, deleted, and replaced at random.

File declaration[1]	Valid statements[2], with options that must appear	Other options that can also be used
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-expr) FROM(variable);	
	LOCATE variable FILE(file-expr);	SET(pointer-variable)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-expr) FROM(variable);	EVENT(event-variable)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-expr) INTO(variable);	
	READ FILE(file-expr) SET(pointer-variable);	
	READ FILE(file-expr) IGNORE(expression);	
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-expr) INTO(variable);	EVENT(event-variable)
	READ FILE(file-expr) IGNORE(expression);	EVENT(event-variable)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-expr) INTO(variable);	
	READ FILE(file-expr) SET(pointer-variable);	
	READ FILE(file-expr) IGNORE(expression);	
	REWRITE FILE(file-expr);	FROM(variable)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-expr) INTO(variable);	EVENT(event-variable)
	READ FILE(file-expr) IGNORE(expression);	EVENT(event-variable)
	REWRITE FILE(file-expr) FROM(variable);	EVENT(event-variable)

[1]The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT

[2]The statement READ FILE (file-expression) ; is a valid statement and is equivalent to:
READ FILE (file-expression) IGNORE (1);

Figure 12.6. Statements and options permitted for creating and accessing CONSECUTIVE data sets

Figure 12.7 lists the data-transmission statements and options that can be used to create and access an INDEXED data set.

KEYS

There are two kinds of keys, recorded keys and source keys. A recorded key is a character string that actually appears with each record in the data set to identify that record; its length cannot exceed 255 characters. A source key is the character-string value of the expression that appears in the KEY or KEYFROM option of a data transmission statement to identify the record to which the statement refers; for direct access of an INDEXED data set, each transmission statement must include a source key.

The length of the recorded keys in an INDEXED data set is defined by the KEYLENGTH environment option or the KEYLEN subparameter of the DD statement that defines the data set. If the length of a source key is greater than the specified length of the recorded keys, the source key is truncated on the right. If the source key is shorter than the specified key length, and GENKEY has not been specified, the source key is padded with blanks on the right to the specified length.

The recorded keys in an INDEXED data set may be separate from, or embedded within, the logical records. If the keys are embedded within the records, either the KEYLOC(n) environment option should be specified when the data set is created, or the subparameter RKP must be included in the DD statement for the associated data set (in the job step in which the data set is created), to give the location of the key within the record.

File declaration[1]	Valid statements[2], with options that must appear	Other options that can also be used
SEQUENTIAL OUTPUT	WRITE FILE(file-expr) FROM(variable) KEYFROM(expression);	
	LOCATE variable FILE(file-expr) KEYFROM(expression);	SET(pointer-variable)
SEQUENTIAL INPUT	READ FILE(file-expr) INTO(variable);	KEY(expression) or KEYTO (character-string-variable)
	READ FILE(file-expr) SET(pointer-variable);	KEY(expression) or KEYTO (character-string-variable)
	READ FILE(file-expr) IGNORE(expression);	
SEQUENTIAL UPDATE	READ FILE(file-expr) INTO(variable);	KEY(expression) or KEYTO (character-string-variable)
	READ FILE(file-expr) SET(pointer-variable);	KEY(expression) or KEYTO (character-string-variable)
	READ FILE(file-expr) IGNORE(expression);	
	REWRITE FILE(file-expr);	FROM(variable)
	DELETE FILE(file-expr);	KEY(expression)
DIRECT INPUT	READ FILE(file-expr) INTO(variable) KEY(expression);	EVENT(event-variable)
DIRECT UPDATE	READ FILE(file-expr) INTO(variable) KEY(expression);	EVENT(event-variable)
	REWRITE FILE(file-expr) FROM(variable) KEY(expression);	EVENT(event-variable)
	WRITE FILE(file-expr) FROM(variable) KEYFROM(expression);	EVENT(event-variable)
	DELETE FILE(file-expr) KEY(expression);	EVENT(event-variable)

Figure 12.7 (Part 1 of 2). Statements and options permitted for creating and accessing INDEXED data sets

Embedded Keys

The KEYLOC option specifies the absolute position of an embedded key from the start of the data in a record, while the RKP subparameter specifies the position of an embedded key relative to the start of the record.

Thus the equivalent KEYLOC and RKP values for a particular byte depends on:

1. The KEYLOC byte count starts at 1; the RKP count starts at 0

2. The record format:

For example, if the embedded key begins at the tenth byte of a record variable, then the specifications are:

```
Fixed length:  KEYLOC(10)
               RKP=9

Variable-length:  KEYLOC(10)
                  RKP=13
```

If KEYLOC is specified with a value equal to or greater than one, embedded keys exist in the record variable and on the data set. If KEYLOC is equal to zero, or is not specified, the RKP value is used; as a result, embedded keys may not always be present in the record variable or the data set. The effect of the use of either option is shown in figure 12.8.

DIRECT UPDATE EXCLUSIVE	READ FILE(file-expr) INTO(variable) KEY(expression);	EVENT(event-variable) and/or NOLOCK
	REWRITE FILE(file-expr) FROM(variable) KEY(expression);	EVENT(event-variable)
	WRITE FILE(file-expr) FROM(variable) KEYFROM(expression);	EVENT(event-variable)
	DELETE FILE(file-expr) KEY(expression);	EVENT(event-variable)
	UNLOCK FILE(file-expr) KEY(expression);	

[1]The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if any of the options KEY, KEYFROM, and KEYTO is used, it must also include the attribute KEYED.

[2]The statement: READ FILE (file-expression); is equivalent to the statement: READ FILE (file-expression) IGNORE (1);

Note: The attribute UNBUFFERED is ignored and BUFFERED is assumed for INDEXED SEQUENTIAL and SEQUENTIAL files.

Use of the DELETE statement is invalid if OPTCD=I (DCB subparameter) was not specified when the data set was created or if the RKP subparameter is zero for FB records, or four for V and VB records.

Figure 12.7 (Part 2 of 2). Statements and options permitted for creating and accessing INDEXED data sets

KEYLOC (n)	RKP	Record Variable	Data Set	
			Unblocked records	Blocked records
n>1	RKP equivalent = n-1+C[1]	Key	Key	Key
n=1		Key	Key[2]	Key
n=0 or not specified	RKP =0	No key	No key	Key[3]
	RKP>0	Key	Key	Key

Note: [1] C = number of control bytes, if any; C=0 for fixed-length records
C=4 for variable-length records
[2] In this instance the key is not recognized by data management
[3] Each logical record in the block has a key

Figure 12.8. Effect of KEYLOC and RKP values of establishing embedded keys in record variables or data sets

Programs written for the PL/I F Compiler which use records with embedded keys can be compiled without alteration to the ENVIRONMENT attribute for the inclusion of the KEYLOC option, if the original RKP subparameter is specified when the recompiled program is executed.

The use of embedded keys obviates the need for the KEYTO option during sequential input, but the KEYFROM option is still required for output. (However, the data specified by the KEYFROM option may be the embedded key portion of the record variable itself.) In a data set with unblocked records, a separate recorded key precedes each record, even when there is already an embedded key; If the records are blocked, the key of only the last record in each block is recorded separately in front of the block.

During the execution of a WRITE statement that adds a record to a data set with embedded keys, the value of the expression in the KEYFROM option is assigned to the embedded key position in the record variable. Note that a record variable can be declared as a structure with an embedded key declared as a structure member, but that such an embedded key must not be declared as a VARYING string.

For a LOCATE statement, the KEYFROM string is assigned to the embedded key when the next operation on the file is encountered.

DUMMY RECORDS

Records within an INDEXED data set are either actual records containing valid data or dummy records. A dummy record is identified by the constant (8)'1'B in its first byte. Dummy records can be inserted by the programmer, or can be created by deleting records. They can be replaced by valid data records having the same keys as the dummy records. The subparameter OPTCD=L must be included in the DD statement that defines the data set when it is created, so that dummy records are recognized and not retrieved by READ statements.

CREATING A DATA SET

When an INDEXED data set is being created, the associated file must be opened for SEQUENTIAL OUTPUT, and the records must be presented in the order of ascending key values. (If there is an error in the key sequence, the KEY condition will be raised.) A DIRECT file cannot be used for the creation of an INDEXED data set.

Once an INDEXED data set has been created, the file that accesses it can be opened for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. In the case of F-format records, it can also be opened for OUTPUT to add records at the end of the data set. The keys for these records must have higher values than the existing keys for that data set and must be in ascending order. The storage allocated for a data set can be increased when it is required for output.

SEQUENTIAL ACCESS

A SEQUENTIAL file that is used to access an INDEXED data set may be opened with either the INPUT or the UPDATE attribute. The data transmission statements need not include source keys, nor need the file have the KEYED attribute. Sequential access is in order of ascending recorded-key values; records are retrieved in this order, and not necessarily in the order in which they were added to the data set. Dummy records are not retrieved if the DD statement that defined the data set included the subparameter OPTCD=L.

Except that the EVENT option cannot be used, rules governing the relationship between the READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses an INDEXED data set are identical to those for a CONSECUTIVE data set (described above).

Embedded keys in a record to be updated must not be altered. The modified record must always overwrite the updated record in the data set.

Additionally, records can be effectively deleted from the data set; a DELETE statement marks a record as a dummy by putting (8)'1'B in the first byte. The DELETE statement should not be used to process a data set with F-format blocked records and either KEYLOC=1 or RKP=0, or V- or VB-format records and either KEYLOC=1 or RKP=4. (The code (8) '1'B would overwrite the first byte of the recorded key.) Note that the EVENT option is not supported for SEQUENTIAL access of INDEXED data sets.

During sequential access of an INDEXED data set, it is possible to read a particular record by supplying a source key in the KEY option of a READ statement, and to continue sequential reading from that point in the data set. (The associated

file must have the KEYED attribute.) Thus, a READ statement that includes the KEY option will cause the record, whose key is supplied, to be read; a subsequent READ statement without the KEY option will cause the record with the next higher recorded key to be read (even if the keyed record has not been found).

The effect of supplying a source key that is shorter than the recorded keys in the data set differs according to whether or not the GENKEY option is specified in the ENVIRONMENT attribute. In the absence of the GENKEY option, the source key is padded on the right with blanks to the length specified in the KEYLENGTH option of the ENVIRONMENT attribute, and the record with this padded key is read (if such a record exists). If the GENKEY option is specified, the source key is interpreted as a generic key, and the first record with a key in the class identified by this generic key is read. (Refer to "Key Classification," above.)

DIRECT ACCESS

A DIRECT file that is used to access an INDEXED data set may be opened with either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

A DIRECT UPDATE file can be used to retrieve, add, delete, or replace records in an INDEXED data set according to the following conventions:

1. Retrieval: If the DD statement that defined the data set included the subparameter OPTCD=L, dummy records are not made available by a READ statement. (The KEY condition is raised.)

2. Addition: A WRITE statement that includes a unique key causes a record to be inserted into the data set. If the key is the same as the recorded key of a dummy record, the new record replaces the dummy record. If the key is the same as the recorded key of a record that is not marked as deleted, or if there is no space in the data set for the record, the KEY condition is raised.

3. Deletion: The record specified by the source key in a DELETE statement is retrieved, marked as deleted, and rewritten into the data set. The effect of the DELETE statement is to insert the value (8) '1'B in the first byte of the data in a record. Deletion is possible only if OPTCD=L was specified in the DD statement that defined the data set when it was created. If the data set has F-format blocked records with RKP=0 or KEYLOC=1, or V-format records with RKP=4 or KEYLOC=1, records cannot be deleted. (The code (8)'1'B would overwrite the embedded keys.)

4. Replacement: The record specified by a source key in a REWRITE statement is replaced by the new record. If the data set contains F-format blocked records, a record replaced with a REWRITE statement causes an implicit READ statement to be executed unless the previous I/O statement was a READ statement that obtained the record to be replaced. If the data set contains V-format records and the updated record has a length different from that of the record read, the whole of the remainder of the track will be moved, and may cause data to be moved to an overflow track.

Regional Organization

A data set with REGIONAL organization is divided into regions, each of which is identified by a region number, and each of which may contain one record or more than one record, depending on the type of REGIONAL organization. The regions are numbered in succession, beginning with zero, and a record may be accessed by specifying its region number, and perhaps a key, in a data transmission statement.

REGIONAL organization of a data set permits the programmer to control the physical placement of records in the data set, and enables him to optimize the access time for a particular application. Such optimization is not available with CONSECUTIVE or INDEXED organization, in which successive records are written either in strict physical sequence or in logical sequence depending on ascending key values; neither of these methods takes full advantage of the characteristics of direct-access storage devices. REGIONAL data sets are confined to direct-access devices.

A REGIONAL data set can be created in a similar manner to a CONSECUTIVE or INDEXED data set, records being presented in the order of ascending region numbers; alternatively, direct access can be used, in which records can be presented in random sequence and inserted directly into preformatted regions. Once a REGIONAL data

set has been created, it can be accessed by a file with the attributes SEQUENTIAL or DIRECT as well as INPUT or UPDATE. Note that neither a region number nor a key need be specified if the data set is associated with a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file. When the file has the DIRECT attribute, records can be retrieved, added, deleted, and replaced at random.

Records within a REGIONAL data set are either actual records containing valid data or dummy records. The nature of the dummy records depends on the type of REGIONAL organization; the three types of REGIONAL organization are described below.

Figure 12.9 lists the data transmission statements and options that can be used to create and access a REGIONAL data set.

KEYS

There are two kinds of keys, recorded keys and source keys. A recorded key is a character string that immediately precedes each record in the data set to identify that record; its length cannot exceed 255 characters. A source key is the character-string value of the expression that appears in the KEY or KEYFROM option of a data transmission statement to identify the record to which the statement refers. When a record in a REGIONAL data set is accessed, the source key gives a region number, and may also give a recorded key.

The length of the recorded keys in a REGIONAL data set is defined by the KEYLENGTH option of the ENVIRONMENT attribute, or the KEYLEN subparameter on the DD statement. Unlike the keys for INDEXED data sets, recorded keys in a REGIONAL data set are never embedded within the record.

TYPES OF REGIONAL ORGANIZATION

There are three types of REGIONAL organization:

1. A REGIONAL(1) data set contains F-format records that do not have recorded keys. Each region in the data set contains only one record; therefore, each region number corresponds with a relative record position within the data set.

2. A REGIONAL(2) data set contains F-format records that have recorded keys. Each region in the data set contains only one record. Direct access to a REGIONAL(2) data set employs the region number specified in a source key to locate the required region. REGIONAL(2) differs from REGIONAL(1) in that records are not necessarily in the specified regions. The specified region identifies a starting-point; a search is then made for a record with the given recorded key starting at the beginning of the track containing the region specified.

3. A REGIONAL(3) data set contains F-format, V-format, VS-format or U-format records with recorded keys. Each region in the data set corresponds with a track on a direct-access device, and can normally contain one or more records. Direct access of a REGIONAL(3) data set employs the region number specified in a source key to locate the required region. Once the region has been located, a sequential search for space to add a record, or for a record that has a recorded key identical with that supplied in the source key, can be made. VS-format records may span more than one region.

REGIONAL(1) ORGANIZATION

In a REGIONAL(1) data set, since there are no recorded keys, the region number serves as the sole identification of a particular record. The character-string value of the source key should represent an unsigned decimal integer that should not exceed 16777215. If the region number exceeds this figure, it is treated as modulo 16777216: 16777226, for instance, is treated as 10. Only the characters 0 through 9 and the blank character are valid in the source key; leading blanks are interpreted as zeros; embedded blanks are not permitted in the number; the first embedded blank, if any, will terminate the region number. If more than eight characters appear in the source key, only the rightmost eight are used as the region number; if there are fewer than eight characters, blanks (interpreted as zeros) are assumed on the left.

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE (file-expr) FROM(variable) KEYFROM(expression); LOCATE variable FILE(file-expr) KEYFROM(expression);	 SET(pointer-variable)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-expr) FROM(variable) KEYFROM(expression);	EVENT(event-variable)
SEQUENTIAL INPUT BUFFERED	READ FILE (file-expr) INTO (variable); READ FILE(file-expr) SET(pointer-variable); READ FILE(file-expr) IGNORE(expression);	KEYTO (character-string-variable) KEYTO (character-string-variable)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-expr) INTO(variable); READ FILE (file-expr) IGNORE(expression);	EVENT(event-variable) and/or KEYTO (character-string-variable) EVENT(event-variable)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-expr) INTO(variable); READ FILE(file-expr) SET(pointer-variable); READ FILE(file-expr) IGNORE(expression); REWRITE FILE(file-expr);	KEYTO (character-string-variable) KEYTO (character-string-variable) FROM(variable)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-expr) INTO(variable); READ FILE(file-expr) IGNORE(expression); REWRITE FILE(file-expr) FROM(variable);	EVENT (event-variable) and/or KEYTO (character-string-variable) EVENT(event-variable) EVENT(event-variable)
DIRECT OUTPUT	WRITE FILE(file-expr) FROM(variable) KEYFROM(expression);	EVENT (event-variable)
DIRECT INPUT	READ FILE(file-expr) INTO(variable) KEY(expression);	EVENT(event-variable)
DIRECT UPDATE	READ FILE(file-expr) INTO(variable) KEY(expression); REWRITE FILE(file-expr) FROM(variable) KEY(expression); WRITE FILE(file-expr) FROM(variable) KEYFROM(expression); DELETE FILE(file-expr) KEY(expression);	EVENT(event-variable) EVENT(event-variable) EVENT(event-variable) EVENT(event-variable)

Figure 12.9 (Part 1 of 2). Statements and options permitted for creating and accessing REGIONAL data sets

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
DIRECT INPUT EXCLUSIVE	READ FILE(file-expr) INTO(variable) KEY(expression);	EVENT(event-variable) and/or NOLOCK
DIRECT UPDATE EXCLUSIVE	READ FILE(file-expr) INTO(variable) KEY(expression);	EVENT(event-variable) and/or NOLOCK
	REWRITE FILE(file-expr) FROM(variable) KEY(expression);	EVENT(event-variable)
	WRITE FILE(file-expr) FROM(variable) KEYFROM(expression);	EVENT(event-variable)
	DELETE FILE(file-expr) KEY(expression);	EVENT(event-variable)
	UNLOCK FILE(file-expr) KEY(expression);	

[1]The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if any of the options KEY, KEYFROM, and KEYTO is used, it must also include the attribute KEYED.
The statement: READ FILE (file-expression); is equivalent to the statement: READ FILE (file-expression) IGNORE(1);

Figure 12.9 (Part 2 of 2). Statements and options permitted for creating and accessing REGIONAL data sets

Dummy Records

Records in a REGIONAL(1) data set are either actual records containing valid data or dummy records. A dummy record in a REGIONAL(1) data set is identified by the constant (8)'1'B in its first byte. Although such dummy records are automatically inserted in the data set either when it is created or when a record is deleted, they are not ignored when the data set is read; the PL/I program must be prepared to recognize them. Dummy records can be replaced by valid data. Note that if the programmer inserts (8)'1'B in the first byte, the record will be lost if the file is copied onto a data set whose dummy records are not retreived.

Creating a REGIONAL(1) Data Set

A REGIONAL(1) data set can be created either sequentially or by direct access.

When a SEQUENTIAL OUTPUT file is used to create the data set, the opening of the file causes all tracks on the data set to be cleared, and a capacity record to be written at the beginning of each track to record the amount of space available on that track. Records must be presented in ascending order of region numbers; any region that is omitted from the sequence is filled with a dummy record. If there is an error in the sequence, or if a duplicate key is presented, the KEY condition will be raised. When the file is closed, any space remaining at the end of the current extent is filled with dummy records.

If a data set is created using a buffered file, and the last WRITE or LOCATE statement before the file is closed attempts to transmit a record beyond the limits of the data set, the CLOSE statement may raise the ERROR condition.

If a DIRECT OUTPUT file is used to create the data set, the whole of the primary extent allocated to the data set is filled with dummy records when the file is opened.

Once a REGIONAL(1) data set has been created, the file that accesses it can be opened for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. It can be opened for OUTPUT only if the existing data set is to be overwritten.

Sequential Access

A SEQUENTIAL file that is used to process a REGIONAL(1) data set may be opened with either the INPUT or UPDATE attribute. The data transmission statements must not include the KEY option; but the file may have the KEYED attribute, since the KEYTO option can be used. If the

character-string variable specified in the KEYTO option has more than eight characters, the value returned (the region number) is padded on the left with blanks; if it has fewer than eight characters, it is truncated on the left.

Sequential access is in the order of ascending region numbers. All records are retrieved, whether dummy or actual, and the PL/I program should be prepared to recognize dummy records.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(1) data set are identical to those for a CONSECUTIVE data set (described above).

Direct Access

A DIRECT file that is used to process a REGIONAL(1) data set may be opened with either the INPUT, or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

A DIRECT UPDATE file can be used to retrieve, add, delete, or replace records in a REGIONAL(1) data set according to the following conventions:

1. Retrieval: All records, whether dummy or actual, are retrieved. The program must be prepared to recognize dummy records.

2. Addition: A WRITE statement substitutes a new record for the existing record (actual or dummy) in the region specified by the source key.

3. Deletion: The record specified by the source key in a DELETE statement is converted to a dummy record.

4. Replacement: The record specified by the source key in a REWRITE statement, whether dummy or actual, is replaced.

REGIONAL(2) ORGANIZATION

In a REGIONAL(2) data set, each record is identified by a recorded key that immediately precedes the record. The actual position of a record in the data set relative to other records is determined not by its recorded key, but by the region number that is supplied in the source key of the WRITE statement that adds the record to the data set.

When a record is added to the data set by direct access, it is written with its recorded key in the first available space after the beginning of the track that contains the region specified. When a record is read by direct access, the search for a record with the appropriate recorded key begins at the start of the track that contains the region specified. Unless it is limited by the LIMCT subparameter of the DD statement that defines the data set, the search for a record or for space to add a record continues right through to the end of the data set and then from the beginning until the whole of the data set has been covered. The closer a record is to the specified region, the more quickly it can be accessed.

Source Keys

The character-string value of the source key can be thought of as having two logical parts, the region number and a comparison key. On output, the comparison key is written as the recorded key; for input, it is compared with the recorded key.

The rightmost eight characters of the source key make up the region number, which must be an unsigned decimal integer that does not exceed 16777215. If the region number exceeds this figure, it is treated modulo 16777216: 16777226 is treated as 10. The region specification can include only the characters 0 through 9 and the blank character; leading blanks are interpreted as zeros; embedded blanks are not permitted in the number; the first embedded blank, if any, will terminate the region number. The comparison key is a character string which occupies the left hand side of the source key, and may overlap or be distinct from the region number, from which it can be separated by other, non-significant , characters. The length of the comparison key is specified by either the KEYLEN subparameter of the DD statement for the data set or the KEYLENGTH option of the ENVIRONMENT attribute. If the source key is shorter than the specified key length, it is extended on the right with blanks. To retrieve a record, the comparison key must exactly match the recorded key of the record. The comparison key can include the region number, in which case the source key and the comparison key are identical; alternatively, part of the source key may not be used. The length of the comparison key is always equal to KEYLENGTH or KEYLEN; if the source key is longer then KEYLEN+8, the characters in the

source key between the comparison key and the region number are ignored.

Consider the following examples of source keys (the character "b" represents a blank):

```
KEY ('JOHNbDOEbbbbbb12363251')
```

The rightmost eight characters make up the region specification, the relative number of the record. Assume that the associated DD statement has the subparameter KEYLEN=14. In retrieving a record, the search will begin with the beginning of the track which contains the region number 12363251, until the record is found having the recorded key of JOHNbDOEbbbbbb.

If the subparameter were KEYLEN=22, the search still would begin at the same place, but since the comparison and the source key are the same length, the search would be for a record having the recorded key 'JOHNbDOEbbbbbb12363251'.

```
KEY ('JOHNbDOEbbbbbbDIVISIONb423bbbb34627')
```

In this example, the rightmost eight characters contain leading blanks, which are interpreted as zeros. The search will begin at region number 00034627. If KEYLEN=14 is specified, the characters DIVISIONb423b will be ignored.

Assume that COUNTER is declared FIXED BINARY (21) and NAME is declared CHARACTER(15). The key might be specified as :

```
KEY (NAME || COUNTER)
```

The value of COUNTER will be converted to a character string of eleven characters. (The rules for conversion specify that a binary value of this length, when converted to character, will result in a string of length 11, three blanks followed by eight decimal digits.) The value of the rightmost eight characters of the converted string will be taken to be the region specification. Then if the keylength specification is KEYLEN=15, the value of NAME will be taken to be the comparison specification.

Dummy Records

A REGIONAL(2) data set can contain dummy records. A dummy record consists of a dummy key and dummy data. A dummy key is identified by the constant (8) '1'B in its first byte. The first byte of the data contains the sequence number of the record on the track.

Dummy records can be replaced by valid data. They are inserted automatically either when the data set is created or when a record is deleted, and they are ignored when the data set is read. (Unlike INDEXED data sets, REGIONAL data sets do not require the subparameter OPTCD=L in the DD statement.)

Creating a Data Set

A REGIONAL(2) data set can be created either sequentially or by direct access. In either case, when the file associated with the data set is opened, the data set is initialized with capacity records specifying the amount of space available on each track.

When a SEQUENTIAL OUTPUT file is used to create the data set, records must be presented in ascending order of region numbers; any region that is omitted from the sequence is filled with a dummy record. If there is an error in the sequence, including an attempt to place more than one record in the same region, the KEY condition will be raised. When the file is closed, any space remaining at the end of the current extent is filled with dummy records.

If a data set is created using a buffered file, and the last WRITE or LOCATE statement before the file is closed attempts to transmit a record beyond the limits of the data set, the CLOSE statement may raise the ERROR condition.

If a DIRECT OUTPUT file is used to create the current extent of a data set, the whole of the primary extent allocated to the data set is filled with dummy records when the file is opened. Records can be presented in random order, and no condition is raised by duplicate keys. Each record is substituted for the first dummy record on the track that contains the region specified in the source key; if there are no dummy records on the track, the record is substituted for the first dummy record encountered on a subsequent track, unless the LIMCT subparameter specifies that the search cannot reach beyond this track. (Note that it is possible to place records with identical recorded keys in the data set.)

Once a REGIONAL(2) data set has been created, the file that accesses it can be opened for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. It cannot be opened for OUTPUT.

Sequential Access

A SEQUENTIAL file that is used to process a REGIONAL(2) data set may be opened with either the INPUT or the UPDATE attribute. The data transmission statements must not include the KEY option, but the file may have the KEYED attribute since the KEYTO option can be used. The KEYTO option specifies that the recorded key only is to be assigned to the specified variable. If the character-string variable specified in the KEYTO option has more characters than are specified in the KEYLEN subparameter, the value returned (the recorded key) is extended on the right with blanks; if it has fewer characters than specified by KEYLEN, the value returned is truncated on the right.

Sequential access is in the physical order in which the records exist on the data set, not necessarily in the order in which they were added to the data set. The recorded keys do not affect the order of sequential access. Dummy records are not retrieved.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(2) data set are identical with those for a CONSECUTIVE data set (described above).

Direct Access

A DIRECT file that is used to process a REGIONAL(2) data set may be opened with either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute. The search for each record is commenced at the start of the track containing the region number indicated by the key.

1. Retrieval: Dummy records are not made available by a READ statement. The KEY condition is raised if a record with the specified recorded key is not found.

2. Addition: A WRITE statement substitutes the new record for the first dummy record on the track containing the region specified by the source key. If there are no dummy records on this track, and an extended search is permitted by the LIMCT subparameter, the new record replaces the first dummy record encountered during the search.

3. Deletion: The record specified by the source key in a DELETE statement is converted to a dummy record.

4. Replacement: The record specified by the source key in a REWRITE statement must exist; a REWRITE statement cannot be used to replace a dummy record. If it does not exist, the KEY condition is raised.

Note that if a track contains records with duplicate recorded keys, the record farthest from the beginning of the track will never be retrieved during direct access.

REGIONAL(3) ORGANIZATION

A REGIONAL(3) data set differs from a REGIONAL(2) data set (described above) only in the following respects:

1. Each region number identifies a track on the direct-access device that contains the data set; the region number should not exceed 32767. A region in excess of 32767 is treated modulo 32768; 32778 is treated as 10.

2. A region can contain one or more records, or a segment of a VS-format record.

3. The data set can contain F-format, V-format, VS-format, or U-format records. Dummy records can be created, but a data set that has V-format, VS-format, or U-format records is not preformatted with dummy records because the lengths of records cannot be known until they are written; however, all tracks in the primary extent are cleared and the operating system maintains a capacity record at the beginning of each track, in which it records the amount of space available on that track.

Source keys for a REGIONAL(3) data set are interpreted exactly as those for a REGIONAL(2) data set, and the search for a record or space to add a record is conducted in a similar manner.

Dummy Records

Dummy records for REGIONAL(3) data sets with F-format records are identical with those for REGIONAL(2) data sets.

V-format, VS-format, and U-format dummy records are identified by the fact that they have dummy recorded keys ((8)'1'B in the first byte). The four control bytes in each V-format and VS-format dummy record are retained, but otherwise the contents of V-format, VS-format, and U-format dummy records are undefined. V-format, VS-format, and U-format format records are converted to dummy records only when a record is deleted; they cannot be reconverted to valid records.

Creating a Data Set

A REGIONAL(3) data set can be created either sequentially or by direct access. In either case, when the file associated with the data set is opened, the data set is initialized with capacity records specifying the amount of space available on each track.

When a SEQUENTIAL OUTPUT file is used to create the data set, records must be presented in ascending order of region numbers, but the same region number can be specified for successive records. If there is an error in the sequence, the KEY condition will be raised. If a track becomes filled by records for which the same region number was specified, the region number is automatically incremented by one; an attempt to add a further record with the same region number will raise the KEY condition (sequence error).

If a data set is created using a buffered file, and the last WRITE or LOCATE statement before the file is closed attempts to transmit a record beyond the limits of the data set, the CLOSE statement may raise the ERROR condition.

If a DIRECT OUTPUT file is used to create the data set, the whole of the primary extent allocated to the data set is initialized when the data set is opened; for F-format records, the space is filled with dummy records, and for V-format, VS-format, and U-format records, the capacity record for each track is written to indicate empty tracks. Records can be presented in random order, and no condition is raised by duplicate keys or duplicate region specifications. If the data set has F-format records, each record is substituted for the first dummy record in the region (track) specified in the source key; if there are no dummy records on the track, and an extended search is permitted by the LIMCT subparameter, the record is substituted for the first dummy record encountered during the search. If the data set has V-format, VS-format, or U-format records, the new record is inserted on the specified track, if sufficient space is available; otherwise, if an extended search is permitted, the new record is inserted in the next available space.

Note that for spanned records space may be required for overflow onto subsequent tracks.

Once a REGIONAL(3) data set has been created, the file that accesses it can be opened for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. It can only be opened for OUTPUT if the entire existing data set is to be deleted and replaced.

Sequential Access

A SEQUENTIAL file that is used to access a REGIONAL(3) data set may be opened with either the INPUT or UPDATE attribute. The data transmission statements must not include the KEY option, but the file may have the KEYED attribute since the KEYTO option can be used. The KEYTO option specifies that the recorded key only is to be assigned to the specified variable. If the character-string variable specified in the KEYTO option has more characters than are specified in the KEYLEN subparameter, the value returned (the recorded key) is extended on the right with blanks; if it has fewer characters than specified by KEYLEN, the value returned is truncated on the right.

Sequential access is in the order of ascending relative tracks. Records are retrieved in this order, and not necessarily in the order in which they were added to the data set; the recorded keys do not affect the order of sequential access. Dummy records are not retrieved.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(3) data set are identical with those for a CONSECUTIVE data set (described above).

Direct Access

A DIRECT file that is used to process a REGIONAL(3) data set may be opened with

either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

1. Retrieval: Dummy records are not made available by a READ statement. The KEY condition is raised if a record with the specified recorded key is not found.

2. Addition: In a data set with F-format records, a WRITE statement substitutes the new record for a dummy record in the region (track) specified by the source key. If there are no dummy records on the specified track, and an extended search is permitted by the LIMCT subparameter, the new record replaces the first dummy record encountered during the search. If the data set has V-format, VS-format, or U-format records, a WRITE statement inserts the new record after any records already present on the specified track if space is available; otherwise, if an extended search is permitted, the new record is inserted in the next available space.

3. Deletion: A record specified by the source key in a DELETE statement is converted to a dummy record. The space formerly occupied by an F-format record can be re-used; space formerly occupied by V-format, VS-format, or U-format records is not available for re-use.

4. Replacement: The record specified by the source key in a REWRITE statement must exist; a REWRITE statement cannot be used to replace a dummy record. When a VS-format record is replaced, the new one must not be shorter than the old.

Note: If a track contains records with duplicate recorded keys, the record farthest from the beginning of the track will never be retrieved during direct access.

Teleprocessing

The teleprocessing facilities of PL/I are provided by an extension of the basic record-oriented transmission facilities with the addition of the TRANSIENT file attribute and the PENDING condition. The implementation provides a communicating link between the PL/I message processing programs using these features, and the teleprocessing facilities of the operating system.

A teleprocessing message control program (MCP) handles messages originating from and destined for a number of remote terminals or a number of PL/I message processing programs. Each origin or destination associated with a message is identified by a name carried within that message. Messages are transmitted to and from a PL/I message processing program via queues in main storage. (These queues are supported by corresponding intermediate queues in a disk data set. The PL/I program has access only to the main storage queues, by means of intermediate buffers for each file.)

The "data set" associated with each TRANSIENT file is in fact an input or output message queue set up by the MCP. A READ statement for the file will take the next message (or the next record from the current message) from the associated queue, assign the data part to the variable named in the READ INTO option (or set a pointer to point to the data in a READ SET buffer), and save the origin name by assigning it to the variable named in the KEYTO option. (The PENDING condition is raised if the input queue is empty when the READ statement is executed.) A WRITE or LOCATE statement will transmit the processed message or record to the output queue, using the element expression specified in the KEYFROM option to identify the destination.

ENVIRONMENT Attribute

A message can consist of one logical record, or several logical records, on the teleprocessing data set. The programmer must specify whether a complete message (which may be several logical records) or only one logical record is to be transmitted to his PL/I program at each I/O operation. He must also specify the size of the record variable (or input and output buffer, for locate mode), and the number of intermediate buffers required for each message. This information can be provided by means of the appropriate options of the ENVIRONMENT attribute.

The options, and their meanings, are:

TP(M): Each I/O operation in the PI/I program transmits a complete message

TP(R): Each I/O operation in the PI/I program transmits one logical record

RECSIZE: Size of the record variable (or input or output buffer, for locate mode) in the PL/I program. If the

TP(M) option is used, this size should, if possible, be equal to the length of all the logical records that constitute the message. If it is smaller, part of the message will be lost. If it is greater, the contents of the last part of the variable (or buffer) are undefined. If the TP(R) option is specified, this size must be the same as the logical record length.

BUFFERS: Number of intermediate buffers required to contain the longest message to be transmitted. If a message is too long for the buffers specified, extra buffers must be obtained before processing can continue, which increases execution time. The extra buffers are obtained by the operating system; the programmer need not take any action.

These are the only options of the ENVIRONMENT attribute that can be specified for a TRANSIENT file.

TRANSIENT Attribute

The TRANSIENT attribute, which is an alternative to the DIRECT and SEQUENTIAL attributes, indicates that the contents of the data set associated with the file are re-established each time the data set is accessed. In effect, this means that records can be continually added to the data set by one program during the execution of another program that continually removes records from the data set. Thus the data set can be considered to be a continuous queue through which the records pass in transit between the message control program and the message processing program. The queue is always accessed sequentially.

The data set associated with a TRANSIENT file differs from those associated with DIRECT and SEQUENTIAL files in that its contents are dynamic; reading a record removes it from the data set. Such a data set can never be created by a DIRECT or SEQUENTIAL file. (It can, however, be accessed as a CONSECUTIVE data set by a SEQUENTIAL file.)

The TRANSIENT attribute can be specified only for RECORD KEYED BUFFERED files with either the INPUT or the OUTPUT attribute. (The EVENT option cannot be used for teleprocessing operations.) The file must also have the ENVIRONMENT attribute with the options appropriate to teleprocessing.

For TRANSIENT files, the file name or title must be the ddname of a DD statement. The message queue data set is identified in the DD statement by the QNAME parameter. For a TRANSIENT OUTPUT file, the element expression specified in the KEYFROM option must have as its value a recognized terminal or program identification.

Error Handling

The conditions that can be raised during teleprocessing transmission are TRANSMIT, KEY, RECORD, ERROR, and PENDING.

The TRANSMIT condition can be raised only on input, and is as described for other types of transmission.

The RECORD condition is raised in the same circumstances as for other types of transmission. (The messages and records are treated as V-format records.)

The ERROR condition is raised as for other types of transmission; it is also raised when the expression in the KEYFROM option is missing or detectably invalid. Note that if the expression is syntactically valid but does not represent an origin or a destination name recognized by the MCP, the KEY condition is raised.

The PENDING condition can be raised only during execution of a READ statement for a TRANSIENT file. It is raised when the associated queue is empty; standard system action is to wait at the READ statement until a message is available. When the PENDING condition is raised, the value returned by the ONKEY built-in function is a null string.

Note: When the TP(R) option is specified in the ENVIRONMENT attribute, a message is transmitted one record at a time. There is no ON-condition or other automatic means for detecting the end of the message; the user is responsible for arranging the indication of the end of the message (possibly by using the first record as a header giving the necessary control information.)

Statements and Options

The READ statement is used for input, with either the INTO option or the SET option; the KEYTO option must be given. The origin name is assigned to the variable named in the KEYTO option. If the origin name is shorter than the character-string variable

named in the KEYTO option, it is padded on the right with blanks. If the KEYTO variable is a varying-length string, its current length is set to that of the origin name. The origin name should not be longer than the KEYTO variable (if it is, it is truncated), but in any case will not be longer than 8 characters. The data part of the message or record is assigned to the variable named in the INTO option, or the pointer variable named in the SET option is set to point to the data in the READ SET buffer.

Either the WRITE or the LOCATE statement may be used for output; either statement must have the KEYFROM option -- the first eight characters of the value of the KEYFROM expression are used to identify the destination. The data part of the message is transmitted from the variable named in the FROM option of the WRITE statement; or, in the case of LOCATE, a pointer variable is set to point to the location of the data in the output buffer. When a message is transmitted by an OUTPUT TRANSIENT file as a number of logical records, the end of the message must be indicated by closing the file.

The list of statements and options permitted for TRANSIENT files is given in tabular form in figure 12.10. Some examples follow:

```
DECLARE (IN INPUT,OUT OUTPUT) FILE
   TRANSIENT ENV(TP(M) RECSIZE(124)),
   (INREC, OUTREC) CHARACTER(120)
   VARYING, TERM CHARACTER(8);

READ FILE(IN) INTO(INREC) KEYTO(TERM);
.
.
.
WRITE FILE(OUT) FROM(OUTREC)
  KEYFROM(TERM);
```

The above example illustrates the use of move mode in teleprocessing applications. Note that the files IN and OUT are given the attributes KEYED and BUFFERED because TRANSIENT implies these attributes. The input buffer for file IN contains the next message (or record from a message, depending on the message format) from the input queue. The input queue will also be named IN unless the file has been opened with a TITLE option specifying a different queue name. The message format is determined by the programmer, and the file declaration for IN must include this information in the ENVIRONMENT attribute.

The READ statement causes the message or record to be moved from the input buffer into the variable INREC; if the buffer is empty when the READ statement is executed (i.e., there are no messages in the queue), the PENDING condition is raised. The standard system action for the condition is to suspend execution and wait until a message is available. The name of the origin is assigned to TERM.

After processing, the message or record is held in OUTREC. The WRITE statement moves it to the output buffer, together with the value of TERM (which will still contain the origin name unless another name has been assigned to it during processing). From the buffer, the message will be automatically transmitted to the correct queue for the destination, as specified by the value of TERM.

Since the output queue is determined from the destination name, the file name OUT has no significance outside the PL/I program. However, the file would need the TRANSIENT, KEYED, and BUFFERED attributes, and the correct message format in the ENVIRONMENT attribute.

```
DECLARE (IN INPUT,OUT OUTPUT) FILE
   TRANSIENT ENV(TP(M) RECSIZE(124)),
   MESSAGE CHARACTER(120) VARYING
   BASED(INPTR),
   TERM CHARACTER(8);

READ FILE(IN) SET(INPTR) KEYTO(TERM);
.
.
.
WRITE FILE(OUT) FROM(MESSAGE)
  KEYFROM(TERM);
```

This example is similar to the previous one, except that locate mode input is used; the message data is processed in the input buffer, using the based variable MESSAGE, which has been declared with the pointer variable INPTR. (The data of the message will be aligned on a doubleword boundary.) Note that the attribute TRANSIENT implies KEYED and BUFFERED. The WRITE statement moves the processed data from the input to the output buffer; otherwise its effect is as described for the WRITE statement in the first example.

The technique used in this example would be useful in applications where the differences between processed and unprocessed messages were relatively simple, since the maximum size of input and output messages would be the same. If the length and structure of the output message could vary widely, depending on the text of the input message, locate mode output could be used to advantage; after the input message had been read in, a suitable based variable could be located in the output buffer (using the LOCATE statement), where further processing would take place. The message would be transmitted immediately

File Declaration	Valid statements, with options that must appear	Other options that can also be used
TRANSIENT INPUT	READ FILE(file-expr) INTO(variable) KEYTO(character-string-variable); READ FILE(file-expr) SET(pointer-variable) KEYTO(character-string-variable);	
TRANSIENT OUTPUT	WRITE FILE(file-expr) FROM(variable) KEYFROM(expression); LOCATE variable FILE(file-expr) KEYFROM(expression);	 SET(pointer-variable)

[1]The complete file declaration would include the attribute FILE, RECORD, KEYED, BUFFERED, and the ENVIRONMENT attribute with either the TP(M) or the TP(R) option.

Figure 12.10. Statements and options permitted for TRANSIENT files

before execution of the next WRITE or LOCATE statement for the file.

Note that although the EVENT option is not permitted, data transmission could be overlapped with processing in an MVT operating system by means of the PL/I multitasking facilities described in chapter 17, "Multitasking". For example, the processing program could consist of a number of subtasks, each associated with a different queue. Each subtask processes the input from its associated queue, and transmits output to the required destination. As soon as the PENDING condition is raised in a subtask, another subtask could receive input or transmit output.

Summary of Record-oriented Transmission

The following points cover the salient features of record-oriented transmission:

1. A SEQUENTIAL file specifies that the accessing, creation, or modification of the data set records is performed in a particular order:

   CONSECUTIVE or REGIONAL data set; from the first record of the data set to the last record of the data set (or from the last to the first if the BACKWARDS attribute has been specified).

   INDEXED or REGIONAL(1) data set; in ascending order of key sequence.

2. A DIRECT file specifies that records may be processed in random order. The particular record is identified by a key.

3. Records in a data set that are accessed, created, or modified by a SEQUENTIAL file may or may not have recorded keys. If they do, the recorded keys may be extracted from the data set or placed into the data set by the KEYTO and KEYFROM options. REGIONAL(1) data sets may also be retrieved or created using the KEYTO and KEYFROM options respectively; the region number is specified as the key.

4. INDEXED KEYED files opened for SEQUENTIAL INPUT and SEQUENTIAL UPDATE may be positioned to a particular record within the data set either by a READ KEY or a DELETE KEY operation that specifies the key of the desired record. Thereafter, successive READ statements without the KEY option will access the following records in the data set sequentially.

5. Existing records of a data set in a SEQUENTIAL UPDATE file can be modified, ignored, or, if the data set is INDEXED, deleted. The DELETE statement used without the KEY option for this type of file specifies that the last record read is to be deleted.[1] The DELETE statement can be used with the KEY option to delete a specific record in a DIRECT UPDATE file; also, records can be added to such a file by means of the WRITE statement. An existing record in an UPDATE file can be replaced through use of a REWRITE statement.

6. When a file has the DIRECT, INPUT or UPDATE, and EXCLUSIVE attributes, it

---

[1]If the DELETE statement is used with a SEQUENTIAL file, the data set must have INDEXED organization.

is possible to protect individual records that are read from the data set. For an EXCLUSIVE file, any READ statement without a NOLOCK option automatically locks the record read. No other task operating upon the same data set can access a locked record until it is unlocked by the locking task. The record is protected from access by tasks in other jobs, as well as by those within the same job as the locking task. Any task referring to a locked record will wait at that point until the record is unlocked. A record can be explicitly unlocked by the locking task through execution of a REWRITE, DELETE, UNLOCK, or CLOSE statement. Records are unlocked automatically upon completion of the locking task. The EXCLUSIVE attribute applies to the data set and not the file. Consequently, record protection is provided even when all tasks refer to the data set through use of different files.

7. A WRITE statement adds a record to a data set, while a REWRITE statement replaces a record. Thus, a WRITE statement may be used with OUTPUT files, and DIRECT UPDATE files, but a REWRITE statement may be used with UPDATE files only. Moreover, for DIRECT files, a REWRITE statement uses the KEY option to identify the existing record to be replaced; a WRITE statement uses the KEYFROM option, which not only specifies where the record is to be written in the data set, but also specifies, except for REGIONAL(1), an identifying key to be recorded in the data set.

8. Records of a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file can be skipped over and ignored by use of the IGNORE option of a READ statement. The expression of the IGNORE option specifies the number of records to be skipped. A READ statement in which only the FILE option appears indicates that one record is to be skipped.

9. Teleprocessing support is provided by an extension of the basic record-oriented transmission facilities. TRANSIENT files are associated with queues of messages either incoming from or outgoing to remote terminals. Such files must be KEYED and BUFFERED, and the ENVIRONMENT attribute must be used to specify the message format. TRANSIENT files can be accessed by READ, WRITE, and LOCATE statements, which cannot have the EVENT option.

Examples of Declarations for Record Files

Following are examples of declarations of file constants including the ENVIRONMENT attribute:

```
DECLARE FILE#3 INPUT DIRECT
    ENVIRONMENT(V BLKSIZE(328)
    REGIONAL(3));
```

This declaration specifies only three file attributes: INPUT, DIRECT, and ENVIRONMENT. Other implied attributes are FILE (implied by any of the attributes) and RECORD and KEYED (implied by DIRECT). Scope is EXTERNAL, by default. The ENVIRONMENT attribute specifies that the data set is of the REGIONAL(3) organization and contains unblocked varying-length records with a maximum length of 328 bytes. Note that a maximum length record will contain only 320 bytes of data to be used by the program, because 8 bytes are required for control information in such V-format records. The KEY option must be specified in each READ statement that refers to this file.

```
DECLARE INVNTRY UPDATE BUFFERED
    ENVIRONMENT (F RECSIZE(100)
     INDEXED BUFFERS(4));
```

This declaration also specifies only three file attributes: UPDATE, BUFFERED, and ENVIRONMENT. Implied attributes are FILE, RECORD, and SEQUENTIAL (the last two attributes are implied by BUFFERED). Scope is EXTERNAL, by default. The data set is of INDEXED organization, and it contains fixed-length records of 100 bytes each. Four buffers are to be allocated for use in accessing the data set. Note that although the data set actually contains recorded keys, the KEYTO option cannot be specified in a READ statement, since the KEYED attribute has not been specified.

Note that for both of the above declarations, all necessary attributes are either stated or implied in the DECLARE statement. None of the attributes can be changed in an OPEN statement or in a DD statement. The second declaration might have been written:

```
DECLARE INVNTRY
    ENVIRONMENT(F RECSIZE(100)
      INDEXED);
```

With such a declaration, INVNTRY can be opened for different purposes. It could, for example, be opened as follows:

```
OPEN FILE (INVNTRY)
    UPDATE SEQUENTIAL BUFFERED;
```

With this OPEN statement, the file attributes would be the same as those specified (or implied) in the DECLARE statement in the second example above (the number of buffers would have to be stated in the associated DD statement). The file might be opened in this way, then closed, and then later opened with a different set of attributes, for example:

```
OPEN FILE (INVNTRY)
    INPUT SEQUENTIAL KEYED;
```

This OPEN statement allows records to be read with either the KEYTO or the KEY option. Because the file is SEQUENTIAL and the data set is INDEXED, the data set may be accessed in a purely sequential manner; or, by means of a READ statement with a KEY option, it may be accessed randomly. A KEY option in a READ statement with a file of this description causes a specified record to be obtained. Subsequent READ statements without a KEY option access records sequentially, beginning with the next record.

Chapter 13: Editing and String Handling

The data manipulations performed by the arithmetic, comparison, and bit-string operators are extended in PL/I by a variety of string-handling and editing features. These features are specified by data attributes, statement options, built-in functions, and pseudovariables.

The following discussions give general descriptions of each feature, along with illustrative examples.

Editing by Assignment

The most fundamental form of editing performed by the assignment statement involves converting the data type of the value on the right-hand side of the assignment symbol to conform to the attributes of the receiving variable. Because the assigned value is made to conform to the attributes of the receiving variable, the precision or length of the assigned value may be altered. Such alteration can involve the addition of digits or characters to and the deletion of digits or characters from the converted item. The rules for data conversion are discussed in chapter 4, "Expressions and Data Conversions", and in section F, "Data Conversion and Expression Evaluation".

ALTERING THE LENGTH OF STRING DATA

When a value is assigned to a string variable, it is converted, if necessary, to the same string type (character or bit) as the receiving string. If necessary, it is truncated or, for fixed-length receiving strings, extended on the right to conform to the declared length of the receiving string. For example, assume SUBJECT has the attributes CHARACTER (10), indicating a fixed-length character string of ten characters. Consider the following statement:

```
SUBJECT = 'TRANSFORMATIONS';
```

The length of the string on the right is fifteen characters; therefore, five characters will be truncated from the right end of the string when it is assigned to SUBJECT. This is equivalent to executing:

```
SUBJECT = 'TRANSFORMA';
```

If the assigned string is shorter than the length declared for the receiving string variable, the assigned string is extended on the right either with blanks, in the case of a character-string variable, or with zeros, in the case of a bit-string variable. Assume SUBJECT still has the attributes CHARACTER (10). Then the following two statements assign equivalent values to SUBJECT:

```
SUBJECT = 'PHYSICS';

SUBJECT = 'PHYSICSbbb';
```

The letter b indicates a blank character.

Let CODE be a bit-string variable with the attributes BIT(10). Then the following two statements assign equivalent values to CODE:

```
CODE = '110011'B;

CODE = '1100110000'B;
```

Note, however, that the following statements do not assign equivalent values to SUBJECT if it has the attributes CHARACTER (10):

```
SUBJECT = '110011'B;

SUBJECT = '1100110000'B;
```

When the first statement is executed, the bit-string constant on the right is first converted to a character string and is then extended on the right with blank characters rather than zero bits. This statement is equivalent to:

```
SUBJECT = '110011bbbb';
```

The second of the two statements requires only a conversion from bit-string to character-string type and is equivalent to:

```
SUBJECT = '1100110000';
```

A string value, however, is not extended with blank characters or zero bits when it is assigned to a string variable that has the VARYING attribute. Instead, the length specification of the receiving string variable is effectively adjusted to describe the length of the assigned string. Truncation will occur, though, if the length of the assigned string exceeds the

maximum length declared for the varying-length string variable.

OTHER FORMS OF ASSIGNMENT

In addition to the assignment statement, PL/I provides other ways of assigning values to variables. Among these are two methods that involve input and output statements: one in which actual input and output operations are performed, and one in which data movement is entirely internal.

Input and Output Operations

Although the assignment statement is concerned with the transmission of data between storage locations internal to a computer, input and output operations can also be treated as related forms of assignment in which transmission occurs between the internal and external storage facilities of the computer.

Record-oriented operations, however, do not cause any data conversion of items in a logical record when it is transmitted. Required editing of the record must be performed within internal storage either before the record is written or after it is read.

Stream-oriented operations, on the other hand, do provide a variety of editing functions that can be applied when data items are read or written. These editing functions are similar to those provided by the assignment statement, except that any data conversion always involves character type, conversion from character type on input, and conversion to character type on output.

STRING Option in GET and PUT Statements

The STRING option in GET and PUT statements allows the statements to be used to transmit data between internal storage locations rather than between the internal and external storage facilities. In both GET and PUT statements, the FILE option, specified by FILE (file-expr) , is replaced by the STRING option, as shown in the following formats:

```
GET STRING
    (character-string-expression)
    data specification;

PUT STRING
    (character-string-variable)
    data specification;
```

The GET statement specifies that data items to be assigned to variables in the data list are to be obtained from the specified character string. The PUT statement specifies that data items of the data list are to be assigned to the specified character-string variable. The "data-specification" is the same as described for input and output. In general, it takes one of the following forms:

```
DATA [(data-list)]

[LIST]  (data-list)

EDIT  {(data-list) (format-list)}...
```

Although the STRING option can be used with each of the three modes of stream-oriented transmission, it is most useful with edit-directed transmission, which considers the input stream to be a continuous string of characters. For list-directed and data-directed GET statements, individual items in the character string must be separated by commas or blanks; for data-directed GET statements, the string must also include the transmission-terminating semicolon, and each data item must appear in the form of an assignment statement. Edit-directed transmission provides editing facilities by means of the format list. Note that the COLUMN control format option may not be used with the STRING option.

The NAME condition is not raised for a GET DATA statement with the STRING option. Instead, the ERROR condition is raised for situations that would cause the NAME condition to be raised for a GET DATA statement with the FILE option.

The STRING option permits data gathering or scattering operations to be performed with a single statement, and it allows stream-oriented processing of character strings that are transmitted by record-oriented statements. Consider the following statement:

```
PUT STRING (RECORD) EDIT
   (NAME, PAY#, HOURS*RATE)
   (A(12), A(7), P'$999V.99');
```

This statement specifies that the character-string value of NAME is to be assigned to the first (leftmost) 12 character positions of the string named RECORD, and that the character-string value of PAY# is to be assigned to the next seven character positions of RECORD. The value of HOURS is then to be multiplied by the

value of RATE, and the product is to be edited into the next seven character positions, according to the picture specification.

Frequently, it is necessary to read records of different formats, each of which gives an indication of its format within the record by the value of a data item. The STRING option provides an easy way to handle such records; for example:

```
READ FILE (INPUTR) INTO (TEMP);
GET STRING (TEMP) EDIT (CODE) (F(1));
IF CODE ¬= 1 THEN GO TO OTHER_TYPE;
GET STRING (TEMP) EDIT (X,Y,Z)
   (X(1), 3 F(10,4));
```

The READ statement reads a record from the input file INPUTR. The first GET statement uses the STRING option to extract the code from the first byte of the record and to assign it to CODE. The code is tested to determine the format of the record. If the code is 1, the second GET statement then uses the STRING option to assign the items in the record to X,Y, and Z. Note that the second GET statement specifies that the first character in the string TEMP is to be ignored (the X(1) format item in the format list). Each GET statement with the STRING option always specifies that the scanning is to begin at the first character of the string. Thus, the character that is ignored in the second GET statement is the same character that is assigned to CODE by the first GET statement.

In a similar way, the PUT statement with a STRING option can be used to create a record within internal storage. In the following example, assume that the file OUTPRT is eventually to be printed.

```
PUT STRING (RECORD) EDIT
   (NAME, PAY#, HOURS*RATE)
   (X(1), A(12), X(10), A(7), X(10),
    P'$999V.99');

WRITE FILE (OUTPRT) FROM (RECORD);
```

The PUT statement specifies, by the X(1) spacing format item, that the first character assigned to the character-string variable is to be a single blank, the ANS carriage-control code that specifies a single space before printing. Following that, the values of the variables NAME and PAY# and of the expression HOURS*RATE are assigned. The format list specifies that ten blank characters are to be inserted between NAME and PAY# and between PAY# and the expression value. The WRITE statement specifies that record transmission is to be used to write the record into the file OUTPRT.

PICTURE SPECIFICATION

Picture specifications extend the editing facilities available in PL/I, and provide the user with greater control over his data formats. A picture specification consists of a sequence of character codes enclosed in quotation marks which is either part of the PICTURE attribute, or part of the P (picture) format-item:

```
DECLARE PRICE          PICTURE'$Z9V99';

PUT FILE(SYSPRINT)     EDIT
                       ('PART NUMBER', PART#)
                       (A(12), P'AAA99X');
```

Picture specifications are of two types:

- numeric character specificatons
- character-string picture specifications

A numeric character specification in a PICTURE attribute indicates that the data item represents a numeric quantity but that it is to be stored as a character-string, and indicates how the numeric value is to be represented in the string. A numeric character specification in a P format item indicates how a numeric value is, or is to be, represented as a character-string on the external medium.

A character-string picture specification is an alternative way of describing a fixed-length character string, with the additional facility of indicating that any position in the string may only contain characters from certain subsets of the complete set of characters available on the operating system.

The concepts of the two types of picture specifications are described separately below, and a detailed description of each picture character, together with examples of its use, appears in section D, "Picture Specification Characters". It is sufficient here to note that the presence of an A or X picture character defines a picture specification as a character-string picture specification; otherwise, it is a numeric character specification.

Numeric Character Specifications

A numeric character specification specifies that the associated data item has a numeric value but is to be stored (or is to be represented in the external medium) as a character string. It also specifies the

form the character string is to take, and exactly how the numeric value is represented in the string. For example:

```
DCL PRICE PICTURE'$Z9V99';
```

This specifies that PRICE is to be represented by a character-string of length 5. The first character is always $, the second will be a blank or non- zero digit, and the third, fourth and fifth characters will be digits. The numeric value is indicated by the four characters which can represent digits, a decimal point being assumed in the position indicated by the V; hence it is regarded as FIXED DECIMAL (4,2), and is always positive. 13.25 is represented as '$1325' and .95 as '$b095'.

The numeric character specification has two major uses:

- For data items which will be concerned with input/output operations (although they may be used anywhere in a program where numeric data can occur). However, most numeric operations on pictured data are considerably less efficient than the same operations on coded numeric data.

- The second use stems from the fact that a pictured data item effectively has two values. When the item is used in a numeric context, the numeric value is obtained from or stored into the character-string by the conversion process defined by the picture specification; when the item is used as source data in a context where a character-string expression is required, the actual character-string which represents the value is used. For example:

```
DCL COUNT PICTURE'999' INITIAL(0),
    STRING CHAR (3);
COUNT = COUNT +1;
STRING = COUNT;
```

The initial representation of COUNT is '000'. In the first assignment statement, this is converted to FIXED DECIMAL (3,0), the addition is performed, and the result is converted back to the pictured form '001'. In the second assignment statement the value of string is set to '001'.

Note particularly that the character context includes defining. A numeric character data-item may be defined on a character-string and vice versa.

Picture Character '9' in Numeric Character Specifications

The picture character '9' is the simplest form of numeric character specification. A string of n '9' picture characters specifies that the item is to be represented by a fixed-length character-string of length n, each character of which is a digit (zero through nine). The numeric value is the value of the digits as an unsigned decimal number (i.e., FIXED DECIMAL (n,0)). For example:

```
DCL DIGIT PICTURE'9'
    COUNT PICTURE'999',
    XYZ PICTURE '(10)9';
```

The last example shows an alternative way of writing the picture specification 9 ten times.

Example of use:

```
DCL 1 CARD_IMAGE,
      2 DATA CHAR(72),
      2 IDENTIFICATION CHAR(3),
      2 SEQUENCE PIC'99999';
  .
  .
  .
SEQUENCE = SEQUENCE + 1;
WRITE FILE(OUTPUT) FROM(CARD_IMAGE);
```

(Note that the definition of '9' in a character-string picture is different in that the corresponding character can be blank or a digit.)

Picture Characters Z *

It is often preferable to replace leading zeros in numbers by blanks. In pictures this is accomplished by using the Z picture specification character. A picture specification containing only Zs and 9s has one or more Zs optionally followed by one or more 9s. The representation of numeric data is as for the '9' picture specification except that if the digit to be held would otherwise be zero and if all digit positions to the left would also be zero, then the character-string will contain a blank in this position. For example:

```
DCL PAGE_NUMBER PICTURE'ZZ9';
```

The value 197 is held as '197', 69 as 'b69', 5 as 'bb5' and zero as 'bb0'. With a picture specification of all Zs, the value zero is held as an all-blank string.

The asterisk picture specification character has the same effect as the Z character except that an * is held in the string instead of a blank. This can be used, for example, when printing checks, when it is desired not to leave blank spaces within fields. For example:

```
DCL CREDIT PICTURE '$**9.99';
```

(The $ and (.) characters are described below.) A value of 95 is held as '$**0.95'; a value of 12350 is held as '$123.50'.

Picture Character V

The V picture specification character indicates the position of an assumed decimal point within the character-string. For example:

```
DCL VALUE PICTURE 'Z9V999';
```

The string '12345' represents the numeric value 12.345. Note that the V character in the picture specification does not specify a character position in the character-string representation. In particular, on assignment to the data item a decimal point is not included in the character string.

Insertion Picture Characters B . , /

A decimal point picture character(.) can appear in a numeric picture specification. It merely indicates that a point is to be included in the character representation of the value. Therefore, the decimal point is a part of its character-string value. The decimal point picture character does not cause decimal point alignment during assignment; it is not a part of the variable's arithmetic value. Only the character V causes alignment of decimal points. For example:

```
DECLARE SUM PICTURE '999V.99';
```

SUM is a numeric character variable representing numbers of five digits with a decimal point assumed between the third and fourth digits. The actual point specified by the decimal point insertion character is not a part of the arithmetic value; it is, however, part of its character-string value. (The decimal point picture character can appear on either side of the character V. In certain contexts the two forms have different meanings but V. is recommended in most cases. See section D, "Picture Specification Characters.") The following two statements assign the same character string to SUM:

```
SUM = 123;

SUM = 123.00;
```

In the first statement, two zero digits are added to the right of the digits 123.

Note the effect of the following declaration.

```
DECLARE RATE PICTURE '9V99.99';
```

Let RATE be used as follows:

```
RATE = 7.62;
```

When this statement is executed, decimal point alignment occurs on the character V and not the decimal point picture character that appears in the picture specification for RATE. If RATE were printed, it would appear as '762.00', but its arithmetic value would be 7.6200.

Unlike the character V, which can appear only once in a picture specification, the decimal point picture character can appear more than once; this allows digit groups within the numeric character data item to be separated by points, as is common in Dewey decimal notation and in the numeric notations of some European countries.

Because a decimal point picture character causes a period character to be inserted into the character-string value of a numeric character data item, it is called an insertion character. PL/I provides three other insertion characters: comma (,), slash(/), and blank(B). 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.

If a point, comma or slash picture character appears with a string of Z or * zero suppression characters, then if all previous digits in the string are suppressed, the insertion character is suppressed to blank or '*'.

The B character differs from the other three in that a blank is always inserted in

the corresponding position of the character string, even within a string of * zero suppression characters.

Picture Character $

The $ picture character controls the appearance of the currency symbol $ in specified positions of numeric character data items. For example a dollar sign can be appended to the left of a numeric character item, as indicated in the following statements:

```
DECLARE PRICE PICTURE '$99V.99';

PRICE = 12.45;
```

The character-string value of PRICE is equivalent to the character-string constant '$12.45'. Its arithmetic value, however, is 12.45.

Sign Specification in Numeric Character Specifications

There are several ways in which signed information may be held in a numeric character data item. The simplest of these is the S character specification. This specifies a character in the character-string representation which contains '+' if the value is positive or zero, and '-' if the value is negative. It must occur either to the right or to the left of all digit positions. For example:

```
DCL ROOT PICTURE 'S999';
```

50 is held as '+050', zero as '+000' and -243 as '-243'. Similarly the '+' picture character specifies a corresponding character position containing '+' for positive or zero, and blank for negative values; the '-' picture character specifies a corresponding character position containing blank for positive or zero, and '-' for negative values.

Overpunched Sign Specification Characters, T I R

An alternative way of representing signed values, which does not require an additional character in the string, is by an overpunched sign specification. This representation has arisen from the custom of indicating signs in numeric data held on punched cards, by superimposing a 12-punch (to represent +) or an 11-punch (to represent -) on top of a column containing a digit (usually the last one in a field). The resulting card-code is, in most cases, the same as that for an alphabetic character (e.g., 12-punch superimposed on 1 through 9 gives A through I, 11-punch superimposed on 1 through 9 gives J through R). The 12-0 and 11-0 combinations are not characters in the PL/I set but are within the set of characters accepted by the operating system.

The T picture specification character specifies a character in the character-string representation which will hold a digit and sign, in the representation described above, i.e., 12-punch superimposed on 0, or on 1 through 9 (A through I) for positive, 11-punch superimposed on 0, or on 1 through 9 (J through R) for negative. It can appear anywhere a '9' picture specification character could have occurred. For example:

```
DCL CREDIT PICTURE 'ZZV9T';
```

The character-string representation of CREDIT is 4 characters. +21.05 is held as '210E'. -0.07 is held as 'bb0P'.

The I picture specification character specifies a character position which holds the representation of a digit overpunched with a 12-punch if the value is positive or zero, but just a digit if the value is negative.

The R picture specification character specifies a character position which hold the representation of a digit overpunched with an 11-punch if the value is negative, but just a digit if the value is positive. For example:

```
GET EDIT (X) (P'R99');
```

will set X to (+) 132 on finding '132' in the next 3 positions of the input stream, but -132 on finding 'J32'.

Other Numeric Character Facilities

Further details on the use of the above picture specification characters, together with details of picture specification characters for floating signs and currency symbols, and floating point values, appear in section D, "Picture Specification Characters".

The full list of numeric character specification characters is:

9,V,Z,*,Y,(.),(,),/,B,S,+,-,$,CR,DB,T,I,R, K,E,F of which all except K,V,F specify the occurrence of a character in the character-string representation.

Character-string Picture Specifications

A character-string picture specification is an alternative way of describing a fixed-length character string, with the additional facility of indicating that any position in the string may only contain characters from certain subsets of the complete set of available characters.

A character-string picture specification is recognized by the occurrence of an A or X picture specification character. The only valid characters in a character-string picture specification are X, A, and 9. Each of these specifies the presence of one character position in the character-string which can contain the following:

X any character recognized by the particular implementation (i.e., all 256 possible bit combinations represented by the 8-bit byte).

A any alphabetic character, #, @, $, or blank.

9 any digit, or blank. Note the difference from the 9 picture specification character in numeric character specifications.

When a character-string value is assigned, or transferred, to a pictured character-string data item, the particular character in each position is checked for validity, as specified by the corresponding picture specification character, and the CONVERSION condition is raised for an invalid character. For example:

```
DECLARE PART# PICTURE 'AAA99X';
```

The following values are valid for assignment to PART#.

```
'ABC12M'
'bbb09/'
'XYZb13'
```

The following values are not (the invalid characters are underscored);

```
'AB123M'
'ABC1/2'
'Mb#A5;'
```

Bit-string Handling

The following examples illustrate some of the facilities of PL/I that can be used in bit-string manipulations.

```
DECLARE 1 PERSONNEL_RECORD,
          2 NAME,
            3 LAST CHARACTER(15),
            3 FIRST CHARACTER(10),
            3 MIDDLE CHARACTER(1),
          2 CODE_STRING,
            3 MALE BIT(1),
            3 SECRETARIAL BIT(1),
            3 AGE,
              4 (UNDER_20,
                 TWENTY_TO_30,
                 OVER_30) BIT(1),
            3 HEIGHT,
              4 (OVER_6,
                 FIVE_SIX_TO_6,
                 UNDER_5_6) BIT(1),
            3 WEIGHT,
              4 (OVER_180,
                 ONE_TEN_TO_180,
                 UNDER_110) BIT(1),
            3 EYES,
              4 (BLUE,
                 BROWN,
                 HAZEL,
                 GREY,
                 OTHER) BIT(1),
            3 HAIR,
              4 (BROWN,
                 BLACK,
                 BLOND,
                 RED,
                 GREY,
                 BALD) BIT(1),
            3 EDUCATION,
              4 (COLLEGE,
                 HIGH_SCHOOL,
                 GRAMMAR_SCHOOL) BIT(1);
```

This structure contains NAME, a minor structure of character-strings, and CODE_STRING, a minor structure of bit-strings. By default, the elements of PERSONNEL-RECORD have the UNALIGNED attribute. Consequently, CODE_STRING is mapped with eight elements per byte, that is, in the same way as a bit-string of length 25.

Each of the first two bits of the string represents only two alternatives: MALE or ¬MALE and SECRETARIAL or ¬SECRETARIAL. The other categories (at level 3) list several alternatives each. (Note that the level number 4 and the attributes BIT(1) are factored for each category.)

The following portion of a program might be used with PERSONNEL_RECORD:

```
INREC: READ FILE(PERSON)
           INTO (PERSONNEL_RECORD);

       IF (¬MALE & SECRETARIAL
           & UNDER_20
           & UNDER_5_6
           & UNDER_110
           & BLUE
           & (HAIR.BROWN|BLOND)
           & HIGH_SCHOOL)
           | (MALE & ¬SECRETARIAL
           & OVER_30
           & OVER_6
           & OVER_180
           & EYES.GREY
           & BALD
           & COLLEGE)

           THEN PUT LIST (NAME);

           GO TO INREC;
```

Another way to program the same information retrieval operation is as follows:

```
     DECLARE PERS_STRING BIT(25) DEFINED
           CODE_STRING;

     IF PERS_STRING
           = '011000010011000010000010'B
           THEN GO TO OUTP;

     IF PERS_STRING
           = '011000010011000000100010'B
           THEN GO TO OUTP;

     IF PERS_STRING
           = '100011001000001000000110'B
           THEN GO TO OUTP;

     GO TO INREC;

OUTP:  PUT LIST (NAME);

     GO TO INREC;
```

In this example, the bit string PERS_STRING is defined on the minor structure CODE_STRING. Bit-string constants are constructed to represent the values of the information being sought. The bit string then is compared, in turn, with each of the bit-string constants. Note that the first and second constants are identical except that the first tests for brown hair and the second tests for blond hair. These two variations are specified in the first example by (HAIR.BROWN|BLOND).

Note that the second method of testing PERSONNEL_RECORD could not be used if the structure were ALIGNED (the base identifier for overlay defining must be UNALIGNED). The first method, if it were used, would be more efficient with an ALIGNED structure.

The second method has the disadvantage that the 25 bits in PERS_STRING have to match the bit-string constant exactly. This means that in an abnormal situation, such as when a man is described as having grey hair and being bald, he would be selected by the first method but not by the second. The second method also has the disadvantage that if a further item of data, such as another colour of hair, were to be added, the bit string constants would have to be changed in every comparison, whereas the first method requires that only the comparisons in which the new item is used need to be changed.

If the second method were used, an improvement could be made by using combinations of bits to denote each characteristic, rather than single bits. For instance, the minor structure HAIR could be replaced by a bit string length 3 at the same level in the structure and, '000'B could represent bald, '001'B grey-haired, '010'B red-haired, etc. This would reduce length required for PERS_STRING from 25 to 16 bits, and would obviate the possibility of conflicts such as that between bald and grey-haired.

The tests might also be made with a series of IF statements, either nested or unnested, in which each bit would be tested with a single IF statement.

String Built-in Functions

PL/I provides a number of built-in functions, some of which also can be used as pseudovariables, to add power to the string-handling facilities of the language. Following are brief descriptions of these functions (more detailed descriptions appear in section G, "Built-in Functions and Pseudovariables").

The BIT built-in function specifies that a data item is to be converted to a bit string. The built-in function allows a programmer to specify the length of the converted string, overriding the length that would result from the standard rules of data conversion.

The CHAR built-in function is exactly the same as the BIT built-in function, except that the conversion is to a character string whose length may be specified by the programmer.

The SUBSTR built-in function, which can also serve as a pseudovariable in a

receiving field, allows a specific substring to be extracted from (or assigned to in the case of a pseudovariable) within a specified string value.

The INDEX built-in function allows a string, either a character string or a bit string, to be searched for the first occurrence of a specified substring, which can be a single character or bit. The value returned is the location of the first character or bit of the substring, relative to the beginning of the string. The value is expressed as a binary integer. If the substring does not occur in the specified string, the value returned is zero.

The LENGTH built-in function gives the current length of a character string or bit string. It is particularly useful with strings that have the VARYING attribute.

The HIGH built-in function provides a string of a specified length that consists of repeated occurrences of the highest character in the collating sequence. For these implementations, the character is hexadecimal FF.

The LOW built-in function provides a string of a specified length that consists of repeated occurrences of the lowest character in the collating sequence. For these implementations, the character is hexadecimal 00.

The REPEAT built-in function permits a string to be formed from repeated occurrences of a specified substring. It is used to create string patterns.

The STRING built-in function, which can also be used as a pseudovariable, concatenates all the elements in an aggregate variable into a single string element.

The BOOL built-in function allows any of the 16 different Boolean operations to be applied to two specified bit strings.

The UNSPEC built-in function, which can also be used as a pseudovariable, specifies that the internal coded representation of a value is to be regarded as a bit string with no conversion. For example:

```
X = ARRAY(UNSPEC('A'));
```

In this statement the internal representation of the character 'A' is for these implementations a string eight bits in length. This bit string is converted to a fixed binary arithmetic value, and used as a subscript for the array. (The decimal value of this particular subscript is 193).

The TRANSLATE built-in function changes specified character elements in a string for specified replacement character elements. The 'replacement' element is inserted into the 'position' in the string occupied by the element to be replaced.

This built-in function enables the programmer to use a translation facility whereby all the characters in a given string are translated according to a translation table contained in two other strings. One of these strings serves as a key to the replacement characters held in the other string. For example:

```
DECLARE (W,X,Y,Z) CHAR (3);

        X='ABC';
        Y='TAR';
        Z='CAB';

W = TRANSLATE (X,Y,Z);

/* W = 'ART' */
```

The VERIFY built-in function compares two strings to check whether the bits or characters in one string occur anywhere in the other string. If all the characters or bits in one string are detected in the second string, a value of '0' is returned. If a character or bit does not occur in the second string, a value representing the position of this character or bit in the first string is returned.

The first string is verified from left to right. A position value for the first unmatched bit or character only is returned.

Chapter 14: Exceptional Condition Handling and Program Checkout

When a PL/I program is executed, a large number of exceptional conditions are monitored by the system and their occurrences are automatically detected whenever they arise. These exceptional conditions may be errors, such as overflow or an input/output transmission error, or they may be conditions that are expected but infrequent, such as the end of a file or the end of a page when output is being printed. When checking out a program, a programmer can also get a selective flow trace and dumps by specifying that the occurrence of any one of a list of identifiers be treated as an exceptional condition.

Each of the conditions for which a test may be made has been given a name, and these names are used by the programmer to control the handling of exceptional conditions. The list of condition names is part of the PL/I language. For keyword names and descriptions of each of the conditions, see section H, "ON-Conditions."

The situations in which these conditions occur is the same for both the optimizing and the checkout compilers. The enabling of these conditions, and the specifying of the action required when a condition is raised, are described in this chapter.

With the checkout compiler, the facilities for making values available to the programmer during execution are greatly extended. These facilities are described in chapter 15, "Execution-time Facilities of the Checkout Compiler".

Enabled Conditions and Established Action

A condition that is being monitored, and the occurrence of which will cause an interrupt, is said to be enabled. Any action specified to take place when an occurrence of the condition causes an interrupt, is said to be established.

Most conditions are checked for automatically, and when they occur, the system will take control and perform some standard action specified for the condition. Such conditions are enabled by default, and the standard system action is established for them.

The most common system action is to raise the ERROR condition. This provides a common condition that may be used to check for a number of different types of errors, rather than checking each error type individually. Standard system action for the ERROR condition depends on the processing mode:

Batch processing (optimizing and checkout compilers): If the condition is raised in the major task, the FINISH condition is raised and the program is subsequently terminated. If it is raised in a subtask, that task is terminated.

Conversational processing (checkout compiler only): Control is passed to the terminal.

The programmer may specify whether or not some conditions are to be enabled, that is, are to be checked for so that they will cause an interrupt when they arise. If a condition is disabled, an occurrence of the condition will not cause an interrupt. Under the checkout compiler, the SIZE, STRINGRANGE, and SUBSCRIPTRANGE conditions are continuously monitored, whether enabled or not.

All input/output conditions and the ERROR, FINISH, and AREA conditions are always enabled and cannot be disabled. All of the computational conditions and the program checkout conditions may be enabled or disabled. The program checkout conditions and the SIZE condition must be explicitly enabled if they are to cause an interrupt; all other conditions are enabled by default and must be explicitly disabled if they are not to cause an interrupt when they occur.

Condition Prefixes

Enabling and disabling can be specified for the eligible conditions by a condition prefix. A condition prefix is a list of one or more condition names, enclosed in parentheses and separated by commas, and connected to a statement (or a statement label) by a colon. The prefix always precedes the statement and any statement labels. For example:

```
(SIZE):  L1:  X=(I**N)/(M+L);
```

A condition name in a prefix list indicates that the corresponding condition is enabled within the scope of the prefix. The name of a condition used in a prefix can be preceded by the word NO, without a separating blank or connector, to indicate that the corresponding condition is disabled. For example:

```
(NOCONVERSION):  Y=A||B;
```

Scope of the Condition Prefix

The scope of the prefix, that is, the part of the program throughout which it applies, is usually the statement to which the prefix is attached. The prefix does not apply to any functions or subroutines that may be invoked in the execution of the statement.

A condition prefix to an IF statement applies only to the evaluation of the expression following the IF; it does not apply to the statements in the THEN or ELSE clauses, although these may themselves have prefixes. Similarly, a prefix to the ON statement has no effect on the statements in the on-unit. A condition prefix to a DO statement applies only to the evaluation of any expressions in the DO statement itself and not to any other statement in the DO-group.

Condition prefixes to the PROCEDURE statement and the BEGIN statement are special (though commonly used) cases. A condition prefix attached to a PROCEDURE or BEGIN statement applies to all the statements up to and including the corresponding END statement. This includes other PROCEDURE or BEGIN statements nested within that block. It does not apply to any procedures lying outside that block.

The enabling or disabling of a condition may be redefined within a block by attaching a prefix to statements within the block, including PROCEDURE and BEGIN statements (thus redefining the enabling or disabling of the condition within nested blocks). Such a redefinition applies only to the execution of the statement to which the prefix is attached. In the case of a nested PROCEDURE or BEGIN statement, it applies only to the block the statement defines, as well as any blocks contained within that block. When control passes out of the scope of the redefining prefix, the redefinition no longer applies. A condition prefix can be attached to any statement except a DECLARE, DEFAULT, or ENTRY statement.

ON Statement

A system action exists for every condition, and if an interrupt occurs, the system action will be performed unless the programmer has specified an alternate action in an ON statement for that condition. The purpose of the ON statement is to establish the action to be taken when an interrupt results from an exceptional condition that has been enabled, either by default or by a condition prefix.

Note: The action specified in an ON statement will not be executed during any portion of a program throughout which the condition has been disabled.

The form of the ON statement is:

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

(See section J, "Statements" for a full description).

The keyword SYSTEM followed by a semicolon specifies standard system action whenever an interrupt occurs. It re-establishes system action for a condition for which some other action has been established. The on-unit is used by the programmer to specify an alternative action to be taken whenever an interrupt occurs.

The SNAP option specifies that, when an interrupt occurs, a list of all blocks in the chain of invocation leading to the current task is written on the standard system file SYSPRINT. If SNAP is specified, the action of the SNAP option precedes the action of the on-unit. If SNAP SYSTEM is specified, the system action message is followed immediately by a list of active blocks.

The on-unit must be either a single,unlabeled, simple statement or an unlabeled begin block. The single statement cannot be a RETURN, FORMAT, DECLARE, or DEFAULT statement. It cannot be either of the two compound statements, IF and ON, or a DO-group. (PROCEDURE, BEGIN, END, and DO statements can never appear as single statements.) The begin block, if it appears, can contain any statement (except that, as with any BEGIN block, a RETURN statement can appear only within a procedure nested in the begin block).

The single statement on-unit, or the begin block on-unit, is executed as though it were a procedure (without parameters) that was called at the point in the program at which the interrupt occurred. If the

on-unit is a single statement it behaves as though it were a single-statement procedure; when execution of the unit is complete, control generally returns to the block from which the on-unit was entered. Just as with a procedure, control may be transferred out of an on-unit by a GO TO statement; in this case, control is transferred to the point specified in the GO TO, and a normal return does not occur.

Note: The specific point to which control returns from an on-unit varies for different conditions. In some cases, it returns to the point that immediately follows the action in which the condition arose, or the statement following the one in which the condition was raised. In other cases, control returns to the actual point of interrupt, and the action is reattempted. An example of the latter case is the return from the on-unit of an ON CONVERSION statement. When an interrupt occurs as the result of a conversion error, control returns from the on-unit to reattempt conversion of the character that caused the error (on the assumption that the invalid character has been changed during execution of the on-unit). If the invalid character is not changed, the ERROR condition is raised.

Null On-unit

A special case of an on-unit is the null statement. The effect of this is the same as a normal return from a begin-block on-unit, except that with the CONVERSION and AREA conditions, there is no retry.

Use of the null on-unit is not the same as disabling, for two reasons: first, a null on-unit may be specified for any condition, but not all conditions can be disabled; and, second, disabling a condition, if possible, may save time by avoiding any checking for this condition. If a null on-unit is specified, the system must still check for occurrence of the condition; the action then taken is the action that would be taken on normal return from an on-unit.

Note: A null on-unit for the CONVERSION condition will not cause a permanent loop if a conversion error occurs, because no conversion is re-attempted unless the invalid character is changed in the on-unit. If it is not changed, the ERROR condition is raised.

Scope of the ON Statement

The execution of an ON statement associates an action specification with the named condition. Once this association is established, it remains until it is overridden or until termination of the block in which the ON statement is executed.

An established interrupt action passes from a block to any block it activates, and the action remains in force for all subsequently activated blocks unless it is overridden by the execution of another ON statement for the same condition. If it is overridden, the new action remains in force only until that block is terminated or until a REVERT statement is executed for the condition. When control returns to the activating block, all established interrupt actions that existed at that point are re-established. This makes it impossible for a subroutine to alter the interrupt action established for the block that invoked the subroutine.

If more than one ON statement for the same condition appears in the same block, each subsequently executed ON statement permanently overrides the previously established condition. No re-establishment is possible, except through execution of another ON statement with an identical action specification (or re-execution, through some transfer of control, of an overridden ON statement).

Dynamically Descendent On-units

It is possible to raise a condition during execution of an on-unit and enter a further on-unit. An on-unit entered due to a condition either raised or signalled in another on-unit is a dynamically-descendant on-unit. A normal return from a dynamically-descendant on-unit reestablishes the environment of the on-unit in which the condition was raised.

On-units for File Parameters and File Variables

File constants or file variables used as arguments and parameters can be specified in input or output condition on-units. The following examples illustrate the rules and uses of this facility:

1. On-units for a particular condition in separate blocks can specify different file identifiers for the same file.

For example:

```
 E:  PROCEDURE;
   DECLARE F1 FILE;
   ON ENDFILE (F1) GOTO L1;
   CALL E1 (F1);
         .
         .
         .
E1:  PROCEDURE (F2);
   DECLARE F2 FILE;
   ON ENDFILE (F2) GO TO L2;
   READ FILE (F1);
   READ FILE (F2);
   END E1;
```

An end-of-file encountered for F1 in E1 causes the on-unit for F2 in E1 to be entered. If the on-unit in E1 were not specified, an end-of-file condition encountered for either F1 or F2 would cause entry to the on-unit for F1 in E.

2. On-units for a particular input or output condition in the same block can specify different file identifiers for the same file. The presence of a second on-unit overrides the first.

   For example:

```
E:  PROCEDURE;
  DECLARE F1 FILE;
  CALL E1 (F1);
        .
        .
        .

E1:  PROCEDURE (F2);
   DECLARE F2 FILE;
   ON ENDFILE (F1) GOTO L1;
   READ FILE (F1) INTO (X1);
         .
         .
         .
   ON ENDFILE (F2) GOTO L2;
   READ FILE (F2) INTO (X2);
         .
         .
         .
   READ FILE (F1) INTO (X3);
         .
         .
         .
   END E;
```

   An end-of-file condition raised by execution of the second READ FILE (F1); statement causes the on-unit for F2 to be entered.

3. If a REVERT statement for a particular condition that specifies a file parameter is executed, any on-unit previously established for the argument corresponding to the file parameter is entered.

   For example:

```
   E:  PROCEDURE;
      DCL F1 FILE;
      ON ENDFILE (F1);
      CALL E1 (F1);
            .
            .
            .
  E1:  PROCEDURE (F2);
      DECLARE F2 FILE;
      ON ENDFILE (F2) GOTO L;
            .
            .
            .
      REVERT ENDFILE (F2);
      /*NULL ON-UNIT IN E ASSOCIATED
        WITH ENDFILE INTERRUPTS FOR F2
        */
      READ FILE (F2) INTO (X1);
            .
            .
            .
      END E;
```

   An end-of-file condition encountered in the execution of the READ statement for F2 does not cause the on-unit for F2 and E1 to be entered. Because of REVERT statement the on-unit for F1 in the containing procedure is entered.

Whenever a file variable is used, the effect is the same as if the current file-constant value of the variable had been used. Thus having an ON statement which specifies a file variable refers to the file constant that is the current value of the variable when the on-unit is established.

For example:

```
DECLARE FV FILE VARIABLE,
        FC1 FILE,
        FC2 FILE;
FV = FC1;
ON ENDFILE(FV) GO TO FIN;
   .
   .
   .
FV = FC2;
READ FILE(FC1) INTO (X1);
READ FILE(FV) INTO (X2);
```

An end-of-file condition raised during the first READ statement will cause the on-unit to be entered, since the on-unit refers to file FC1. If the condition is raised in the second READ statement, however, the on-unit is not entered, since this READ refers to file FC2.

If an ON statement specifying a file variable is executed more than once, and the variable has a different value each time, then a different on-unit will be

established at each execution. For example.

```
DECLARE FV FILE VARIABLE,
        FC1 FILE,
        FC2 FILE;
   .
   .
   .
DO FV=FC1,FC2;
   ON ENDFILE(FV) GO TO FIN;
END;
```

REVERT Statement

The REVERT statement is used to cancel the action specification of all the ON statements for the named condition that have been executed in the same block in which the REVERT statement is executed. It then re-establishes the action that was in force at the time of activation of that block. This statement has the form:

```
REVERT condition-name;
```

A REVERT statement that is executed in a block in which no on-unit has been established for the named condition is treated as a null statement.

SIGNAL Statement

The programmer may simulate the occurrence of an ON condition by means of the SIGNAL statement. An interrupt will occur unless the named condition is disabled. This statement has the form:

```
SIGNAL condition-name;
```

The SIGNAL statement causes execution of the interrupt action currently established for the specified condition. The principal use of this statement is in program checking, to test the action of an on-unit, and to determine that the correct action is associated with the condition.

If the signaled condition is not enabled, the SIGNAL statement is treated as a null statement.

CONDITION Condition

The ON-condition of the form:

```
CONDITION (identifier)
```

allows a programmer to establish an on-unit that will be executed whenever a SIGNAL statement is executed specifying CONDITION and that identifier.

As a debugging aid, this condition can be used to establish an on-unit whose execution results in printing information that shows the current status of the program. The advantage of using this technique is that the statements of the on-unit need be written only once. They can be executed from any point in the program through placement of a SIGNAL statement. Following is an example of how the CONDITION condition might be included in a program:

```
ON CONDITION (TEST) BEGIN;
                      .
                      .
                      .
                    END;
```

Execution of the begin block would be caused wherever the following statement appears:

```
SIGNAL CONDITION (TEST);
```

The identifier can be declared contextually (as in the example given above) or explicitly. An explicit declaration of a condition name could be as follows:

```
DCL CNAME CONDITION;
ON CONDITION(CNAME) BEGIN;
                      .
                      .
                      .
                    END;
```

The CONDITION condition is always enabled, but it can be raised only by the SIGNAL statement.

CHECK Condition

The CHECK condition is an important tool provided in PL/I for program testing. It is raised during execution of the program whenever the value of a designated variable is modified, or whenever control is transferred to a statement prefixed by a designated label or entry constant. Variables, label constants, and entry constants for which the CHECK condition is to be raised are designated explicitly in an optional name list given with the CHECK prefix that enables the CHECK condition. If the CHECK prefix is given without the name list, all variables, label constants, and entry constants that are within the scope of the CHECK prefix can cause the CHECK condition to be raised. Variables

that can raise the CHECK condition include array and structure variables, label variables, entry variables, event variables, area variables, file variables, task variables, based and defined variables, and locator variables. Subscripted and locator-qualified names are not allowed but qualified names (i.e., members of structures) can be used. iSUB-defined variables are not allowed.

The interrupt occurs immediately after assignment to the variable being checked. An interrupt will take place, for instance, after the assignment of each element of a checked array. Exceptions are as follows.

1. If arguments specified in a CALL statement are being passed directly (as opposed to being passed by means of dummy arguments), then CHECK for these names is raised on return from the subroutine.

2. With label and entry constants, the interrupt occurs immediately before the execution of the statement or the invocation of the entry name.

The system action for problem variables is to print the identifier causing the interrupt and its new value in the form of data-directed output. For program control variables, the information provided is:

Checkout compiler: As for PUT DATA

Optimizing compiler: Name of the identifier

If the CHECK condition is raised by a SIGNAL CHECK, the standard system action is to print the identifiers (and their values, where applicable) given in the name list of the CHECK prefix. If the CHECK prefix does not have a name list, the standard action is to print all the identifiers (and their values, where applicable), that are within the scope of the CHECK prefix.

Thus, by preceding a block with a CHECK prefix, as shown in the example in figure 14.1, the programmer can obtain selective dumps in a readable format by specifying variables; he can obtain a flow trace by specifying labels and entry names.

The CHECK condition may also be specified in an ON statement, if other than system action is required. This gives the user all the facilities of PL/I, including the simplicity of data-directed output for controlling and editing his debugging information.

SIZE condition

The SIZE condition is not enabled unless it appears in a condition prefix. It is raised if high-order significant digits are lost from an arithmetic value during assignment to a variable or compiler-created intermediate storage location, or in an input/output operation. An error message is printed, and the ERROR condition is raised; in the absence of an appropriate on-unit, this leads to termination of the task. The checkout compiler will detect a SIZE error and take standard system action whether or not the condition is enabled, although a SIZE on-unit can be entered only when the condition has been enabled.

SUBSCRIPTRANGE Condition

Another ON condition that is used principally for program checkout, but that may also be used in production, is SUBSCRIPTRANGE. For the optimizing compiler, the condition needs to be enabled by a condition prefix. The checkout compiler will detect a SUBSCRIPTRANGE error and take standard system action, whether or not the condition is enabled, although a SUBSCRIPTRANGE on-unit can be entered only when the condition has been enabled.

Since this checking involves a substantial overhead in both storage space and execution time, it usually is used only in program testing - it is removed for production programs, SUBSCRIPTRANGE being a normally-disabled condition.

STRINGRANGE Condition

The STRINGRANGE condition is not enabled unless it appears in a condition prefix. It is raised by an invalid reference to the SUBSTR built-in function and pseudovariable, the arguments to which must lie within certain bounds (see "SUBSTR String Built-in Function" in section G, "Built-in Functions and Pseudovariables"). It allows execution to continue with a SUBSTR reference that has been revised either automatically (that is, by standard system action) or by the programmer using an on-unit. The checkout compiler will detect a STRINGRANGE error and take standard system action whether or not the condition has been enabled, although a STRINGRANGE on-unit can be entered only when the condition has been enabled.

Condition Built-in Functions and Condition Codes

When an on-unit is invoked, it is as if it were a procedure without arguments. It is therefore impossible to pass to the on-unit any information about the interrupt (other than the name of the condition). To assist the programmer in making use of on-units, some special functions are provided that may be used to inquire about the cause of an interrupt and possibly to attempt to correct the error.

These condition built-in functions can be used only in on-units or in blocks invoked by on-units. They are listed in section G, "Built-In functions and Pseudovariables".

The condition built-in functions provide information such as the name of the procedure in which the interrupt occurred, the character or character string that caused a conversion interrupt, the value of the key used in the last record transmitted, and so on. Some can be used as pseudovariables for error correction.

The ONCODE function provides a binary integer whose value depends on the cause of the last interrupt. This function, used in conjunction with the ERROR condition, allows the programmer to deal with errors that may be detected by the implementation, but that are not represented by condition names in the language. It can also be used to distinguish between the various circumstances under which a particular condition (for instance the KEY condition) can be raised.

Example of Use of On-conditions

The routine shown in figure 14.1 illustrates the use of the ON statement, the SIGNAL and REVERT statements, and condition prefixes. The routine reads batches of cards containing test readings. Each batch has a header card with a sample number, called SNO, of zero and a trailer card with SNO equal to 9999. If a conversion error is found, one retry is attempted with the error character set to zero. Two data fields are used to calculate a subscript; if the subscript is out of range, the sample is ignored. If there is more than one subscript error or more than one conversion error in a batch, that batch is then ignored.

The numbers to the right of each line are card sequence numbers, which are not part of the program itself.

The CHECK prefixes in cards 1 and 25 are included to help with debugging; in a production program, they would be removed. The prefix in card 1 specifies that interrupts will occur at the following times: just before the statements HEADER, NEWBATCH, and BADBATCH are executed; just before the procedure INPUT is invoked; and whenever the value of an element of the variable SAMPLE changes. Since no ON statement has been executed for the CHECK condition, system action is performed. This will result in the appropriate name being written on SYSPRINT (together with the new value in the case of SAMPLE).

Since the labels used within the internal procedure INPUT are not known in DIST, they cannot be specified in a CHECK list for DIST. A separate CHECK prefix is therefore inserted just before the procedure statement heading INPUT. This check list specifies the labels in INPUT, and the array TABLE.

The first statement executed is the ON ENDFILE statement in card 9. This specifies that the external procedure SUMMARY is to be called when an ENDFILE interrupt occurs. This action applies within DIST and within INPUT and within all other procedures called by DIST, unless they establish their own action for ENDFILE.

```
(CHECK(HEADER,NEWBATCH,INPUT,BADBATCH,SAMPLE)):     /*DEBUG*/                01
    DIST: PROCEDURE;                                                          02
          DECLARE 1 SAMPLE EXTERNAL,                                          03
                    2 BATCH CHARACTER(6),                                     04
                    2 SNO PICTURE '9999',                                     05
                    2 READINGS CHARACTER(70),                                 06
                  TABLE(15,15) EXTERNAL, (ONCHAR, ONCODE) BUILTIN;
   /* ESTABLISH INTERRUPT ACTIONS FOR ENDFILE & CONVERSION */                 08
          ON ENDFILE (PDATA) CALL SUMMARY;                                    09
          ON CONVERSION BEGIN;  CALL SKIPBCH;                                 10
                                GO TO NEWBATCH;                               11
                        END;                                                  12
          ON ERROR DISPLAY(BATCH||SNO||READINGS);                             13
   /* MAIN LOOP TO PROCESS HEADER & TABLE */                                  14
  HEADER: READ INTO (SAMPLE) FILE (PDATA);                                    15
          /* CHECK ACTION LISTS INPUT DATA FOR DEBUG */                       16
          IF SNO ¬= 0 THEN SIGNAL CONVERSION;                                 17
NEWBATCH: GET LIST (OMIN,OINT,AMIN,AINT) STRING (READINGS);                   18
          TABLE = 0;                                                          19
          CALL INPUT;                                                         20
          CALL PROCESS;                                                       21
          GO TO HEADER;                                                       22
   /* ERROR RETURN FROM INPUT */                                              23
BADBATCH: SIGNAL CONVERSION;                                                  24
(CHECK(IN1,IN2,ERR2,ERR1,TABLE)):           /*DEBUG*/                         25
           INPUT: PROCEDURE;                                                  26
           /* ESTABLISH INTERRUPT ACTIONS FOR CONVERSION & SUBRG */           27
                 ON CONVERSION BEGIN;                                         28
                               IF ONCODE = 624 & ONCHAR = ' '                 29
                                  THEN DO; ONCHAR = '0';                      30
                                           GO TO ERR1;                        31
                                       END;                                   32
                                  ELSE GO TO BADBATCH;                        33
                               END;                                           34
                 ON SUBSCRIPTRANGE GO TO ERR2;                                35
           /* LOOP TO READ SAMPLE DATA AND ENTER IN TABLE */                  36
             IN1: READ INTO (SAMPLE) FILE (PDATA);                            37
                  IF SNO = 9999 THEN RETURN;        /*TRAILER CARD*/         38
             IN2: GET EDIT (R,OMEGA,ALPHA)(3 P'999')                          39
                      STRING (READINGS);                                      40
(SUBSCRIPTRANGE): TABLE((OMEGA-OMIN)/OINT,(ALPHA-AMIN)/AINT) = R;             41
                  GO TO IN1;                                                  42
           /* FIRST CONVERSION & SUBSCRIPTRANGE ERROR IN THIS BATCH */        43
            ERR2: ON SUBSCRIPTRANGE GO TO BADBATCH;  /*FOR NEXT ERROR*/       44
                  CALL ERRMESS(SAMPLE,02);                                    45
                  GO TO IN1;                                                  46
            ERR1: REVERT CONVERSION;       /*SWITCH FOR NEXT ERROR*/          47
                  CALL ERRMESS(SAMPLE,01);                                    48
                  GO TO IN2;                                                  49
                  END INPUT;                                                  50
          END DIST;                                                           51
```

Figure 14.1. A program checkout routine

Throughout the procedure, any conditions except SIZE, SUBSCRIPTRANGE, STRINGRANGE, STRINGSIZE, and CHECK are enabled by default; and for all conditions except those mentioned explicitly in ON statements, the system action applies. This system action, in most cases, is to generate a message and then to raise the ERROR condition. The action specified for the ERROR condition in card 13 is to display the contents of the card being processed. When the ERROR action is completed, the FINISH condition is raised, and execution of the program is subsequently terminated.

The statement in card 10 specifies action to be taken whenever a CONVERSION interrupt occurs. Since this action consists of more than one statement, it is bracketed by BEGIN and END statements.

The main loop of the program starts with the statement HEADER. Since the CHECK

condition is enabled for HEADER, an interrupt will occur before HEADER is executed. The READ statement with the INTO option will cause a CHECK condition to be raised for each element of the variable SAMPLE; consequently, the input is listed in the form of data-directed output.

The card read is assumed to be a header card. If it is not, the SIGNAL CONVERSION statement causes execution of the BEGIN block, which in turn calls a procedure (not shown here) that reads on, ignoring cards until it reaches a header card. The begin block ends with a GO TO statement that terminates the on-unit.

The GET statement labeled NEWBATCH uses the STRING option to get the different test numbers that have been read into the character string READINGS, which is an element of SAMPLE. Since the variables named in the data list are not explicitly declared, their appearance causes implicit declaration with the attributes FLOAT DECIMAL (6).

The array TABLE is initialized to zero before the procedure INPUT is called. This procedure inherits the on-units already established in DIST, but it can override them.

The first statement of INPUT establishes a new action for CONVERSION interrupts. Whenever an interrupt occurs, the ONCODE is tested to check that the interrupt is due to an illegal P format input character and that the illegal character is a blank. If the illegal character is a blank, it is replaced by a zero, and control is transferred to ERR1.

ERR1 is internal to the procedure INPUT. The statement, REVERT CONVERSION, nullifies the ON CONVERSION statement executed in INPUT and restores the action specified for conversion interrupts in DIST (which causes the batch to be ignored).

After a routine is called to write an error message, control goes to IN2, which retries the conversion. If another conversion error occurs, the interrupt action is that specified in cards 10 and 11.

The second ON statement in INPUT establishes the action for a SUBSCRIPTRANGE interrupt. This condition must be explicitly enabled by a SUBSCRIPTRANGE prefix for an interrupt to occur. If an interrupt does occur, the on-unit causes a transfer to ERR2, which establishes a new on-unit for SUBSCRIPTRANGE interrupts, overriding the action specified in the ON statement in card 35. Any subsequent subscript errors in this batch will, therefore, cause control to go to BADBATCH, which signals the CONVERSION condition as it existed in the procedure DIST. Note that on leaving INPUT, the on-action reverts to that established in DIST, which in this case calls SKIPBCH to get to the next header card.

After establishment of a new on-unit, a message is printed, and a new sample card is read.

The statement labeled IN1 reads an 80-column card image into the structure SAMPLE. A READ statement does not cause input data to be checked for validity, so the CONVERSION condition cannot arise.

The statement IN2 checks and edits the data in card columns 11 through 19 according to the picture format item. A non-numeric character (including blank) in these columns will cause a conversion interrupt, with the results discussed above.

The next statement (card 41) has a SUBSCRIPTRANGE prefix. The data just read is used to calculate a double subscript. If either subscript falls outside the bounds declared for TABLE, an interrupt occurs. If both fall outside the range, two interrupts occur.

Chapter 15: Execution-time Facilities of the Checkout Compiler

Introduction

This chapter describes language features which can provide various facilities to help the programmer at execution time. These features are implemented by the PL/I checkout compiler only. If they are included in a source program to be processed by the PL/I optimizing compiler, they are checked for correct syntax and then ignored; their presence in such a program is not regarded as an error.

In order that the working time of both the programmer and the computer shall be used with the maximum efficiency, it is essential that program turnaround should be as rapid as possible. The most important way of achieving this, as far as the programmer is concerned, is to reduce the time he spends finding out how well his program works, and to allow him to correct any syntactic or logical errors with the minimum delay. The PL/I checkout compiler supports this aim by providing execution-time facilities that:

1. Provide the programmer with information about designated items. This information comprises:

   a. A trace of the items, that is, information is put out whenever these items are referenced in pre-defined situations throughout execution.

   b. A list showing the current status of the designated items at any specified point during execution.

   The items to be traced or listed, and the points at which this output occurs, are specified by statements in the source program. The pre-defined situations are specified in the language.

2. Allow the programmer to initiate the trace dynamically.

3. Provide him with the opportunity, in the appropriate processing environment, for amending his program. The amendments apply only to the current execution of the program, and are not incorporated in the source program.

The extent to which these facilities are applicable to a particular program depends on the processing mode:

1. Batch processing:

   The programmer does not control the time at which execution begins, and cannot intervene during execution to initiate a trace or a current-status list, or to modify his program. If a trace or a current-status list is required, the appropriate statements must have been included in the source program. Output from these facilities is not available until execution has terminated.

2. Conversational processing:

   The programmer initiates execution of his program at the terminal and can intervene during execution to initiate a trace or a current-status list, or to temporarily modify his program. Statements to initiate a trace or status-list can also be included in the source program. Output from these facilities is immediately available and can be printed at the terminal.

   If the SYSPRINT file is associated with a device other than the programmer's terminal, some of SYSPRINT output will appear on both devices. That part of the SYSPRINT output which is not normally available at the terminal can be copied onto it by means of the appropriate terminal instruction.

Conversational processing requires a keyboard terminal as the input/output device. This enables the programmer to:

1. Transmit and receive data at a rate fast enough to allow him to maintain a train of thought, and

2. Have control passed to him at the terminal, or obtain it by calling attention from the terminal.

Processing at the terminal is performed in immediate mode, that is, any instruction entered can be executed immediately. If a permitted PL/I statement or statement group is entered, it can be translated and interpreted (executed) immediately.

PL/I includes statements and options that support these facilities. These provide:

1. Tracing facilities:

   Information about designated items can be written on the SYSPRINT file.

2. Current status list:

   The current status of problem data or program-control data can be written on the SYSPRINT file.

3. Program amending:

   Extra PL/I statements can be included in the program during execution. These are processed in immediate mode, that is, they are entered at the terminal and can be immediately translated and interpreted. These statements are not incorporated permanently in the source program.

Note the relationship of PL/I and the processing mode:

1. Any statement in a PL/I source program submitted for batch processing can also be included in a source program submitted for conversational processing, and vice versa.

2. There are some language items that have or can have a different usage in batch processing from that in conversational processing. They are:

   a. GO TO statement: In batch processing, control is transferred to a statement identified by a label.
   In conversational processing, control can be transferred to a statement identified by either a label or, in a GO TO statement entered in immediate mode, a number.

   b. HALT statement: In batch processing, this statement is a null operation.
   In conversational processing, this statement causes execution of the current task to be suspended and control passed to the terminal.

   c. ERROR condition: In batch processing, if the ERROR condition is raised, the standard system action is to raise the FINISH condition and terminate the task or, if the condition is raised in the major task, terminate the program.
   In conversational processing, if the ERROR condition is raised, the standard system action is to pass control to the terminal.

   d. FINISH condition: In batch processing, the standard system action in the absence of an on-unit is simply to continue processing from the point where the interruption occurred.

   In conversational processing, the standard system action is to pass control to the terminal.

3. There are a few PL/I statements that cannot be used in immediate mode; these are described in the section "Program Amending," below.

Tracing Facilities

The tracing mechanism is activated by the following statements:

Item or feature	Statements
Data	CHECK/NOCHECK
Transfer of control	FLOW/NOFLOW

CHECK and NOCHECK Statements

A CHECK statement provides, for every statement that comes within its range, dynamic enabling of the CHECK condition. As a result, the standard system action for the CHECK condition can be taken for these statements; this action provides that information is written, on the SYSPRINT file, about the names specified or assumed in the prefix whenever these names appear in pre-defined situations during program execution. If an on-unit has been established for the CHECK condition the on-unit is executed and standard system action is not performed.

A CHECK statement can specify a list of names. If it has such a list, the CHECK condition is enabled for the specified names only. If it does not have a list, the CHECK condition is enabled for all the names in the program.

A CHECK statement remains effective until the program terminates or an appropriate NOCHECK statement is executed. The NOCHECK statement suppresses the CHECK condition for specified or assumed names. If no names are specified in a NOCHECK statement, then the CHECK condition is

suppressed for all names in the program. CHECK and NOCHECK statements executed in a procedure compiled by the checkout compiler have no effect in any procedures compiled by the optimizer that may form part of the same program.

The range of a CHECK or NOCHECK condition statement is:

1. In the external block that contains the CHECK or NOCHECK statement: all the statements executed after the execution of the CHECK or NOCHECK statement.

   In this context, "contains" means that the CHECK or NOCHECK statement is in the external block or in a block internal to the external block.

2. In an external, separately compiled block invoked from a block to which the CHECK or NOCHECK statement applies: all references to names associated with the inherited prefixes if these names are known in both the invoking and the invoked blocks. Thus, when a name in the CHECK or NOCHECK statement name list appears in the invoked external procedure, it will be within the range of the statement only if it is declared in both procedures to be EXTERNAL.

The names can be unsubscripted, non-locator-qualified variables, label constants, or entry constants.

The effect of the use of both CHECK and NOCHECK statements in a program is shown by the following example:

```
      .
      .
      .
   CHECK (A,B,C,D);
      .
      .
      .
   NOCHECK (A,D);
      .
      .
      .
   CHECK (D,E);
      .
      .
      .
   NOCHECK (B,E);
```

The first CHECK statement establishes the names A, B, C, and D as members of the name list.

The first NOCHECK statement deletes A and D from this list; after this point, the CHECK condition is raised for B and C only.

The second CHECK statement restores D to the name list and adds a new name E, to the list. After this point, the CHECK condition is raised for B, C, D, and E.

The second NOCHECK statement deletes B and E from the name list. After this point, the CHECK condition is raised for C and D only.

The CHECK and NOCHECK statements effectively modify actual or inherited CHECK or NOCHECK prefixes and add CHECK or NOCHECK prefixes to currently unprefixed statements. A statement inherits a prefix either from an actual prefix to a PROCEDURE or BEGIN block or from a previously-executed CHECK or NOCHECK statement; in both cases, the CHECK or NOCHECK keyword effectively adds a corresponding prefix to every statement within its range. To determine the effect of a CHECK or NOCHECK statement, carry out, conceptually, the following steps.

1. When a CHECK statement without a name-list is executed, delete all actual or inherited CHECK and NOCHECK prefixes within its range, then allow every statement within its range to inherit a CHECK prefix without a name-list.

2. When a NOCHECK statement without a name-list is executed, delete all actual or inherited CHECK and NCCHECK prefixes within its range.

3. When a CHECK or NOCHECK statement with a name-list is executed, carry out the following steps on all statements within its range.

   a. Delete from all actual or inherited prefixes (both CHECK and NOCHECK) all names that appear in the CHECK or NOCHECK statement name-list (except where the same name appears in the statement and the prefix but refers to a different data item in each case). In both cases, treat a prefix with no name-list as having a name-list that includes all known names.

   b. If the statement is a CHECK add a CHECK prefix having the same name-list to every statement. If the statement is a NOCHECK, add a NOCHECK prefix having the same name-list to every statement. In both cases, exclude any names that are not known at the statement being prefixed.

Note: Before carrying out (a), expand into their element names any structure names in

the CHECK or NOCHECK statement and in any prefixes that may be modified.

The action of CHECK and NOCHECK statements in combination with prefixes is illustrated by the following examples. They show how the effects of prefixes written by the programmer are modified by the execution of a CHECK or NOCHECK statement.

Example 1:

```
CHECK;              /* NAMES CHECKED FOR:   */
   .
   .
   .
(CHECK(A)):...;     /* ALL                  */
(CHECK(D,E)):....;/* ALL                    */
(NOCHECK(A)):....;/* ALL                    */
(NOCHECK):....;     /* ALL                  */
(CHECK):....;       /* ALL                  */
```

Example 2:

```
CHECK(A,B,C);       /* NAMES CHECKED FOR:   */
   .
   .
   .
(CHECK(A)):....;    /* A,B,C                */
(CHECK(D,E)):....;/* A,B,C,D,E              */
(NOCHECK(A)):....;/* A,B,C                  */
(NOCHECK):....;     /* A,B,C                */
(CHECK):....;       /* ALL                  */
```

Example 3:

```
NOCHECK(C,D);       /* NAMES CHECKED FOR:   */
   .
   .
   .
(CHECK(A)):....;    /* A                    */
(CHECK(D,E)):....;/* E                     */
(NOCHECK(A)):....;/* NONE                   */
(NOCHECK):....;     /* NONE                 */
(CHECK):....;       /* ALL EXCEPT C,D       */
```

The situations in which the CHECK condition is raised are described in "CHECK Condition" in section H, "ON-Conditions." Some of them are illustrated in the figure 15.1.

The CHECK condition is raised when AUTOMATIC, BASED, or CONTROLLED variables are initialized by means of the INITIAL attribute (with or without the CALL option). If standard system action is taken, CHECK output is produced as follows:

AUTOMATIC: Only if the CHECK statement has been executed before the establishment of the prologue for the block containing the initialization. Thus in the example given, CHECK output is never produced for the variable C because, in the example, no assignment is made to C. Such output would be produced if, for example, the CHECK statement was executed in a block that contained the procedure PR, or if the procedure PR was invoked recursively. In the latter instance, CHECK output would be produced at the second and all succeeding invocations.

BASED or CONTROLLED: When the variable is allocated by means of an ALLOCATE or LOCATE statement. CHECK is never raised by the INITIAL attribute of a BASED variable that is never explicitly allocated.

Note: The CHECK condition is never raised for the initialization of STATIC variables.

FLOW Statement

The FLOW statement causes information about the transfer of control during execution of a task to be written on the SYSPRINT file. When a FLOW statement has been executed, it remains effective until the task terminates or until a NOFLOW statement is executed in the same task. The FLOW statement has no effect outside procedures compiled by the checkout compiler.

When a FLOW statement has been executed in a task, every transfer of control that occurs subsequently in that task causes a flow comment to be put out before the transfer takes place. This comment consists of:

1. The number of the statement that causes the transfer of control.
2. The number of the statement to which control is transferred.

The transfer of control can only be to a point within the task that contains the FLOW statement. If the FLOW statement is in a nested block, the point can be in any containing block, including external blocks. However, the FLOW statement cannot be inherited across tasks; for example, if task A, which contains a FLOW statement, attaches a task (B) which does not contain a FLOW statement, no flow comments are put out during the execution of task B.

```
PR:PROC OPTIONS(MAIN);
      DCL (A,B) DECIMAL(5),
          C CHAR(10) INIT('DAILYRATES') AUTO,
          C1 CHAR(25) BASED(P),
          D(10) LABEL,
          G ENTRY;
               .
               .
               .;
      CHECK(A,B,C,C1,D,F,P);
               .
               .
               .
      A=1;                              /*CHECK OUTPUT FOR A*/
      B=2;                              /*CHECK OUTPUT FOR B*/
      ALLOCATE C1;                      /*CHECK OUTPUT FOR P*/
               .
               .
               .
      D(1): READ FILE (X) INTO(C1); /*CHECK OUTPUT FOR D(1) AND C1*/
               .
               .
               .
      E: GET DATA(A,B);                 /*CHECK OUTPUT FOR A AND B*/
               .
               .
               .
      DISPLAY (C) REPLY(C1);            /*CHECK OUTPUT FOR C1*/
               .
               .
               .
      CALL F(A,B);                      /*CHECK OUTPUT FOR F AT TIME OF CALL*/
               .                        /*CHECK OUTPUT FOR A, B ON RETURN
               .                          FROM F (EVEN IF VALUES OF Y, Z
               .                          NOT CHANGED IN F)*/
      CALL G;
               .
               .
               .
      F: PROC(Y,Z);
          DCL (Y,Z) DECIMAL(5)
               .
               .
               .;
      Y=20;
               .
               .
               .
      END F;
      END PR;
```

Notes:

1. If the CHECK statement had been CHECK;, output would have been as indicated with, in addition, CHECK output for E, G, and Y.

2. If dummy arguments had been created for A and B, no CHECK output would have been produced.

Figure 15.1. Example of use of CHECK statement

While it is always clear why a particular statement is specified in the flow comment as the statement that caused the transfer of control, it is not always so obvious why control was transferred to the statement given as the destination.

Consider the following program:

```
Statement
number
          .
          .
          .
   3      FLOW;
            .
            .
            .
  12      ON CONVERSION GO TO L12;
            .
            .
            .
  24      GO TO L19;
            .
            .
            .
  35      L12:CALL ABS;
  36      ...;
            .
            .
            .
  42      X = F(A,B);
            .
            .
            .
  57      L19:DO I = 1 TO 99;
            .
            .
            .
  65      END;
  66      A = B**2;
            .
            .
            .
 117      ABS:PROC;
              .
              .
              .
 124        RETURN;
              .
              .
              .
 130      END ABS;
            .
            .
            .
 150      SIGNAL CONVERSION;
            .
            .
            .
 192      F:PROC(Y,Z) RETURNS(DECIMAL);
              .
              .
              .
 197        RETURN (M);
 198      END F;
```

In this program, the statement numbers in the flow comments produced by transfers of control are shown in figure 15.2.

Statement numbers are derived from a count of semicolons, for both simple and compound statements. In the example above,

```
ON CONVERSION GO TO L12;
```

is counted as one statement.

NOFLOW Statement

The NOFLOW statement suppresses the action of a FLOW statement executed earlier in the same task.

Current Status List

Information about selected items in a program can be put into the output stream by means of a PUT statement with one of the options LIST, DATA, SNAP, FLOW, or ALL. This information can comprise names and values of both problem-data and program-control variables and details of data relating to flow of control and ON conditions. Note that only the PL/I checkout compiler can provide all this information. The PL/I optimizing compiler can provide only the names and values of problem-data variables, and the names of program control variables.

The information provided by the options specified in the PUT statement is summarized in figure 15.3.

Details of the output provided by the use of each of these options is given in the sections below.

PUT Variables

The data list for the LIST and DATA options can specify both problem and program-control variables. Only problem variables can be specified in an EDIT data list. If DATA is specified without a data list, the data is assumed to be all problem and program-control variables known in the block.

Statement	Transferred from	Transferred to
GO TO in on-unit	12	35
GO TO	24	57
CALL	35	117
Function reference	42	192
DO	57	When iteration is complete, statement number of statement after matching END statement, that is, 66
END for iterative DO	65	57
RETURN in procedure invoked by CALL	124	35
END in procedure invoked by CALL	130	35
SIGNAL	150	12
RETURN in procedure invoked as function reference	197	42

Figure 15.2. Flow comments produced by various transfers of control

Option	Information
[LIST] (data-list) DATA [(data-list)] EDIT (data-list)(format-list) [,(data-list)(format-list)]...	Variables
SNAP	Active blocks and on-units
FLOW[(n)]	Last n transfers of control
ALL[(character-string-expression)]	Variables, active blocks and on-units, transfers of control, ON built-in functions

Figure 15.3. Program-item information provided by the PUT statement options

The information provided for the problem variables specified or assumed depends on the data-transmission option selected:

Option	Output
LIST	Value
DATA	Name of variable, and value
EDIT	Value as specified

If a variable specified or assumed for a PUT DATA statement is not initialized or is not allocated, the checkout compiler includes a comment to this effect in the output.

The information provided for a program-control variable specified or assumed in a PUT LIST or PUT DATA statement depends on the variable. The name of the variable is put out only if DATA (with or without a data list) is specified. A program-control variable does not have a value in the sense that a problem variable has one. Instead, the output for a program-control variable comprises information related to the current situation of the variable. For example,

the output for a file variable states whether the file is open or closed, and the output for an event variable states whether the event is active or inactive.

Under the optimizing compiler a PUT DATA statement specifying a program control variable will cause only the name of the variable to be printed. A PUT LIST or PUT EDIT statement must not specify program control data under the optimizing compiler.

The value output for each type of program-control variable is:

```
AREA
  Area size
  Area extent
  List of freed allocations within the
    extent

ENTRY
  Entry constant assigned to the variable
    (if any)
  If the entry constant is internal and is
    in a procedure that is not the current
    procedure:
    Statement number
    A list of the currently active
      procedures invoked in the process of
      activating the block containing the
      entry constant. If the list of
      active blocks cannot be produced,
      because the entry variable no longer
      has a valid value, a comment to this
      effect is made.
```

Note: The above output is provided by a PUT DATA statement specifying any entry variable or a PUT LIST statement specifying an entry variable that has been declared as having a non-null argument list. If a PUT LIST statement specifies an entry variable that has been declared as not requiring an argument list, the entry is invoked. In such a case, only PUT DATA may be used to put out the value of the entry variable.

```
EVENT
  Description of the event:
    Task or I/O event
    Active or inactive
    Complete or incomplete
    Status
  If the event is active:
    Indication of whether task or I/O event
    Absolute priority
    If a task event:
      Entry name specified in the CALL
        statement that activated the event
        variable
      Statement number of this CALL
      statement
    If an I/O event
      File name or 'DISPLAY'
      Statement number of I/O statement
        that activated the event variable
      Statement numbers of the WAIT
        statements associated with this
        event

FILE (variable or constant)
  If item is a variable:
    File constant assigned to variable
  Whether the file is open or closed
  If the file is open:
    List of attributes other than the
      ENVIRONMENT attribute
    Number of records transmitted
  If the file is a STREAM file:
    Already-transmitted items in the
      current record

LABEL
  If variable has valid label constant
    value
    Label constant assigned to the variable
    Statement number
    If the label constant is in a procedure
      that is not the current procedure:
      A list of the currently active
        procedures invoked in the process
        of activating the block containing
        the label constant. If the list of
        active blocks cannot be produced,
        because the label variable no
        longer has a valid value, a comment
        to this effect is made.
  If label variable does not have a valid
    value:
      Comment to this effect
```

Note: Label variables can be initialized without having constants assigned to them. In the program:

```
DCL L(3) LABEL;
     .
     .
     .
L(1):...;
     .
     .
     .
```

L(1) has a value and can appear in a GO TO statement; but it is not a label constant. In this case, the full output for a label variable with a label constant value is transmitted, except for the label constant value itself.

```
OFFSET
  Whether the offset has a null value
  If the offset is not null and the long
    form of the offset variable is used:
    Name of the based variable addressed by
      the offset
    Name of the area, and value (in bytes)
      of the offset
    Whether it is invalid; for example,
      because the based variable previously
      associated with it had been freed
```

If the offset is not null and the short form of the offset is used:
The byte-address value of the offset

POINTER
Whether the pointer has a null value
If the pointer is not null and the long form of pointer is used:
Name of the based variable addressed by the pointer
If the last value assigned to the pointer is the value of an offset:
Name of the area, and value (in bytes) of the offset[1]
If the last usage of the pointer was in a READ...SET or a LOCATE statement:
Name of the file
Record number of the record with which the pointer is associated
Name of based variable (if a LOCATE statement)
Whether it is invalid; for example, because the based variable previously associated with it has been freed
If the pointer is not null and the short form of pointer is used:
The byte-address of the pointer

TASK
Description of the task:
Active or inactive
Absolute priority
If the task is active:
Entry name specified in the CALL statement that activated the task variable
Statement number of this CALL statement

PUT SNAP Statement

The PUT statement with the SNAP option causes the following data to be put into the stream:

1. The current statement number.

2. A list of the currently active blocks and on-units invoked in the process of activating the block in which the PUT statement was executed. Routines compiled by the optimizing compiler, and FORTRAN and COBOL routines, are included in the list.

---

[1]Except that if the area is based or is an element of an array of areas, the value of the offset is not transmitted; and if the area is an element of a based structure, neither the offset nor the name of the area is transmitted.

PUT FLOW Statement

The PUT statement with the FLOW option causes a list of the last n transfers of control to be put into the stream. In each transfer of control, the statements involved are:

1. The statement that caused the transfer of control.

2. The statement to which control is transferred.

The rules for identifying these statements are the same as for the FLOW statement. The value of n is any value specified by the programmer; it may be specified in the PUT FLOW statement or in the appropriate compiler option. If there are conflicting values for n in the PUT statement and the compiler option, the smaller is used. If no value is given in either place, then a default of 25 is assumed.

Under the optimizing compiler, the syntax of a PUT FLOW statement is checked, then it is ignored. A PUT FLOW statement has no effect outside procedures compiled by the checkout compiler.

PUT ALL Statement

The PUT ALL statement provides the maximum amount of debugging information obtainable without a dump of main storage. Options may be specified to select a part only of the total information available.

The information transmitted by PUT ALL with no options is as shown in figure 15.4. The content of each item is as follows.

SNAP information: The information provided by a PUT SNAP statement. In a multitasking program, the chain of invocation is followed back through all attaching tasks to the main procedure of the program.

FLOW information: The information provided by a PUT FLOW statement without a number-of-statements option.

Condition built-in functions: Values of the following built-in functions in data-directed format.

DATAFIELD
ONCHAR
ONCODE
ONCOUNT
ONFILE
ONKEY
ONLOC
ONSOURCE

This information is given for the current task only, because the values of the built-in functions are no different in the contexts of other tasks.

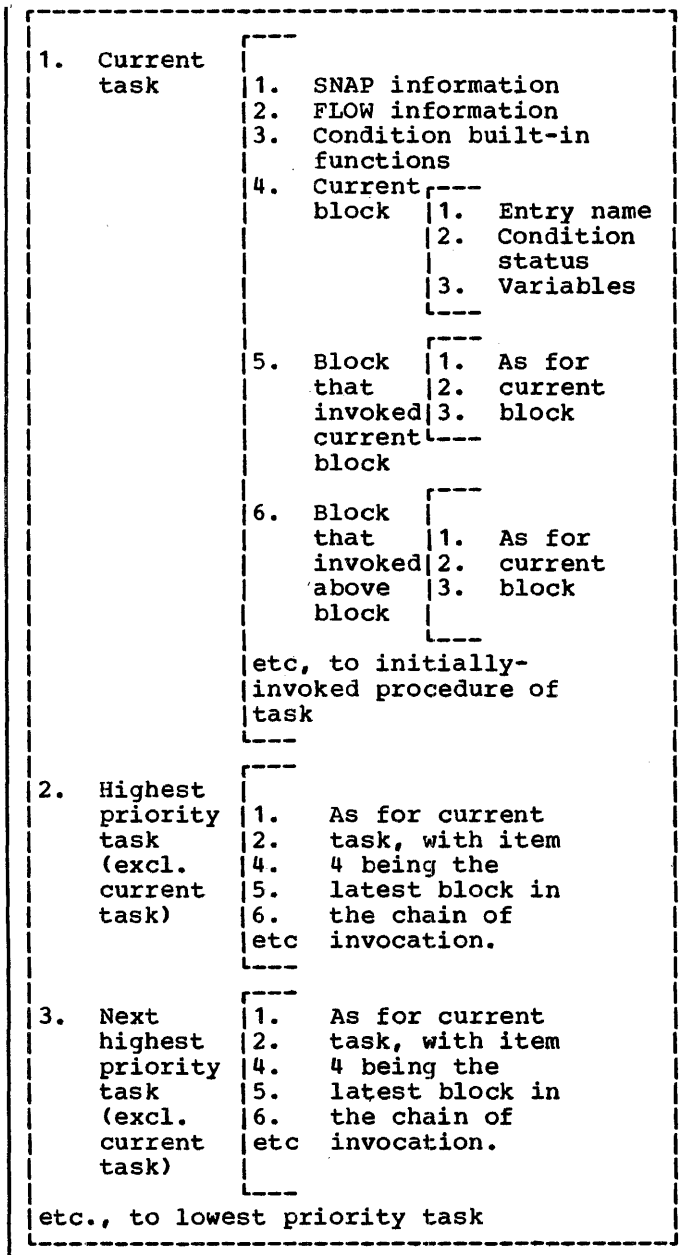


Figure 15.4. Information transmitted by PUT ALL statement

Entry name: The name of the entry point through which the procedure was entered on its current invocation. This is the same as the corresponding name in the SNAP information.

Condition status: For each PL/I on-condition:

Whether it is enabled.

Variables: The value of every problem data and program control variable and every file constant declared in that block, in data-directed format. Controlled variables have every generation transmitted, starting with the latest. If a variable is uninitialized or unallocated a comment is printed.

The options are specified as a character string following the ALL option. The full list of options is as follows.

```
PUT ALL('DSFCTn')
```

where n is a number 1 through 9999. Any or all of the options may be specified together.

When one or more of the options is specified, SNAP and FLOW information is given for each task for which other information is transmitted, except that it is always omitted for the current task. The other information is transmitted if one or more of the options D, S, F, or C is specified. The information is given for all tasks and blocks unless limited by one or both of the options T and n.

The meaning of each option is as follows.

D: Values of problem and program control variables, as defined under "Variables" above, are transmitted. Values of file constants are not transmitted.

S: The same meaning as D, except that array variables are not transmitted.

F: Values of file constants are transmitted.

C: Values of the condition built-in functions and the condition status of each block are transmitted.

T: Limits the output to the current task.

n: Limits the output to this number of blocks.

The options D and S conflict. There is also a conflict if two or more numbers corresponding to two values of n appear in the string. In these cases, the option specified latest in the string overrides any earlier conflicting option.

Under the optimizing compiler, the syntax of the PUT ALL statement is checked, then it is ignored.

Program Amending

When processing in conversational mode, the programmer can suspend program execution and obtain control at the terminal. He does this by striking the appropriate key at the terminal. This raises the ATTENTION condition, which causes processing to be interrupted, and in the absence of an ATTENTION on-unit, control to be passed to the terminal. If the interrupt takes place within the scope of an ATTENTION on-unit, control goes to the terminal upon normal return from the on-unit.

The programmer can then do one of the following:

1. He can enter PL/I statements for immediate execution. When these statements have been processed, he can cause program execution to be resumed at any specified point.

2. He can enter PL/I statements for execution at a specified point once program execution has been resumed. These statements are not executed immediately they are entered, as in the preceding situation, but are stored for later use. Program execution is resumed; when the specified point in execution is reached, then, without further action by the programmer, program execution is again suspended and the extra statements are executed. In order to achieve this, the extra statements must be preceded by an appropriate terminal subcommand. This is an instruction to the checkout compiler. In this instance, the subcommand specifies the point at which program execution will automatically be suspended so that the extra statements entered with it can be executed.

3. He can enter, by itself, one of a number of terminal subcommands.

The subcommands provide various aids to debugging that do not involve entering extra statements.

Full details of the terminal subcommands and their usage are given in OS Time Sharing Option: PL/I Checkout Compiler.

The extra PL/I statements are always executed in immediate mode; this applies whether they are inserted while the programmer has control at the terminal or whether he has used a terminal command to cause them to be inserted at a specified point once program execution has been resumed. A reference in these statements to a name must be to a name known in the current block or known in a specified external procedure. An immediate-mode statement cannot alter an existing declaration, nor can a new declaration be introduced. Similarly, the block structure cannot be altered, by, for example, the creation of a new block.

The restrictions imposed on PL/I statements used in immediate mode are:

1. The following statements cannot be used in immediate-mode:

   BEGIN

   DECLARE

   DEFAULT

   ENTRY

   FETCH[1]

   FORMAT

   ON

   PROCEDURE

   RELEASE[1]

2. If an immediate-mode statement refers to a name that is not known in the block, the name is given the BUILTIN attribute.

3. An unmatched END statement cannot be used.

4. Statements cannot be labeled or have condition prefixes.

---

[1]Except that if the original source program contains FETCH or RELEASE statements, FETCH and RELEASE are valid in immediate mode if they specify procedures specified in the original program statements.

Chapter 16: Compile-time Facilities

Introduction

Compile time is generally defined as that time during which a user's source program is compiled, or translated, into an executable object program. Ordinarily, changes to a source program may not be made at this time.

However, with PL/I, the programmer does have some control over his source program during compile time. His source program can contain special statements (identified by a leading %) that can cause parts of the source program to be altered in various ways:

1. Any identifier appearing in the source program can be changed.
2. If conditional compilation is desired, the programmer can indicate which sections of his program are to be compiled.
3. Strings of text residing in a user library or a system library can be incorporated into the source program.

PL/I makes source program alteration at compile time possible by providing two stages:

1. The Preprocessor Stage -- During this stage, the user's source program is scanned for preprocessor statements, special statements that cause the preprocessor to alter the text being scanned. These statements are considered part of the source program, and appear freely intermixed with the statements and other text of the source program. The altered source program, resulting from the action of the preprocessor statements, then serves as input to the second stage. Note that the preprocessor stage is optional.
2. The Processor Stage -- During this stage, the output from the first stage is compiled into an executable object program.

This chapter is concerned with the first stage; the actual compilation of a program is not discussed.

In addition to the preprocessor statements, the programmer has at his disposal listing control statements which allow him to control the layout of the printed listing of his program. These do not employ the preprocessor stage.

Preprocessor Input and Output

The preprocessor interprets preprocessor statements and acts upon the source program accordingly. Input to the preprocessor is a sequence of characters that is the user's source program. It contains preprocessor statements freely intermixed with the rest of the user's source program. Preprocessor statements are identified by a leading percent symbol (%) and are executed as they are encountered by the preprocessor (with the exception of preprocessor procedures, which must be invoked in order to be executed). One or more blanks may separate the percent symbol from the statement.

In addition to interpreting the preprocessor statements, the preprocessor checks the source program for unmatched delimiters on comments and character-string constants. It also checks non-preprocessor statements for invalid characters, and replaces them with blanks. This is the only checking done at this stage on non-preprocessor statements.

Output from the preprocessor consists of a string of characters called the preprocessed text, which consists of the altered source program and which serves as input to the processor stage.

The programmer can specify compiler options that cause the input to, and the output from, the preprocessor to be printed. The listing of the input to the preprocessor, which represents the program as coded by the programmer, is known as the insource listing, and the listing of the input to the compiler - the preprocessed text if the preprocessor is used - is known as the source listing.

PREPROCESSOR SCAN

The preprocessor starts its scan of the input text at the beginning of the string and scans each character sequentially. As long as a preprocessor statement is not encountered, the characters are placed into

the preprocessed text in the same order and general form in which they were scanned. However, when a preprocessor statement is encountered, it is executed. This execution can cause the scanning of the source program and the subsequent formation of preprocessed text to be altered in either of two ways:

1. The executed statement may cause the preprocessor to continue the scan from a different point in the program. This new point may very well be one that has already been scanned.

2. The executed statement may initiate replacement activity. That is, it may cause an identifier not appearing in a preprocessor statement to be replaced when that identifier is subsequently encountered in the scan. The replacement value will then be written into the preprocessed text in place of the old identifier (see "Rescanning and Replacement" below for details).

The scan is terminated when an attempt is made to scan beyond the last character in the source program. The preprocessed text is completed and the second stage of compilation can then begin.

Rescanning and Replacement

For an identifier to be replaced by a new value, the identifier must first be activated for replacement. Initially, an identifier is activated by its appearance in a preprocessor DECLARE statement (i.e., a % DECLARE statement). (It can be deactivated by appearing in a % DEACTIVATE statement and it can be reactivated by appearing in a % ACTIVATE statement.) After it has been activated initially, it must be given a replacement value. This is usually done via the execution of a preprocessor assignment statement. Once an identifier has been activated and been given a value, any occurrence of that identifier in text other than preprocessor statements is replaced by that value, provided that the identifier is still active when it is encountered by the scan. Preprocessor variables can be activated with either the RESCAN or the NORESCAN options. If the NORESCAN option applies, the value is immediately inserted into the program text. If the RESCAN option applies, a rescan is made during which the value is tested to determine whether it, or any part of it, should be replaced by another value. If it cannot be replaced, it is inserted into the preprocessed text; if it can be replaced, replacement activity continues until no further replacements can be made. Thus, insertion of a value into preprocessed text takes place only after all possible replacements have been made. Note that the deactivation of an identifier causes it to lose its replacement capability but not its value. Hence, the subsequent reactivation of such an identifier need not be accompanied by the assignment of a replacement value.

For example, if the source program contained the following sequence of statements:

```
%DECLARE A CHARACTER, B FIXED;
%A = 'B+C';
%B = 2;
X = A;
```

then the following would be inserted into the preprocessed text in place of the above sequence:

```
X = 2+C;
```

In this example, the first statement is a preprocessor DECLARE statement that activates A and B and also activates them as preprocessor variables with the default RESCAN. The second and third statements are preprocessor assignment statements; the second assigns the character string 'B+C' to A, and the third assigns the constant 2 to B. The fourth statement is a nonpreprocessor statement[1] and, therefore, is not executed at this stage. However, because this statement contains A, and A is a preprocessor variable that has been activated for replacement, the current value of A will replace it in that statement. Thus, the string 'B+C' replaces A in the statement. But this string contains the preprocessor variable B. Upon checking B, the preprocessor finds that it has been activated and that it has been assigned a value of 2. Hence, the value 2 replaces B in the string. Further checking shows that 2 cannot be replaced; scanning resumes with +C which, again, cannot be replaced. Thus, the chain of replacements comes to an end and the resulting statement is inserted into the preprocessed text.

---

[1]For the purpose of this discussion, a nonpreprocessor statement is any statement or sequence of text that appears in the source program but is not contained in a preprocessor statement, nor in a preprocessor procedure, nor in a comment.

If, in the above example, the preprocessor variable A has been activated by this statement:

```
%ACTIVATE A NORESCAN;
```

the statement inserted into the text would be:

```
X = B + C;
```

since the NORESCAN option for preprocessor variable A suppresses the rescanning of the result 'B+C' substituted for A.

Note that the preprocessor variable B has a default precision of (5,0) and, therefore, actually contains 2 preceded by four zeros. When this value replaces B in the string 'B+C' it is converted to a character string and becomes 2 preceded by seven blanks (the rules for conversion of decimal fixed-point values to character string are followed). See the section "Preprocessor Expressions" in this chapter, for details.

Replacement values must not contain percent symbols, unmatched quotation marks, or unmatched comment delimiters.

The following example illustrates how compile-time facilities can be used to speed up the execution of a DO-loop.

A programmer might include the following loop in his program:

```
DO I=1 TO 10;
Z(I)=X(I)+Y(I);
END;
```

The following sequence would accomplish the same thing, but without the requirements of incrementing and testing during execution of the compiled program:

```
%DECLARE I FIXED;
%I=1;
%LAB:;
Z(I)=X(I)+Y(I);
%I=I+1;
%IF I<=10 %THEN %GO TO LAB;
%DEACTIVATE I;
```

The first statement activates I and establishes it as a preprocessor variable. The second statement assigns the value 1 to I. This means that subsequent encounters of the identifier I in non-preprocessor statements will be replaced by 1 (provided that I remains activated). The third statement is a preprocessor null statement that is used as the transfer target for the preprocessor GO TO statement appearing later.

The fourth statement, not being a preprocessor statement, is only scanned for replacement activity; it is not executed. The first time that this statement is scanned, I has the value 1 and has been activated. Therefore, each occurrence of I in this statement is replaced by 1 and the following is inserted into the preprocessed text being formed:

```
Z(       1)=X(       1)+Y(       1)
```

Note that each 1 is preceded by seven blanks.

The fifth statement increments the value of I by 1 and the sixth statement, a preprocessor IF statement, tests the value of I. If I is not greater than 10, the scan is resumed at the statement labeled LAB; otherwise, the scan continues with the text immediately following the %GO TO statement. Hence, for each increment of I, up to and including 10, the assignment statement is rescanned and each occurrence of I is replaced by its current value. As a result, the following statements are inserted into the preprocessed text:

```
Z(       1)=X(       1)+Y(       1);
Z(       2)=X(       2)+Y(       2);
          .
          .
          .
Z(      10)=X(      10)+Y(      10);
```

As before, each number from 1 through 9 is preceded by seven blanks; the number 10 is preceded by six blanks.

When the value of I reaches 11, control falls through to the %DEACTIVATE statement. This statement is interpreted as follows: subsequent encounters of the identifier I in source program text are not to be replaced by the value 11 in the preprocessed text being formed; each I will be left unmodified, either for the remainder of the scan or at least until I is reactivated by a %ACTIVATE statement. If I is again activated, it will still have the value 11 (unless an intervening preprocessor assignment statement has established a new value for I).

Preprocessor Variables

A preprocessor variable is an identifier that has been specified in a %DECLARE statement with either the FIXED or CHARACTER attribute. No other attributes can be declared for a preprocessor variable. Other attributes are supplied by the compiler, however. A preprocessor

variable declared with the FIXED attribute is also given the attributes DECIMAL and precision (5,0); a CHARACTER preprocessor variable is given the VARYING attribute with no maximum length. No contextual or implicit declaration of identifiers is allowed in preprocessor statements.

The scope of a preprocessor variable encompasses all text except those preprocessor procedures that have redeclared that variable. The scope of a preprocessor variable that has been declared in a preprocessor procedure is the entire procedure (there is no nesting of preprocessor procedures).

When a preprocessor variable has been given a value, that value replaces all occurrences of the corresponding identifier in text other than preprocessor statements during the time that the variable is active. If the preprocessor variable is inactive, replacement activity cannot occur for the corresponding identifier.

A preprocessor variable is activated by its appearance in the %DECLARE statement. It can be deactivated and subsequently reactivated by its appearance in %DEACTIVATE and %ACTIVATE statements, respectively. Deactivation of a preprocessor variable does not strip it of its value; in other words, an inactive preprocessor variable retains the value it had while it was active and can be altered by a preprocessor statement or procedure if so desired.

Preprocessor Expressions

Preprocessor expressions are written and evaluated in the same way as source program expressions, with the following exceptions:

1. The operands of a preprocessor expression can consist only of preprocessor variables, references to preprocessor procedures, decimal integer constants, bit-string constants, character-string constants, and references to the built-in functions SUBSTR, INDEX, and LENGTH. Repetition factors are not allowed with the string constants and the arguments to a built-in function reference must be preprocessor expressions.

2. The exponentiation symbol (**) cannot be used as an arithmetic operator.

3. For arithmetic operations, only decimal integer arithmetic of precision (5,0) is performed; that is, each operand is converted to a decimal fixed-point value of precision (5,0) before the operation is performed, and the decimal fixed-point result is converted to precision (5,0) also. Any character string being converted to an arithmetic value must be in the form of an optionally signed decimal integer constant. Note that the properties of the division operator are affected. For example, the expression 3/5 evaluates to 0, rather than to 0.6.

4. The conversion of a fixed-point decimal number to a character string always results in a string of length 8. (Leading zeros in the number are replaced by blanks and an additional three blanks are appended to the left end of the number, one of which is replaced by a minus sign if the number is negative.)

A character string in an expression being assigned to a preprocessor variable may include preprocessor variables, references to preprocessor procedures, constants, and operators; preprocessor statements cannot be included in such strings.

Preprocessor Procedures

A preprocessor procedure is an internal function procedure that can be executed only at the preprocessor stage. Its syntax differs from other function procedures in that its PROCEDURE and END statements must each have a leading percent symbol. The format of a preprocessor procedure is as follows:

```
%label:  [label:]...  PROCEDURE
        [(identifier
        [,identifier] ...)]
        RETURNS[(CHARACTER|FIXED)];
            .
            .
            .
[label:]...RETURN
          (preprocessor-expression);
            .
            .
            .
% [label:] ...   END [label];
```

More than one RETURN statement may appear. The general rules governing the statements that can appear within a preprocessor procedure are given in the description of the %PROCEDURE statement in section J, "Statements". One thing should be noted, however: no statement appearing

within a preprocessor procedure can have a leading percent symbol.

INVOCATION OF PREPROCESSOR PROCEDURES

A preprocessor procedure is invoked by a function reference in the usual sense; i.e., by the appearance of the entry name and its associated argument list (if any) in an expression. The function reference can appear in a preprocessor statement or in a nonpreprocessor statement.

A condition must be met if the reference to the preprocessor procedure is made in a nonpreprocessor statement: the entry name used in the reference must be active at the time the reference is encountered. Entry names of preprocessor functions are the same as preprocessor variables as far as activation and deactivation is concerned; i.e., they can be activated initially by a %DECLARE statement or by a %ACTIVATE statement.

Provided its entry name is active, a preprocessor procedure need not be scanned before it is invoked. It must, however, be either present in the main text, or in included text (by a %INCLUDE statement) at the point of invocation. Preprocessor procedure entry names need not be specified in %DECLARE statements.

The value returned by a preprocessor function (i.e., the value of the preprocessor expression in the RETURN statement) always replaces the function reference and its associated argument list. Note that for a reference made in a preprocessor statement, the replacement is only for that particular execution of the statement; a subsequent scanning of the statement would again result in the invocation of the function.

ARGUMENTS AND PARAMETERS FOR PREPROCESSOR FUNCTIONS

The number of arguments in the procedure reference and the number of parameters in the %PROCEDURE statement need not be the same. The arguments are interpreted according to the type of statement (preprocessor or nonpreprocessor) in which the function reference appears. The arguments in the argument list are evaluated before any match is made with the parameter list. If there are more arguments than parameters, the excess arguments on the right are ignored. (Note that for a function reference argument, the function is invoked and executed, even if the argument is ignored later.) If there are fewer arguments than parameters, the excess parameters on the right are given values of zero, for FIXED parameters, or the null string, for CHARACTER parameters. The usual rules concerning the creation of dummy arguments apply if the function reference is in a preprocessor statement, but dummy arguments are always created if the function reference occurs in a nonpreprocessor statement.

If the function reference appears in a nonpreprocessor statement, the arguments are interpreted as character strings and are delimited by the appearance of a comma or a right parenthesis occurring outside of balanced parentheses. For example, the argument list (A(B,C),D) has two arguments, namely, the string A(B,C) and the string D. Each argument is then scanned for possible replacement activity. Both the procedure name and its argument list must be found at one replacement level. Thus, only the commas and parentheses seen in the text being scanned when the procedure name is encountered are considered in this context. After all replacements have been made, each resulting argument is converted to the type indicated by the corresponding parameter attribute in the preprocessor procedure statement for the function entry name.

If the function reference appears in a preprocessor statement, the arguments are associated with the parameters in the normal fashion. If there is a disagreement, the arguments are converted to the attributes of the corresponding parameters. Only preprocessor variables, character- string constants, and fixed-point decimal constants can be passed to a preprocessor function invoked by a preprocessor statement.

Returned Value

The value returned by a preprocessor function to the point of invocation is represented by the preprocessor expression in the RETURN statement of that function. Before being returned, this value is converted (if necessary) to the attribute (CHARACTER or FIXED) specified in the RETURNS option of the function's %PROCEDURE statement. If the point of invocation is in a nonpreprocessor statement, and the entry name has not been activated with the NORESCAN option, the value is scanned for replacement activity after it has replaced the function reference.

Note that the rules for preprocessor expressions do not permit the value

returned by a preprocessor procedure to contain preprocessor statements.

Example of Preprocessor Functions

In the statements below, VALUE is a preprocessor function procedure that returns a character string of the form 'arg1(arg2)', where arg1 and arg2 represent the arguments that have been passed to the function.

Assume that the source program contains the following sequence:

```
%DECLARE A CHARACTER;
DECLARE (Z(10), Q) FIXED;
%A='Z';
%VALUE:  PROCEDURE (ARG1,ARG2) RETURNS
              (CHARACTER);
         DECLARE ARG1 CHARACTER,
              ARG2 FIXED;
         RETURN(ARG1||'('||ARG2||')');
         %END VALUE;
Q = 6+VALUE(A,3);
```

When the scan encounters the last statement, A is active and is thus eligible for replacement. Since VALUE is also active, the reference to it in the last statement causes the preprocessor to invoke the preprocessor function procedure of that name. However, before the arguments A and 3 are passed to VALUE, A is replaced by its value Z (assigned to A in a previous assignment statement), and 3 is converted to fixed-point to conform to the attribute of its corresponding parameter. VALUE then performs a concatenation of these arguments and the parentheses and returns the concatenated value, that is, the string Z (3), to the point of invocation. The returned value replaces the function reference and the result is inserted into the preprocessed text. Thus, the preprocessed text generated by the above sequence is as follows:

```
DECLARE (Z(10),Q) FIXED;
Q = 6+Z(       33);
```

The preprocessor function procedure GEN defined in the following example can generate a GENERIC declaration for up to 99 entry names with up to 99 parameter descriptors in the parameter descriptor lists. Only four are generated in this example, however.

Assume that the source program contains the following sequence:

```
DCL A GEN (A,2,5,FIXED),...;
      .
      .
      .
   %GEN:  PROC (NAME,LOW,HIGH,ATTR)
               RETURNS (CHAR);
     DCL (NAME, SUFFIX, ATTR, STRING)
       CHAR, (LOW, HIGH, I, J) FIXED;
     STRING='GENERIC(';
     DO I=LOW TO HIGH; /* ENTRY NAME
                  LOOP */
       IF I>9
         THEN SUFFIX=SUBSTR(I, 7, 2);
           /* 2 DIGIT SUFFIX*/
         ELSE SUFFIX=SUBSTR(I, 8, 1);
           /*1 DIGIT SUFFIX*/
       STRING=STRING||NAME||SUFFIX||
         ' WHEN (';
       DO J=1 TO I; /* DESCRIPTOR
         LIST*/
         STRING=STRING||ATTR;
         IF J<I /* ATTRIBUTE
                   SEPARATOR */
           THEN STRING=STRING||',';
           ELSE STRING=STRING||')';
           /* LIST SEPARATOR */
       END;
       IF I<HIGH /* ENTRY NAME
                   SEPARATOR*/
           THEN STRING=STRING||',';
           ELSE STRING=STRING||')';
           /* END OP LIST */
     END;
     RETURN (STRING);
   % END;
```

The following text is produced by this preprocessor procedure:

```
DCL A GENERIC(A2 WHEN (FIXED,FIXED),
              A3 WHEN (FIXED, FIXED,
                FIXED),
              A4 WHEN (FIXED, FIXED,
                FIXED, FIXED),
              A5 WHEN (FIXED, FIXED,
                FIXED, FIXED, FIXED));
```

Use of the SUBSTR, LENGTH, and INDEX Built-in Functions

A reference to SUBSTR, LENGTH, or INDEX in a nonpreprocessor statement is executed by the preprocessor only if these names are active. These built-in functions can be activated only by a %DECLARE or %ACTIVATE statement. If any of the identifiers SUBSTR, LENGTH, and INDEX appear in prefixes to %PROCEDURE statements, it is assumed that references are to be to user-defined preprocessor procedures of that name. In such cases the identifiers SUBSTR, LENGTH, and INDEX may be re-declared with the BUILTIN attribute when the built-in functions are to be used within a preprocessor procedure.

The built-in functions behave in the same way as user preprocessor functions when encountered in source text or in preprocessor statements.

The first argument of SUBSTR, LENGTH and INDEX is, if necessary, converted to character; the second and third arguments of SUBSTR are, if necessary, converted to decimal; and the second argument of INDEX is, if necessary, converted to character. The returned value is CHARACTER for SUBSTR, and FIXED for LENGTH or INDEX.

Preprocessor DO-group

The preprocessor DO-group can provide iterative execution of the preprocessor statements contained within the group. The format of the preprocessor DO-group is as follows:

```
%[label:]... DO[ i=m1[TO m2[BY m3]]
                    [BY m3[TO m2]]];
                .
                .
                .
%[label:]... END[label];
```

In the above format, i must be a preprocessor variable and m1, m2, and m3 must be preprocessor expressions. The label that can follow the keyword END must be one of the labels preceding the keyword DO. Preprocessor DO-groups may be nested and multiple closure is allowed.

Control cannot be transferred into a preprocessor DO-group specifying iteration, except by way of a return from a preprocessor procedure invoked from within the group.

Both preprocessor statements and text other than preprocessor statements can appear within a preprocessor DO-group. However, only the preprocessor statements are executed; nonpreprocessor statements are scanned but only for possible replacement activity.

Noniterative preprocessor DO-groups are useful as THEN or ELSE clauses of %IF statements.

The expansion of a preprocessor DO-group is similar to that shown under the nonpreprocessor DO statement section J, "Statements".

The example below results in the same expansion generated for the example of preprocessor loop expansion in the section "Rescanning and Replacement" in this chapter:

```
%DECLARE I FIXED;
%DO I=1 TO 10;
Z(I)=X(I)+Y(I);
%END;
%DEACTIVATE I;
```

The second example under "Returned Value" shows how preprocessor DO-groups can be used within a preprocessor procedure (percent symbols must be omitted, of course).

INCLUSION OF EXTERNAL TEXT

Strings of external text can be incorporated into the source program at the preprocessor stage by use of the %INCLUDE statement. Such text, once incorporated, is called included text and may consist of both preprocessor and nonpreprocessor statements. Hence, included text can contribute to the preprocessed text being formed.

If the included text contains any preprocessor declarations, the scope of the names declared as preprocessor variables is all the included text and any text which follows the included text, except preprocessor procedures in which the name is redeclared.

The general format and the rules governing the use of the %INCLUDE statement are presented in section J, "Statements".

The text specified by a %INCLUDE statement is incorporated into the source program immediately after the point at which the statement is executed. The scan therefore continues with the first character in the included text. All preprocessor statements in this text are executed and replacements are made where required.

Preprocessor procedures whose declarations appear in external text can be invoked only after that external text becomes included text. The result of a preprocessor procedure reference encountered before that procedure has been incorporated into the source program is undefined.

Assume that PAYRL is a member of the data set SYSLIB and contains the following structure declaration:

```
DECLARE 1 PAYROLL,
   2 NAME,
     3 LAST CHARACTER (30) VARYING,
     3 FIRST CHARACTER (15) VARYING,
     3 MIDDLE CHARACTER (3) VARYING,
   2 MAN_NO,
     3 REGLR FIXED DECIMAL (8,2),
     3 OVERTIM FIXED DECIMAL (8,2),
   2 RATE,
     3 REGLAR FIXED DECIMAL (8,2),
     3 OVERTIME FIXED DECIMAL (8,2);
```

Then the following sequence of preprocessor statements:

```
%DECLARE PAYROLL CHARACTER;
%PAYROLL='CUM_PAY';
%INCLUDE PAYRL;
%DEACTIVATE PAYROLL;
%INCLUDE PAYRL;
```

will generate two identical structure declarations into the preprocessed text, the only difference being their names, CUM_PAY and PAYROLL. Execution of the first %INCLUDE statement causes the text in PAYRL to be incorporated into the source program. When the scan encounters the identifier PAYROLL in this included text, it replaces it by the current value of the active preprocessor variable PAYROLL, namely, CUM_PAY. Further scanning of the included text results in no additional replacements. The scan then encounters the %DEACTIVATE statement. Execution of this statement deactivates the preprocessor variable PAYROLL and makes the identifier ineligible for replacement. When the second %INCLUDE statement is executed, the text in PAYRL once again is incorporated into the source program. This time, however, scanning of the included text results in no replacements whatsoever, because none of the identifiers in the included text are active. Thus, two structure declarations, differing in name only, are inserted into preprocessed text.

Preprocessor Statements

This section lists those statements that can be used at the preprocessor stage and briefly discusses those preprocessor statements that have not yet been explained in this chapter. All of the preprocessor statements, their formats, and the rules governing their use are described in the sub-section "Preprocessor Statements" in section J, "Statements".

But first, some unrelated comments pertaining to preprocessor statements in general should be made:

1. Some keywords appearing in preprocessor statements can be abbreviated as shown in section C, "Keywords and Abbreviations".

2. Comments can appear within preprocessor statements wherever blanks can appear; however, such comments are never inserted into preprocessed text.

3. All preprocessor statements can be labeled. Such labels must appear immediately following the % (only blanks can intervene). All labels must be unsubscripted statement label constants. (Labels on %DECLARE statements are ignored.)

The functions performed by the following preprocessor statements have already been discussed in this chapter:

```
%ACTIVATE
%DEACTIVATE
%DECLARE
%DO
%END
%INCLUDE
%PROCEDURE
RETURN
```

Note that the preprocessor RETURN statement cannot have a leading % because it can be used only within a preprocessor procedure, and all percent symbols must be omitted therein.

Four other statements can be executed at the preprocessor stage:

```
%assignment
%GO TO
%IF
%null
```

The preprocessor assignment statement is used to evaluate preprocessor expressions and to assign the result to a preprocessor variable. All of the examples shown in this section make use of this statement.

The % GO TO statement causes the preprocessor to interrupt its sequential scanning and continue it elsewhere in the source program, specifically at the label specified in the % GO TO. Thus, it can be useful for rescanning or avoiding text.

The % IF statement can be used to control the sequence of the scan according to the value of a preprocessor expression. It must have a THEN clause and it can have an ELSE clause. Each clause, as well as

each preprocessor statement within the clause, must be preceded by a %. Nesting of %IF statements is allowed and must follow the same rules that apply for the nesting of nonpreprocessor IF statements.

The preprocessor null statement is the same as a nonpreprocessor null statement (except for the %). It can be used to provide transfer targets for %GO TO statements or it can be used in nested %IF statements to balance the %ELSE clauses. For example, %ELSE%; is a null ELSE clause.

Listing Control Statements

There are three statements that give the programmer control over the layout of his printed listing:

```
%SKIP[(n)];
%PAGE;
%CONTROL(FORMAT|NOFORMAT);
```

%SKIP specifies that a number of lines in the listing are to be skipped, and %PAGE specifies that the listing is to be continued on the following page. %CONTROL activates and de-activates the FORMAT option of the checkout compiler.

Although the statements have the initial % sign, they do not necessitate the use of the preprocessor. If the preprocessor is used, however, the %PAGE and %SKIP statements are applied to both the insource listing (the input to the preprocessor) and the source listing (the preprocessed text). The %CONTROL statement applies only to the formatted listing produced by the checkout compiler.

Application of the statement %SKIP(n); to either the insource listing or the source listing cause n blank lines to be inserted before the next line in the listing. The statement %SKIP; is equivalent to %SKIP(1);.

The statement %PAGE; causes the next line of text in the listing to be printed on the first line of the next page. There are no options that can be specified in a %PAGE statement.

The %CONTROL statement is executed with one of the options FORMAT and NOFORMAT. When FORMAT is specified as a compiler option, a %CONTROL(NOFORMAT) statement suspends the compiler option's formatting action. A subsequent %CONTROL(FORMAT) statement restores the formatting action. The %CONTROL statement has no effect when the FORMAT compiler option is not specified.

After being put into effect, a %SKIP or %PAGE statement is not printed by the preprocessor and is deleted from the text by the compiler; it does not appear in the formatted listing. %CONTROL statements do not appear in formatted listings.

To cause formatting of the listing, the listing control statement must be on a line of text with no other statements.

When the preprocessor is used, a %SKIP, %PAGE, or %CONTROL statement with text other than comments on the same line is moved by the preprocessor onto a line of its own, so that it is put into effect when the compiler listing is printed. Hence the preprocessor listing will be exactly as coded, but the source listing will have line or page skips in accordance with any %PAGE and %SKIP statements, and the FORMAT compiler option will be put in effect or not in accordance with any %CONTROL statements.

When the preprocessor is used, a keyword or other identifier split accross the end of a line that contains a %PAGE or %SKIP has its two parts concatenated to form a complete identifier in the source listing. If the deletion of the %SKIP or %PAGE statement provides sufficient space, the complete word appears on the same line as the first part; otherwise it is used to start a new line.

If n in a %SKIP(n); statement is greater than the number of lines remaining on the page, the equivalent of a %PAGE; statement is executed in its place.

The effects of the %SKIP and %PAGE statements in various situations are shown in figure 16.1.

Programmer's Code	Insource Listing or Source Listing without Preprocessor	Source Listing after Preprocessing
A=B; %SKIP(2); C=D;	A=B; C=D;	A=B; C=D;
X=2; %SKIP(1); Y=0;	X=2; %SKIP(1); Y=0;	X=2; Y=0;
B1='1'B; %SKIP;B2='0'B;	B1='1'B; %SKIP;B2='0'B;	B1='1'B; B2='0'B;
P=0;...%SKIP(1);SIG NAL CONVERSION; (G of SIGNAL in last posn. in line)	P=0;...%SKIP(1);SIG NAL CONVERSION;	P=0;... SIGNAL CONVERSION;
RES=SQRT(X); %PAGE; DCL Z FLOAT;	DCL Z FLOAT; at start of new page	DCL Z FLOAT; at start of new page
OPEN FILE (F1); %PAGE; PUT (REC_1);	No skip to new page	PUT (REC_1); at start of new page
END LOOP_3B;%PAGE;A LLOCATE A_3; (A of ALLOCATE in last posn. in line)	No skip to new page	ALLOCATE A_3; at start of new page

Figure 16.1. Effects of %PAGE and %SKIP

Chapter 17: Multitasking

The use of a computing system to execute a number of operations concurrently is broadly termed multiprogramming. The PL/I programmer can make use of the multiprogramming capability of the system by means of the multitasking facilities described in this chapter.

Introduction

A PL/I program is a set of one or more procedures, each of which consists of one or more blocks of PL/I statements. The execution of these procedures constitutes one or more tasks, each of which can be identified by a different task name. A task is dynamic; it exists only while the procedure is being executed. This distinction between the procedure and its execution is essential to the discussion of multitasking. A procedure could be executed several times in different tasks.

When the multitasking facilities are not used, the execution of a program comprising one or more procedures constitutes a single task, with a single flow of control; when a procedure invokes another procedure, control is passed to the invoked procedure, and execution of the invoking procedure is suspended until the invoked procedure passes control back to it. This serial type of operation is said to be synchronous; when the programmer is concerned only with synchronous operations, the distinction between program and task is relatively unimportant.

With multitasking, the invoking procedure does not relinquish control to the invoked procedure. Instead, an additional flow of control is established, so that both procedures can be executed (in effect) concurrently. This process is known as attaching a task. The attached task is a subtask of the attaching task. Any task can attach a number of subtasks. The task that has control at the outset is called the major task. This parallel type of operation is said to be asynchronous.

The diagram shown in figure 17.1 illustrates the difference between synchronous and asynchronous operations. The arrowed lines represent the control flows. Procedures A and B are executed synchronously; C and D are executed asynchronously.

When several procedures are executed as asychronous tasks, individual statements are not necessarily executed simultaneously by different tasks; whether this occurs depends on the state and resources of the system. Hence, at any given time, it may be necessary for the system to select its next action from a number of different tasks. Each task has a priority value associated with it, which governs this selection process. The programmer can control the priority of the task, within limits, if he wishes to do so; otherwise, the priority value is set automatically.

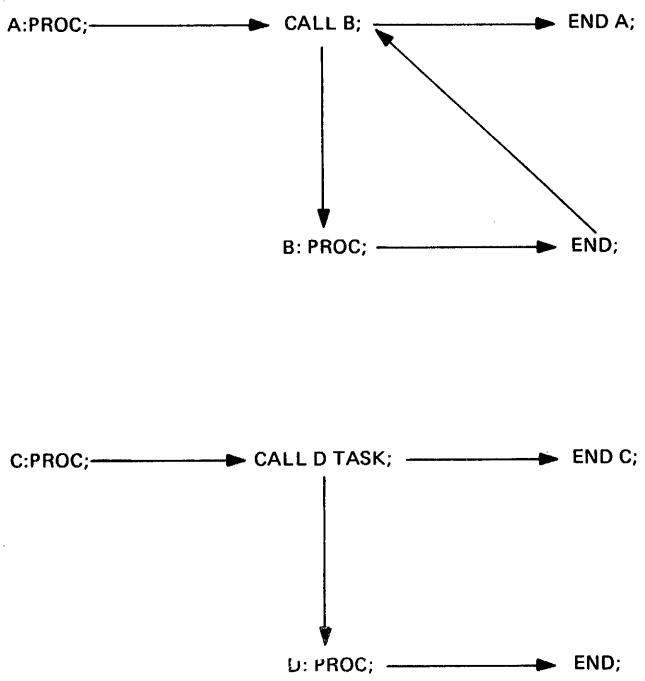


Figure 17.1. Synchronous and asynchronous operation

A task can have control of either the CPU or the systems's I/O resources. I/O may be performed in one task while CPU operations are being carried out in another, and there may, at the same time, be other tasks waiting for one or other of the resources. Operation of the CPU can be interrupted if a task of higher priority than the current one requires CPU facilities. Interruption can occur, for instance, if a higher-priority task completes an I/O operation or if the current task attaches a subtask of higher

priority. An I/O operation is never interrupted; I/O resources can only be re-allocated after completion of an I/O operation. When an I/O operation is completed, the system searches amongst all the active tasks that require the I/O resources to find the one with the highest priority.

The checkout and optimizing compilers implement the multitasking features of PL/I in different ways. The optimizing compiler utilizes the operating system tasking facilities, whereas the checkout compiler maintains full control of all processing, whether synchronous or asynchronous. The results produced by a multitasking program will, however, be the same under both compilers.

It may be that one task is to run independently of other concurrent tasks for some time, but then become dependent on some other task (for example, one task may require the result of another task before it can be completed). To allow for this, provision has been made for one task to await the completion of an operation at any stage of another task before carrying on. This process is known as task synchronization. Information about the state of an operation can be held by an event variable, to which an event name refers. By specifying an event name in a WAIT statement, the programmer can cause the task to wait for completion of the associated operation before proceeding.

The programmer can apply the EVENT option to tasks and certain input/output operations, in which case the value of the event variable is set automatically as a result of the operation concerned; or he can set the value explicitly.

The EVENT option allows an input/output operation to proceed asynchronously with the task that initiated it; at any time subsequent to the initiation of the input/output operation, the task can await its completion. For example, a task can display a message to the operator and, instead of waiting for a reply, can immediately proceed, pausing later to deal with the reply.

In general, the rules associated with the synchronous invocation of procedures apply equally to the asynchronous attachment of tasks. For example, on-units established prior to attachment of a subtask are inherited by the subtask, just as if the initial block of the subtask had been synchronously invoked. However, asynchronous operation introduces some extra considerations, such as the fact that a number of concurrent tasks can independently refer to one variable. This necessitates some extra rules, which are described in this chapter.

Multitasking also requires some extra rules and provisions for input/output. For example, without special provision, there would be nothing to prevent one task from operating on a record in a DIRECT UPDATE file while another task was operating on the same record; to cope with this, the EXCLUSIVE attribute is provided. The protection of records on EXCLUSIVE files is described in chapter 12, "Record-Oriented Transmission".

Tasks can be terminated in a number of different ways. Normal termination occurs when control for the task reaches a RETURN or END statement for the procedure attached as a task. The EXIT statement specifies abnormal termination of the task and its subtasks, while the STOP statement specifies abnormal termination of the major task (even if STOP is executed in a subtask).

Multitasking may allow the central processing unit and input/output channels to be used more efficiently, by reducing the amount of waiting time. It does not necessarily follow that an asynchronous program will be more efficient than an equivalent synchronous program (although it may be easier to write). It depends on the amount of overlap possible between operations with varying amounts of input/output; if the overlap is slight, multitasking could be the less efficient method, because of the increased system overheads.

Specifying Tasking and Reentrability

If multitasking is required, there is no need to specify the TASK keyword in the OPTIONS option of the PROCEDURE statement for the main procedure. Under the optimizing compiler, the multitasking modules of the PL/I library are made available by means of the SYSLIB DD job control statement; and under the checkout compiler, the multitasking facilities are always available. It is not an error to specify TASK in the PROCEDURE statement; if present, the keyword is ignored. This allows programs written for the PL/I(F) compiler to be compiled without error messages being generated.

Under the optimizing compiler, it is, however, necessary to specify the REENTRANT option if the procedure could possibly be attached as more than one task, to be executed concurrently. The code generated by the compiler might otherwise not be reentrant.

When REENTRANT is specified, the compiler will generate code that is reentrable as far as machine instructions and compiler-created storage is concerned. However the programmer must ensure that the logic of his PL/I source code is such that the program remains reentrable. In particular, he must not overwrite static storage.

Creation of Tasks

The programmer specifies the creation of an individual task by using one or more of the multitasking options with a CALL statement. Once a procedure has been activated by execution of such a CALL statement, all blocks synchronously activated as a result of its execution become part of the created task, and all tasks attached as a result of its execution become subtasks of the created task. The created task itself is a subtask of the task executing the CALL statement. All programmer-created tasks are subtasks of the major task.

Note: A task can be attached by a procedure entered as a result of a function reference in a PUT statement for the standard file SYSPRINT.

CALL STATEMENT

The CALL statement for asynchronous operation has the same form as that for synchronous operation, except for the addition of one (or any combination) of the multitasking options, TASK, EVENT, or PRIORITY. These options, in addition to their individual meanings (listed below), all specify that the invoked procedure is to be executed concurrently with the invoking procedure.

The CALL statement for asynchronous operation can specify arguments to be passed to the invoked procedure, just as it could if the operation were to be synchronous.

TASK Option

TASK option has the following format:

```
TASK [(element-task-name)]
```

The task name can be subscripted and/or qualified. Without the task name, the option merely specifies asynchronous operation. If the task is to have a name, the option must appear complete with the task name, which is thus contextually declared to have the TASK attribute, unless an explicit declaration exists. This is the only way in which a task can acquire a name. (Explicit declaration of a task variable does not associate the task name with any task.) The name can be used to control the priority of the task at some other point, by means of the PRIORITY pseudovariable and built-in function. The task name has no other use to the PL/I programmer.

EVENT Option

The EVENT option has the following format:

```
EVENT (element-event-name)
```

The event name can be subscripted and/or qualified. When this option is used, the event name is contextually declared to have the EVENT attribute (unless an explicit declaration exists) and is associated with the completion of the task created by the CALL statement. Another task can then be made to wait for completion of this task by specifying the event name in a WAIT statement of the other task.

An event variable has two separate values: a completion value that indicates whether or not the event is complete, and a status value that indicates whether the event has been abnormally completed. The completion value is a single bit, and the status value is a fixed binary number of precision (15,0). When the CALL statement is executed, the completion value of the event variable is set to '0'B (for "incomplete") and the status value to zero (for "not abnormally completed"). On termination of the created task, the completion value is set to '1'B, and, in the case of abnormal termination, the status value is set to 1 (if it is still zero).

The EVENT option can also be specified on the READ, WRITE, REWRITE, and DELETE statements, and on the DISPLAY statement with the REPLY option (see chapter 8, "Input and Output"). In these cases, it allows other processing to continue while the input/output operation is being executed.

PRIORITY Option

When a number of tasks simultaneously require attention, a choice has to be made Under the optimizing compiler, this choice is made by the operating system, based on the relative importance of the various tasks: a task that has a higher priority value than the others will receive attention first. Note that tasks, other than those executing the user's program and those in a wait state, may require attention from the system, and may have a higher priority than any of the user's tasks. Under the checkout compiler, the choice is made by the compiler, but when processing in one task is interrupted, the compiler always gives control to the task within the same program that has the highest priority.

The PRIORITY option has the following format:

```
PRIORITY (expression)
```

If this option appears in the CALL statement, the expression is evaluated to a binary integer m, of precision (n,0), where n is implementation-defined (15 for this implementation). The priority of the created task is then made m relative to the task executing the CALL statement. The lowest absolute priority possible is 0; the highest absolute priority possible is 234. (See "Priority of Tasks," in this chapter)

If the option does not appear, the priority of the attached task is equated to that of the task variable named in the TASK option, if any, or else equated to the priority of the attaching task. If the programmer employs a task variable, he must specify a priority for the task (by means of either the PRIORITY pseudovariable or the PRIORITY option of the CALL statement), otherwise the priority will be undefined.

Examples

```
1.  CALL PROCA TASK(T1);

2.  CALL PROCA TASK(T2) EVENT(ET2);

3.  CALL PROCA TASK(T3) EVENT(ET3)
    PRIORITY(-2);

4.  CALL PROCA PRIORITY(1);
```

The CALL statements in the above examples create four different tasks that execute one procedure, PROCA. In example 3, the subtask T3 has a lower priority than the attaching task, while in example 4, the unnamed subtask has a higher priority than the attaching task. (It is assumed that the priorities of the attached tasks would lie within the range 0 to the highest on the current job step).

PRIORITY OF TASKS

A priority specified in a PL/I source program is a relative value; the actual value depends on factors outside the source program.

Under OS, the priority associated with each job step is provided by the programmer, using the PRTY parameter in the JOB statement. This priority can have any number from 0 through 14: the higher the number, the higher the priority. The priority of the major task of the PL/I program when it is first entered is given by

```
Priority=(16*(job step priority))+10
```

This is the maximum priority for the program; that is, the highest priority that any task of the PL/I program can have. If an attempt is made to create a subtask with a higher priority than the maximum priority, the subtask will be executed at the maximum priority. Priority can be reduced to zero, but not below (a priority of less than zero, will be treated as zero priority). A task can change its own priority and that of any other task.

PRIORITY BUILT-IN FUNCTION AND PSEUDOVARIABLE

The PRIORITY pseudovariable provides a method of setting the priority of a named task relative to the current task. The effect of the statement

```
PRIORITY(T)=N;
```

is to set the priority of the task T equal to the priority of the current task plus the integral value of the expression N. If the priority thus calculated would be higher than the maximum priority or less than zero, the implementation ensures that the priority is set to the maximum, or zero, respectively.

The PRIORITY built-in function returns the relative priority of the named task (that is, the difference between the actual priority of the named task and the actual priority of the current task). Consider a task, T1, that attaches a subtask, T2, that

itself attaches a subtask, T3. If task T2 executes the sequence of statements

```
PRIORITY(T3)=3;
X=PRIORITY(T3);
```

X will not necessarily have the value 3. If, for example, task T2 had an actual priority of 24, and the maximum priority were 26, then execution of the first statement would result in task T3 having a priority of 26, not 27. Relative to task T2, task T3 would have a priority of 2; hence, after execution of the second statement, X would have a value of 2.

Between execution of the two statements, control could pass to task T1, which could change the priority of task T2, in which case the value of X would depend on the new priority. For example, given the same original priorities as before, task T3 would have a priority of 26 after execution of the first statement. If the priority of task T2 were now changed to 20 by its attaching task, T1, execution of the second statement would result in X having a value of 6.

A task name may have a priority assigned to it before it is associated with a procedure. It is not an error if such a name is never associated with a procedure. Thus, when a program is being developed, task names may be introduced into the program before the corresponding tasks are introduced.

Coordination and Synchronization of Tasks

The rules for scope of names apply to blocks in the same way whether or not they are invoked as, or by, subtasks; thus, data and files can be shared between asynchronously executing tasks. Hence, a high degree of cooperation is possible between tasks, but this necessitates some coordination. Certain additional rules are introduced to deal with sharing of data and files between tasks, and the WAIT statement is provided to allow task synchronization.

SHARING DATA BETWEEN TASKS

It is the programmer's reponsibility to ensure that two references to the same variable cannot be in effect at one instant if either reference would cause the value of the variable to be changed. He can do so by including an appropriate WAIT statement at a suitable point in his source program to force temporary synchronization of the tasks involved. Subject to this qualification, and the normal rules of scope, the following additional rules apply:

1. Static variables can be referred to in any task in which they are known.

2. Regardless of task boundaries, an automatic variable can be referred to in any block in which it is known, or to which it is passed as an argument, or in which it is referred to using an appropriate based variable. (Note that unless a dummy argument is created, the value of an argument can change at any time; the current value is used when any reference is made by any task.)

3. Controlled variables can be referred to in any task in which they are known. However, not all generations are known in each task. When a task is initiated, only the latest generation, if any, of each controlled variable known in the attaching task is known to the attached task. Both tasks may refer to this generation. Subsequent generations in the attached task are known only within the attached task; subsequent generations within the attaching task are known only within the attaching task. A task can free only its own allocations; an attempt to free allocations made by another task will have no effect. No generations of the controlled variable need exist at the time of attaching. It is not permissible for a task to free a controlled generation shared with a subtask if the subtask will later refer to the generation. When a task is terminated, all generations of controlled storage made within that task are freed.

4. Based variables allocated within an area are freed when the area is freed; unless contained in an area allocated in another task, all based variable allocations (including areas) are freed on termination of the task in which they were allocated.

5. Any generation of a variable of any storage class can be referred to in any task by means of an appropriate based variable reference. The programmer must ensure that the required variable has been allocated at the time of reference.

A task may allocate and free based variables in any area to which it can refer. A task can only free an allocation of a based variable not allocated in an

area if the based variable was allocated by that task.

SHARING FILES BETWEEN TASKS

A file is shared between a task and its subtask if the file is open at the time the subtask is attached. If a subtask shares a file with its attaching task, the subtask must not attempt to close the file. A subtask must not access a shared file while its attaching task is closing the file. The subtask may re-open a file closed by the attaching task, but it will not then be shared.

If a file name is known to a task and its subtask, and the associated file was not open when the subtask was attached, then the file is not shared; the effect is as if the task and its subtask were separate tasks to which the file name were known. That is, each task may separately open, access, and close the file. This type of operation is guaranteed only for files that are DIRECT in both tasks. Note that if one task opens a file, no other task can provide the corresponding close operation.

It is possible that two or more tasks may attempt to operate simultaneously on the same record in a data set opened for direct access; this can be synchronized by use of the EXCLUSIVE file attribute. This attribute is described in chapter 12, "Record-Oriented Transmission" and section D, "Attributes".

WAIT STATEMENT

The WAIT statement has the following format:

```
WAIT (event-name [,event-name]...)

   [(element-expression)];
```

Full details of the WAIT statement are given in section J, "Statements"; the following is a shorter description, providing background to the present discussion.

The WAIT statement specifies that the task executing it will go into a waiting state and execution of another task may be started or resumed until such time as the required events have been completed. An event is complete when its completion value is '1'B. Note that the WAIT statement specifies event names, not task names.

An event variable may be associated with an input/output operation that has been initiated by the task executing the WAIT statement. In this case, execution of the WAIT statement has the following effect:

1. If transmission ends (or has ended) normally, the event variable is set complete.

2. If the transmission ends (or has ended) requiring input/output conditions to be raised, the event variable is set abnormal (i.e., its status value is set to 1) and all the required conditions are raised. The event variable is set complete on return from the last on-unit.

If an abnormal return is made from an on-unit entered from the WAIT operation, the associated event variable is set complete, the WAIT operation is terminated, and control for the task passes to the point specified by the abnormal return.

Example

```
P1:  PROCEDURE;
       .
       .
       .
     CALL P2 EVENT(EP2);
     CALL P3 EVENT(EP3);
     WAIT (EP2);
     WAIT (EP3);
       .
       .
       .
     END P1;
```

In this example, the task executing P1 will proceed until it reaches the first WAIT statement; it will then await the completion of the task executing P2, and then the completion of the task executing P3, before continuing.

TESTING AND SETTING EVENT VARIABLES

The two values, completion and status, of an event variable can be retrieved by the built-in functions COMPLETION and STATUS.

The COMPLETION function returns the current completion value of the event variable named in the argument. This value is '0'B if the event is incomplete, or '1'B if the event is complete.

The STATUS function returns the current status value of the event variable named in the argument. This value is nonzero if the event variable has been set abnormal, or 0 if it is normal.

These two built-in functions can also be used as pseudovariables; thus, either of the two values of an event variable can be set independently. Alternatively, it is possible to assign the composite value of one event variable to another by specifying the event variables in an assignment statement. Thus, the setting of an event variable can be controlled by the programmer. By this means, he can mark the stages of a task; and, by using a WAIT statement in one task and an event assignment (from the COMPLETION built-in function or another event variable) in another task, he can synchronize any stage of one task with any stage of another.

The programmer should not attempt to assign a completion value to an event variable currently associated with an active task or with an input/output event. An input/output event is never complete until an associated WAIT statement is executed, the WAIT being in the same task as the EVENT option.

Other ways in which an event variable can be set have already been discussed (such as specifying the event name in the EVENT option of a CALL statement). Full details of event variables will be found under "EVENT Attribute" in section I, "Attributes". See also "EVENT Option" in chapter 12, "Record-Oriented Transmission".

Note:

When tasks are being synchronized, the following points should be kept in mind:

1. An input/output event must be waited for in the task that initiates the input/output operation. The event can also be waited for in any other task, but in this case this task will wait until the event has been set complete by a WAIT statement in the initiating task.

2. There is a very real danger that two tasks could interlock and enter a permanent wait state. The programmer must ensure that this cannot happen in a program. For example:

```
Task T1                 Task T2 (Event E2)

.                       COMPLETION(EV)='0'B;
.                       .
.                       .
WAIT (E2);              .
.                       WAIT (EV);
.                       .
.                       .
COMPLETION(EV)          .
 ='1'B;                 .
.                       RETURN;
```

Task T1 would wait for the completion of task T2, and task T2 would wait for task T1 to execute the completion pseudovariable to set the event variable EV complete.

Under the checkout compiler this condition is detected and causes termination of processing. Under the optimizing compiler, the program waits until canceled by the operating system or the operator.

DELAY STATEMENT

The DELAY statement (see section J, "Statements") allows a task to wait for a specified period, without reference to an event variable.

Termination of Tasks

A task is terminated by the occurrence of one of the following:

1. Control for the task reaches a RETURN or END statement for the initial procedure of the task.

2. Control for the task reaches an EXIT statement.

3. Control for the task, or for any other task, reaches a STOP statement.

4. The block in which the task was attached is terminated (either normally or abnormally).

5. The attaching task itself is terminated.

6. Standard system action for the ERROR condition or the action on normal return from an ERROR on-unit is carried out.

Termination is normal only if item (1) of the above list applies. In all other cases, termination is abnormal.

To avoid unintentional abnormal termination of a subtask, an attaching task should always wait for completion of the subtask in the same block that attached the subtask before the task itself is allowed to be terminated.

When a task is terminated, the following actions are performed:

1. All input/output events that have been initiated in the task and are not yet complete are set complete, and their status values (if still zero) are set to 1; the results of the input/output operations are not defined.

2. All files that have been opened during the task and have not yet been closed are closed; all input/output conditions are disabled while this action is taking place.

3. All allocations of controlled variables made by the task are freed.

4. All allocations of based variables made by the task are freed, except those it has allocated within an area allocated by another task (these are freed when the area is freed).

5. All active blocks (including all active subtasks) in the task are terminated.

6. If the EVENT option was specified when the task was attached, the completion value of the associated event variable is set to '1'B. If the status value is still zero, and termination is abnormal, the status value is set to

7. All records locked by the task are unlocked.

Note: If a task is terminated while it is assigning a value to a variable, the value of the variable is undefined after termination. Similarly, if a task is terminated while it is creating or updating an OUTPUT or UPDATE file, the effect on the associated data set is undefined after termination. It is the responsibility of the programmer to ensure that assignment and transmission are properly completed before termination of the task performing these operations.

Programming Example

This example shows an application of multitasking to a banking system. The program is divided into a batch section and a real-time section. Each section constitutes a subtask of the major task; each subtask has other subtasks attached to it that perform the various data processing routines necessary in each section. The use of several subtasks increases the program efficiency by permitting overlap between the input/output operations and the operations performed by the central processing unit.

The batch section of the program processes batches of cards that contain account information (such as cheques cashed, deposits made, or loan account details) and, after a certain number of transactions, produces a statement.

The real-time section of the program provides a means of communication between itself and the operator, using the DISPLAY statement with the REPLY option. This facility permits the user to issue commands to the program through the operator's console. These commands can:

1. Cause management or credit information, bank statements, or similar information to be made immediately available.

2. Initiate or terminate processing. Thus the user can initiate the processing of card batches, terminate a section of processing, terminate the entire program, or reply to a call for clarification of mispunched data.

The functions of the various tasks that make up the program, and their relationship to each other, are shown in figure 17.2. Suggested coding for the ONLINE and PROCESS procedures is given below. These procedures are internal to the BANKER procedure, as are all the procedures in the program in this case.

```
ONLINE:  PROCEDURE;
         DECLARE COMMAND CHARACTER(30) VARYING,
                 COMTYPE(8) CHARACTER(30) VARYING,
                 COUNT(8) FIXED BINARY INITIAL ((8)0),
                 ID CHARACTER (72) VARYING,
                 XL(8) LABEL,
                 ENDBEVT EVENT EXTERNAL;
         COMTYPE(1) = 'CREDIT';
         COMTYPE(2) = 'STATEMENT';
         COMTYPE(3) = 'INFORMATION';
         COMTYPE(4) = 'CALL BATCH';
         COMTYPE(5) = 'END BATCH';
         .
         .
         .
         COMTYPE(8) = 'END PROGRAM';

START:   DISPLAY ('NEXT COMMAND') REPLY (COMMAND);
         /*TASK IS IN WAITING STATE UNTIL REPLY IS RECEIVED*/
X:       DO I = 1 TO 8;
            IF COMMAND = COMTYPE (I)
              THEN GO TO XL(I);
         END;
         DISPLAY ('UNRECOGNIZABLE COMMAND, REPEAT')
            REPLY (COMMAND);
         GO TO X;

XL(1):   DISPLAY ('ACCOUNT ID') REPLY (ID);
         COUNT(1) = COUNT(1) + 1;
         CALL CREDIT (ID) PRIORITY (-1); /*ATTACH CREDIT TASK*/
         GO TO START;

XL(2):   .
         .
         .

XL(5):   COMPLETION (ENDBEVT) = '1'B;
         /*SETS EVENT COMPLETE IN BATCH. BATCH
         WILL TERMINATE WHEN ALL CARDS READ IN*/
         GO TO START;
         .
         .
         .
         END ONLINE;

PROCESS: PROCEDURE;
         DECLARE ANS CHARACTER (30) VARYING,
                 (READEVT, ENDEVT, TEVREAD,
                 TEVUPDT, TEVRED) EVENT EXTERNAL;

  WS:    WAIT (READEVT, ENDBEVT) (1);
         IF COMPLETION(READEVT)='1'B THEN GO TO READIN;
         WAIT (TEVREAD, TEVUPDT, TEVRED) (3);

  EXS:   EXIT;
         /*IF 'END BATCH' COMMAND WAIT FOR ASSOCIATED
         TASKS BEFORE BATCH IS TERMINATED*/

READIN:  COMPLETION (READEVT) = '0'B;
         CALL READER TASK (PR1) PRIORITY (-1) EVENT (TEVREAD);
         CALL UPDATE TASK (PR2) PRIORITY (-2) EVENT (TEVUPDT);
         CALL RED TASK (PR4) PRIORITY (-3) EVENT (TEVRED);
         WAIT (TEVREAD, TEVUPDT, TEVRED) (3);
         DISPLAY ('CARDS PROCESSED') REPLY (ANS);
         IF ANS = 'WAIT' THEN GO TO WS; /*WAIT FOR COMMAND*/
         IF ANS = 'READ' THEN GO TO READIN; /*PROCESS NEXT BATCH*/

END PROCESS;
```

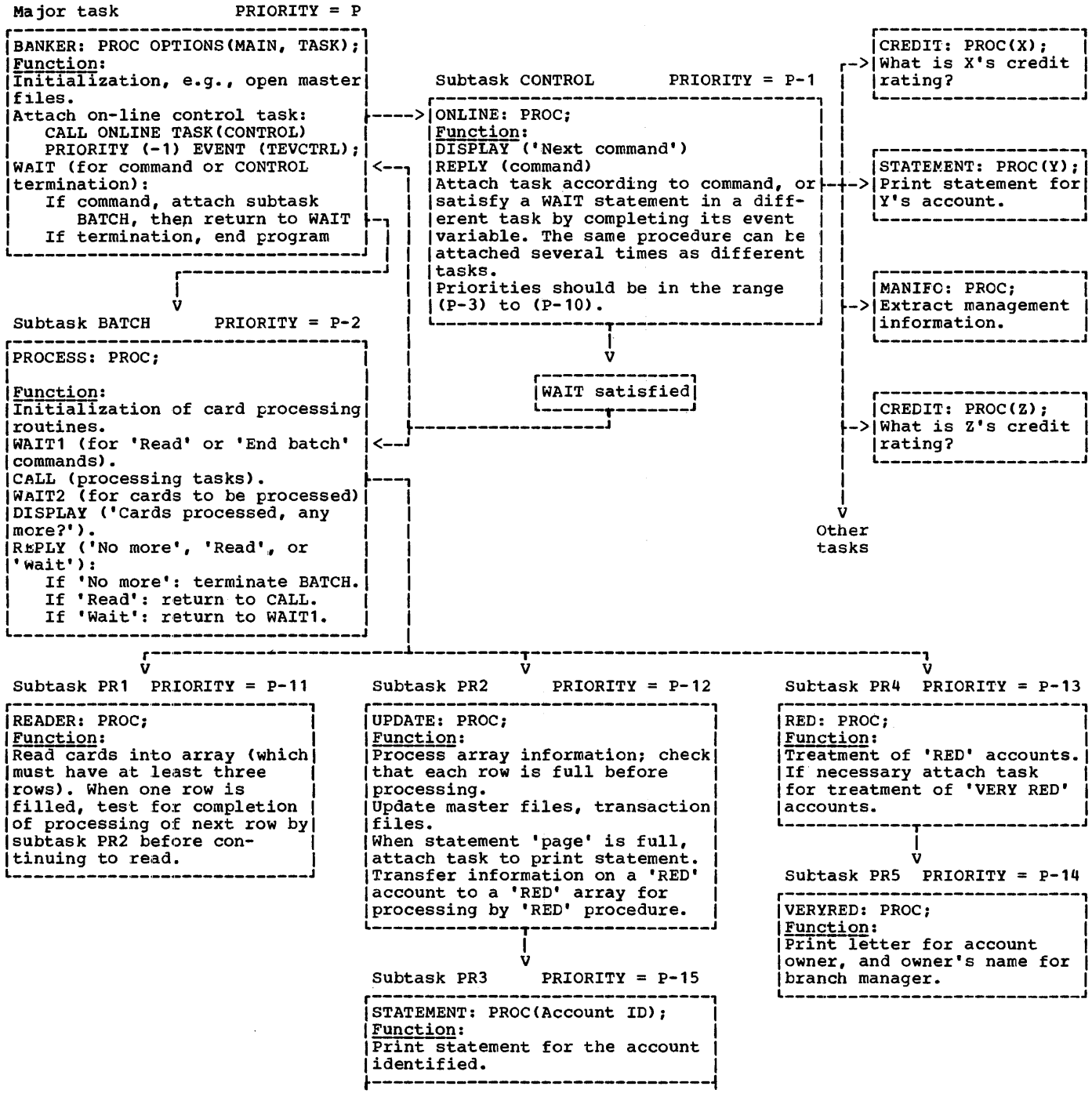


Figure 17.2. Flow diagram for programming example of multitasking

Chapter 18: Efficient Programming

This chapter contains three sections: the first provides a description of the optimization facilities of the optimizing compiler; the second gives suggestions for some coding practices to improve efficiency of programs for the optimizing compiler; the third lists some of the errors and pitfalls most likely to be encountered by programmers when first using PL/I.

OPTIMIZATION

The optimization facilities of PL/I are available only for programs processed by the PL/I optimizing compiler. The PL/I checkout compiler does not provide optimization of object programs; it implements the optimization language items by checking the syntax and then ignoring them.

The main purpose of optimization is to generate object programs which execute as fast as possible and which occupy as little space as possible during execution. In many cases this will involve generating efficient code for the statements written by the programmer; in other cases, however, the optimizing compiler may alter the sequence of statements or operations to improve the performance whilst producing the same result.

The following types of optimization are carried out by the optimizing compiler:

- Elimination of common expressions
- Transfer of invariant expressions out of loops
- Elimination of redundant expressions
- Simplification of expressions
- Initialization of arrays and structures
- In-line code for conversions
- In-line code for record I/O transmission statements
- Reduction of key conversion for REGIONAL data sets
- Matching format lists with data lists
- In-line code for string manipulation
- In-line code for many of the built-in functions
- Special-case code for DO statements
- Structure assignments
- Register and address optimization, including maintenance of values in registers for as long as possible and producing efficient address arithmetic based on optimal flow-paths
- Program branches kept as much as possible to the same base address
- Packing library routines into logical units to minimize space requirements
- Elimination of common constants and program control data to minimize space usage

Certain of the more important features are described further in the following sections.

COMMON EXPRESSIONS

The term "common expression" is used to describe an expression such as B+C in:

```
A = B+C
    .
    .
    .
D = B+C
```

in which the variables B and C are not reset between the two occurrences of the expressions. In a case like this it is necessary to evaluate the expression only once.

The technique of avoiding repeated evaluation of the same expression is called common expression elimination.

An important application of common expression elimination occurs in statements containing subscripted variables where the same subscript value is used for each variable. For example:

```
PAYROLL_TAX(MANNO) = PAY_CODE(MANNO) *
  WEEKPMNT(MANNO);
```

The value of the subscript expression MANNO is computed once only when the statement is

executed (the computation would involve the conversion of a value from decimal to binary if MANNO were declared a decimal variable).

Interrupt Handling for Programs with Common Expression Elimination

The order of most operations in each PL/I statement is dependent on the priority of the operators involved. However, the order of evaluation of those sub-expressions whose results form the operands of operators of lower priority, such as subscript expressions, locator qualifier expressions, and function references, is not defined beyond the rule that an operand must be fully evaluated before its value can be used in another operation. Therefore on-units associated with interrupts which occur during the evaluation of such sub-expressions can be entered in an unpredictable order. Consequently, an expression might have several possible values, according to the order of, and action taken by, the on-units that are entered. When a computational on-unit is entered:

1. The values of all variables set by the execution of previous statements are guaranteed to be the latest values assigned to the variables, and can be used by the on-unit. For instance the PUT DATA statement can be used to record the values of all variables on entry to an on-unit.

2. The value of any variable set in an on-unit resulting from a computational interrupt is guaranteed to be the latest value assigned to the variable, for any part of the program.

Where there is a possibility that variables might be modified as the result of a computational interrupt, either in the associated on-unit, or as the result of the execution of a branch from the on-unit, common expression elimination is inhibited. For example:

```
ON ZERODIVIDE B,C=1;
   .
   .
   .
X=A*B+B/C;
Y=A*B+D;
```

The compiler would normally attempt to eliminate the re-evaluation of the sub-expression A*B in the second assignment statement. However, in this example, if the ZERODIVIDE condition is raised during the evaluation of B/C the two values for A*B could be different. This optimization is inhibited to allow for this possibility.

Note that the above discussion applies only when the optimization option ORDER is specified or assumed. If the programmer does not require the guarantees described above, the optimization option REORDER can be specified. In this case, common expression elimination is not inhibited. The ORDER and REORDER options are discussed later in this chapter.

TRANSFER OF INVARIANT EXPRESSIONS OR STATEMENTS

An expression or statement occuring within a loop is said to be invariant if the compiler can detect that the expression value or statement action would be identical for each iteration of the loop.

An invariant expression or statement can be moved from within a loop to a point in the program outside the loop, so that it is executed once only, rather than for each iteration of the loop. For example:

```
DO I = 1 TO N;
   .
   .
   .
 J = 3;
   .
   .
   .
END;
```

The statement J=3 is invariant and can be moved outside the loop. It can be moved forwards or backwards, according to circumstances.

If the programmer wishes to take advantage of this type of optimization, he must specify the optimization option REORDER on a BEGIN or PROCEDURE block which contains the loop with reorderable statements or operations. If the option is not specified, the default option ORDER is assumed and the optimization is inhibited. The ORDER and REORDER options are discussed below.

ORDER AND REORDER OPTION

ORDER and REORDER are optimization options specified for a procedure or begin block in a PROCEDURE or BEGIN statement.

The standard default is ORDER, but REORDER is inherited by all contained

blocks unless they explicitly specify ORDER.

ORDER Option

The ORDER option should be specified for a procedure or begin block if the programmer requires that the most recently assigned values of variables that are modified in the block are guaranteed for use in on-units entered because of computational interrupts during the execution of statements and expressions in the block. In a block to which the ORDER option applies, common expressions may be eliminated by the compiler. If so, the occurrence of computational interrupts during execution of the block may be less than would occur if common expressions had not been eliminated. However, if an interrupt occurs during execution of an ORDER block, the values assigned to variables in statements which precede the interrupt are guaranteed to be the most recent values assigned when reference is made to them in the on-unit for the interrupt. Other forms of optimization are permitted in an ORDER block except for forward or backward move-out of any expression which can cause an interrupt. Since it would be necessary to disable all the possible conditions which might be encountered, the use of ORDER virtually suppresses any move-out of statements or expressions from loops.

REORDER Option

The REORDER option permits the compiler to generate optimized code to produce the result specified by the source program, when error-free execution takes place. Move-out is permitted for any invariant statements and expressions from inside a loop to a point in the source program either preceding or following such a loop. Thus the statement or expression is executed once only, either before or after the loop.

More efficient execution of loops can be achieved by maintaining in registers the values of variables which are subject to frequent modification during the execution of the loops. When error-free execution permits, values can be kept in registers, and considerable efficiency can be achieved by dispensing with time-consuming load-and-store operations to reset the values of variables in their storage locations. If the latest value of a variable is required after a loop has been executed, the value is assigned to the storage location of the variable when control passes out of the loop.

Register allocation can be more significantly optimized if REORDER is specified for the block. However, the values of variables that are reset in the block are not guaranteed to be the latest assigned values when a computational interrupt occurs, since the latest value of a variable may be present in a register but not in the storage location of the variable. Thus, any on-unit entered for a computational interrupt must not refer to variables set in the reorder block. However, use of the built-in functions ONSOURCE and ONCHAR is still valid in this context.

A program is in error if during execution there is a computational or system action interrupt in a REORDER block followed by the use of a variable whose value is not guaranteed.

Since these restrictions preclude the correction of erroneous data, except by using ONSOURCE and ONCHAR pseudovariables for a CONVERSION on-unit, the programmer must either depend on the standard system action, thereby terminating execution of the program, or use the on-unit to perform error recovery and to restart the program by obtaining fresh data for computation. The second approach should ensure that all valid data is processed, and that invalid data is noted, while still taking advantage of any possible optimization. For example:

```
ON OVERFLOW PUT DATA;
DO J = 1 TO M;
DO I = 1 TO N;
X(I,J) = Y(I) + Z(J) *L + SQRT(W);
P = I*J;
END;
END;
```

When the above statements appear in a reorder block, the source code compiled is interpreted as follows:

```
ON OVERFLOW PUT DATA;
TEMP1 = SQRT(W);
DO J = 1 TO M;
TEMP2 = Z(J) * L + TEMP1;
DO I = 1 TO N;
X(I,J) = Y(I) + TEMP2;
END;
END;
P = N*M;
```

TEMP1 and TEMP2 are temporary variables created to hold the values of expressions moved backwards out of the loops. The statement P = N * M; can be moved forwards out of the loops since the value of P can only be required after the last iteration

of the outer loop. If an overflow interrupt occurs, the values of the variables used in the loops cannot be guaranteed to be the most recent values assigned before the occurrence of the interrupt, since the current values may be held in registers, and not in the storage location to which the on-unit must refer.

ELIMINATION OF REDUNDANT EXPRESSION

A redundant expression is an expression that need not be evaluated in order to continue executing the program correctly.

The effect of this optimization is to make the use of logical expressions in IF statements more efficient than a series of nested IF statements. For example:

```
IF (A = D) | (C = D) THEN
   X = Y + Z;
```

is more efficient than:

```
IF A = D THEN X = Y+Z;
   ELSE IF C=D THEN X = Y + Z;
```

If A or C does equal D, the THEN clause in the first example is executed, without the expression ever being resolved to a single bit.

EXPRESSION SIMPLIFICATION

Expression simplification is the process of changing the form of source statement expressions without changing the intended effect so that they can be compiled into more efficient object code.

Two forms of expression simplification are carried out by the compiler. Both involve the use of arithmetic constants in operational expressions. The simplifications are as follows: expressions such as 3*B are transformed into B+B+B; and in subscript expressions, expressions such as I+2 are transformed into I*MULT + 2 *MULT where MULT is a constant multiplier. The 2*MULT is then used as an offset factor in the addressing calculations.

CODING SOURCE PROGRAMS FOR THE OPTIMIZING COMPILER

This section contains details of coding practices which should be observed or avoided in order to take advantage of the optimization facilities.

Common Expression Elimination

Common expression elimination is inhibited by:

1. The use in expressions of variables whose values can be reset in either an input-output or computational on-unit.

2. If a based variable is, at some point in the program, overlaid on top of a variable used in the common expression, then assigning a new value to the based variable in between the two occurences of the common expression, inhibits optimization.

   For instance, the common expression X+Z, in the following example, is not eliminated because the based variable A which, earlier in the program, is overlaid on the variable X, is assigned a value in between the two occurences of X+Z.

   ```
   DCL A BASED(P);
   P=ADDR(X);
       .
       .
   P=ADDR(Y);
       .
       .
   B=X+Z;
   P->A=2;
   C=X+Z;
   ```

3. When ORDER is specified, any variable which satisfies the following definition of an aliased-variable: an aliased variable is any variable whose value can be modified by references to identifiers other than its own identifier, such as variables with the DEFINED attribute, variables used as the base for defined variables, parameters, arguments to the ADDR built-in function, and based variables.

   Variables whose addresses are known to an external procedure by means of pointers that are either external or used as arguments are also assumed to be aliased variables.

   The effect of an aliased variable is not to prevent common expression elimination completely, but to inhibit it slightly. For all aliased variables the compiler builds a list of all the variables which could possibly reference the aliased

variable. The list is the same for each member of the list, and in a given program there may be many such lists.

When an expression containing an aliased variable is being checked for its use as a common expression, the possible flow paths along which related common expression could occur are searched for assignments, not only to the variable referenced in the expression, but also for all the members of the alias list to which that variable belongs. If the program contains an external pointer variable, it is assumed that this pointer could be set to all variables whose addresses are known to external procedures, i.e., all external variables, all arguments passed to external procedures, and all variables whose addresses could be assigned to the external pointer. Thus variables addressed by the external pointer, or by any other pointer which has a value assigned to it from the external pointer, are assumed to belong to the same alias list as the external variables, etc.

4. The form of an expression. If the expression B+C could be treated as a common expression, the compiler would not be able to detect it as a common expression in the following statement:

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

   The compiler processes the expression A+B+C from left to right. Consequently it only recognizes the expressions A+B and (A+B)+C. However, by coding the expression D=A+(B+C), the programmer can ensure that it is recognized, since the compiler must process the expression with the highest priority first.

5. The scope of a common expression. In order to determine the presence of common expressions, the program is analyzed and the existence of flow-units is determined. A flow-unit is a unit of compiled code representing all or part of a block that can only be entered at the first instruction and left at the last. Common expressions are recognized across individual flow units. However, if the program flow-paths between flow units are complex, the recognition of common expressions is inhibited across flow-units.

Common expression elimination is assisted by these points:

1. Variables in expressions should not be external or associated with external pointers, or arguments to ADDR built-in functions.

2. The source program should not contain external procedures, external label variables, or label constants known to external procedures. Where possible a label list should be supplied in declarations of label variables.

3. Variables in expressions should not be set or accessed in on-units if possible.

Transfer of Invariant Expressions

Transfer of invariant expressions out of loops is inhibited by:

1. ORDER specified for the block. However, transfer is not entirely prevented by the ORDER option. It is only inhibited for operations which can cause computational interrupts. Such operations do not include array subscript manipulation where the subscripts are represented by binary halfword integers; such subscripts cannot cause overflow unless they are uninitialized, in which case the program is in error anyway.

2. The use of variables whose values can be set or used by input or output statements.

3. The use in expressions of variables whose values can be set in input/output or computational on-units, or which are aliased-variables.

4. A complicated program flow, involving external procedures, external label variables and label constants, and the absence of a label list in a label variable declaration.

Transfer is assisted by:

1. Specifying REORDER for the block

2. Avoidance of points 2-4 above

Redundant Expression Elimination

Redundant expression elimination is inhibited or assisted by the same factors as for transfer of invariant expressions, described above.

Other Optimization Features

Optimized code can be generated for the following items:

1. For a do-group control variable except when its value can be modified either explicitly or by an on-unit during execution of a do-loop.

2. For do-loops that do not contain other do-loops, provided that, if the scope of the control variable extends beyond the block containing the do-loop, then it is given a definite value after the do-loop and before the end of the block.

3. For assignment of arrays or structures unless non-contiguous storage is used.

4. For array initialization where the same value is assigned to each element unless the array occupies non-contiguous storage.

5. For in-line conversions unless they involve complicated picture or character to arithmetic conversions.

6. For in-line code for the string built-in functions SUBSTR and INDEX unless the ON-conditions STRINGSIZE or STRINGRANGE are enabled.

7. For register allocation and addressing schemes unless the program flow is complicated by use of external procedures, external label variables, label constants known to external procedures, or unless label lists are not supplied in label variable declarations. Optimized register usage is also inhibited by the use of aliased variables and variables that are referenced or set in an on-unit.

Programming Techniques for the Optimizing Compiler

In PL/I there are often several different ways of producing a given effect. One of these ways will usually be more efficient from a particular point of view than another, depending largely on the method of implementation of the language features concerned. However, it should be realized at the outset that a primary cause of program inefficiency occurs at the problem definition stage, before any actual programming is done: PL/I cannot be used to full advantage unless the problem is defined in terms of PL/I facilities.

The purpose of this section is to help the programmer make the best use of the optimizing compiler. The first two parts are presented from two different viewpoints:

Improving the speed of compilation

Improving the speed of execution

The remainder of the section is of common interest, and deals with the use of storage; use of compile-time facilities; use of input/output facilities; and additional hints.

No techniques for improving the efficiency of code generated by the checkout compiler are given, since this compiler is designed primarly to give efficient use of programmer's time, rather than to produce optimum code.

IMPROVING SPEED OF COMPILATION

The NOOPTIMIZE optimizing compiler option requests that compilation be as fast as possible. In addition, the following measures are suggested.

1. Allocate as much storage to the compiler as possible. This minimizes the time consumed by the compiler spilling into auxiliary storage.

2. Keep the number of begin blocks and procedures to a minimum. Do not use BEGIN-END to effect statement grouping; this is more simply obtained by use of DO-groups.

3. Try to avoid using those compiler options that produce listings, e.g., object listing.

4. On re-runs, further slight increases in efficiency can be obtained by:

   a. removing all unreferenced labels and data;

   b. correcting all source errors, and, where possible, removing the causes of diagnostic messages including such messages as "FILE/STRING option missing in GET/PUT statement".

IMPROVING SPEED OF EXECUTION

The OPTIMIZE(TIME) optimizing compiler option requests that execution be as fast as possible. In addition, the following measures are suggested.

1. Avoid unnecessary program segmentation and block structure; all procedures, on-units and begin blocks need prologues and epilogues, the initialization and housekeeping for which carry an overhead. The use of GOTO or IF statements to control program logic, is more efficient than using the CALL statement, provided that the number of GOTO or IFs required is not excessive.

2. The following measures apply to labels.

   a. A GO TO statement is more efficient if it refers to a label within the same block rather than to a label outside the block.

   b. A GO TO statement that refers to a label variable is more efficient if the declaration of the label variable includes a complete list of label constants that the variable can represent.

3. Avoid extensive use of adjustable arrays and/or CONTROLLED storage.

4. Use constants wherever possible instead of expressions.

5. Exercise care in specifying precision. For example:

```
DCL A FIXED DEC(8,4),
    B FIXED DEC(10,2),
    C FIXED DEC(10,1);
    .
    .
    .
C=A+B;
```

   This requires almost twice as much code as it would if B had been declared (10,4), because the evaluation of A+B requires a scale factor of 4.

6. Use the PICTURE attribute only when necessary. For example, use FIXED DECIMAL(5,2) instead of PIC'999V99'. If a picture field is used in more than one arithmetic operation, convert it once and then use the new form in each operation. This holds for any conversion required more than once.

   If it is necessary to use data with the PICTURE attribute in arithmetic expressions, use pictures that will be handled in-line, as this considerably reduces execution time. Pictures with all 9s, a V and a non-drifting sign are particularly useful. For example:

```
'999'
'$99V99'
'S99'
'V999'
```

7. Internal switches and counters, and data involved in substantial computation or used for subscripts, should be declared BINARY; data required for output should be kept in DECIMAL form.

8. Keep data conversions to a minimum. Some possible methods follow:

   a. Use additional variables. For example, if a problem specifies that a character variable has to be regularly incremented by 1,

```
DCL CTLNO CHAR(8);
.
.
.
CTLNO = CTLNO+1;
```

   requires two conversions, while

```
DCL CTLNO CHAR(8),
   DCTLNO DEC FIXED;
.
.
.
DCTLNO=DCTLNO+1;
CTLNO=DCTLNO;
```

   requires only one conversion.

   b. Take special care to make structures match when it is intended to move data from one structure to another.

   c. Avoid mixed mode arithmetic, especially the use of character strings in arithmetic calculations.

9. The following measures apply to data aggregates:

   a. In general, array expressions are expanded into iterative DO-groups and structure expressions are expanded into a series of element expressions. However, for a simple assignment of the form: A = B;, where A and B are arrays or structures, B is copied to A all in one piece. A and B must not

have adjustable bounds, lengths, or sizes and the references must be to connected storage.

b. Avoid declaring redundant levels in a structure. For example:

```
DCL 1 A,
      2 B,
        3 C(10),
        3 D CHAR(20);
```

In this example B is redundant.

c. Avoid references to pseudovariables with aggregate arguments in stream-oriented input.

d. Avoid references to user functions or to array-handling built-in functions in aggregate expressions.

e. Avoid nested references to functions, particularly those with aggregate arguments.

f. Wherever possible, declare aggregates in the procedure in which they are used, instead of passing them as arguments. If an aggregate is required in two or more procedures, then it should be declared EXTERNAL, if the procedures are external, or else it should be declared in the outermost block in which it is used.

g. Declare subscript variables in the block in which they are used as FIXED BINARY (15,0).

10. The following measures apply to strings:

a. Bit strings should, if possible, be specified as multiples of eight bits. However, bit strings used as logical switches should be specified according to the number of switches required. In the following examples, (a) is preferable to (b), and (b) to (c):

Example 1:

Single Switches

```
(a)  DCL SW BIT(1) INIT('1'B);
       .
       .
       .
     IF SW THEN DO;
       .
       .
       .
(b)  DCL SW BIT(8) INIT('1'B);
       .
       .
       .
     IF SW THEN DO;
       .
       .
       .
(c)  DCL SW BIT(8) INIT ('1'B);
       .
       .
       .
     IF SW = '10000000'B THEN
     DO;
       .
       .
       .
```

Example 2:

Multiple Switches

```
(a)  DCL B BIT(8);
       .
       .
       .
     B = '11100000'B;
       .
       .
       .
     IF B = '11100000'B THEN DO
     ;
       .
       .
       .
(b)  DCL B BIT(3);
       .
       .
       .
     B = '111'B;
       .
       .
       .
     IF B = '111'B THEN DO;
       .
       .
       .
(c)  DCL (SW1,SW2,SW3) BIT(1);
       .
       .
       .
     SW1, SW2, SW3, = '1'B;
       .
       .
       .
     IF SW1&SW2&SW3 THEN DO;
       .
       .
       .
```

If bit-string data whose length is not a multiple of 8 is to be held in structures, such structures should be declared ALIGNED.

Note: The use of bit strings in a multitasking program can

occasionally cause incorrect results. When the program references the bit string, it may be necessary for a PL/I library routine to access adjacent storage, as well as the string itself. If another task accesses this adjacent storage at the same time, then the results may be unpredictable. The problem is less likely to arise with aligned bit strings than unaligned.

b. Note that concatenation operations on bit-strings are time-consuming.

c. Varying-length strings are generally less efficient than fixed-length strings.

d. Fixed-length strings are not efficient if their length is not known at compile time, as in the following example:

```
DCL A CHAR(N);
```

11. Avoid using the SIZE, SUBSCRIPTRANGE, STRINGRANGE and CHECK ON-conditions, except during debugging. Debugging aids should be removed from the program before running it as a production job.

12. Do not refer to the DATE built-in function more than once in a run; it is expensive. Instead, refer to the function once and save the value in a variable for subsequent use; e.g. instead of:

```
PAGEA= TITLEA||DATE;
PAGEB= TITLEB||DATE;
```

it is more efficient to write

```
DTE=DATE;
PAGEA=TITLEA||DTE;
PAGEB=TITLEB||DTE;
```

13. The following measures apply to input/output:

a. Allocate sufficient buffers to prevent the program becoming I/O bound.

b. Use blocked output records.

c. Open a number of files in a single OPEN statement.

d. In STREAM input/output, use long data lists instead of splitting up input/output statements.

e. Use edit-direct input/output in preference to list- or data-directed.

f. Consider the use of overlay defining to simplify transmission to or from a character string structure. For example:

```
DCL 1 IN,
      2 TYPE CHAR(2),
      2 REC,
        3 A CHAR(5),
        3 B CHAR(7),
        3 C CHAR(66);
GET EDIT(IN)
 (A(2),A(5),A(7),A(66));
```

In the above example, each format-item/data-field pair is matched separately, code being generated for each matching operation. It would be more efficient to define a character string on the structure and apply the GET statement to the string:

```
DCL STRNG CHAR(80) DEF IN;
.
.
.
GET EDIT (STRNG) (A(80));
```

g. If a file is declared DIRECT INDEXED, the ENVIRONMENT options INDEXAREA, NCWRITE, and ADDBUFF should be applied if possible.

h. When creating or accessing a CONSECUTIVE data set, use file and record variable declarations that cause in-line code to be generated, if possible. Details of the declarations are given in chapter 12, "Record-Oriented Transmission".

i. Conversion of source keys for REGIONAL data sets can be avoided if the following special cases are observed.

(1) For REGIONAL(1): When the source key is a fixed binary element variable or constant with precision in the range (12,0) to (23,0).

(2) For REGIONAL(2) and (3): When the source key is of the form (character-string-expression||r), where r is a fixed binary element variable or constant with precision in the range (12,0) to (23,0).

j. Direct update of an INDEXED data set is slowed down if an I/O

operation on the same file intervenes between a READ and a REWRITE for the same key. This can cause the REWRITE statement to issue an extra READ.

k. When creating or accessing a data set having fixed-length records, use standard formatting, that is, specify FS or FBS record format, whenever possible.

(Input/Output is also discussed under "Use of Input/Output Facilities" later in this chapter.)

14. The following measures apply to interlanguage communication:

a. Where possible, ensure that PL/I aggregate arguments will be mapped the same as those for COBOL or FORTRAN.

b. The compiler cannot always detect when a structure in PL/I and COBOL will map identically. Each element in the base structure has an alignment requirement for example, a CHAR(4) item can be aligned on any byte in main storage, whereas a FIXED BIN(31) item must be fullword aligned. The compiler creates a dummy argument for the structure whenever the first base element has a less stringent alignment requirement than any other base element. This rule is applied independently to each minor structure at level 2. The NOMAP option should be specified if the actual lengths of items do not require padding bytes to be inserted. For example:

```
DCL 1 S, 2 X CHAR(4), 2 Y FIXED
BIN(31);
```

The compiler will create a dummy argument for this structure because Y has a greater alignment stringency than X. However, the structure will map identically in PL/I and COBOL because no padding is required.

c. When arguments do map differently, use the NOMAPIN option to avoid unnecessary initialization of dummy arguments, and use the NOMAPOUT option to avoid unnecessary assignment from dummy arguments if the final value is not required (e.g., if the value is unchanged).

d. Avoid multiple initialization of the PL/I environment by ensuring:

(1) that the main procedure is PL/I, or

(2) that a PL/I procedure is called from the main routine, or

(3) that the structure of the program is such that the PI/I environment is not destroyed between calls to PL/I procedures.

IN-LINE OPERATIONS

Many operations are handled in-line. It will repay the user, therefore, to recognise which operations are performed in-line and which require a library call, and to arrange his program to use the former wherever possible. The majority of these in-line operations are concerned with data conversion and string handling.

Data Conversion

The data conversions performed in-line are shown in figure 18.1. Whether a particular conversion is done in-line or not can depend on whether optimization for minimum execution time has been requested. The column 'Optimization' has the entry 'none' if the level is immaterial, and 'time' if execution-time optimization is necessary. A conversion outside the range or condition given, or marked 'Not done' is performed by a library call.

Not all the picture characters available may be used in a picture involved in an in-line arithmetic conversion. The only ones permitted are:

V and 9

Drifting or non-drifting characters $ S + -

Zero suppression characters Z *

Punctuation characters , . / B

Conversion		Comments and Conditions	Optimization	
Source	Target		SIZE Disabled	SIZE Enabled
FIXED BINARY	FIXED BINARY	-	none	none
	FIXED DECIMAL	If either scale factor = 0 and the other scale factor ≤ 0, then the optimization can be 'none'	time	time
	FLOAT	If source scale factor = 0, then the optimization can be 'none' (whether SIZE is enabled or not)	time	time
	Bit string	String must be fixed-length, ALIGNED, and with length ≤2048	none	Not done
	Character string or Picture	Source scale factor must be ≥ 0 String must be fixed-length with length ≤256 Picture types 1, 2 or 3	time	Not done
FIXED DECIMAL	FIXED BINARY	If source and target scales have the same sign and are non-zero, then the optimization (SIZE disabled) must be 'time'.	none	time
	FIXED DECIMAL	-	none	none
	FLOAT	Source precision must be <10	time	time
	Bit String	Source scale factor must be zero String must be fixed-length, ALIGNED, and with length ≤2048	none	Not done
	Character string	Source scale factor must be ≥ 0 String must be fixed-length and length ≤256	time	time
	Picture	Picture types 1, 2 and 3 For picture types 1 and 2 with no sign, optimization can be 'none'	time	Not done
FLOAT	FIXED BINARY	-	time	Not done
	FIXED DECIMAL	Target precision must be ≤9	time	Not done
	FLOAT	Source and target may be single or double length	none	none
	Bit string	String must be fixed-length, ALIGNED, and with length ≤2048	time	Not done
Bit string	FIXED BINARY	Source string must be fixed-length, ALIGNED, and with length ≤2048	none	Not done
	FIXED DECIMAL and FLOAT	Source must be fixed-length, ALIGNED, and with length <32	time	Not done

Figure 18.1 (Part 1 of 2). Implicit data conversion performed in-line

Conversion		Comments and Conditions	Optimization	
Source	Target		SIZE Disabled	SIZE Enabled
Picture	Character string	String must be fixed-length with length ≤256	none	none
	Picture	Pictures must be identical	none	none
Picture type 1	FIXED BINARY	Source precision must be <10	time	Not done
	FIXED DECIMAL	If picture has a sign, then the optimization must be 'time'	none	Not done
	FLOAT	Source precision must be <10	time	Not done
	Picture	Picture types 1, 2 or 3	time	Not done
Label	Label	-	none	none
locator	locator	-	none	none

Figure 18.1 (Part 2 of 2). Implicit data conversions performed in-line

For in-line conversions, pictures with this subset of characters are divided into three types:

Picture type 1: Pictures of all 9s with (optionally) a V and a leading or trailing sign. For example:

'99V999', '99', 'S99V9', '99V+', '$999'

Picture type 2: Pictures with zero suppression characters and (optionally) punctuation characters and a sign character. Also, type 1 pictures with punctuation characters. For example:

'ZZZ', '**/**9', 'ZZ9V.99', '+ZZ.ZZZ', '$///99', '9.9'

Picture type 3: Pictures with drifting strings and (optionally) punctuation characters and a sign character. For example:

'$$$$', '-,--9', 'S/SS/S9', '+++9V.9','$$$9-'

Sometimes a picture conversion is not performed in-line even though the picture is one of the above types. This may be because:

1. The level of optimization is too low.
2. SIZE is enabled.
3. There is no overlap between the digit positions in the source and target. For example:

   DECIMAL (6,8) or DECIMAL (5, -3) to PIC '999V99' will not be performed

4. The picture may have certain characteristics that make it difficult to handle in-line. example:

   a. Punctuation between a drifting Z or a drifting * and the first 9 is not preceded by a V. For example:

      'ZZ.99'

   b. Drifting or zero expression characters to the right of the decimal point. For example:

      'ZZV.ZZ', '++V++'

String Handling

The string functions and operations performed in-line are shown in figures 18.2 and 18.3. It should be noted that even the string functions indicated as always being performed in-line may sometimes call a library routine. For example, if the expression in the BIT or CHAR functions requires an implicit conversion not handled in-line, the appropriate library routines will be called.

String Operation	Comments and Conditions		
	Source	Target	Comments
Assign	Non-adjustable, ALIGNED, fixed-length bit string	Non-adjustable, ALIGNED, bit string	No length restriction if OPTIMIZE(TIME) is specified; otherwise maximum length of 8192 bits
	Adjustable or VARYING, ALIGNED bit string	Non-adjustable, ALIGNED bit string ≤ 2048 bits	Only if OPTIMIZE(TIME) is specified
	Non-adjustable, UNALIGNED, fixed-length bit string that is an element of an AUTOMATIC, BASED, or STATIC structure with no adjustable bounds or extents	(same as source)	Only if OPTIMIZE(TIME) is specified. Maximum length = 57 bits
	Non-adjustable, fixed-length character string	Non-adjustable character string	
	Adjustable or VARYING character string	Non-adjustable character string of length ≤ 256	
'and', 'not', 'or'	As for bit string assignments, but no adjustable or varying-length operands are handled		
Compare	As for string assignment with the two comparands taking the roles of source and target, but no adjustable or varying-length operands are handled		
Concatenate	As for string assignments, but no adjustable or varying-length source strings are handled		
STRING function	Element variables and non-adjustable array and structure variables in connected storage.		
Notes: 1. the maximum lengths specified refer to the lengths of operations rather than operands. If the target is fixed-length, the operation length is the target length. If the target is VARYING, the operation length is the lesser of the operand lengths. 2. UNALIGNED bit strings that are parameters, defined variables, or part of aggregate variables are not handled.			

Figure 18.2. Conditions under which string operations are handled in-line

USE OF STORAGE

When reduction of storage space is more important than reduction of execution time, the following measures may be employed.

1. Wherever possible, fixed-point data for computation should be binary, rather than decimal.

2. If a file declared as INDEXED is to be used for DIRECT UPDATE but will not have records added to it, the use of the ENVIRONMENT option NOWRITE will save data management about 5000 bytes of storage.

3. Alignment Attributes: These allow the user to provide alignment for arithmetic and string data as follows:

```
ALIGNED:
     Arithmetic:
          FIXED DECIMAL: byte
          FLOAT(DOUBLE): doubleword
          FIXED BINARY(p,q)
          where p≤15:halfword
          Other:  word
     String:  byte

UNALIGNED:
     Arithmetic and character
       string:  byte
     Bit string:  bit
```

Thus the UNALIGNED attribute can be used to obtain denser packing of data, with the minimum of padding.

String Function	Comments and Conditions
BIT	Always
BOOL	The third argument must be a constant. The first two arguments must satisfy the conditions for 'and', 'or', and 'not' operations in figure 18.2.
CHAR	Always
HIGH	Always
INDEX	Second argument must be a non-adjustable character string <256 characters long
LENGTH	Always
LOW	Always
REPEAT	Second argument must be constant
SUBSTR	STRINGRANGE must be disabled
TRANSLATE	First argument must be fixed-length, second and third arguments must be constant
UNSPEC	Always
VERIFY	First argument must be fixed-length; if CHARACTER it must be ≤256 characters, if BIT it must be ALIGNED, ≤2048 bits. Second argument must be constant

Figure 18.3. Conditions under which the string functions are handled in-line

Area, event and task data are always word- or doubleword-aligned. They can never be unaligned.

In data aggregates, the explicitly declared alignment for the aggregate applies to each element in the aggregate. In structures, however, this alignment can be overridden by an alignment specified for a particular base element. For example

```
DCL 1 STR UNALIGNED,
        2 A,
        2 B ALIGNED,
        2 C;
```

Here A and C will be UNALIGNED and B will be ALIGNED.

Default attributes depend on the data type of the element concerned, both for data items and for data aggregates. These defaults are:

UNALIGNED	All string data and PICTURE items
ALIGNED	All arithmetic data i.e., BINARY DECIMAL FIXED FLOAT

For example:

```
DCL A BIT(4),
    I,
    (B CHAR(10), X) UNALIGNED,
    (C BIT(12), Y FIXED) ALIGNED;
```

Here A is UNALIGNED by default, I is ALIGNED by default, and B, C, X and Y, are as explicitly declared.

```
DCL (A1(80) CHAR(6), A2(80) BINARY)
    ALIGNED,
   B1 (3,3) BIT(2),
   C1 (3,3) CHAR(4),
   D1 (100) DECIMAL;
```

Here A1 and A2 are as explicitly declared, B1 and C1 are UNALIGNED by default, and D1 is ALIGNED by default.

```
DCL 1 A,
      2 B,
      2 C BIT(4) UNALIGNED,
      2 D ALIGNED,
        3 E BIT(2),
        3 F,
      2 G CHAR(10);
```

Here A is a major structure
B is FLOAT DECIMAL ALIGNED by default
C is explicitly UNALIGNED
D is a minor structure
E is BIT ALIGNED (inherited from D)

```
F is FLOAT DECIMAL ALIGNED by
  default(here ALIGNED is
  inherited from D, but it is
  also the default for F if D had
  not been declared ALIGNED)
G is CHAR UNALIGNED
```

The user must take care that the alignment attributes are correct when matching variables for:

a. Use of the DEFINED attribute

b. Arguments and associated parameters

4. A DO-group, like that in the following example, is expensive in terms of storage, although not in terms of time:

```
DO I = 1,2,6,9;
 .
 .
 .
END;
```

An alternative that saves storage is:

```
DCL VALUES(4) FIXED BIN STATIC
INIT(1,2,6,9);

DO J=1 TO 4;
I=VALUES(J);
 .
 .
 .
END;
```

5. After debugging, disable any normally-disabled conditions that were enabled for debugging purposes by removing the relevant prefixes, rather than by including NO-condition prefixes. For instance, disable the SIZE condition by removing the SIZE prefix, rather than by adding a NOSIZE prefix. The former method allows the compiler to eliminate code that checks for the condition, whereas the latter method necessitates the generation of extra code to prevent the checks being carried out.

Many PL/I facilities and data conversions are handled by PL/I resident and transient library subroutines. Some of these library subroutines require considerably more storage space than others and should be avoided if the size of the object program is to be kept to a minimum. They include subroutines that handle the following:

1. Conversions between character and arithmetic data.

2. Conversions involving numeric character data.

3. List- and data-direct input/output, except for character string variables associated with character data.

4. Edit-directed input/output, conversions between character and E- and F-format data.

5. Edit-direct output, numeric character to B-format conversion, character or numeric character to P-format conversion (except where pictures are identical).

6. CHECK prefix option.

7. ONCODE, ONLOC, and COMPLETION built-in functions.

8. WAIT statement.

USE OF INPUT/OUTPUT FACILITIES

The characteristics of a data set (record format, length, blocking factor, etc.) will significantly alter the time overhead of programs performing a large amount of input/output. In general, the blocking of records will save both time and space on a data set.

It is advantageous to open a number of files in a single OPEN statement, since separate opening activities (either explicit or implicit) will require the reloading of the OPEN modules from the transient library. It should also be considered that when a file is open, buffers or workspace and data management interface modules are occupying storage. Accordingly, if storage space is a prime consideration, the judicious use of the OPEN and CLOSE statements will help to control the available storage.

Of the three STREAM transmission techniques available, data-directed will generally be the most costly, both in time and space (symbol table entries exist at object time for each data variable transmitted). List-directed is available for free format input-output, but the greatest degree of control is available by using edit-directed, which is generally the most efficient technique, both in object time space and execution.

Repeated execution of a stream I/O operation is best achieved by means of a repetitive DO specification within the I/O statement, rather than by placing the statement within a DO-loop. The repetitive

specification obviates the need for repeated library calls to perform initial housekeeping operations such as obtaining buffer space.

RECORD transmission offers facilities for handling data aggregates as single data entities, rather than element by element. Certain advantages, accordingly, are available in terms of efficient access to data set records and efficient use of data set space, since the data is unconverted and unedited.

For other than spanned records, the use of locate mode I/O saves movement between buffers and other locations. With spanned records, this movement is necessary since the segments must be transferred between buffers and work areas. It is inefficient to use both locate and move modes on the same file, since considerable overheads are incurred in obtaining and releasing work areas.

When using the INDEXED data set organization, accessing records in an overflow area can slow processing considerably. It is recommended that INDEXED data sets are regularly recreated so that logically-deleted but physically-present records are purged from the data set and records are collected from the overflow areas into the prime data areas.

Consideration should be given especially to the choice between the BUFFERED and UNBUFFERED file attributes, available for sequential access. A BUFFERED file permits the object program to perform "anticipatory buffering", that is, overlap of device transmission and computing time. An UNBUFFERED file, though sometimes saving space required for buffers, does not permit such overlap, unless the EVENT option is used. (It should be noted, in any case, that an UNBUFFERED file will require hidden buffers if the record format is V, or if the data set is INDEXED, or REGIONAL(2) or (3)).

Input/Output Error Recovery

When I/O transmission errors are encountered, exhaustive recovery procedures are performed automatically by data management, so that, upon entry to a TRANSMIT on-unit, it is unnecessary and impossible to perform further transmission error correction procedures. Synchronization of transmission errors and entry to relevant on-units can only be guaranteed for input errors. An output operation error may be detected in a succeeding output operation, the particular one being dependent upon the blocking factor and the number of allocated buffers.

ADDITIONAL HINTS

Operating System and Job Control

External procedures, but not internal procedures, are treated as separate control sections.

PROCEDURE Statement

Be careful to differentiate between options of the OPTIONS option, and other options of PROCEDURE. For example:

```
P:PROC OPTIONS(MAIN, RECURSIVE);
```

is incorrect. It should be:

```
P:PROC OPTIONS(MAIN) RECURSIVE;
```

Declarations and Attributes

1. Do not rely too heavily on default attributes. Explicit declarations help to clarify the source program logic, and in some cases (for example, precision) reduce the chance of error.
2. Variables declared FIXED BINARY or FLOAT BINARY are automatically aligned on the proper word boundary (unless they are declared UNALIGNED), regardless of whether they are single or part of an aggregate. FIXED DECIMAL variables are stored in packed decimal format and the decimal instructions are used in operations involving them. FLOAT DECIMAL variables are stored in floating-point format; operations involving them are carried out using the floating-point instruction set.

Assignments and Initialization

1. High order zeros will be inserted if required on assignment to or initialization of an arithmetic variable:

```
DCL A FIXED DECIMAL (5,2) INIT (12);
/*A HAS VALUE 012.00*/

DCL B FIXED BINARY (15,0);
B=12;
/*B HAS VALUE 000000000001100B*/
```

2. Arrays may be initialized by assignment from an element expression:

```
DCL A(10);
A=0;
```

The element value will be assigned to each element of the array. Similarly, when an element expression is assigned to a structure, its value will be assigned to each element of the structure:

```
DCL 1B,
    2 C BIT(1),
    2 D CHAR(1),
    2 E CHAR(4);
B=0;
```

As a result of this assignment, the values of the various elements will be:

```
C  '0'B
D  'b'
E  'bbb0'
```

If it is required to assign zero to all arithmetic and bit elements and blanks to all string elements in a structure containing two or all three data types, the structure may have a null string assigned to it, thus:

```
B='';
```

The values of the elements would then be:

```
C  '0'B

D  'b'

E  'bbbb'
```

DO Loops

Iterations can step backwards, and the expression in the WHILE option can refer to the control variable, e.g.,

```
DO I=N+2*L BY -X WHILE (I>0);
END;
```

The control variable can be modified within the loop.

It is possible to transfer from within a DO loop to a label on the END statement for the group. This has the effect of incrementing the control variable without intermediate processing; control will not fall through. It is also possible to transfer out of an iterative DO group before the terminating value of the control variable is reached.

Functions

The arguments in a function reference can be modified by the function.

ON-conditions and on-units

Note the scope of condition prefixes:

```
     (SIZE):A:PROC;
            .
            .
            .
(NOSIZE):IF M>N THEN DO;
           J=E+F;
           END;
            .
            .
            .
           END A;
```

In the above example, SIZE is disabled only during the evaluation of the expression M>N; SIZE is enabled for the assignment J=E+F.

Comparison of Aggregates

After evaluation of the comparison in the IF statement, a single element must result, not an aggregate. Hence aggregates cannot be compared in an IF statement. However, the equality of two aggregates of string data can be tested by using the STRING

built-in function and pseudovariable. For example:

```
DECLARE (A,B) (10) CHAR(10);
      .
      .
      .
IF STRING(A) = STRING(B) THEN ...
```

Common Errors and Pitfalls

This is a list of the errors and pitfalls most likely to be encountered when writing a PL/I source program. Some of the items concern misunderstood or overlooked language rules, while others result from failure to observe the implementation conventions and restrictions.

The warnings apply particularly to programs compiled by the optimizing compiler. Although a source program is in error under the optimizing compiler will in general be in error under the checkout compiler as well, the checkout compiler detects many of the errors listed and takes appropriate action.

OPERATING SYSTEM AND JOB CONTROL

A STATIC variable in an overlay segment could be overwritten during an overlay operation unless it is contained in the root segment.

SOURCE PROGRAM AND GENERAL SYNTAX

1. Keypunch transcription errors may occur unless particular care is taken when writing the following characters:

   1 (numeral), I (letter), | (or),
   / (slash), ' (quotation mark);

   ¬ (not), 7 (seven),
   > (greater than);

   L (letter), < (less than).

   O (letter), 0 (zero);

   S (letter), 5 (five);

   Z (letter), 2 (two);

   _ (break character),
   - (minus sign);

2. Ensure that the source program is completely contained within the margins specified by the MARGINS option.

3. Inadvertent omission of certain symbols may give rise to errors that are difficult to trace. Common errors are: unbalanced quotation marks; unmatched parentheses; unmatched comment delimiters (e.g., /* punched instead of */ when closing a comment); and missing semicolons.

4. Reserved keyword operators in the 48-character set (e.g., GT, CAT) must in all cases be preceded and followed by a blank or comment.

5. Care should be taken to ensure that END statements correctly match the appropriate DO, BEGIN, and PROCEDURE statements.

6. In some situations, parentheses are required when their necessity is not immediately obvious. In particular, the expression following WHILE and RETURN must be enclosed in parentheses.

PROGRAM CONTROL

1. The procedure to be given initial control at execution time must have the OPTIONS(MAIN) attribute. If more than one procedure has the MAIN option, the first one encountered by the linkage-editor gets control.

2. When a procedure of a program is invoked while it is still active in the same task, it is said to be used recursively. Under the optimizing compiler, attempting the recursive use of a procedure that has not been given the RECURSIVE attribute may result in a program interrupt after exit from the procedure. This will occur if reference is made to automatic data of an earlier invocation of the procedure.

3. When a procedure may be invoked while it is still active in another task, the REENTRANT option must be specified.

DECLARATIONS AND ATTRIBUTES

1. DECLARE statements for AUTOMATIC variables are in effect executed at

entry to a block; sequences of the following type should not be used:

```
A: PROC;
   N=4;
   DCL B(N) FIXED;
   .
   .
   .
   END;
```

2. Missing commas in DECLARE statements are a common source of error. For example, a comma must follow the entry for each element in a structure declaration.

3. External identifiers should neither contain more than seven characters, nor start with the letters IKN.

4. In a PICTURE declaration, the V character indicates the scale factor, but does not in itself produce a decimal point on output. The point picture character produces a point on output, but is purely an editing character and does not indicate the scale factor. In a decimal constant, however, the point does indicate the scale factor. For example:

```
DCL A PIC'99.9',
    B PIC'99V9',
    C PIC'99.V9';
A,B,C=45.6;
PUT LIST (A,B,C);
```

This will cause the following values to be put out for A, B, and C, respectively:

```
04.5    456     45.6
```

If these values were now read back into the variables by a GET LIST statement, A, B, and C would be set to the following respective values:

```
004     56.0    45.6
```

If the PUT statement were then repeated, the result would be:

```
00.4    560     45.6
```

5. Separate external declarations for the same identifier must not specify conflicting attributes, either explicitly or by default. If this occurs the compiler will not be able to detect the conflict.

6. An identifier cannot be used for more than one purpose within its scope. Thus, the use of X in the following sequence of statements would be in error:

```
   PUT FILE (X) LIST (A,B,C); X=Y+Z;
   X:  M=N;
```

7. The precision of decimal integer constants should be taken into account when such constants are passed. For example:

```
CALL ALPHA(6);

ALPHA: PROCEDURE(X);
       DCL X FIXED DECIMAL;
       END;
```

If ALPHA is an external procedure, the above example is incorrect because X will be given a default precision, while the constant, 6, will be passed with precision (1,0).

8. When a data item requires conversion to a dummy, and the called procedure alters the value of the parameter, note that the dummy is altered, not the original argument. For example:

```
  DCL A FIXED,
      B FLOAT;
  CALL X(A,B);

X:PROC(Y,Z);
  DCL (Y,Z) FIXED;
  Y=Z**100; /*A IS ALTERED IN
              CALLING PROC*/
  Z=Y**3;   /*B IS UNALTERED IN
              CALLING PROC*/
  END X;
```

9. When the attributes for a given identifier are incompletely declared, the rest of the required attributes are supplied by default. The following default assumptions should be carefully noted.

FLOAT DECIMAL(6) REAL is assumed for implicitly declared arithmetic variables, unless the initial letter is in the range I through N, when FIXED BINARY(15,0) REAL is assumed.

If a variable is explicitly declared and any of the base, scale, or mode attributes is specified, the others are assumed to be from the set FLOAT/DECIMAL/REAL. For example:

```
DCL I;          /*I IS FIXED BINARY
                  (15,0) REAL
                  AUTOMATIC*/

DCL J REAL;     /*J IS FLOAT DECIMAL
                  (6) REAL
                  AUTOMATIC*/

DCL K STATIC; /*K IS FIXED BINARY
                  (15,0) REAL
                  STATIC*/
```

```
DCL L FIXED;  /*L IS FIXED DECIMAL
                (5,0) REAL
                AUTOMATIC*/
```

10. The precision of complex expressions is not obvious. For example, the precision of 1 + 1I is (2,0), that is, the precision follows the rules for expression evaluation.

11. When a procedure contains more than one entry point, with different parameter lists on each entry, make sure that no references are made to parameters other than those associated with the point at which control entered the procedure. For example:

```
A: PROCEDURE(P,Q);
   P=Q+8; RETURN;
B: ENTRY(R,S);
   R=P+S; /*THE REFERENCE TO P
            IS AN ERROR*/
   END;
```

12. Based storage is allocated in terms of doublewords; therefore, even for the smallest item, at least eight bytes are required.

13. The variable used in the REFER option must be referred to unambiguously. For example:

```
DCL 1 A,
      2 Y FIXED BIN,
      2 Z FLOAT,
    1 B,
      2 Y FIXED BIN,
      2 T(1:N REFER(B.Y));
```

    In any references to this declaration, Y must be fully qualified to prevent a possible ambiguity.

14. Conflicting contextual declarations must be avoided. P is often used as the name of a pointer; it must not, therefore, assume by default the characteristics of another data type. For example:

```
DCL B BASED (P),
    .
    .
    .
    P AUTO,
    .
    .
    .;
```

    The explicit declaration of P is processed first by the compiler and the default attributes, FLOAT and DECIMAL are added; the contextual declaration of P is then conflicting.

15. Parameters may not be given one of the storage class attributes AUTOMATIC, BASED, or STATIC; a parameter must either be CONTROLLED or have no storage class.

ASSIGNMENTS AND INITIALIZATION

1. When a variable is accessed, it is assumed to have a value which has been previously assigned to it and which is consistent with the attributes of the variable. If this assumption is incorrect, either the program will proceed with incorrect data or a program interrupt will occur. Such a situation can result from failure to initialize the variable, or it can occur as a result of the variable having been set in one of the following ways:

    a. by the use of the UNSPEC pseudovariable

    b. by record-oriented input

    c. by overlay defining a picture on a character string, with subsequent assignment to the character string and then access to the picture

    d. by passing as an argument a variable assigned in a different procedure, without matching the attributes of the parameter.

    e. by assignment to a based variable with different attributes, but at the same location.

    Failure to initialize a variable will result in the variable having an unpredictable value at execution time. Do not assume this value to be zero.

    Failure to initialize a subscript can be detected by enabling SUBSCRIPTRANGE, when debugging the program (provided the uninitialized value does not lie within the range of the subscript).

2. Under the optimizing compiler, any attempt to put out a variable or array that has not been initialized may well cause a data interrupt to occur. For example:

```
DCL A(10) FIXED;
A(1)=10;
PUT LIST (A);
```

To avoid the data interrupt, the array should be initialized before the assignment statement, thus:

```
A=0;
```

Note that this problem can also occur under the optimizing compiler as a result of CHECK system action for an uninitialized array. If the CHECK condition were enabled for the array in the above example, and system action were taken, the results, and the way in which the program terminates, would be unpredictable. The same problem arises when PUT DATA is used.

3. Note the distinction between = (assignment) and = (comparison). The statement

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

means "compare B with C and assign the result (either '1'B or '0'B) to A, performing type conversion if necessary."

4. Assignments that involve conversion should be avoided if possible (see section 2. under "Arithmetic and Logical Operations" later in this chapter).

5. In the case of initialization of or assignment to a fixed length string: if the assigned value is shorter than the string, it is extended on the right with blanks (for a character string) or zeros (for bit strings). For example:

```
DCL A CHAR(6),
    B CHAR(3) INIT('CR');
    A=B;
```

After the execution of the above statements, B would contain CRb, and A would contain CRbbbb.

6. It is not possible to reference a cross section of an array of structures in an assignment statement or any other single statement; the whole of an array of structures, or a single element may be referenced, but not a cross section.

7. When SIZE is disabled, the result of an assignment which would have raised SIZE is unpredictable:

FIXED BINARY: The result of an assignment here -- which includes, for instance, source language assignments and the conversions implied by parameter matching -- may be to raise FIXEDOVERFLOW.

FIXED DECIMAL: Truncation to the nearest byte may occur, without raising an interrupt. If the target precision is even, an extra digit may be inserted in the high-order byte.

ARITHMETIC AND LOGICAL OPERATIONS

1. The rules for expression evaluation should be carefully noted, with particular reference to priority of operations. The following examples show the kind of mistake that can occur:

```
X>Y|Z is not equivalent to X>Y|X>Z
but is equivalent to (X>Y)|Z

X>Y>Z is not equivalent to X>Y&Y>Z
but is equivalent to (X>Y)>Z
```

All operation sequences of equal priority are evaluated left to right, except for **, prefix +, prefix -, and ¬, which are evaluated right to left. Thus, the statement

```
A=B**-C**D;
```

is equivalent to

```
A=B**(-(C**D));
```

The normal use of parentheses is to modify the rules of priority; however, it may be convenient to use redundant parentheses as a safeguard or to clarify the operation.

2. Conversion is governed by comprehensive rules which must be thoroughly understood if unnecessary trouble is to be avoided. Some examples of the effect of conversion follow.

   a. DECIMAL FIXED to BINARY FIXED can cause unexpected results if fractions are involved:

```
DCL I FIXED BIN(31,5) INIT(1);
    I = I+.1;
```

The value of I is now 1.0625. This is because .1 is converted to FIXED BINARY(5,4), so that the nearest binary approximation is 0.0001B (no rounding occurs). The decimal equivalent of this is .0625. A better result would have been achieved by specifying .1000 in place of .1.

b. If arithmetic is performed on character string data, the intermediate results are held in the maximum fixed decimal precision (15,0):

```
DCL A CHAR(6) INIT('123.45');
DCL B FIXED(5,2);
B=A;    /*B HAS VALUE 123.45*/
B=A+A; / *B HAS VALUE 246.00*/
```

c. The rules for arithmetic to bit string conversion affect assignment to a bit string from a decimal constant:

```
DCL A BIT(1),
    D BIT(5);
A=1; /*A HAS VALUE '0'B*/
D=1; /*D HAS VALUE '00010'B*/
D='1'B; /*D HAS VALUE
          '10000'B*/
IF A=1 THEN GO TO Y;
       ELSE GO TO X;
```

The branch will be to X, because the assignment to A resulted in the following sequence of actions:

(1) The decimal constant, 1, is assumed to be FIXED DECIMAL (1,0) and is assigned to temporary storage with the attributes FIXED BINARY(4,0), taking the value 0001B;

(2) This value is now treated as a bit string of length (4), so that it becomes '0001'B;

(3) The resultant bit string is assigned to A. Since A has a declared length of 1, and the value to be assigned has acquired a length of 4, truncation occurs at the right, and A has a final value of '0'B.

To perform the comparison operation in the IF statement, '0'B and 1 are converted to FIXED BINARY and compared arithmetically. They are unequal, giving a result of "false" for the relationship A=1.

In the first assignment to D, a sequence of actions similar to that described for A takes place, except that the value is extended at the right with a zero, because D has a declared length that is 1 greater than that of the value to be assigned.

d. Assignment of arithmetic values to character strings involves conversion according to the rules given in section F, "Data Conversion and Expression Evaluation".

Example 1

```
DCL A CHAR(4),
    B CHAR(7);
A='0'; /*A HAS VALUE '0bbb'*/
A=0;   /*A HAS VALUE 'bbb0'*/
B=1234567; /*B HAS VALUE
             'bbb1234'*/
```

Note: The three blanks are necessary to allow for the possibility of a minus sign, a decimal or binary point, and provision for a single leading zero before the point.

Example 2

```
DCL CTLNO CHAR(8) INIT('0');
DO I=1 TO 100;
    CTLNO=CTLNO+1;
    .
    .
    .
END;
```

In this example, a conversion error occurs because of the following sequence of actions:

(1) The initial value of CTLNO, that is, '0bbbbbbb', is converted to FIXED DECIMAL(15,0).

(2) The decimal constant, 1, assumed to be FIXED DECIMAL(1,0), is added; in accordance with the rules for addition, the precision of the result is (16,0).

(3) This value is now converted to a character string of length 18 in preparation for the assignment back to CTLNO.

(4) Because CTLNO has a length of 8, the assignment causes truncation at the right; thus, CTLNO has a final value that consists entirely of blanks. This value cannot be successfully converted to arithmetic type for the second iteration of the loop.

e. FIXED division can result in unexpected overflows or truncation. For example, the expression

```
25+1/3
```

would yield a value of 5.33...3. To obtain a result of 25.33...3, it would be necessary to write

```
25+01/3
```

The explanation is that constants have the precision and scale factor with which they are written, while FIXED division results in a value of maximum implementation- defined precision. The results of the two evaluations are reached as follows:

Item	Precn/ Scale Factor	Result
1	(1,0)	1
3	(1,0)	3
1/3	(15,14)	0.33333333333333
25	(2,0)	25
25+1/3	(15,14)	5.33333333333333 (truncation on left; FIXEDOVERFLOW would be raised unless disabled)
01	(2,0)	01
3	(1,0)	3
01/3	(15,13)	00.3333333333333
25	(2,0)	25
25+01/3	(15,13)	25.3333333333333

Alternatively, the PRECISION built-in function could be used:

```
25+PREC(1/3,15,13)
```

f. Checking of a picture is performed only on assignment into the picture variable:

```
DCL A PIC'999999',
    B CHAR(6) DEF A,
    C CHAR(6);
B='ABCDEF';
C=A; /*WILL NOT RAISE CONV
       CONDITION*/
A=C; /*WILL RAISE CONV*/
```

Note also (A, B, C as declared above):

```
A=123456; /*A HAS VALUE
            123456*/
          /*B HAS VALUE
            '123456'*/
C=123456; /*C HAS VALUE
            'bbb123'*/
C=A; /*C HAS VALUE '123456'*/
```

g. A FIXED DECIMAL element with a declared even precision (P,Q) may have an effective precision of (P+1,Q), as the high-order byte may not be non-zero. The SIZE condition can be used to eliminate this effect:

```
DCL (A,B,C) FIXED DECIMAL (6,0);
    ON SIZE;
      .
      .
      .
(SIZE): A = B + C;
```

This ensures that the high-order byte of A is zero after the assignment.

DO GROUPS

1. The scope of a condition prefix applied to a DO statement is limited to execution of the statement itself; it does not apply to execution of the entire group.

2. An iterative DO group is not executed if the terminating condition is satisfied at initialization:

```
I=6;
DO J=I TO 4;
   X=X+J;
END;
```

X is not altered by this group, since BY 1 is implied. Iterations can step backwards, and if BY -1 had been specified, three iterations would have taken place.

3. Expressions in a DO statement are assigned to temporaries with the same characteristics as the expression, not the variable. For example:

```
DCL A DECIMAL FIXED(5,0);
    A=10;
    DO I=1 TO A/2;
    .
    .
    .
    END;
```

This loop will not be executed, because A/2 has decimal precision (15,10), which, on conversion to binary (for comparison with I), becomes binary (31,34).

Five iterations would result if the DO statement were replaced by

```
ITEMP=A/2;
DO I=1 TO ITEMP;
or
DO I=1 TO PREC(A/2,6,1)
```

4. DO groups cannot be used as on-units; a BEGIN block should be used for an on-unit of more than one statement.

5. Upper and lower bounds of iterative DO groups are computed once only, even if the variables involved are reassigned within the group. This applies also to the BY expression.

   Any new values assigned to the variables involved would take effect only if the DO group were started again.

6. In a DO group with both a control variable and a WHILE option, the evaluation and testing of the WHILE expression is carried out only after determination (from the value of the control variable) that iteration may be performed. For example, the following group would be executed at most once:

```
DO I=1 WHILE(X>Y);
.
.
.
END;
```

7. I is frequently used as the control variable in a DO group, for example:

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

   Within the scope of this implicit declaration, I might be contextually declared as a pointer, for example:

```
DCL X BASED(I);
```

   The two statements are in conflict and will produce a diagnostic message. When I is a pointer variable, it can only be used in a DO group in one of the following ways:

   a.
```
DCL (I, IA, IB, IC) POINTER;
   .
   .
   .
DO I=IA,IB,IC;
```

   b.
```
DCL (I, IA) POINTER;
   .
   .
   .
DO WHILE(I=IA);
```

8. If the control variable is used as a subscript within the do-group, care must be taken not to let the variable run beyond the bounds of the array dimension. For instance:

```
DECLARE A(10);
DO I = 1 TO N;
     .
     .
     .
A(I) = X;
     .
     .
     .
END;
```

   If N is greater than 10 then the assignment statement may overwrite data beyond the storage allocated to the array A. Such a bug can be difficult to find, particularly if th the overwritten storage happens to contain object code. The error can be prevented by enabling SUBSCRIPTRANGE.

DATA AGGREGATES

1. Array arithmetic should be thought of as a convenient way of specifying an iterative computation. For example:

```
DCL A(10,20);
.
.
.
A=A/A(1,1);
```

   has the same effect as

```
DCL A(10,20);
.
.
.
DO I=1 TO 10;
DO J=1 TO 20;
A(I,J)=A(I,J)/A(1,1);
END; END;
```

   Note that the effect is to change the value of A(1,1) only, since the first iteration would produce a value of 1 for A(1,1). If the programmer wished to divide each element of A by the original value of A(1,1), he could write

```
B=A(1,1);
A=A/B;
```

or alternatively,

```
DCL A(10,20),
    B(10,20);
.
.
.
B=A/A(1,1);
```

2. Note the effect of array multiplication:

```
DCL (A,B,C) (10,10);
.
.
.
A=B*C;
```

This does not effect matrix multiplication; it is equivalent to:

```
DCL (A,B,C) (10,10);
.
.
.
DO I=1 TO 10;
DO J=1 TO 10;
A(I,J)=B(I,J)*C(I,J);
END; END;
```

STRINGS

1. Assignments made to a varying string by means of the SUBSTR pseudovariable do not set the length of the string. A varying string initially has an undefined length, so that if all assignments to the string are made using the SUBSTR pseudovariable, the string still has an undefined length and cannot be successfully assigned to another variable or written out.

2. The user must ensure that the lengths of intermediate results of string expressions do not exceed 32767 bytes. This applies particularly to strings of varying lengths, as there is no object-time length checking.

FUNCTIONS AND PSEUDOVARIABLES

1. When UNSPEC is used as a pseudovariable, the expression on the right is converted to a bit string. Consequently, the expression must not be invalid for such conversion; for example, if the expression is a character string containing characters other than 0 or 1, a conversion error will result.

ON-CONDITIONS AND ON-UNITS

1. Note the correct positioning of the ON statement. If the specified action is to apply when the named condition is raised by a given statement, the ON statement must be executed before that statement. The statements:

```
GET FILE (ACCTS) LIST (A,B,C);
ON TRANSMIT (ACCTS) GO TO TRERR;
```

would result in the ERROR condition being raised in the event of a transmission error during the first GET operation, and the required branch would not be taken (assuming that no previous ON statement applies). Furthermore, the ON statement would be executed after each execution of the GET statement.

2. An on-unit cannot be entered by means of a GOTO statement. To execute an on-unit deliberately, the SIGNAL statement can be used.

3. CONVERSION on-units entered as a result of an invalid conversion (as opposed to SIGNAL) should either change the invalid character (by means of the ONSOURCE or ONCHAR pseudovariable), or else terminate with a GOTO statement. Otherwise, the system will print a message and raise the ERROR condition.

4. At normal exit from an AREA on-unit the standard system action is to try again to make the allocation. As a result the on-unit will be entered again, and an indefinite loop will be created. To avoid this, the amount allocated should be modified in the on-unit, for example, by using the EMPTY built-in function or by changing a pointer variable.

5. Do not use on-units to implement the program's logic: use them only to recover from truly exceptional conditions. Whenever an on-unit is entered, considerable error-handling overheads are incurred. To implement the logic, the programmer should perform the necessary tests, rather than relying on the compiler's condition-detecting facilities.

   For example, in a program using record-oriented output to a keyed data set, the programmer might wish to eliminate certain keys because they would not fit into the limits of the data set. He may rely on the raising of the KEY condition to detect unsuitable keys, but it is

considerably more efficient for him to test each key himself.

6. After debugging, disable any normally-disabled conditions that were enabled for debugging purposes by removing the relevant prefixes, rather than by including NO-condition prefixes. For instance, disable the SIZE condition by removing the SIZE prefix, rather than by adding a NOSIZE prefix. The former method allows the compiler to eliminate code that checks for the condition, whereas the latter method necessitates the generation of extra code to prevent the checks being carried out.

INPUT/OUTPUT

1. The UNDEFINEDFILE condition is raised not only by conflicting language attributes (such as DIRECT with PRINT), but also by the following:

   a. Block size smaller than record size (except when records are spanned).

   b. LINESIZE exceeding the permitted maximum.

   c. KEYLENGTH zero or not specified for creation of INDEXED, REGIONAL(2), or REGIONAL(3) data sets.

   d. Specifying a KEYLOC option, for an INDEXED data set, with a value resulting in KEYLENGTH + KEYLOC exceeding the record length.

   e. Specifying a V-format logical record length of less than 18 bytes for STREAM data sets.

   f. Specifying, for FB-format blocked records, a block size which is not an integral multiple of the recordsize.

   g. Specifying, for VB-format records, a logical record length that is not at least four bytes smaller than the specified block size.

2. If a file is to be used for both input and output, it must not be declared with either the INPUT or the OUTPUT attribute. The required option can be specified on the OPEN statement.

3. Input/output lists must be surrounded by a pair of parentheses; so must iteration lists. Therefore, two pairs of outer parentheses are required in

```
GET LIST ((A(I) DO I=1 TO N));
```

4. The last eight bytes of a source key to access a regional data set must be the character string representation of a fixed decimal integer. When generating the key, the rules for arithmetic to character string conversion should be considered. For example, the following group would be in error:

```
DCL KEYS CHAR(8);
DO I=1 TO 10;
   KEYS=I;
   WRITE FILE(F) FROM (R)
     KEYFROM (KEYS);
END;
```

   The default for I is FIXED BINARY(15,0), which requires not 8 but 9 characters to contain the character string representation of the arithmetic values.

5. Note that the file must have the KEYED attribute if the KEY, KEYFROM, or KEYTO options are to be used in any input/output statement referring to that file.

6. The standard file names SYSIN and SYSPRINT are implicit only in GET and PUT statements. Any other reference, such as those in ON statements or record-oriented input/output statements, must be explicit.

7. PAGESIZE and LINESIZE are not file attributes, that is, they cannot be included in a declaration for the file; they are options on the OPEN statement.

8. When an edit-directed data list is exhausted, no further format items will be processed, even if the next format item does not require a matching data item. For example:

```
DCL A FIXED(5),
    B FIXED(5,2);
GET EDIT (A,B) (F(5),F(5,2),X(70));
```

   The X(70) format item will not be processed. To read a following card with data in the first ten columns only, the SKIP option can be used:

```
GET SKIP EDIT (A,B) (F(5), F(5,2));
```

9. The number of data items represented by an array or structure name

appearing in a data list is equal to the number of elements in the array or structure; thus if more than one format item appears in the format list, successive elements will be matched with successive format items. For example:

```
DCL 1 A,
      2 B CHAR(5),
      2 C FIXED(5,2);
.
.
.
PUT EDIT (A) (A(5),F(5,2));
```

B will be matched with the A(5) item, and C will be matched with the F(5,2) item.

10. Arrays are transmitted in row major order (e.g., A(1,1), A(1,2), A(1,3), ... A(2,1), ... etc.)

11. Strings used as input data for GET DATA and GET LIST must be enclosed in quotation marks.

12. The 48-character representation of a semicolon (,.) is not recognized as a semicolon if it appears in a data-directed input stream; the 11-8-6 punch must be used.

13. The user must be aware of a limitation of PUT DATA; (i.e., without a data list): its effect when used with an ON statement is restricted because the data known to PUT DATA would be the data known at the point of the on-unit.

    If the ON statement

    ```
    ON ERROR PUT DATA;
    ```

    is used in an outer block, it must be remembered that variables in inner blocks are not known and therefore will not be dumped. It would be a good practice, therefore, to repeat the on-unit in all inner blocks during debugging.

    If an error occurs during execution of the PUT DATA statement, and this statement is within an ERROR on-unit, the program will recursively enter the ERROR on-unit until no more storage remains. Since this could be wasteful of machine time and printout, the ERROR on-unit should be turned off once it is activated. Instead of:

    ```
    ON ERROR PUT DATA;
    ```

    better code would be:

    ```
    ON ERROR BEGIN;
            ON ERROR SYSTEM;
            PUT DATA;
            END;
    ```

    When PUT DATA is used without a data-list every variable known at that point in the program is transmitted in data-directed output format to the specified file. Users of this facility, however, should note that:

    a) Uninitialized FIXED DECIMAL data may raise the CONVERSION condition or a data interrupt.

    b) Unallocated CONTROLLED data will cause arbitrary values to be printed and, in the case of FIXED DECIMAL, may raise the CONVERSION condition or a data interrupt.

14. A pointer set in READ SET or LOCATE SET is not valid beyond the next operation on the file, or beyond a CLOSE statement. In OUTPUT files, WRITE and LOCATE statements can be freely mixed.

    When it is required to rewrite a record that has been read into a buffer (using a READ SET statement that specifies a SEQUENTIAL UPDATE file) and then updated, the REWRITE statement without a FROM option may be used. The result of a REWRITE after a READ SET is always to cause the contents of the last buffer to be rewritten onto the data set. For example:

    ```
    3  READ FILE (F) SET (P);
                  .
                  .
                  .
    5  P->R = S;
                  .
                  .
                  .
    7  REWRITE FILE (F);
                  .
                  .
                  .
    11  READ FILE (F) INTO (X);
                  .
                  .
                  .
    15  REWRITE FILE (F);
                  .
                  .
                  .
    19  REWRITE FILE (F) FROM (X);
    ```

    Notes:

    Statement 7 will rewrite a record updated in the buffer.

Statement 15 will not change the record on the data set at all.
Statement 19 will raise ERROR, since there is no preceding READ statement.

There is one case where it is not possible to check for the KEY condition on a LOCATE statement until transmission of a record is attempted.

This is:

> When there is insufficient room in the specified region to output the record on a REGIONAL(3) V- or U-format file. Neither the record raising the condition nor the current record is transmitted.

If a LOCATE is the last I/O statement to be executed before the file is closed, the record is not transmitted and the condition may be raised by the CLOSE statement.

15. If a reference is made, at object time, to a based variable that has not been allocated storage, an unpredictable interrupt (protection, addressing, or specification) may occur.

16. Areas, pointers, offsets and structures containing any of these cannot be used with STREAM I/O.

17. When a based variable is freed, the associated pointer no longer contains useful information.

18. A based variable allocated in an area must be freed in that area. For example:

```
DCL A AREA, B BASED (X);
ALLOCATE B IN (A);


FREE B;              /* INVALID */
FREE B IN (A);       /* VALID   */
```

19. The alignment in the buffer of the first byte of the first record in a block that has been read from an ASCII data set is not necessarily on a doubleword. The block prefix is doubleword aligned, but the alignment of the first record depends on the length of the block prefix.

Chapter 19: Interlanguage Communication Facilities

The PL/I interlanguage facilities permit communication, at execution time, between programs compiled by the PL/I checkout and optimizing compilers and programs compiled by one of the following compilers, and executed using the corresponding library. The compilers and libraries have all been developed by IBM for OS.

Compiler or Library	Program No.
FORTRAN E	360S-FO-092
FORTRAN G	360S-FO-520
FORTRAN H Version II	360S-FO-500
FORTRAN Library	360S-LM-501
COBOL E	360S-CO-503
COBOL E Library	360S-LM-504
COBOL F	360S-CB-524
COBOL F Librar.	360S-LM-525
American National Standard COBOL	360S-CB-545
American National Standard COBOL Library	360S-LM-546

Communication between a PL/I program, and a program compiled by one of the FORTRAN or COBOL compilers, can be achieved in two ways:

1. By using a conversion data set for the PL/I and COBOL/FORTRAN routines.

2. By invoking a COBOL/FORTRAN routine from a PL/I routine, or vice versa, and by passing data either as arguments or in the form of static storage.

If a common data set is used to communicate between a PL/I and a COBOL routine, the COBOL option of the ENVIRONMENT attribute may be required. For further details, see the section "Data Interchnage (COBOL)" in chapter 12.

A PL/I procedure can invoke a COBOL routine by use of the CALL statement, or can invoke a FORTRAN routine by use of the CALL statement or a function reference. Alternatively, a PL/I procedure can be invoked by use of the corresponding language features in a COBOL or a FORTRAN main program or routine. Arguments can be passed on invocation, and a value can be returned for function references.

A COMMON block in FORTRAN has storage equivalent to that of a STATIC EXTERNAL variable in PL/I. If a COMMON block and a STATIC EXTERNAL variable are given the same name, then they will be allocated the same block of storage, in the same way as two identical STATIC EXTERNAL variables in PL/I. Assigning a value to one variable causes the same value to be assigned to the other. There is no similar equivalence in COBOL - no COBOL variable can have common storage with a PL/I variable other than as an argument or parameter.

The interlanguage facilities are entirely provided by the PL/I compiler; they are obtained by specifying the appropriate language items in the invoking or invoked PL/I procedure. Existing COBOL or FORTRAN programs or routines generally do not need modification or recompiling for interlanguage use; new programs or routines can be written in these languages and compiled as before, without the need to anticipate interlanguage communication. Thus existing COBOL or FORTRAN application programs can be extended by the use of PL/I procedures, while COBOL or FORTRAN libraries can be made available to new or existing PL/I procedures.

Note: In the context of this Chapter, "routine" includes a COBOL subprogram, or a FORTRAN subroutine or function, including a FORTRAN library function. The conventions that exist in these languages for handling subroutines and functions apply normally, and are not modified for interlanguage use. In particular, the restriction that a FORTRAN function cannot be invoked without passing an argument or arguments still applies when the invocation is from a PL/I routine.

Interlanguage Facilities

While a detailed knowledge of COBOL or FORTRAN is not essential for use of the interlanguage facilities, the programmer may need to be aware of the equivalents in data organization in PL/I and the other two languages. These equivalents must be understood in order to achieve argument/parameter matching.

The interlanguage facilities automatically resolve differences in the mapping for equivalent data organizations, when matching arguments and parameters; the programmer can, if he wishes, override this action.

Facilities are provided to extend PL/I interrupt-handling to cover invoked COBOL or FORTRAN routines.

Passing Arguments to a COBOL or FORTRAN Routine

When an argument is passed to a COBOL or a FORTRAN routine, the data type is determined in the normal PL/I manner, that is, from the parameter descriptor list of the associated entry declaration, or from the argument itself. The interlanguage facilities ensure, however, that the addressing mechanism for the argument is that used by the invoked language, and that, unless otherwise required, the mapping of any aggregates passed is that used by the invoked language. Note that since the interlanguage facilities provided by PL/I cannot look at the parameter in the invoked routine, it is the programmer's responsibility to ensure that the parameter in the invoked routine corresponds in data type and organization to the argument description in PL/I.

If the PL/I compiler can determine, at compile-time, that the mapping of a structure or array argument is the same in PL/I as in the invoked language, the argument is passed directly to the invoked routine. However, where such mapping equivalence does not exist, the interlanguage facilities provide for a dummy argument to be passed, where the dummy is mapped according to the rules of the invoked language. See section K, "Data Mapping".

If the PL/I data types of arguments passed to FORTRAN or COBOL have no equivalents in these languages, a warning message is produced at compile-time. At execution-time the results are undefined, and may include abnormal termination.

Data types: PL/I has more data types than either COBOL or FORTRAN; some have no equivalents in these languages. The extent to which PL/I data types have equivalents in COBOL or FORTRAN, and therefore can be passed as arguments, is summarized here.

Problem data: Most of the PL/I data types have equivalents in either COBOL or FORTRAN. Tables of data equivalents for PL/I-COBOL and PL/I-FORTRAN are given below, in "COBOL Interface" and "FORTRAN Interface" respectively.

Program-control data: Arguments of any program-control data type can be passed to an invoked COBOL or FORTRAN routine. However, only an entry argument can be passed and used within the invoked routine, and then only if the routine is a FORTRAN routine. Arguments of any other data type should not be used in the invoked routine except to be passed in turn to a PL/I procedure.

Note: The COBOL option in the ENVIRONMENT attribute can be specified for a file that is to be used in certain input/output operations. Although this option initiates remapping of PL/I structures, it is in no way associated with the interlanguage facilities described here; a file with this option cannot be used as a file argument or a file parameter. For use of the COBOL option of the ENVIRONMENT attribute, see "ENVIRONMENT Attribute" in chapter 12, "Record-Oriented Transmission."

Data-mapping: In order that an argument can be successfully passed to a COBOL or FORTRAN routine, the mapping of the actual argument passed must correspond to the mapping assumed for the parameter by COBOL or FORTRAN.

For an element argument, the only requirement is that the alignments of argument and parameter are compatible. In PL/I the alignment of variables is determined by the ALIGNED and UNALIGNED attributes. The equivalent specifications in COBOL and FORTRAN are:

PL/I	COBOL	FORTRAN
ALIGNED	SYNCHRONIZED	Normal alignment
UNALIGNED	Unsynchronized	No equivalent

The alignment of a PL/I argument is deduced, like the data type, from the parameter descriptor list or from the argument itself. Only ALIGNED elements may be passed to SYNCHRONIZED COBOL parameters, or to FORTRAN parameters. Either ALIGNED or UNALIGNED elements can be passed to COBOL unsynchronized parameters. It is the programmer's responsibility to ensure that these alignments are compatible.

The problem is more complicated for data aggregates. A PL/I or a COBOL structure for example can have either of the alignment stringencies given above. In addition, each member can have its own alignment stringency or all members can have the same alignment stringency. Padding bytes are inserted by the mapping algorithm for the particular language, in order to preserve the required alignment for each member. In a PL/I structure, the alignments are adjusted, where possible, to minimize the amount of padding required; this adjustment does not occur in a COBOL structure. The result is that a structure mapped with the PL/I mapping algorithm may not have the same layout in main storage as a structure mapped with the COBOL algorithm.

Similarly, the mapping of arrays is different in PL/I and FORTRAN. PL/I stores arrays of more than one dimension in

row-major-order, while FORTRAN stores them in column-major-order. Hence, for arrays with more than one dimension, a reference to an element in PL/I is obtained by reversing the order of the subscripts that would be used in FORTRAN to refer to the same element.

The interlanguage facilities resolve these problems by creating dummy arguments for PL/I data aggregates passed as arguments to COBOL or FORTRAN routines. When a PL/I structure is passed as an argument to a COBOL routine, the mapping of the argument in both languages is considered. If the compiler can determine that the mappings are identical, the argument is passed directly to the COBOL routine.

However, if the compiler cannot determine that the mappings are identical, a dummy argument is created, mapped according to the COBOL mapping algorithm. The values of the members of the PL/I structure are assigned to the corresponding members in the dummy argument; the dummy is then passed as an argument to the COBOL routine. On return to the PL/I procedure, the values in the dummy argument (which may or may not have been changed) are assigned to the corresponding members of the original PL/I argument.

Similarly, when a PL/I array is passed as an argument to a FORTRAN routine, the mapping of the array in both languages is considered. If the arrays are unidimensional, and are in connected storage and are aligned identically, the argument is passed directly to the invoked FORTRAN routine. If either the arrays are unidimensional and do not meet the above conditions, or are multidimensional, a dummy argument is created, mapped according to FORTRAN array handling. (In effect, this means the subscripts are reversed). The values of the PL/I array elements are assigned to the corresponding elements in the dummy argument. The dummy is then passed as an argument to the FORTRAN routine. On return to the PL/I procedure, the values in the dummy argument (which may or may not have been changed) are assigned to the appropriate elements of the PL/I argument.

The programmer can specify certain options that inhibit or restrict the effect of the interlanguage facilities for remapping data aggregates. If several are passed at an invocation, he can, for example, inhibit the facilities for one argument, allow them for another argument, or restrict them for a third argument.

Invocation

Invocation of a COBOL or FORTRAN routine is performed by a CALL statement or (in the case of a FORTRAN routine only) function reference that specifies an entry constant or variable whose value corresponds to the entry point of a COBOL or FORTRAN routine. The entry point must not be that of a FORTRAN main program. The entry constant or variable must be identified as invoking COBOL or FORTRAN by use of the appropriate options in the OPTIONS attribute in the declaration of the entry in the PL/I program. The programmer may also specify, in this declaration, options which suppress re-mapping of data aggregates and an option which allows PL/I to deal with certain interrupts in the COBOL or FORTRAN routine.

The options are:

COBOL: This specifies that the designated entry point is in a COBOL routine.

FORTRAN: This specifies that the designated entry point is in a FORTRAN routine.

NOMAP: This specifies that a dummy argument is not created; the aggregate argument is passed directly to the invoked routine.

NOMAPIN: This specifies that, if a dummy argument is created, it is not initialized with the values of the aggregate argument.

NOMAPOUT: This specifies that, if a dummy argument is created, then, on return, the values in the dummy argument are not assigned to the aggregate argument.

INTER: This specifies that any interrupts occurring during the execution of a COBOL or FORTRAN routine that are not dealt with by the COBOL or FORTRAN interrupt-handling facilities are dealt with by the PL/I interrupt-handling facilities (see also "Interrupt Handling" later in this chapter).

The NOMAPIN and NOMAPOUT options should be used if initialization is not required whenever program efficiency is important, because they allow the compiler to omit unnecessary initialization code.

ARGn: This is an option of NOMAP, NOMAPIN, and NOMAPOUT which specifies which arguments the option applies to. If no ARGn is specified, the option is applied to all arguments.

The following points should be noted in the declaration of the entry name:

1. Either COBOL or FORTRAN (but not both) can appear in the declaration. One or more of the options NOMAP, NOMAPIN and NOMAPOUT can appear in the same declaration.

2. The RETURNS attribute cannot be used with the COBOL option, as COBOL subprograms do not return values.

3. An entry variable or a parameter can be declared with the interlanguage options.

4. An entry name with the interlanguage options can appear in a GENERIC attribute specification.

5. The entry constant name of the COBOL or FORTRAN routine may have one through eight characters. If more than eight characters are specified, the leftmost eight only are taken.

Examples:

1.
```
DCL COBOL ENTRY (CHAR(5))
     OPTIONS(COBOL INTER),

    COBOLB ENTRY (1, 2 FIXED, 2 FLOAT)
     OPTIONS(COBOL NOMAPIN),
    COBOLBXX OPTIONS(COBOL) EXTERNAL
     ENTRY(...);
```

2.
```
DCL FORTA ENTRY(FIXED BINARY)
     OPTIONS(FORTRAN) RETURNS
     (FLOAT (5));
```

3.
```
DCL A EXTERNAL ENTRY(...)  VARIABLE
     OPTIONS (FORTRAN),

    B OPTIONS(FORTRAN);
        .
        .
        .
    A=B;
    CALL A(...);
```

4.
```
DCL A GENERIC (COBOLZ
     WHEN(CHARACTER),
       FORTZ WHEN(FIXED BINARY)),

    COBOLZ OPTIONS(COBOL),

    FORTZ OPTIONS(FORTRAN);
```

5.
```
DCL A ENTRY;

CALL X(A);
        .
        .
        .
X:PROC(B);
DCL B OPTIONS(COBOL);
```

6.
```
DCL COBSUB ENTRY(...,...,...,)
     OPTIONS(COBOL,NOMAP(ARG1,ARG3));
        .
        .
        .
CALL COBSUB(A,B,C);
        .
        .
        .
CALL COBSUB(X,Y,Z);
```

Passing Arguments to a PL/I Procedure

When an argument is passed to a PL/I procedure from COBOL or FORTRAN, the data type is determined in the normal PL/I manner, that is from the declaration of the parameter. The interlanguage facilities ensure that the addressing mechanism used for the parameter is that used by PL/I, and that, unless otherwise required, the mapping of any aggregate parameters passed is also that used by PL/I. Note that since the interlanguage facilities provided by PL/I cannot look at the argument in the routine invoking PL/I, it is the programmer's responsibility to ensure that the argument passed to PL/I corresponds in data type and organization to the parameter declared in PL/I.

Data mapping: The situation is similar to that which occurs on invocation of COBOL or FORTRAN by PL/I. The mapping of the argument on entry to the PL/I procedure must correspond to the mapping used by PL/I in addressing the parameter.

For element arguments and parameters, this means that a SYNCHRONIZED or unsynchronized COBOL argument may be passed to an UNALIGNED PL/I parameter, or that a SYNCHRONIZED COBOL argument or a FORTRAN argument can be passed to an ALIGNED PL/I parameter.

For aggregate arguments and parameters where the mapping of the argument in COBOL or FORTRAN differs from the mapping of the parameter in PL/I, the interlanguage facilities resolve the problem by creating a dummy argument which is passed to the PL/I procedure.

The dummy argument is mapped according to PL/I rules, and, before invocation of

the PL/I procedure, the values of the members of the COBOL or FORTRAN argument are assigned to the corresponding members of the dummy argument. On return from the PL/I procedure, the values of the members of the dummy argument are assigned back to the original argument.

If the compiler can recognize that the mapping in COBOL or FORTRAN and PL/I are equivalent, no such dummy is created. Alternatively, the programmer can inhibit the creation of the dummy, or the assignments between the original argument and the created dummy, by means of options.

Invocation

The entry points in a PL/I procedure that are to be invoked from COBOL or FORTRAN must be identified by the appropriate options in the corresponding PROCEDURE or ENTRY statement. The programmer may also specify options that suppress re-mapping of data aggregates.

COBOL: This specifies that the entry point can only be invoked by a COBOL routine.

FORTRAN: This specifies that the entry point can only be invoked by a FORTRAN routine.

NOMAP: This specifies that a dummy argument is not created; the COBOL or FORTRAN aggregate argument is passed directly to PL/I.

NOMAPIN: This specifies that, if a dummy argument is created, it is not initialized with the values of the aggregate argument.

NOMAPOUT: This specifies that, if a dummy argument is created its values are not assigned back to the aggregate argument on return. The NOMAPIN and NOMAPOUT options should be used, if initializations not required, whenever program efficiency is important, since they allow the compiler to omit unnecessary initialization code.

Parameter list: The parameter or parameters to which the NOMAP, NOMAPIN, or NOMAPOUT options apply can be specified in a list. If no list is specified, the option is applied to all parameters.

The following points should be noted when coding the PROCEDURE or ENTRY statement:

1. Only one of the options MAIN, COBOL, or FORTRAN can appear in the same statement. One or more of the options NOMAP, NOMAPIN, or NOMAPOUT can appear in the same statement.

2. If the parameters for the procedure include strings, areas or arrays, the lengths, sizes or bounds for these must be specified as decimal integer constants.

3. The RETURNS option cannot be specified for any entry point invoked by a CCBOL routine.

Examples:

1.
```
P1:PROC(A,B,C) OPTIONS(FORTRAN
NOMAPIN(C) NOMAPOUT(A));
     DCL A(3,4) FLOAT BIN(20),
         B FIXED BIN(31),
         C(5,6) FLOAT DEC(6);
```

2.
```
P2:PROC(R,S,T) OPTIONS (FORTRAN
NOMAP);
```

3.
```
P3:PROC(X,Y) OPTIONS(COBOL NOMAPIN(X)
NOMAPOUT(Y));
     DCL 1 X, 2...2...3...,
         1 Y, 2...2...3...;
```

Using Common Storage

A variable in a PL/I program can be allocated the same block of storage as a group of variables in a FORTRAN routine. This storage can then be used to communicate between the two routines. Allocation of common storage is achieved by declaring a PL/I variable to be STATIC EXTERNAL and to have the same name as a COMMON block in the FORTRAN routine. The STATIC EXTERNAL variable and the COMMON block will then be equivalent to two declarations of a STATIC EXTERNAL variable in different external PL/I procedures. The number of variables using common storage is not limited to two: any number of identical STATIC EXTERNAL variables in different PL/I procedures may be used together with any number of identical COMMON blocks in different FORTRAN routines, if all the procedures and routines are link-edited into a single program. Methods of link-editing are given in the compilers' programmers' guides.

The STATIC EXTERNAL variables must follow the normal PL/I rules relating to these attributes, and they must be of a

data type that corresponds to the data type of the COMMON variables (see "FORTRAN Interface" later in this chapter for a table of corresponding data types). Also, the PL/I variables must be aligned to meet the requirements of the corresponding FORTRAN data type.

The PL/I variables may be initialized using the INITIAL attribute, and the FORTRAN variables may be initialized using a block data subprogram. If the PL/I variables on the one hand and the FORTRAN variables on the other are not initialized to the same value, the procedure or routine that is encountered first by the linkage-editor determines the initial value of all the variables. It is not an error to initialize a PL/I variable to a different value from a corresponding FORTRAN variable, or to initialize one and not the other.

The PL/I variable may have further variables overlayed upon it by means of the DEFINED attribute, provided that the defined variable meets the data type and alignment requirements of the FORTRAN variable. If the requirements are not met, execution errors may occur.

Common storage cannot be used for a PL/I and a COBOL variable; the only facility provided by PL/I for communication between a PL/I procedure and a COBOL routine is that for passing arguments.

INTERLANGUAGE ENVIRONMENT

For a PL/I procedure to be executed, a PL/I environment must first be established. If the program contains a PL/I main procedure, this environment is established when the program is first entered. If the main routine is COBOL or FORTRAN, the interlanguage facilities will establish the required PL/I environment when necessary. This section describes the conventions and restrictions in the interlanguage context.

Establishing the PL/I Environment

If the main routine of the program is a PL/I main procedure, the PL/I environment is established on entry to the program. Even if this program contains a mixture of PL/I and COBOL or FORTRAN routines, the normal rules for freeing PL/I storage and closing PL/I files apply.

If the main routine of the program is not a PL/I main procedure, the PL/I environment is established when the first PL/I procedure is invoked. The extent of this environment includes the routine that invoked the PL/I procedure (see figure 19.1), and the environment remains in existence until that routine is terminated. The environment can be reestablished and terminated as frequently as required. Whenever the PL/I environment is destroyed, all PL/I controlled and based storage is released, and all PL/I files are closed.

For reasons of efficiency and of programming convenience, the PL/I environment should be destroyed as infrequently as possible during execution of a program. This can be ensured if the main routine is a PL/I main procedure, or if a PL/I procedure, no matter what it contains, is invoked from the main routine. The latter alternative, however, has the disadvantage that if the main routine is in FORTRAN, the PL/I environment will not be ended normally when the final FORTRAN RETURN is executed to return control to the operating system (see "Termination of FORTRAN and COBOL Routines" later in this Chapter).

Interrupt Handling

COBOL and FORTRAN routines handle certain of the hardware interrupts that may occur during their execution, but there are some that they do not handle. The interlanguage communication facilities of PL/I allow any interrupt not dealt with by a COBOL or FORTRAN routine to be handled by any PL/I procedure from which that routine is dynamically descendent.

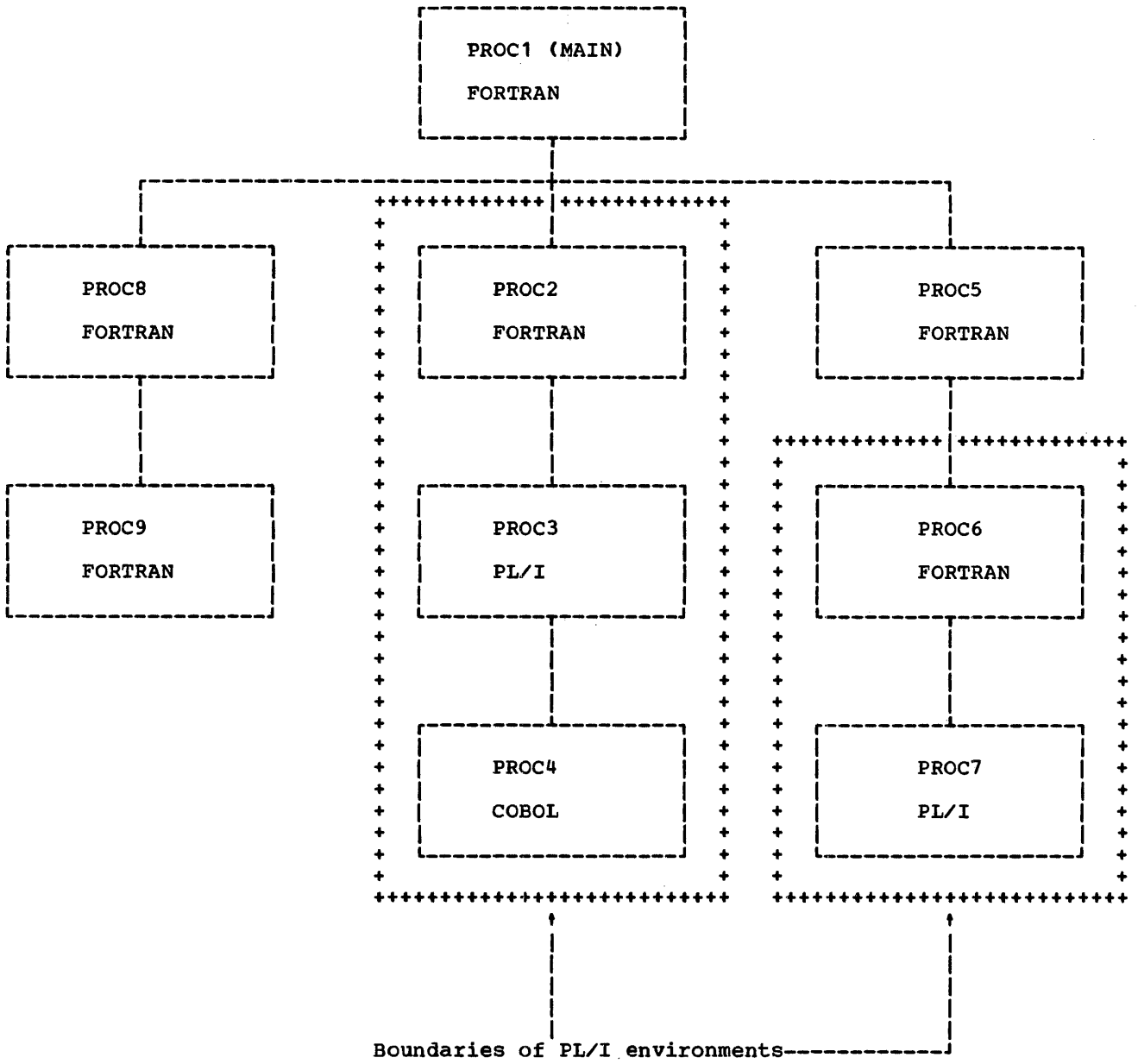


Figure 19.1. Extent of PL/I environment

The programmer specifies the INTER option of the OPTIONS attribute when declaring the COBOL or FORTRAN entry name. (See also the INTER option under "Passing Arguments to COBOL or FORTRAN Routine" earlier in this chapter.) This allows the interrupts not dealt with by the invoked COBOL or FORTRAN routine to be handled by either a PL/I on-unit or by PL/I standard system action.[1] In PL/I, an on-unit, while established, applies not only to the procedure in which it was created, but also to all procedures that are dynamically descendent from it. If there occurs, during the execution of a COBOL or FORTRAN routine, an interrupt that will not be

---

[1]Except that PL/I cannot handle a ZERODIVIDE interrupt in a division of COMPUTATIONAL-3 data in a routine compiled by a COBOL compiler other than the COBOL E compiler. Such an interrupt will cause termination of the program.

handled by that routine, and if the routine was invoked by a PL/I procedure in which the INTER option was specified for the COBOL or FORTRAN entry name, then a search is made through all invoking procedures for an appropriate on-unit. If none is found, standard system action is taken. If INTER is not specified, no search is made, and the interrupt is dealt with by the operating system control program.

Note that the search passes through all routines in the invoking chain, as far as the limit of the PL/I environment. It is therefore possible for the search to include COBOL and FORTRAN routines. Such routines have no effect on the results of the search, since only PL/I on-units are searched for, unless one of them is a COBOL routine that has been compiled by a compiler that does not implement Americam National Standard COBOL or that was made available prior to Release 19 of System/360 Operating System. In these cases, the result of the search and the effects of the interrupt are undefined, and may include abnormal termination of the program.

GO TO Statement

The GO TO statement must not be used to transfer control across more than one interlanguage boundary, where an interlanguage boundary is defined as an invocation in which one routine calls another of a different language. Such transfers of control may be initiated inadvertently if the programmer uses a GO TO statement in an on-unit. (Note that entry to an on-unit is not considered as transferring control outside the block or routine in which the statement that caused the on-unit to be entered was executed; the on-unit may be regarded as being appended to the procedure or routine from which it is entered. This applies even if the on-unit is entered from a COBOL or FORTRAN routine). Consider the following example:

```
P:PROCEDURE;
   DECLARE LAB LABEL(L1,L2)
   EXTERNAL,
   FORT ENTRY OPTIONS(FORTRAN
   INTER);
   ON ERROR GO TO LAB;
      .
      .
   CALL FORT;
      .
      .
   L1:.......;
      .
      .
END P;

Q:PROCEDURE OPTIONS(FORTRAN);
   DECLARE LAB LABEL(L1,L2)
   EXTERNAL;
      .
      .
   L2:.......;
      .
      .
END Q;
```

Assume that the CALL FORT; statement is executed, and that FORT then calls Q. Assume further that an error occurs in Q which initiates entry to the on-unit established in P. At this stage control is still with procedure Q because the on-unit is regarded as being appended to the procedure from which it was entered. If LAB has the value L1, then the GO TO branch is in error because it transfers control back to procedure P and in doing so crosses the interlanguage boundaries between Q and FORT and between FORT and P. If LAB has the value L2, the GO TO is not in error because control remains in procedure Q. If an interrupt in FORT caused the on-unit to be entered before Q was called, then the GO TO would not have been in error, if LAB had the value L1: only one interlanguage boundary would be crossed, namely the FORTRAN-PL/I boundary between FORT and P. (LAB should not have the value L2 in this case because procedure Q is not active).

Termination of FORTRAN and COBOL Routines

A routine may be terminated by either executing a statement that terminates the whole program, or by handing control back to the calling routine.

The statements that terminate the whole program are STOP in FORTRAN and STOP RUN in COBOL. They are equivalent to the PL/I STOP statement. The effects of these statements are unchanged in a mixed language program; they still terminate the whole program.

If a FORTRAN STOP is executed in a routine that is within a PL/I environment, that environment is not ended in the normal way. If a COBOL STOP RUN is executed in a routine that is within a PL/I environment, that environment is ended in the normal way only if it includes the main routine of the program; otherwise the termination will be abnormal. The main difference, from the programmer's point of view, between a normal and an abnormal ending is that in the abnormal ending, open files in PI/I procedures are not closed. This could cause output data to be lost. Considering the example in figure 19.1, a STOP in PROC2 or a STOP RUN in PROC4 would not close any

files that may be open in PROC3, and a STOP in PROC6 would not close any files in PROC7.

A RETURN executed in a FORTRAN subroutine or function that is inside a PL/I environment and which returns control to a routine outside that environment (in other words, a RETURN statement in a FORTRAN routine that directly invokes a PL/I routine but which is not dynamically descendent from any PL/I routine), ends the PL/I environment and causes all files in dynamically descendent PL/I procedures to be closed. However, a RETURN statement in a FORTRAN main routine is effectively a STOP statement; control is passed to the operating system without any files being closed.

When a COBOL main routine that is within a PL/I environment passes control back to the operating system, the environment is ended normally.

Multitasking

A PL/I procedure cannot invoke a COBOL or a FORTRAN routine as a task, that is, the CALL statement must not specify the TASK, EVENT, or PRIORITY options.

Only one task of a PL/I program can have active COBOL or FORTRAN routines at any one time. If a PL/I program has more than one task active at the same time, then, if one of these tasks has invoked a COBOL or a FORTRAN routine, the programmer must ensure that the other tasks wait until control has returned to the PL/I program before another non-PL/I routine is invoked.

COBOL INTERFACE

Argument/parameter matching across a PL/I-COBOL interface requires a knowledge of the equivalence of data types and of data organization in the two languages. The PL/I equivalents of the COBOL data types are shown in figure 19.2. These are the PL/I data types that should appear in PL/I parameter descriptors associated with COBOL arguments or parameters respectively.

While a knowledge of the equivalent data types is sufficient for specifying COBOL items in terms of PL/I element variables, the specification of equivalent data aggregates (group items in COBOL, structures or arrays in PL/I) requires a knowledge of the data-organization descriptions of the two languages. The example given in figure 19.3 shows how a COBOL data aggregate is described in PL/I terms.

In COBOL, the OCCURS clause cannot appear more than three times in any one group-item description. This imposes a restriction on any PL/I array within a structure passed as an argument to a COBOL routine. Also, the OCCURS clause cannot appear on a level-01 entry. This precludes the use of a level-01 array in a PL/I structure passed to or from a COBOL routine.

A PL/I structure that contains an area or a bit-string variable should not be passed as an argument to a COBOL routine. If it is, a diagnostic message is produced and the structure is not automatically remapped.

A bit or character string with the VARYING attribute may be passed to a COBOL routine, although there is no equivalent attribute in COBOL. The address of the start of the two-byte length prefix is passed, so that the prefix consitutes the first two bytes of the COBOL string. Conversely, when COBOL data is passed to a PL/I string parameter with the VARYING attribute, the first two bytes of the argument form the parameter's length prefix.

COBOL				PL/I			
Data type	Length (bytes)	Alignment: Synch. (aligned)	Alignment: Unsynch. (unaligned)	Data Type	Length (bytes)	Alignment: Aligned	Alignment: Unaligned
COMPUTATIONAL[1] dec. length: 1-4	2	Halfword	Byte	FIXED BINARY(15,0) (halfword integer)	2	Halfword	Byte
5-9	4	Fullword	Byte	FIXED BINARY(31,0) (fullword integer)	4	Fullword	Byte
10-18	8	Fullword	Byte	No equivalent	-	-	-
COMPUTATIONAL-1	4	Fullword	Byte	FLOAT DEC(6) (short float)	4	Fullword	Byte
COMPUTATIONAL-2	8	Doubleword	Byte	FLOAT DEC(16) (long float)	8	Doubleword	Byte
COMPUTATIONAL-3	1[2]	Byte	Byte	FIXED DEC	1[2]	Byte	Byte
DISPLAY	any	Byte	Byte	CHARACTER	any	Byte	Byte

Notes: [1]Decimal length is equal to the number of 9s in the picture.

[2]The length of 1 byte applies to the smallest fixed decimal value (i.e., 1 digit). For other values, the length is given by CEIL((number of digits + 1)/2) bytes.

Figure 19.2. COBOL-PL/I data equivalents

```
01  A SYNCHRONIZED.                          1  A ALIGNED,
    02  B OCCURS 3 TIMES.                       2  B(3),
        03  C OCCURS 4 TIMES.                      3  C(4),
            04  D OCCURS 5 TIMES USAGE COMP-3          4  D(5) FIXED DECIMAL(7,3),
                PIC S9999V999.
    02  E USAGE DISPLAY.                        2  E,
        03  F PIC X(8).                            3  F CHAR(8),
        03  G PIC 9(8).                            3  G PIC '(8)9',
    02  DUMMY OCCURS 6 TIMES.                   2  H(6,7) FIXED BINARY (15,0);
        03  H OCCURS 7 TIMES USAGE COMP
                PIC S9999.
```

Figure 19.3. Declaration of a data aggregate in COBOL and PL/I

FORTRAN INTERFACE

Argument/parameter matching across a PL/I-FORTRAN interface, and the use of common storage for PL/I and FORTRAN variables, require a knowledge of the equivalence of data types and of data organizations in the two languages. The PL/I equivalents of the FORTRAN data types are shown in figure 19.4. These are the PL/I data types that should appear in PL/I parameters or parameter descriptors associated with FORTRAN arguments or parameters respectively, and in the declaration of STATIC EXTERNAL variables with the same names as FORTRAN COMMON blocks.

The specification of equivalent data aggregates in PL/I and FORTRAN is simpler than in PL/I and COBOL, as the only data aggregates that exist in FORTRAN are arrays. The problems arise when using non-connected unidimensional arrays or multidimensional arrays as PL/I arguments.

Generally, when passing arguments between PL/I and FORTRAN, the interlanguage facilities pass a unidimensional array directly to the invoked routine, without the creation of a dummy argument. However, if a PL/I unidimensional array in non-connected storage is passed as an argument to a FORTRAN routine, the interlanguage facilities create a dummy argument into which the unconnected array is mapped. The dummy is then passed as the argument. On return, the values in the dummy are assigned to the corresponding elements in the array.

A dummy argument is always created for a multidimensional array passed between PL/I and FORTRAN routines, unless the NOMAP option specified.

If a PL/I array of bit strings is passed as an argument to a FORTRAN routine, only 8 or 32 should be specified for the string lengths. If values other than these are specified, a diagnostic message is produced and the array is not automatically remapped. Similarly, only these lengths should be used for PL/I variables having storage common with FORTRAN variables.

FORTRAN			PL/I			
Data Type	Length (bytes)	Alignment[1]	Data Type	Length (bytes)	Alignment	
					Aligned	Unaligned
INTEGER*2	2	Halfword	REAL FIXED BINARY(15,0)	2	Halfword	Byte
INTEGER*4	4	Fullword	REAL FIXED BINARY(31,0)	4	Fullword	Byte
REAL*4	4	Fullword	REAL FLOAT DEC(6) (real short float)	4	Fullword	Byte
REAL*8	8	Doubleword	REAL FLOAT DEC(16) (real long float)	8	Doubleword	Byte
REAL*16	16	Doubleword	REAL FLOAT DEC(33) (real extended float)	16	Doubleword	Byte
COMPLEX*8	8	Fullword	COMPLEX FLOAT DEC(6) (complex short float)	8	Fullword	Byte
COMPLEX*16	16	Doubleword	COMPLEX FLOAT DEC(16) (complex long float)	16	Doubleword	Byte
COMPLEX*32	32	Doubleword	COMPLEX FLOAT DEC(33) (complex extended float)	32	Doubleword	Byte
LOGICAL*1	1	Byte	BIT(8)	1	Byte	Bit[2]
LOGICAL*4	4	Fullword	BIT(32)	4	Byte	Byte

Notes: [1]Generally FORTRAN data is held in main storage with these alignments. COMMON data, however, is always byte-aligned. This could cause a specification interrupt if the items in the COMMON area are not stored in order of decreasing stringency.
[2]The fact that the alignment required of unaligned bit strings is bit rather than byte does not affect PL/I-FORTRAN data interchange, since the FORTRAN string will always take up an integral number of bytes.

Figure 19.4. FORTRAN-PL/I data equivalents

PL/I Attribute	COBOL		FORTRAN	
	Argument	Parameter	Argument	Parameter
ALIGNED	0000	0000	0000	0000
AREA	Note 1	Note 1	Note 1	Note 1
BINARY	0000	0000	0000	0000
BIT	Note 1	Note 1	Note 2	Note 2
CHARACTER	0000	0000	0004	0004
COMPLEX	0004	0004	Note 4	Note 4
CONNECTED	0000	0000	0000	0000
CONTROLLED	0000	0012	0000	0012
DECIMAL	0000	0000	Note 3	Note 3
DEFINED	0000	-	0000	-
Dimension	Note 8	Note 8	0000	0000
ENTRY	0004	0004	0004	0004
EVENT	0004	0004	0004	0004
FILE	0004	0004	0004	0004
FIXED	0000	0000	0000	0000
FLOAT	0000	0000	0000	0000
LABEL	0004	0004	0004	0004
Non-connected	Note 5	0000	Note 5	0000
OFFSET	0004	0004	0004	0004
PICTURE	0000	0000	0004	0004
POINTER	0004	0004	0004	0004
Precision	Note 6	Note 6	Note 7	Note 7
REAL	0000	0000	0000	0000
Structure	0000	0000	Note 1	Note 1
TASK	0004	0004	0004	0004
UNALIGNED	Note 9	0000	Note 9	0000
VARYING	0004	0004	0004	0004

Figure 19.5 (Part 1 of 2). Return codes produced by PL/I data types

Notes

1. Checkout compiler:0004
   Optimizing compiler: 0008
   In both cases, creation of a dummy argument is suppressed

2. BIT(8) or BIT(32): 0000
   Any other length: 0008
   In latter case, creation of a dummy argument is suppressed.

3. FLOAT DECIMAL: 0000
   FIXED DECIMAL: 0004

4. FLOAT COMPLEX: 0000
   FIXED COMPLEX: 0008

5. If creation of temporary suppressed by NOMAP option: 0012
   If no NOMAP option: 0000

6. Variable is FIXED(p,0), or is short or long FLOAT:0000
   Variable is BINARY FIXED (p,q) with q¬=0, or is extended FLOAT: 0004

7. Variable is FLOAT, or is FIXED BINARY with precision (p,0): 0000
   Variable is FIXED DECIMAL, or is BINARY(p,q) with q¬=0: 0004

8. If item is element of a structure or is a minor structure: 0000
   All other cases: 0008

9. If argument is an aggregate and creation of temporary is suppressed by NOMAP, or if argument is scalar: 0012
   If argument is an aggregate and no NOMAP: 0000

Figure 19.5 (Part 2 of 2). Return codes produced by PL/I data types

RETURN CODES

Diagnostic messages are provided at compile time as an aid to debugging the program. The messages are classified according to the type of the information they provide. A numeric value, the return code, is associated with each class of message, as a guide to the severity of the error or possible error that has been diagnosed. The highest return code generated during compilation constitutes the return code of that compilation, and its value is printed on the compile-time listing. Diagnostic messages and return codes are described in the compilers' programmers' guides. The classes of message and corresponding return codes are as follows.

Informatory	return code 0000
Warning	return code 0004
Error	return code 0008
Severe error	return code 0012

If no messages are produced, a code of 0000 is returned.

A return code of 0000 indicates that the compiler found no possible sources of error. A code of 0004 indicates that execution will probably be successful. A code of 0008 indicates that an error has been found but that execution nevertheless might be successful. A code of 0012 indictes that execution will probably not be successful.

As part of the interlanguage facilities of PL/I, diagnostic messages are produced, and the return code set appropriately, if the programmer specifies arguments or parameters whose attributes are such that errors may occur at execution time. The compiler will never prevent data being passed, nor will it attempt to correct errors; although it produces messages to indicate likely sources of error to the programmer, it will always allow him to attempt to pass any type of data he specifies.

Figure 19.5 shows the return codes generated by various types of PL/I data.

Part 11: Rules and Syntactic Descriptions

Section A: Syntax Notation

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

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

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

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

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

   b. Either all lower-case letters or a combination of lower-case and upper-case letters. In the latter case, there must be one portion in all lower-case letters and one portion in all upper-case letters, and the two portions must be separated by a hyphen.

   All such variables used are defined in the manual either syntactically, using this notation, or are defined semantically. For example:

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

   b. file-expression. This denotes the occurrence of a reference to a file.

   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. For example:

   ```
   DECLARE identifier FIXED;
   ```

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

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

   a. A single notation variable or notation constant.

   b. Any collection of notation variables, notation constants, syntax-language symbols, and keywords surrounded by braces or brackets.

4. Braces {} are used to denote grouping of more than one element into a syntactic unit.

   Example:

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

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

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

   Example:

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

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

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

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

This denotes the occurrence of either a literal asterisk or the variable "upper-bound," but not both. If "upper-bound" appears, it can optionally be preceded by the syntactic unit composed of the variable "lower-bound" and the literal colon.

7. Three dots ... denote the occurrence of the immediately preceding syntactic unit one or more times in succession. For example:

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

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

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

```
operand {&|⊥} operand
```

This denotes that the two occurrences of the variable "operand" are separated by either an "and" (&) or an "or" (|). The operator | is underlined to indicate that it is an "or" symbol in the PL/I language rather than an "or" symbol in the syntax language.

Section B: Character Sets with EBCDIC and Card-punch Codes

60-CHARACTER SET

Character	Card-Punch	8-Bit EBCDIC Code
blank	no punches	0100 0000
.	12-8-3	0100 1011
<	12-8-4 (12-8-6)	0100 1100
(	12-8-5 (0-8-4)	0100 1101
+	12-8-6 (12)	0100 1110
|	12-8-7 (NA)	0100 1111
&	12 (NA)	0101 0000
$	11-8-3	0101 1011
*	11-8-4	0101 1100
)	11-8-5 (12-8-4)	0101 1101
;	11-8-6	0101 1110
¬	11-8-7 (NA)	0101 1111
-	11	0110 0000
/	0-1	0110 0001
,	0-8-3	0110 1011
%	0-8-4 (NA)	0110 1100
_	0-8-5 (NA)	0110 1101
>	0-8-6 (8-6)	0110 1110
?	0-8-7 (12-0)	0110 1111
:	8-2 (8-5)	0111 1010
#	8-3 (NA)	0111 1011
@	8-4 (NA)	0111 1100
'	8-5 (8-4)	0111 1101
=	8-6 (8-3)	0111 1110
A	12-1	1100 0001
B	12-2	1100 0010
C	12-3	1100 0011
D	12-4	1100 0100
E	12-5	1100 0101
F	12-6	1100 0110
G	12-7	1100 0111
H	12-8	1100 1000
I	12-9	1100 1001
J	11-1	1101 0001
K	11-2	1101 0010
L	11-3	1101 0011
M	11-4	1101 0100
N	11-5	1101 0101
O	11-6	1101 0110
P	11-7	1101 0111
Q	11-8	1101 1000
R	11-9	1101 1001
S	0-2	1110 0010
T	0-3	1110 0011
U	0-4	1110 0100
V	0-5	1110 0101
W	0-6	1110 0110
X	0-7	1110 0111
Y	0-8	1110 1000
Z	0-9	1110 1001
0	0	1111 0000
1	1	1111 0001
2	2	1111 0010
3	3	1111 0011
4	4	1111 0100
5	5	1111 0101
6	6	1111 0110
7	7	1111 0111
8	8	1111 1000
9	9	1111 1001

Composite Symbols	Card-Punch
<=	12-8-4, 8-6 (12-8-6, 8-3)
||	12-8-7, 12-8-7 (NA)
**	11-8-4, 11-8-4
¬<	11-8-7, 12-8-4 (NA)
¬>	11-8-7, 0-8-6 (NA)
¬=	11-8-7, 8-6 (NA)
>=	0-8-6, 8-6 (8-6, 8-3)
/*	0-1, 11-8-4
*/	11-8-4, 0-1
->	11, 0-8-6 (11, 8-6)

The card-punch codes given in brackets are BCDIC codes that differ from the corresponding EBCDIC codes. NA indicates that the symbol has no representation in BCDIC. BCDIC codes can be used in the PL/I source program provided that the BCD compiler option is specified. No BCDIC 8-bit codes are given here since, when the BCD option is specified, all BCDIC codes in the source program are converted by the compiler into EBCDIC. Note that although the full PL/I 60-character set is not available in BCDIC, all the 48-character set (see next page) is available, so if the 48-character-set option is specified as well as the BCD option, all PL/I operations can be performed.

48-CHARACTER SET

Character	Card-Punch	8-Bit EBCDIC Code
blank	no punches	0100 0000
.	12-8-3	0100 1011
(	12-8-5 (0-8-4)	0100 1101
+	12-8-6 (12)	0100 1110
$	11-8-3	0101 1011
*	11-8-4	0101 1100
)	11-8-5 (12-8-4)	0101 1101
-	11	0110 0000
/	0-1	0110 0001
,	0-8-3	0110 1011
'	8-5 (8-4)	0111 1101
=	8-6 (8-3)	0111 1110
A	12-1	1100 0001
B	12-2	1100 0010
C	12-3	1100 0011
D	12-4	1100 0100
E	12-5	1100 0101
F	12-6	1100 0110
G	12-7	1100 0111
H	12-8	1100 1000
I	12-9	1100 1001
J	11-1	1101 0001
K	11-2	1101 0010
L	11-3	1101 0011
M	11-4	1101 0100
N	11-5	1101 0101
O	11-6	1101 0110
P	11-7	1101 0111
Q	11-8	1101 1000
R	11-9	1101 1001
S	0-2	1110 0010
T	0-3	1110 0011
U	0-4	1110 0100
V	0-5	1110 0101
W	0-6	1110 0110
X	0-7	1110 0111
Y	0-8	1110 1000
Z	0-9	1110 1001
0	0	1111 0000
1	1	1111 0001
2	2	1111 0010
3	3	1111 0011
4	4	1111 0100
5	5	1111 0101
6	6	1111 0110
7	7	1111 0111
8	8	1111 1000
9	9	1111 1001

Composite Symbols	Card Punch	60-Character Set Equivalent
..	12-8-3, 12-8-3	:
LE	11-3, 12-5	<=
CAT	12-3, 12-1, 0-3	||
**	11-8-4, 11-8-4	**
NL	11-5, 11-3	¬<
NG	11-5, 12-7	¬>
NE	11-5, 12-5	¬=
//	0-1, 0-1	%
,.	0-8-3, 12-8-3	;
AND	12-1, 11-5, 12-4	&

Composite Symbols	Card Punch	60-Character Set Equivalent
GE	12-7, 12-5	>=
GT	12-7, 0-3	>
LT	11-3, 0-3	<
NOT	11-5, 11-6, 0-3	¬
OR	11-6, 11-9	|
/*	0-1, 11-8-4	/*
*/	11-8-4, 0-1	*/
PT	11-7, 0-3	->

The card-punch codes given in brackets are BCDIC codes that differ from the corresponding EBCDIC codes. BCDIC codes can be used in the PL/I source program provided that the BCD compiler option is specified. No BCDIC 8-bit codes are given here since, when the BCD option is specified, all BCDIC codes in the source program are converted by the compiler into EBCDIC.

Note: When using the 48-character set, the following rules should be observed:

1. The two periods that replace the colon must be immediately preceded by a blank if the preceding character is a period.

2. The two slashes that replace the percent symbol must be immediately preceded by a blank if the preceding character is an asterisk, or immediately followed by a blank if the following character is an asterisk.

3. The sequence "comma period" represents a semicolon except when it occurs in a comment, a character string, or a picture specification or when it is immediately followed by a digit.

4. If the compiler option that specifies the 48-character set is included in the compilation, 60-character set symbols may be freely intermixed with 48-character set symbols and will be accepted by the compiler as valid input.

5. 48-character set "reserved" words (e.g., GT,LE,CAT, etc.,) must be preceded and followed by a blank or a comment. If they are not, the interpretation by the compiler is undefined and may not therefore, be what the user intended.

   A record containing part or all of a 48-character set reserved word must be 3 characters or more in length.

Section C: Keywords and Keyword Abbreviations

Keyword	Abbreviation	Use of Keyword
A[(w)]		format item
ABS(x)		built-in function
ACOS(x)		built-in function
%ACTIVATE	%ACT	preprocessor statement
ADD(x_1,x_2,x_3[,x_4])		built-in function
ADDBUFF		option of ENVIRONMENT attribute
ADDR(x)		built-in function
ALIGNED		attribute
ALL [(character-string-expression)]		option of PUT statement
ALL(x)		built-in function
ALLOCATE	ALLOC	statement
ALLOCATION(x)	ALLOCN(x)	built-in function
ANY(x)		built-in function
AREA		condition
AREA[(size)]		attribute
ARGn		option of NOMAP, NOMAPIN, NOMAPOUT options
ASCII		option of the ENVIRONMENT attribute
ASIN(x)		built-in function
ATAN(x_1[,x_2])		built-in function
ATAND(x_1[,x_2])		built-in function
ATANH(x)		built-in function
ATTENTION	ATTN	condition
AUTOMATIC	AUTO	attribute
B[(w)]		format item
BACKWARDS		attribute, option of OPEN statement
BASED[(locator-expression)]		attribute
BEGIN		statement
BINARY	BIN	attribute
BINARY(x_1[,x_2[,x_3]])	BIN(x_1[,x_2[,x_3]])	built-in function
BIT[(length)]		attribute
BIT(x_1[,x_2])		built-in function
BLKSIZE(block-size)		option of ENVIRONMENT attribute
BOOL(x_1,x_2,x_3)		built-in function
BUFFERED	BUF	attribute
BUFFERS(n)		option of ENVIRONMENT attribute
BUFOFF[(n)]		option of ENVIRONMENT attribute
BUILTIN		attribute
BY		option of DO statement, option of repetitive input/output specification
BY NAME		option of assignment statement
C(real-format-item [,real-format-item])		format item
CALL		statement, option of INITIAL attribute
CEIL(x)		built-in function
CHAR(x_1[,x_2])		built-in function
CHARACTER[(length)]	CHAR[(length)]	attribute
CHECK		statement
CHECK[(name-list)]		condition, condition prefix
CLOSE		statement
COBOL		option of ENVIRONMENT attribute, or OPTIONS option/attribute
COLUMN(n)	COL(n)	FORMAT ITEM
COMPLETION(x)	CPLN(x)	built-in function, pseudovariable
COMPLEX	CPLX	attribute
COMPLEX(x_1,x_2)	CPLX(x_1,x_2)	built-in function, pseudovariable
CONDITION	COND	attribute
CONDITION(name)	COND(name)	condition
CONJG(x)		built-in function
CONNECTED	CONN	attribute
CONSECUTIVE		option of ENVIRONMENT attribute

Keyword	Abbreviation	Use of Keyword
%CONTROL		listing control statement
CONTROLLED	CTL	attribute
CONVERSION	CONV	condition, condition prefix
COPY[(file-expression)]		option of GET statement
COS(x)		built-in function
COSD(x)		built-in function
COSH(x)		built-in function
COUNT(file-expression)		built-in function
CTLASA		option of ENVIRONMENT attribute
CTL360		option of ENVIRONMENT attribute
D		option of ENVIRONMENT attribute
DATA		option of GET or PUT statement
DATAFIELD		built-in function
DATE		built-in function
DB		option of ENVIRONMENT attribute
%DEACTIVATE	%DEACT	preprocessor statement
DECIMAL	DEC	attribute
DECIMAL(x_1[,x_2[,x_3]])	DEC(x_1[,x_2[,x_3]])	built-in function
DECLARE	DCL	statement
%DECLARE	%DCL	preprocessor statement
DEFAULT	DFT	statement
DEFINED	DEF	attribute
DELAY		statement
DELETE		statement
DESCRIPTORS		option of DEFAULT statement
DIM(x_1,x_2)		built-in function
DIRECT		attribute
DISPLAY		statement
DIVIDE(x_1,x_2,x_3[,x_4])		built-in function
DO		statement, repetitive input/output data specification
%DO		preprocessor statement
E(w,d[,s])		format item
EDIT		option of GET or PUT statement
ELSE		clause of IF statement
%ELSE		clause of %IF statement
EMPTY		built-in function
END		statement
%END		preprocessor statement
ENDFILE(file-expression)		condition
ENDPAGE(file-expression)		condition
ENTRY		attribute, statement
ENVIRONMENT	ENV	attribute, option of CLOSE statement
ERF(x)		built-in function
ERFC(x)		built-in function
ERROR		condition
EVENT		attribute, option of CALL, DELETE, DISPLAY, READ, REWRITE, and WRITE statements
EXCLUSIVE	EXCL	attribute
EXIT		statement
EXP(x)		built-in function
EXTERNAL	EXT	attribute
F(w,[,d[,s]])		format item
F		option of ENVIRONMENT attribute
FB		option of ENVIRONMENT attribute
FBS		option of ENVIRONMENT attribute
FETCH		statement
FILE		attribute
FILE(file-expression)		option of CLOSE, DELETE, GET, LOCATE, OPEN, PUT, READ, REWRITE, UNLOCK, and WRITE statements
FINISH		condition
FIXED		attribute
FIXED(x_1[,x_2[,x_3]])		built-in function
FIXEDOVERFLOW	FOFL	condition, condition prefix
FLOAT		attribute

Keyword	Abbreviation	Use of Keyword
FLOAT(x_1[,x_2])		built-in function
FLOOR(x)		built-in function
FLOW		statement, option of PUT statement
FORMAT		statement, option of %CONTROL statement
FORTRAN		option of OPTIONS option/attribute
FREE		statement
FROM(variable)		option of WRITE or REWRITE statements
FS		option of ENVIRONMENT attribute
GENERIC		attribute
GENKEY		option of ENVIRONMENT attribute
GET		statement
GO TO	GOTO	statement
%GO TO	%GOTO	preprocessor statement
HALT		statement
HBOUND(x_1,x_2)		built-in function
HIGH(x)		built-in function
IF		statement
%IF		preprocessor statement
IGNORE(n)		option of READ statement
IMAG(x)		built-in function, pseudovariable
IN(element-area-variable)		option of ALLOCATE and FREE statements
%INCLUDE		preprocessor statement
INDEX(x_1,x_2)		built-in function
INDEXAREA [(index-area-size)]		option of ENVIRONMENT attribute
INDEXED		option of ENVIRONMENT attribute
INITIAL	INIT	attribute
INPUT		attribute, option of OPEN statement
INTER		option of OPTIONS option/attribute
INTERNAL	INT	attribute
INTO(variable)		option of READ statement
IRREDUCIBLE	IRRED	attribute
KEY(file-expression)		condition
KEY(x)		option of READ, DELETE, and REWRITE statements
KEYED		attribute, option of OPEN statement
KEYFROM(x)		option of WRITE statement
KEYLENGTH(n)		option of ENVIRONMENT attribute
KEYLOC(n)		option of ENVIRONMENT attribute
KEYTO(variable)		option of READ statement
LABEL		attribute
LBOUND(x_1,x_2)		built-in function
LEAVE		option of ENVIRONMENT attribute
LENGTH(x)		built-in function
LIKE		attribute
LINE(n)		format item, option of PUT statement
LINENO(x)		built-in function
LINESIZE(expression)		option of OPEN statement
LIST		option of GET or PUT statement
LOCATE		statement
LOG(x)		built-in function
LOG2(x)		built-in function
LOG10(x)		built-in function
LOW(x)		built-in function
MAIN		option of OPTIONS option
MAX(x_1,x_2...x_n)		built-in function
MIN(x_1,x_2...x_n)		built-in function
MOD(x_1,x_2)		built-in function
MULTIPLY(x_1,x_2,x_3[,x_4])		built-in function
NAME(file-expression)		condition
NCP(n)		option of ENVIRONMENT attribute
NOCHECK		statement
NOCHECK[(name-list)]		condition prefix
NOCONVERSION	NOCONV	condition prefix
NOFIXEDOVERFLOW	NOFOFL	condition prefix
NOFLOW		statement
NOFORMAT		option of %CONTROL statement

Keyword	Abbreviation	Use of Keyword
NOLOCK		option of READ statement
NOMAP[(arg-list)]		option of OPTIONS attribute
NOMAPIN[(arg-list)]		option of OPTIONS attribute
NOMAPOUT[(arg-list)]		option of OPTIONS attribute
NOOVERFLOW	NOOFL	condition prefix
NORESCAN		option of %ACTIVATE statement
NOSIZE		condition prefix
NOSTRINGRANGE	NOSTRG	condition prefix
NOSTRINGSIZE	NOSTRZ	condition prefix
NOSUBSCRIPTRANGE	NOSUBRG	condition prefix
NOUNDERFLOW	NOUFL	condition prefix
NOWRITE		option of ENVIRONMENT attribute
NOZERODIVIDE	NOZDIV	condition prefix
NULL		built-in function
OFFSET[(area-name)]		attribute
OFFSET(x_1,x_2)		built-in function
ON		statement
ONCHAR		built-in function, pseudovariable
ONCODE		built-in function
ONCOUNT		built-in function
ONFILE		built-in function
ONKEY		built-in function
ONLOC		built-in function
ONSOURCE		built-in function, pseudovariable
OPEN		statement
OPTIONS(list)		attribute, option of ENTRY and PROCEDURE statements
ORDER		option of BEGIN and PROCEDURE statements
OUTPUT		attribute, option of OPEN statement
OVERFLOW	OFL	condition, condition prefix
P 'picture specification'		format item
PAGE		format item, option of PUT statement
%PAGE		listing control statement
PAGESIZE(w)		option of OPEN statement
PENDING(file-expression)		condition
PICTURE	PIC	attribute
POINTER	PTR	attribute
POINTER(x_1,x_2)	PTR(x_1,x_2)	built-in function
POLY(x_1,x_2)		built-in function
POSITION (expression)	POS (expression)	attribute
PRECISION(x_1,x_2[,x_3])	PREC(x_1,x_2[,x_3])	built-in function
PRINT		attribute, option of OPEN statement
PRIORITY(x)		option of CALL statement
PRIORITY[(x)]		built-in function, pseudovariable
PROCEDURE	PROC	statement
%PROCEDURE	%PROC	preprocessor statement
PROD(x)		built-in function
PUT		statement
R(x)		format item
RANGE		option of DEFAULT statement
READ		statement
REAL		attribute
REAL(x)		built-in function, pseudovariable
RECORD		attribute, option of OPEN statement
RECORD(file-expression)		condition
RECSIZE(record-length)		option of ENVIRONMENT attribute
RECURSIVE		option of PROCEDURE statement
REDUCIBLE	RED	attribute
REENTRANT		option of OPTIONS option
REFER(element-variable)		option of BASED attribute
REGIONAL(1|2|3)		option of ENVIRONMENT attribute
RELEASE		statement
REORDER		option of BEGIN and PROCEDURE statements
REPEAT(x_1,x_2)		built-in function
REPLY(c)		option of DISPLAY statement
REREAD		option of ENVIRONMENT attribute
RESCAN		option of %ACTIVATE statement

Keyword	Abbreviation	Use of Keyword
RETURN		statement, preprocessor statement
RETURNS(attribute-list)		attribute, option of PROCEDURE statement
REVERT		statement
REWRITE		statement
ROUND(x_1,x_2)		built-in function
SCALARVARYING		option of ENVIRONMENT attribute
SEQUENTIAL	SEQL	attribute
SET(locator-variable)		option of ALLOCATE, LOCATE, and READ statements
SIGN(x)		built-in function
SIGNAL		statement
SIN(x)		built-in function
SIND(x)		built-in function
SINH(x)		built-in function
SIZE		condition
SKIP[(n)]		format item, option of GET and PUT statements
%SKIP		listing control statement
SNAP		option of ON and PUT statements
SQRT(x)		built-in function
STATIC		attribute
STATUS(x)		built-in function, pseudovariable
STOP		statement
STREAM		attribute, option of OPEN statement
STRING(x)		built-in function, pseudovariable
STRING(string-name)		option of GET and PUT statements
STRINGRANGE	STRG	condition, condition prefix
STRINGSIZE	STRZ	condition, condition prefix
iSUB		dummy variable of DEFINED attribute
SUBSCRIPTRANGE	SUBRG	condition, condition prefix
SUBSTR(x_1,x_2[,x_3])		built-in function, pseudovariable
SUM(x)		built-in function
SYSIN		name of standard system input file
SYSPRINT		name of standard system output file
SYSTEM		option of ON or DECLARE statements
TAN(x)		built-in function
TAND(x)		built-in function
TANH(x)		built-in function
TASK		attribute, option of OPTIONS option
TASK[(task-name)]		option of CALL statement
THEN		clause of IF statement
%THEN		clause of %IF statement
TIME		built-in function
TITLE(element-expression)		option of OPEN statement
TO		option of DO statement, option of repetitive input/output specification
TOTAL		option of ENVIRONMENT attribute
TP(M|R)		option of ENVIRONMENT attribute
TRANSIENT		attribute
TRANSLATE(x_1,x_2[,x_3])		built-in function
TRANSMIT(file-expression)		condition
TRKOFL		option of ENVIRONMENT attribute
TRUNC(x)		built-in function
U		option of ENVIRONMENT attribute
UNALIGNED	UNAL	attribute
UNBUFFERED	UNBUF	attribute, option of OPEN statement
UNDEFINEDFILE (file-expression)	UNDF (file-expression)	condition
UNDERFLOW	UFL	condition, condition prefix
UNLOCK		statement
UNSPEC(x)		built-in function, pseudovariable
UPDATE		attribute, option of OPEN statement
V		option of ENVIRONMENT attribute
VALUE		option of DEFAULT statement
VARIABLE		attribute
VARYING	VAR	attribute
VB		option of ENVIRONMENT attribute

Keyword	Abbreviation	Use of Keyword
VBS		option of ENVIRONMENT attribute
VERIFY(x_1,x_2)		built-in function
VS		option of ENVIRONMENT attribute
WAIT		statement
WHEN(generic-descriptor-list)		option in GENERIC declaration
WHILE		option of DO statement
WRITE		statement
X(w)		format item
ZERODIVIDE	ZDIV	condition, condition prefix

Section D: Picture Specification Characters

Picture specification characters appear in either the PICTURE attribute or the P-format item for edit-directed input and output. In either case, an individual character has the same meaning. A discussion of the concepts of picture specifications appears in chapter 13, "Editing and String Handling".

Picture characters are used to describe the attributes of the associated data item, whether it is the value of a variable or a data item to be transmitted between the program and external storage.

A picture specification always describes a character representation that is either a character-string data item or a numeric character data item. A character-string pictured item is one that can consist of alphabetic characters, decimal digits, and blanks. A numeric character pictured item is one in which the data itself can consist only of decimal digits, a decimal point, the letter E, and, optionally, a plus or minus sign. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified, but they are not a part of the arithmetic value of the numeric character variable, although the characters are stored with the digits and are considered to be part of the character-string value of the variable. Under the optimizing compiler, the maximum length of character string pictured data that is guaranteed to be handled is 1023, though longer items may be handled if sufficient storage is available. The checkout compiler will accept items of length not exceeding 32767 characters. Under both compilers, the maximum length of numeric character item is 255.

Arithmetic data assigned to a numeric character variable is converted to character representation. Editing, such as zero suppression and the insertion of other characters, can be specified for a numeric character data item.

Data assigned to a variable declared with a numeric picture specification (or data to be written with a numeric picture format item) must be either internal coded arithmetic data or data that can be converted to coded arithmetic. Thus, assigned data can contain only digits and, optionally, a decimal point, a sign and the exponent delimeter E. It should not contain any editing characters, for example, a currency symbol; if it does, the CONVERSION condition is raised.

Numeric character data to be read using the P-format item must conform to the specification contained in the P-format item, including editing characters. If the indicated character does not appear in the input stream, the CONVERSION condition is raised.

Data assigned to a variable declared with a character-string picture specification (or data to be written with a character-string picture format item) should conform, character by character (or be convertible, character by character) to the picture specification; if it does not, the CONVERSION condition is raised. Character string data read in using the P format item must conform to the specification given in the format item. If the indicated character does not appear in the input stream, the CONVERSION condition is raised.

Figures in this section illustrate how different picture specifications affect the representation of values when assigned to a pictured variable or when printed using the P-format item. Each figure shows the original value of the data, the attributes of the variable from which it is assigned (or written), the picture specification, and the character-string value of the numeric character or pictured character-string variable.

Picture Characters for Character-string Data

Only three picture characters can be used in character-string picture specifications:

X specifies that the associated position can contain any character whose internal bit configuration can be recognized by the computer in use.

A specifies that the associated position can contain any alphabetic character or a blank character.

9 specifies that the associated position can contain any decimal digit or a blank character.

A character picture specification must contain at least one A or X.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
CHARACTER(5)	'9B/2L'	XXXXX	9B/2L
CHARACTER(5)	'9B/2L'	XXX	9B/
CHARACTER(5)	'9B/2L'	XXXXXXX	9B/2Lbb
CHARACTER(5)	'ABCDE'	AAAAA	ABCDE
CHARACTER(5)	'ABCDE'	AAAAAA	ABCDEb
CHARACTER(5)	'ABCDE'	AAA	ABC
CHARACTER(5)	'12/34'	99X99	12/34
CHARACTER(5)	'L26.7'	A99X9	L26.7

[1]A variable declared with a character-string picture specification has a character-string value only.

Figure D.1. Pictured character-string examples

Figure D.1 gives examples of character-string picture specifications. In the figure, the letter b indicates a blank character. Note that assignments are left-adjusted, and any necessary padding with blanks is on the right.

Picture Characters for Numeric Character Data

Numeric character data must represent numeric values; therefore, the associated picture specification cannot contain the characters X or A. The picture characters for numeric character data can specify detailed editing of the data.

A numeric character variable can be considered to have two different kinds of value, depending upon its use. They are (1) its arithmetic value and (2) its character-string value.

The arithmetic value is the value expressed by the decimal digits of the data item, the assumed location of a decimal point, and possibly a sign. The arithmetic value of a numeric character variable is used whenever the variable appears in an expression that results in a coded arithmetic value or whenever the variable is assigned to a coded arithmetic, numeric character, or bit-string variable. In such cases, the arithmetic value of the numeric character variable is converted to internal coded arithmetic representation.

The character-string value is the value expressed by the decimal digits of the data item, as well as all of the editing and insertion characters appearing in the picture specification. The character-string value does not, however, include the assumed location of a decimal point, as specified by the picture character V. The character-string value of a numeric character variable is used whenever the variable appears in a character-string expression operation or in an assignment to a character-string variable, whenever the data is printed using list-directed or data-directed output, or whenever a reference is made to a character-string variable that is defined on the numeric character variable. In such cases, no data conversion is necessary.

The picture characters for numeric character specifications may be grouped into the following categories:

- Digit and Decimal-Point Specifiers
- Zero Suppression Characters
- Insertion Characters
- Signs and Currency Symbol
- Credit, Debit, and Overpunched Signs
- Exponent Specifiers
- Scaling Factor

A numeric character specification consists of one or more fields, each field

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(5)	12345	99999	12345
FIXED(5)	12345	99999V	12345
FIXED(5)	12345	999V99	34500[2]
FIXED(5)	12345	V99999	00000[2]
FIXED(7)	1234567	99999	34567[2]
FIXED(3)	123	99999	00123
FIXED(5,2)	123.45	999V99	12345
FIXED(7,2)	12345.67	9V9	56[2]
FIXED(5,2)	123.45	99999	00123

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.
[2]In this case, PL/I does not define the result since significant digits have been truncated on the left. The result shown, however, is that given for these implementations. The SIZE condition will be raised, if enabled.

Figure D.2. Pictured numeric character examples

describing a fixed-point number. A floating-point specification has two fields - one for the mantissa and one for the exponent. A field may be divided into subfields by inserting a V picture specification character; the portion preceding the V and that following it (if any) are subfields of the specification.

A major requirement of the picture specification for numeric character data is that each field must contain at least one picture character that specifies a digit position. This picture character, however, need not be the digit character 9. Other picture characters, such as the zero suppression characters (Z or * or Y), also specify digit positions. At least one of these characters must be used to define a numeric character specification.

DIGIT AND DECIMAL-POINT SPECIFIERS

The picture characters 9 and V are used in the simplest form of numeric character specifications that represent fixed-point decimal values.

9 specifies that the associated position in the data item is to contain a decimal digit.

V specifies that a decimal point is assumed at this position in the associated data item. However, it does not specify that an actual decimal point is to be inserted. The integer and fractional parts of the assigned value are aligned on the V character; therefore, an assigned value may be truncated or extended with zero digits at either end. (Note that if significant digits are truncated on the left, the result is undefined and a SIZE interrupt will occur, if SIZE is enabled.) If no V character appears in the picture specification of a fixed-point decimal value (or in the first field of a picture specification of a floating-point decimal value), a V is assumed at the right end of the field specification. This can cause the assigned value to be truncated, if necessary, to an integer. The V character cannot appear more than once in a picture specification.

Figure D.2 gives examples of numeric character specifications.

ZERO SUPPRESSION CHARACTERS

The zero suppression picture characters specify conditional digit positions in the character-string value and may cause

leading zeros to be replaced by asterisks or blanks and nonleading zeros to be replaced by blanks. Leading zeros are those that occur in the leftmost digit positions of fixed-point numbers or in the leftmost digit positions of the two parts of floating-point numbers, that are to the left of the assumed position of a decimal point, and that are not preceded by any of the digits 1 through 9. The leftmost nonzero digit in a number and all digits, zeros or not, to the right of it represent significant digits. Note that a floating-point number can also have leading zeros in the exponent field.

Figure D.3 gives examples of the use of zero suppression characters. In the figure, the letter b indicates a blank character.

Z specifies a conditional digit position and causes a leading zero in the associated data position to be replaced by a blank character. When the associated data position does not contain a leading zero, the digit in the position is not replaced by a blank character. The picture character Z cannot appear in the same field as the picture character * or a drifting character, nor can it appear to the right of any of the picture characters 9, T, I, R, or Y in a field.

* specifies a conditional digit position. It is used the way the picture character Z is used, except that leading zeros are replaced by asterisks. The picture character * cannot appear in the same subfield as the picture character Z or a drifting character, nor can it appear to the right of any of the picture characters 9, T, I, R, or Y in a field.

Y specifies a conditional digit position and causes a zero digit, leading or nonleading, in the associated position to be replaced by a blank character. When the associated position does not contain a zero digit, the digit in the position is not replaced by a blank character.

Figure D.3 gives examples of the use of zero suppression characters. In the figure, the letter b indicates a blank character.

Note: If one of the picture characters Z or * appears to the right of the picture character V, then all fractional digit positions in the specification, as well as all integer digit positions, must employ the Z or * picture character, respectively. When all digit positions to the right of the picture character V contain zero suppression picture characters, fractional zeros of the value are suppressed only if all positions in the fractional part contain zeros and all integer positions have been suppressed. The entire character-string value of the data item will then consist of blanks or asterisks. No digits in the fractional part are replaced by blanks or asterisks if the fractional part contains any significant digit.

INSERTION CHARACTERS

The picture characters comma (,), point (.), slash (/), and blank (B) are insertion characters; they cause the specified character to be inserted into the associated position of the numeric character data. They do not indicate digit or character positions, but are inserted between digits or characters. Each does, however, actually represent a character position in the character-string value, whether or not the character is suppressed. The comma, point, and slash are conditional insertion characters; within a string of zero suppression characters, they, too, may be suppressed. The blank (B) is an unconditional insertion character; it always specifies that a blank is to appear in the associated position.

Note: Insertion characters are applicable only to the character-string value. They specify nothing about the arithmetic value of the data item.

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)	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(5,2)	000.08	ZZZVZ9	bbb08
FIXED(5,2)	000.00	ZZZVZZ	bbbbb
FIXED(5)	00100	*****	**100
FIXED(5)	00000	*****	*****
FIXED(5,2)	000.01	***V**	***01
FIXED(5)	00100	YYYYY	bb1bb
FIXED(5)	10203	9Y9Y9	1b2b3
FIXED(5,2)	000.04	YYYVY9	bbbb4

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

[2]If SIZE is enabled, it would be raised in this case, and the result would be as shown. If SIZE is not enabled, the result is undefined.

Figure D.3. Examples of zero suppression

causes a comma to be inserted into the associated position of the numeric character data when no zero suppression occurs. If zero suppression does occur, the comma is inserted only when an unsuppressed digit appears to the left of the comma position, or when a V appears immediately to the left of it and the fractional part contains any significant digits, or when the comma is at the start of a string or is preceded only by characters not specifying digit positions. In all other cases where zero suppression occurs, the comma insertion character is treated as though it were a zero suppression character identical to the one immediately preceding it.

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 elsewehre. The point (or the comma or slash) can be used in conjunction with the V to cause insertion of the point (or comma or slash) in the position that delimits the end of the integer portion in and the beginning of the fractional portion of a fixed-point (or floating-point) number,

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(4)	1234	9,999	1,234
FIXED(6,2)	1234.56	9,999V.99	1,234.56
FIXED(4,2)	12.34	ZZ.VZZ	12.34
FIXED(4,2)	00.03	ZZ.VZZ	bbb03
FIXED(4,2)	00.03	ZZV.ZZ	bb.03
FIXED(4,2)	12.34	ZZV.ZZ	12.34
FIXED(4,2)	00.00	ZZV.ZZ	bbbbb
FIXED(9,2)	1234567.89	9,999,999.V99	1,234,567.89
FIXED(7,2)	12345.67	**,999V.99	12,345.67
FIXED(7,2)	00123.45	**,999V.99	***123.45
FIXED(9,2)	1234567.89	9.999.999V,99	1.234.567,89
FIXED(6)	123456	99/99/99	12/34/56
FIXED(6)	123456	99.9/99.9	12.3/45.6
FIXED(6)	001234	ZZ/ZZ/ZZ	bbb12/34
FIXED(6)	000012	ZZ/ZZ/ZZ	bbbbbb12•
FIXED(6)	000000	ZZ/ZZ/ZZ	bbbbbbbb
FIXED(6)	000000	**/**/**	********
FIXED(6)	000000	**B**B**	**b**b**
FIXED(6)	123456	99B99B99	12b34b56
FIXED(3)	123	9BB9BB9	1bb2bb3
FIXED(2)	12	9BB/9BB	1bb/2bb

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D.4. Examples of insertion characters

as might be desired in printing, since the V does not cause printing of a point. The point must immediately precede or immediately follow the V. If the point precedes the V, it will be inserted only if an unsuppressed 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 significant fractional digits (along with any intervening zeros).

is used the same way the comma picture character is used, except that a slash (/) is inserted in the associated position.

B specifies that a blank character always be inserted into the associated position of the character-string value of the numeric character data.

Figure D.4 gives examples of the use of insertion character. In the figure, the letter b indicates a blank character.

SIGNS AND CURRENCY SYMBOL

The picture characters S, +, and - specify signs in numeric character data. The picture character $ specifies a currency symbol in the character-string value of numeric character data.

These picture characters may be used in either a static or a drifting manner. The static use specifies that a sign, a currency symbol, or a blank always appears in the associated position. The drifting use specifies that leading zeros are to be suppressed. In this case, the rightmost suppressed position associated with the picture character will contain a sign, a blank, or a currency symbol (except that where all digit positions are occupied by drifting characters and the value of the data item is zero, the drifting character is not inserted).

A drifting character is specified by multiple use of that character in a picture field. Thus, if a field contains one currency symbol ($), it is interpreted as static; if it contains more than one, it is interpreted as drifting. The drifting character must be specified in each digit position through which it may drift.

Drifting characters must appear in strings. A string is a sequence of the same drifting character, optionally containing a V and one of the insertion characters comma, point, slash, or B. Any of the insertion characters slash, comma, or point within or immediately following the string is considered part of the drifting string. The character B always causes insertion of a blank, wherever it appears. A V terminates the drifting string, except when the arithmetic value of the data item is zero; in that case, the V is ignored. A field of a picture specification can contain only one drifting string. A drifting string cannot be preceded by a digit position nor can it occur in the same field as the picture characters * and Z.

The position in the data associated with the characters slash, comma, and point appearing in a string of drifting characters will contain one of the following:

- slash, comma, or point 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. Two different picture characters cannot be used in a drifting manner in the same field.

If a drifting string contains a V within it, the V delimits the preceding portion as a subfield, and all digit positions of the subfield following the V must also be part of the drifting string that commences the second subfield.

Only one type of sign character can appear in each field. An S, +, or - used as a static character can appear to the right or left of all digits in the mantissa and exponent fields of a floating-point specification, and to the right or left of all digit positions of a fixed-point specification.

In the case in which all digit positions after the V contain drifting characters, suppression in the subfield will occur only if all of the integer and fractional digits are zero. The resulting edited data item will then be all blanks (except for any insertion characters at the start of the field). If there are any significant fractional digits, the entire fractional portion will appear unsuppressed.

$ specifies the currency symbol. If this character appears more than once, it is a drifting character; otherwise it is a static character. The static character specifies that the character is to be placed in the associated position. The static character must appear either to the left of all digit positions in a field of a specification or to the right of all digit positions in a specification. See details above for the drifting use of the character.

S specifies the plus sign character (+) if the data value is ≥0, otherwise it specifies the minus sign character (-). The character may be drifting or static. The rules are identical to those for the currency symbol.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(5,2)	123.45	$999V.99	$123.45
FIXED(5,2)	012.00	99$	12$
FIXED(5,2)	001.23	$ZZZV.99	$bb1.23
FIXED(5,2)	000.00	$ZZZV.ZZ	bbbbbbb
FIXED(1)	0	$$$.$$	bbbbbb
FIXED(5,2)	123.45	$$$9V.99	$123.45
FIXED(5,2)	001.23	$$$9V.99	bb$1.23
FIXED(2)	12	$$$,999	bbb$012
FIXED(4)	1234	$$$,999	b$1,234
FIXED(5,2)	2.45	SZZZV.99	+bb2.45
FIXED(5)	214	SS,SS9	+214
FIXED(5)	-4	SS,SS9	-4
FIXED(5,2)	-123.45	+999V.99	b123.45
FIXED(5,2)	-123.45	-999V.99	-123.45
FIXED(5,2)	123.45	999V.99S	123.45+
FIXED(5,2)	001.23	++B+9V.99	bbb+1.23
FIXED(5,2)	001.23	---9V.99	bbb1.23
FIXED(5,2)	-001.23	SSS9V.99	bb-1.23

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D.5. Examples of drifting picture characters

+ specifies the plus sign character (+) if the data value is ≥0, otherwise it specifies a blank. The character may be drifting or static. The rules are identical to those for the currency symbol.

- specifies the minus sign character (-) if the data value is <0, otherwise it specifies a blank. The character may be drifting or static. The rules are identical to those for the currency symbol.

Figure D.5 gives examples of the use of drifting picture characters. In the figure, the letter b indicates a blank character.

CREDIT, DEBIT, AND OVERPUNCHED SIGNS

The character pairs CR (credit) and DB (debit) specify the signs of real numeric character data items and usually appear in business report forms.

Any of the picture characters T, I, or R specifies an overpunched sign in the associated digit position of numeric character data. An overpunched sign is a 12-punch (for plus) or an 11-punch (for minus) punched into the same column as a digit. It indicates the sign of the arithmetic data item. Only one overpunched sign can appear in a specification for a fixed-point number. A floating-point specification can contain two, one in the mantissa field and one in the exponent field. The overpunch character can, however, be specified for any digit

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(3)	-123	$Z.99CR	$1.23CR
FIXED(4,2)	12.34	$ZZV.99CR	$12.34bb
FIXED(4,2)	-12.34	$ZZV.99DB	$12.34DB
FIXED(4,2)	12.34	$ZZV.99DB	$12.34bb
FIXED(4)	1021	999I	102A
FIXED(4)	-1021	Z99R	102J
FIXED(4)	1021	99T9	10I1

[1]The arithmetic value is the value expressed by the digits and the actual or assumed location of the V in the specification.

Figure D.6. Examples of CR, DB, T, I, and R picture characters

position within a field. The overpunched number then will appear in the specified digit position.

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 a 12-punch ≥0 or an 11-punch if the value is <0. It also specifies that an overpunch is to be indicated in the character-string value.

I specifies that the associated position, on input, will contain a digit overpunched with a 12-punch if the value is ≥0; otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character-string value if the data value is ≥0.

R specifies that the associated position, on input, will contain a digit overpunched with an 11-punch if the value is <0; otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character-string value if the data value is <0.

Figure D.6 gives examples of the CR, DB, and overpunch characters. In the figure, the letter b indicates a blank character.

Note: The picture characters CR, DB, T, I, and R cannot be used with any other sign characters in the same field.

EXPONENT SPECIFIERS

The picture characters K and E delimit the exponent field of a numeric character specification that describes floating-point decimal numbers. The exponent field is always the last field of a numeric character floating-point picture specification. The picture characters K and E cannot appear in the same specification.

K specifies that the exponent field appears to the right of the associated position. It does not specify a character in the numeric character data item.

E specifies that the associated position contains the letter E, which indicates the start of the exponent field.

The value of the exponent is adjusted in the character-string value so that the first significant digit of the first field (the mantissa) appears in the position associated with the first digit specifier of the specification (even if it is a zero suppression character).

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FLOAT(5)	.12345E06	V.99999E99	.12345E06
FLOAT(5)	.12345E-06	V.99999ES99	.12345E-06
FLOAT(5)	.12345E+06	V.99999KS99	.12345+06
FLOAT(5)	-123.45E+12	S999V.99ES99	-123.45E+12
FLOAT(5)	001.23E-01	SSS9.V99ESS9	+123.00Eb-3
FLOAT(5)	001.23E+04	ZZZV.99KS99	123.00+02
FLOAT(5)	001.23E+04	SZ99V.99ES99	+123.00E+02
FLOAT(5)	001.23E+04	SSSSV.99E-99	+123.00Eb02

[1]The arithmetic value is the value expressed by the mantissa, multiplied by 10 to the power indicated in the exponent field.

Figure D.7. Examples of floating-point picture specifications

Figure D.7 gives examples of the use of exponent delimeters. In the figure, the letter b indicates a blank character.

SCALING FACTOR

The picture character F specifies a scaling factor for fixed-point decimal numbers. It appears at the right end of the picture specification and is used in the following format:

F ([+|-] decimal-integer-constant)

F specifies that the optionally signed decimal integer constant enclosed in parentheses is the scaling factor. The scaling factor specifies that the decimal point in the arithmetic value of the variable is that number of places to the right (if the scaling factor is positive) or to the left (if negative) of its assumed position in the character-string value.

The scaling factor must not imply a scale outside the range -128 to 127.

Figure D.8 shows examples of the use of the scaling factor picture character.

Source Attributes	Source Data (in constant form)	Picture Specification	Character-String Value[1]
FIXED(4,0)	1200	99F(2)	12
FIXED(7,0)	-1234500	S999V99F(4)	-12345
FIXED(5,5)	.00012	99F(-5)	12
FIXED(6,6)	.012345	999V99F(-4)	12345

[1]The arithmetic value is the same as the character-string value, multiplied by 10 to the power of the scaling factor.

Figure D.8. Examples of scaling factor picture characters

Section E: Edit-directed Format Items

This section describes each of the edit-directed format items that can appear in the format list of a GET or PUT statement.

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

In this section, the three categories are discussed separately and the format items are listed under each category. The remainder of the section contains detailed discussions of each of the format items, with the discussions appearing in alphabetic order.

Data Format Items

A data format item describes the external format of a single data item.

For input, the data in the stream is considered to be a continuous string of characters; all blanks are treated as characters in the stream, as are quotation marks. Each data format item in a GET statement specifies the number of characters to be obtained from the stream and describes the way those characters are to be interpreted. Strings should not be enclosed in quotation marks, nor should the letter B be used to identify bit strings. If the characters in the stream cannot be interpreted in the manner specified, the CONVERSION condition is raised.

For output, the data in the stream takes the form specified by the format list. Each data format item in a PUT statement specifies the width of a field into which the associated data item in character form is to be placed and describes the format that the value is to take. Enclosing quotation marks are not inserted, nor is the letter B to identify bit strings.

Leading blanks are not inserted automatically to separate data items in the output stream. String data is left-adjusted in the field, whose width is specified. Arithmetic data is right-adjusted. Because of the rules for conversion of arithmetic data to character type, which can cause up to three leading blanks to be inserted (in addition to any blanks that replace leading zeros), there generally will be at least one blank preceding an arithmetic item in the converted field. Leading blanks will not appear in the stream, however, unless the specified field width allows for them. Truncation, due to inadequate field-width specification is on the left for arithmetic items, on the right for string items.

Note that the value of binary data both on input and output is always represented in decimal form for edit-directed transmission.

Following is a list of data format items:

Fixed-point format item	F
Floating-point format item	E
Complex format item	C
Picture format item	P
Bit-string format item	B
Character-string format item	A

Control Format Items

The control format items specify the layout of the data set associated with a file. The following is a list of control format items:

Paging format item	PAGE
Line skipping format item	SKIP
Line position format item	LINE
Column position format item	COLUMN
Spacing format item	X

A control format item has no effect unless it is encountered before the data list is exhausted.

The PAGE and LINE format items apply only to output and only to files with the PRINT attribute. The SKIP, COLUMN and X format items apply to both input and output.

The PAGE, SKIP, and LINE format items have the same effect as the corresponding options of the PUT statement (and of the GET statement, in the case of SKIP), except that the format items take effect only when

they are encountered in the format list, while the options take effect before any data is transmitted.

The COLUMN format item positions the file to the specified character position in the current or following line. It cannot be used in a GET STRING or PUT STRING statement.

The spacing format item specifies relative horizontal spacing. On input, it specifies a number of characters in the stream to be skipped over and ignored; on output, it specifies a number of blanks to be inserted into the stream.

For the effects of control format items when specified in the first GET or PUT statement following the opening of a file, see "OPEN Statement" in section J, "Statements".

Remote Format Item

The remote format item specifies the label of a FORMAT statement that contains a format list which is to be taken to replace the remote format item.

The remote format item is:

```
R(statement-label-designator)
```

The "statement-label-designator" is a label constant or scalar label variable.

Use of Format Items

Most of the format items listed below are followed by a specification. In all cases except the picture and remote items, any expression contained in the specification can be given as decimal integer constants, as element variables, or as other element expressions. The value assigned to a variable during an input operation can be used in an expression in a format item that is associated with a later data item. An expression is evaluated and converted to an integer each time the format item is used.

Alphabetical List of Format Items

A-Format Item

The A-format item is:

```
A [(field-width)]
```

The character-string format item describes the external representation of a string of characters.

General rules:

1. The "field-width" is an expression that is evaluated and converted to an integer, which must be non-negative, each time the format item is used. It specifies the number of character positions in the data stream that contain (or will contain) the string.
2. On input, the specified number of characters is obtained from the data stream and assigned, with any necessary conversion, truncation, or padding, to the associated element in the data list. The field width is always required on input, and if it has a value equal to zero, a null string is assumed. If quotation marks appear in the stream, they are treated as characters in the string.
3. On output, the associated element in the data list is converted, if necessary, to a string of characters and is truncated or extended with blanks on the right to the specified field width before being placed into the data stream. If the field width is equal to zero, the format item and its associated element in the data list are skipped, and no characters are placed into the data stream. Enclosing quotation marks are never inserted. If the field width is not specified, it is assumed to be equal to the character-string length of the element named in the data list (after conversion, if necessary, according to the rules given in section F, "Data Conversion and Expression Evaluation").

B-format Item

The B-format item is:

```
B [(field-width)]
```

The bit-string format item describes the external representation of a bit string. Each bit is represented by the character 0 or 1.

General rules:

1. The "field-width" is an expression that is evaluated and converted to an integer, which must be non-negative, each time the format item is used. It specifies the number of data-stream character positions that contain (or will contain) the bit string.

2. On input, the character representation of the bit string may occur anywhere within the specified field. Blanks, which may appear before and after the bit string in the field, are ignored. Any necessary conversion occurs when the bit string is assigned to the associated element in the data list. The field width is always required on input, and if it is equal to zero, a null string is assumed. Any character other than 0 or 1 in the string, including embedded blanks, quotation marks, or the letter B, will raise the CONVERSION condition.

3. On output, the character representation of the bit string is left-adjusted in the specified field, and necessary truncation or extension with blanks occurs on the right. Any necessary conversion to bit-string is performed. No quotation marks are inserted, nor is the identifying letter B. If the field width is equal to zero, the format item and its associated element in the data list are skipped, and no characters are placed into the data stream. If the field width is not specified, it is assumed to be equal to the bit-string length of the element named in the data list (after conversion, if necessary, according to the rules given in section F "Data Conversion and Expression Evaluation").

C-format Item

The C-format item is:

```
C(real-format-item[,real-format-item])
```

The complex format item describes the external representation of a complex data item.

General rules:

1. Each "real-format-item" is specified by one of the F-, E-, or P-format items. The P-format item must describe numeric character data; it cannot describe character-string data.

2. On input, the complex format item describes the real and imaginary parts of the complex data item within adjacent fields in the data stream. If the second real format item is omitted, it is assumed to be the same as the first. The letter I will cause the CONVERSION condition to be raised.

3. On output, the real format items describe the forms of the real and imaginary parts of the complex data item in the data stream. If the second real format item is omitted, it is assumed to be the same as the first. The letter I is never appended to the imaginary part. If the second real format item (or the first, if only one appears) is an F or E item, the internal sign will be printed only if the value of the imaginary part is less than zero. If the real format item is a P item, the sign will be printed only if the S or - or + picture character is specified. If the I is to be appended, it must be specified as a separate data item in the data list, immediately following the variable that specifies the complex item. The I, then, must have a corresponding format item (either A or P).

COLUMN Format Item

The COLUMN format item is:

```
COLUMN (character-position)
```

The column position format item positions the file to a specified character position within the current or following line. It can be used with either input or output files.

General rules:

1. The "character-position" is an expression which is evaluated and converted to an integer, which must be non-negative, each time the format item is used.

2. The file is positioned to the specified character position in the current line, provided it has not already passed this position. On input, intervening character positions are ignored; on output, they are filled with blanks. If the file is already positioned after the specified character position, the current line is completed and a new line is started; the format item is then applied to the following line.

3. If the specified character position lies beyond the rightmost character position of the current line, or if the value of the expression for the character position is less than one, then the character position is assumed to be one.

   Note: The rightmost character position is determined as follows:

   a. For output files, it is determined by the line size.

   b. For input files, the compiler uses the length of the current logical record to determine the line size and, hence, the rightmost character position. In the case of V-format records, this line size is equal to the logical record length minus the number of bytes containing control information.

4. The COLUMN format item has no effect unless it is encountered before the data list is exhausted.

5. The COLUMN format item must not be used in a GET STRING or PUT STRING statement.

E-format Item

The E-format item is:

```
E(field-width,number-of-fractional-digits
        [,number-of-significant-digits])
```

The floating-point format item describes the external representation of decimal arithmetic data in floating-point format.

General rules:

1. The "field-width", "number-of-fractional-digits", and "number-of-significant-digits" can be represented by expressions, which are evaluated and converted to integers when the format item is used.

   "Field-width" specifies the total number of characters in the field.

   "Number-of-fractional-digits" specifies the number of digits in the mantissa that follow the decimal point.

   "Number-of-significant-digits" specifies the number of digits that must appear in the mantissa.

2. On input, the data item in the data stream is the character representation of an optionally signed decimal floating-point or fixed-point constant located anywhere within the specified field. If the data item is a fixed-point number, an exponent of zero is assumed.

   The external form of a floating-point number is:

```
[+|-] mantissa [E]{+|-} exponent
               E [+|-]
```

   The mantissa must be a decimal fixed-point constant.

   a. The number can appear anywhere within the specified field; blanks may appear before and after the number in the field and are ignored. If the entire field is blank, the CONVERSION condition is raised. When no decimal point appears, the expression for the number of fractional digits specifies the number of character positions in the mantissa to the right of the assumed decimal point. If a decimal point does appear in the number, it overrides the specification of the number of the fractional digits.

      The value expressed by "field-width" includes trailing blanks, the exponent position, the positions for the optional plus or minus signs, the position for the optional letter E, and the position for the optional decimal point in the mantissa.

   b. The exponent is a decimal integer constant. Whenever the exponent and preceding sign or letter E are omitted, a zero exponent is assumed.

3. On output, the internal data is converted to floating-point, and the external data item in the specified field has the following general form:

```
[-] {s-d digits}.{d digits}
    E {+|-} exponent
```

In this form, s represents the number of significant digits, and d represents the number of fractional digits. The value is rounded if necessary. If the data item is fractional, the character 0, rather than s-d digits, appears before the decimal point.

a. The exponent is a two-digit decimal integer constant, which may be two zeros. The exponent is automatically adjusted so that the leading digit of the mantissa is nonzero. When the value is zero, zero suppression is applied to all digit positions (except the first) to the left of the decimal point. All other digit positions contain zero.

b. If the above form of the number does not fill the specified field on output, the number is right-adjusted and extended on the left with blanks. If the number of significant digits is not specified, it is taken to be 1 plus the number of fractional digits. The field width for non-negative values of the data item must be greater than or equal to 5 plus the number of significant digits. For negative values of the data item, the field width must be greater than or equal to 6 plus the number of significant digits. However, if the number of fractional digits is zero, the decimal point is not written, and the above figures for the field width are reduced by 1.

c. The rounding of internal data is as follows: if truncation causes a digit to be lost from the right, and this digit is greater than or equal to 5, then 1 is added to the digit to the left of the truncated digit.

d. If the field width is such that significant digits or the sign are lost, the SIZE condition is raised.

F-format Item

The F-format item is:

```
F(field-width[,number-of-fractional-digits
                [,scaling-factor]])
```

The fixed-point format item describes the external representation of a decimal arithmetic data item in fixed-point format.

General rules:

1. The "field-width", "number-of fractional-digits", and "scaling-factor" can be represented by element expressions, which are evaluated and converted to integers when the format item is used. The evaluated field width and number of fractional digits must both be non-negative.

2. On input, the data item in the data stream is the character representation of an optionally signed decimal fixed-point constant located anywhere within the specified field. Blanks may appear before and after the number in the field and are ignored. If the entire field is blank, it is interpreted as zero.

   The number of fractional digits, if not specified, is assumed to be zero.

   If no scaling factor is specified and no decimal point appears in the field, the expression for the number of fractional digits specifies the number of digits in the field to the right of the assumed decimal point. If a decimal point actually does appear in the data, it overrides the expression for the number of fractional digits.

   If a scaling factor is specified, it effectively multiplies the value of the data item in the data stream by 10 raised to the integral value (p) of the scaling factor. Thus, if p is positive, the number is treated as though the decimal point appeared p places to the right of its given position. If p is negative, the number is treated as though the decimal point appeared p places to the left of its given position. The given position of the decimal point is that indicated either by an actual point, if it appears, or by the expression for the number of fractional digits, in the absence of an actual point.

3. On output, the internal data is converted, if necessary, to fixed-point; the external data is the character representation of a decimal

fixed-point number, rounded if necessary, and right-adjusted in the specified field.

If only the field width is specified in the format item, only the integer portion of the number is written; no decimal point appears.

If both the field width and number of fractional digits are specified, but the scale factor is not, both the integer and fractional portions of the number are written. If the value (d) of the number of fractional digits is greater than zero, a decimal point is inserted before the rightmost d digits. Trailing zeros are supplied when the number of fractional digits is less than d (the value d must be less than the field width). Suppression of leading zeros is applied to all digit positions (except the first) to the left of the decimal point.

The rounding of internal data is as follows: if truncation causes a digit to be lost from the right, and this digit is greater than or equal to 5, then 1 is added to the digit to the left of the truncated digit.

The integer value (p) of the scaling factor effectively multiplies the value of the associated element in the data list by 10 raised to the power of p, before it is edited into its external character representation. When the number of fractional digits is zero, only the integer portion of the number is used.

On output, if the value of the fixed-point number is less than zero, a minus sign is prefixed to the external character representation; if it is greater than or equal to zero, no sign appears. Therefore, for negative values of the fixed-point number, the field width specification must include a count of both the sign and the decimal point.

If the field width is such that any character is lost, the SIZE condition is raised.

LINE Format Item

The LINE format item is:

LINE (line-number)

The line position format item specifies the particular line on the current or following page of a PRINT file upon which the next data item is to be printed.

General rules:

1. The "line-number" can be represented by an expression, which is evaluated and converted to an integer, which must be non-negative, each time the format item is used.
2. Blank lines are inserted, if necessary.
3. If the specified line number is less than or equal to the current line number, or if the specified line is beyond the limits set by the PAGESIZE option of the OPEN statement (or by default), the ENDPAGE condition is raised. An exception is that if the specified number is equal to the current line number, and the column one character has not yet been transmitted, the effect is as for a SKIP(0) item-carriage return with no line spacing.
4. If "line-number" is equal to zero, it is assumed to be one.
5. The LINE format item has no effect unless it is encountered before the data list is exhausted.

P-format Item

The P-format item is:

P 'picture-specification'

The picture format item describes the external representation of numeric character data and of character-string data.

The "picture-specification" is discussed in detail in section D, "Picture Specification Characters" and in the discussion of the PICTURE attribute in section I, "Attributes".

On input, the picture specification describes the form of the data item expected in the data stream and, in the case of a numeric character specification, how the item's arithmetic value is to be interpreted. Note that the picture specification should accurately describe the data in the input stream, including characters represented by editing characters. If the indicated character does not appear in the stream, the CONVERSION condition is raised.

On output, the value of the associated element in the data list is converted to the form specified by the picture specification before it is written into the data stream.

PAGE Format Item

The PAGE format item is:

```
PAGE
```

The paging format item specifies that a new page is to be established. It can be used only with PRINT files.

General rules:

1. The establishment of a new page implies that the file be positioned to line one of the next page.
2. The PAGE format item has no effect unless it is encountered before the data list is exhausted.

R-format Item

The R-format item is:

```
R (statement-label-designator)
```

The remote format item allows format items in a FORMAT statement to replace the remote format item.

General rules:

1. The "statement-label-designator" is a label constant, or an element label variable, or a function reference that has as its value the statement label of a FORMAT statement. The FORMAT statement includes a format list that is taken to replace the format item.
2. The R-format item and the specified FORMAT statement must be internal to the same block. (If the procedure is executed recursively, they must be in the same invocation.)
3. There can be no recursion within a FORMAT statement. That is, a remote FORMAT statement cannot contain an R-format item that names itself as a statement label designator, nor can it name another remote FORMAT statement that will lead to the naming of the original FORMAT statement. Avoidance of recursion can be assured if the FORMAT statement referred to by a remote format item does not itself contain a further remote format item.
4. Any conditions enabled for the GET or PUT statement must also be enabled for the remote FORMAT statement(s) that are referred to.
5. If the GET or PUT statement is a single statement of an on-unit, it cannot contain a remote format item.

SKIP Format Item

The SKIP format item is:

```
SKIP[(relative-position-of-next-line)]
```

The line skipping format item specifies that a new line is to be defined as the current line.

General rules:

1. The "relative-position-of-next-line" can be specified by an element expression, which is evaluated and converted to an integer, w, which must be non-negative, each time the format item is used. It must be greater than zero for non-PRINT files. If it is not, or if it is omitted, 1 is assumed.
2. The new line is the wth line after the present line.
3. If w is greater than one, then on input, one or more lines will be ignored; on output, one or more blank lines will be inserted.
4. w may be equal to zero for PRINT files only; the effect is that of a carriage return without line spacing. Characters previously written may be overprinted.
5. For PRINT files, if the specified relative position is beyond the limit set by the PAGESIZE option of the OPEN statement (or the default), the ENDPAGE condition is raised.
6. If the SKIP format item is the first item to be executed after a file has been opened, output commences on the wth line of the first page. If w is zero or 1, it commences on the first line of the first page.

7. The SKIP format item has no effect unless it is encountered before the data list is exhausted.

X-format Item

The X-format item is:

X (field-width)

The spacing format item controls the relative spacing of data items in the data stream. It is not limited to PRINT files.

General rules:

1. The "field-width" is an expression, which is evaluated and converted to an integer, which must be non-negative, each time the format item is used. The integer specifies the number of blanks before the next field of the data stream, relative to the current position in the stream.

2. On input, the specified number of characters is spaced over in the data stream and not transmitted to the program.

3. On output, the specified number of blank characters are inserted into the stream.

4. The spacing format item has no effect unless it is encountered before the data list is exhausted.

Missing from orig document

Missing from orig document

Section F: Data Conversion and Expression Evaluation

The purpose of this section is to help the user analyze a mixed expression, involving problem data, to determine the conversions that will occur, and the effect these conversions will have on the final result. In this context, an assignment is considered as a special case of a mixed expression. An expression is termed "mixed" for one of two reasons:

1. Operands have attributes that differ. For example,

   ```
   DCL A CHAR(6),B FIXED BINARY(31);

   A=B;
   ```

   B is converted to character-string form before assignment to A.

2. Operands have attributes that are not compatible with the operation to be performed. For example,

   ```
   DCL C BIT(10) VARYING;

   C=C+C;
   ```

   C is converted to an arithmetic value before the addition is performed. The arithmetic result of the addition is converted to bit-string form before assignment back to C.

This section gives all the circumstances and rules under which such conversions take place.

Conversions may also occur in situations other than assignment or expression evaluation. However, the rules given here are directly applicable to these situations. Figure F.3 lists all the circumstances under which conversion may occur.

Section Organization

The conversion rules are presented by figures F.1, F.4, and F.5, and by source to target rules.

Figure F.1 shows the operations that may be performed, gives their priority in expression evaluation, and contains references to figures F.4 and F.5, and to source to target rules where further information may be obtained.

Figure F.4 shows, for all arithmetic operations, the conversions that will take place.

Figure F.5 shows, for all problem data comparisons, the conversions that will take place.

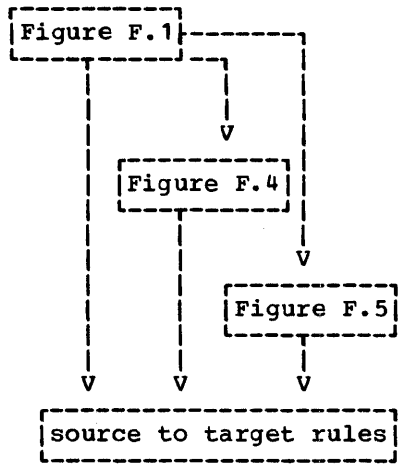
Source to target rules are given for each of the following data types:

Coded Arithmetic:

```
FIXED BINARY
FIXED DECIMAL
FLOAT BINARY
FLOAT DECIMAL

PICTURE (numeric character)

CHARACTER

BIT
```

The following pages take each of these data types as a target and give the conversion rules for all the others taken as sources to that target. The source to target rules are used directly for all conversions that do not involve operators. See figure F.3.

The relationship between figures F.1, F.4, and F.5, and the source to target rules is illustrated as follows:



One other figure, figure F.2, is an aid to calculating the new precision resulting from a conversion.

The placement of the figures in this section is designed so that the user can

have access to most of the information at one page opening. Figures F.1 and F.2 are together on a foldout page so that they are clear of the normal page. Figures F.4 and F.5 are each on a foldout page and placed at the back of the section. Thus, figures F.1, F.2, and F.4 or F.5, and specific source to target rules can all be viewed simultaneously.

EXAMPLE OF USE OF THE CONVERSION RULES

The following example illustrates how information is retrieved from the figures and the source to target rules.

```
DCL FB FIXED BINARY(5),
    FD FIXED DECIMAL(5,2),
    CH CHARACTER(12);
CH=4.2*FB+FD;
```

From the priority rules in figure F.1, the assignment statement is executed in the following steps:

1. result1=4.2*FB
2. result2=result1+FD
3. CH=result2

The attributes of the result at each step are determined as follows.

Step 1

The constant 4.2 has implied attributes of FIXED DECIMAL(2,1).

Refer to figure F.4a using the attributes of the constant as the first operand and the attributes of FB as the second operand. This gives the code reference 5. In figure F.4d, code reference 5 gives the attributes of the result as FIXED BINARY(p,q), where the precision for multiplication is:

```
p=1+(1+2*3.32)+5=14
q=(1*3.32)+0=4
```

- result1 has attributes FIXED BINARY(14,4)

Step 2

Refer to figure F.4a using the attributes of result1 as the first operand and the attributes of FD as the second operand. This gives the code reference 7. In figure F.4c, code reference 7 gives the attributes of the result as FIXED BINARY(p,q), where the precision for addition is:

```
p=1+MAX(14-4,((1+5*3.32)-2*3.32))+7=19
q=MAX(4,(2*3.32))=7
```

- result2 has attributes FIXED BINARY(19,7)

Step 3

Refer to the rules for FIXED BINARY source to CHARACTER target. The source is first converted to FIXED DECIMAL(p,q), where

```
p=1+19/3.32=7
q=7/3.32=3
```

Refer to the rules for FIXED DECIMAL source to CHARACTER target. The decimal constant is assigned to an intermediate string of length 10.

The intermediate string is assigned to CH, which is padded on the right with two blank characters.

Coded Arithmetic

The four data types FIXED BINARY, FIXED DECIMAL, FLOAT BINARY, and FLOAT DECIMAL are all coded arithmetic data. Rules for conversion between them are given under each data type taken as a target. However, the following general points should be noted:

- Small changes in value may occur due to truncation on the right in conversion from decimal to binary, and between fixed-point decimal and floating-point decimal.
- If a complex value is converted to a real value, the imaginary part is ignored. If a real value is converted to a complex value, the imaginary part is zero.

PICTURE (numeric character)

Data is first interpreted as decimal with scale and precision determined by the corresponding PICTURE specification. The item is then converted to the base, scale, mode, and precision of the target. See under specific target types of coded arithmetic data using FIXED DECIMAL or FLOAT DECIMAL as the source.

CHARACTER

The source string must represent a valid arithmetic constant or complex expression; otherwise, the CONVERSION condition will be raised, if enabled. The constant can be signed, and can be surrounded by blanks, but cannot contain blanks between the sign and the value, or between the end of the real part and the sign preceding the imaginary part of a complex expression.

A null string gives the value zero.

The constant will have base, scale, mode, and precision attributes. It will be converted to the attributes of the target when they are independent of the source attributes, as in the case of assignment. See under specific target types of coded arithmetic data using the attributes of the constant as the source.

However, if an intermediate target is necessary, as is the case in evaluation of an operational expression, the attributes of the intermediate target are those it would have if a decimal fixed-point integer of precision (15,0) had appeared in place of the string. (This allows the compiler to generate code to handle all cases, regardless of the attributes of the contained constant.) Consequently, any fractional portion of the constant is lost. See under specific target types of coded arithmetic data using FIXED DECIMAL as the source.

BIT

The source string is interpreted as an unsigned binary integer whose precision is (31,0) if the conversion occurs during evaluation of an operational expression, or whose precision is (56,0) if the conversion occurs during an assignment. The greater precision allowed for in an assignment is possible because the compiler can readily determine the final target. See under specific target types of coded arithmetic data using FIXED BINARY as the source.

If the source string is longer than the allowable precision, bits on the left are ignored; if nonzero bits are lost, the result is undefined and the SIZE condition will be raised if enabled.

A null string gives the value zero.

Source: **Target: FIXED BINARY**

FIXED BINARY

The binary point alignment is maintained during precision conversion, and therefore padding or truncation can occur on the left or the right. If nonzero bits on the left are lost, the result is undefined; the SIZE condition will be raised, if enabled.

FIXED DECIMAL

If the precision of the source is (p_1,q_1), the precision of the result is (p_2,q_2), where $p_2=1+CEIL(p_1*3.32)$ and $q_2=CEIL(q_1*3.32)$. If the calculated value of p_2 exceeds 31, significant digits on the left may be lost, this will cause the SIZE condition to be raised, if enabled, and the result is undefined.

FLOAT BINARY

This conversion can occur only when data is assigned. The precision conversion is the same as that given for FIXED BINARY to FIXED BINARY with p_1 as declared or indicated and q_1 as indicated by the binary point position and modified by the value of the exponent.

FLOAT DECIMAL

This conversion can occur only when data is assigned. The precision conversion is the same as that given for FIXED DECIMAL to FIXED BINARY with p_1 as declared or indicated and q_1 as indicated by the decimal point position and modified by the value of the exponent.

PICTURE (numeric character)
CHARACTER
BIT

See under Coded Arithmetic Target.

Source: **Target: FIXED DECIMAL**

FIXED BINARY

If the precision of the source is (p_1,q_1), the precision of the result is (p_2,q_2), where $p_2=1+CEIL(p_1/3.32)$ and $q_2=CEIL(q_1/3.32)$.

FIXED DECIMAL

The decimal point alignment is maintained during precision conversion, and therefore padding or truncation can occur on the left or the right. If nonzero bits on the left are lost, the result is undefined; the SIZE condition will be raised, if enabled.

FLOAT BINARY

This conversion can occur only when data is assigned. The precision conversion is the same as that given for FIXED BINARY to FIXED DECIMAL with p_1 as declared or indicated and q_1 as indicated by the binary point position and modified by the value of the exponent.

FLOAT DECIMAL

This conversion can occur only when data is assigned. The precision conversion is the same as that given for FIXED DECIMAL to FIXED DECIMAL with p_1 as declared or indicated and q_1 as indicated by the decimal point position and modified by the value of the exponent.

PICTURE (numeric character)
CHARACTER
BIT

See under Coded Arithmetic Target.

Source: **Target: FLOAT BINARY**

FIXED BINARY

If the precision of the source is (p_1,q_1), the precision of the result is p_2, where $p_2=p_1$. The exponent will indicate any fractional part of the value.

FIXED DECIMAL

If the precision of the source is (p_1,q_1), the precision of the result is p_2, where $p_2=CEIL(p_1*3.32)$. The exponent will indicate any fractional part of the value.

FLOAT BINARY

The precision of the result may be converted from short to long precision by padding with zeros on the right, or may be converted from long to short precision by truncation on the right.

FLOAT DECIMAL

If the precision of the source is (p_1,q_1), the precision of the result is p_2, where $p_2=CEIL(p_1*3.32)$.

PICTURE (numeric character)
CHARACTER
BIT

See under Coded Arithmetic Target.

Source: **Target: FLOAT DECIMAL**

FIXED BINARY

If the precision of the source is (p_1,q_1), the precision of the result is p_2, where p_2=CEIL(p_1/3.32). The exponent will indicate any fractional part of the value.

FIXED DECIMAL

If the precision of the source is (p_1,q_1), the precision of the result is p_2, where $p_2=p_1$. The exponent will indicate any fractional part of the value.

FLOAT BINARY

If the precision of the source is (p_1), the precision of the result is p_2, where p_2=CEIL(p_1/3.32).

FLOAT DECIMAL

The precision of the result may be converted from short to long precision by padding with zeros on the right, or may be converted from long to short precision by truncation on the right.

PICTURE (numeric character)
CHARACTER
BIT

See under Coded Arithmetic Target.

Source: Target: PICTURE (Numeric Character)

Upon conversion to numeric character form, the source data acquires attributes that depend entirely on the known attributes of the target variable. Any PICTURE specification implies coded arithmetic data, which is either FIXED DECIMAL or FLOAT DECIMAL. The following rules for different source to numeric character target show those target attributes that are necessary to permit error-free assignment.

FIXED BINARY

If the precision of the source is (p_1,q_1), the target must imply,

FIXED DECIMAL (1+x+q-y,q) or
FLOAT DECIMAL (x)
where x>=CEIL(p_1/3.32),y=CEIL(q_1/3.32), and q>=y.

FIXED DECIMAL

If the precision of the source is (p_1,q_1), the target must imply,

FIXED DECIMAL (x+q-p1,q) or
FLOAT DECIMAL (x)
where x>=p and q>=q_1.

FLOAT BINARY

If the precision of the source is (p_1), the target must imply,

FIXED DECIMAL (p,q) or
FLOAT DECIMAL (p)
where p>=CEIL(p_1/3.32) and the values of p and q take account of the range of values that may be held by the exponent of the source.

FLOAT DECIMAL

If the precision of the source is (p_1), the target must imply,

FIXED DECIMAL (p,q) or
FLOAT DECIMAL (p)
where p>=p_1 and the values of p and q take account of the range of values that may be held by the exponent of the source.

PICTURE (numeric character)

The implied attributes will be either FIXED DECIMAL or FLOAT DECIMAL. See the respective entries for this target.

CHARACTER

The target must imply a character string that will accommodate the source string character-for-character (excluding insertion characters) with no conversion being required. Insertion characters are removed during the assignment to the target.

Source :

Target : PICTURE (numeric character)

BIT

If the length of the source string is (n), the target must imply,

```
FIXED DECIMAL (1+x+q,q) or
FLOAT DECIMAL (x)
where x>=CEIL(n/3.32) and q>=0.
```

Coded Arithmetic

The arithmetic value is converted to a decimal constant. The constant is inserted into an intermediate character-string whose length is derived from the attributes of the source. The intermediate string is assigned to the target according to the rules given for CHARACTER to CHARACTER.

Note that the rules for coded arithmetic to character-string conversion are also used for list-directed and data-directed output, and for evaluating keys (even for REGIONAL files).

FIXED BINARY

The binary precision (p_1,q_1) is first converted to the equivalent decimal precision (p,q), where p=1+CEIL(p_1/3.32) and q=CEIL(q_1/3.32). Thereafter the rules are the same as those given for FIXED DECIMAL to CHARACTER.

FIXED DECIMAL

A decimal fixed-point source with precision (p,q) is converted as follows:

1. If p>=q>=0 then:
   - The constant is right adjusted in a field of width p+3.
   - Leading zeros are replaced by blanks, except for a single zero that immediately precedes the decimal point of a fractional number.
   - A minus sign will precede the first digit of a negative number. A positive value is unsigned.
   - Unless the source is an integer, the constant has q fractional digits.
2. If p<q or q<0, a scaling factor is appended to the right of the constant. The scaling factor has the form:

   F{+|-}nnn, where {+|-}nnn has the value of q.

   The length of the intermediate string is p+k+3, where k is the number of digits necessary to hold the value of q (not including the sign or the letter F).

If the arithmetic value is complex, the intermediate string consists of the imaginary part concatenated to the real part. The left-hand, or real, part is generated exactly as a real source. The right-hand, or imaginary, part is always signed, and it has the letter I appended. The generated string is a complex expression with no blanks between its elements. The length of the intermediate string is:

2*p+7 for p>=q>=0 and
2*(p+k)+7 for p<q or q<0.

The following examples show the intermediate strings that are generated from several real and complex fixed-point decimal values:

precision:	(5,0)	(4,1)	(4,-3)	(2,1)
value :	2947	-121.7	-3279000	1.2+0.3I
string :	'bbbb2947'	'b-121.7'	'-3279F+3'	'bbb1.2+0.3I'

FLOAT BINARY

The floating-point binary precision (p_t) is first converted to the equivalent floating-point decimal precision (p), where $p=CEIL(p_t/3.32)$. Thereafter the rules are the same as those given for FLOAT DECIMAL to CHARACTER.

FLOAT DECIMAL

A decimal floating-point source with precision (p) is converted as if it were transmitted by an E-format item of the form E(w,d,s) where:

w, the length of the intermediate string, is p+6.
d, the number of fractional digits, is p-1.
s, the number of significant digits, is p.

An E-format item generates a floating-point decimal constant with a signed 2-digit exponent (see section E, "Edit Directed Format Items").

The conversion differs from that performed for an E- format item in one respect: the last significant digit is rounded for an E- format item, but not for other conversions from floating-point to character.

If the arithmetic value is complex, the intermediate string consists of the imaginary part concatenated to the real part. The left-hand, or real, part is generated exactly as a real source. The right-hand, or imaginary, part is always signed, and it has the letter I appended. The generated string is a complex expression with no blanks between its elements. The length of the intermediate string is:

2*p+13.

The following examples show the intermediate strings that are generated from several real and complex floating-point decimal values:

```
precision:      (5)              (5)              (3)
value    :      1735x10          -.001663         1
string   :      'b1.7350E+08'    '-1.6630E-03'    'b1.00E+00'

precision:      (5)
value    :      17.3+1.5I
string   :      'b1.7300E+01+1.5000E+00I'
```

PICTURE (numeric character)

A real numeric character field is interpreted as a character string and assigned to the target string according to the rules given for CHARACTER to CHARACTER. If the numeric character field is complex, the real and imaginary parts are concatenated before assignment to the target string.

Insertion characters will be included in the target string.

CHARACTER

The source string is assigned to the target string from left to right. If the source string is longer than the target, excess characters on the right are ignored, and the STRINGSIZE condition will be raised, if enabled. If the target is longer than the source, the target is padded on the right with blanks.

BIT

Bit 0 becomes character 0 and bit 1 becomes character 1. A null bit string becomes a null character string. The generated character string is assigned to the target string according to the rules given for CHARACTER to CHARACTER.

Coded Arithmetic

If necessary, the arithmetic value is converted to binary and both the sign and any fractional part are ignored. (If the arithmetic value is complex, the imaginary part is also ignored.) The resulting binary integer is treated as a bit string. It is assigned to the target according to the rules given for BIT to BIT.

FIXED BINARY

If the precision of the source is (p,q), the length of the intermediate bit string is given by:

```
MIN(31,(p-q)).
```

If (p-q) is negative or zero, the result is a null bit string.

The following examples show the intermediate strings that are generated from several fixed-point binary values:

```
precision:      (1)             (3)             (4,2)
value    :      1               -3              1.25
string   :      '1'B            '011'B          '01'B
```

FIXED DECIMAL

If the precision of the source is (p,q), the length of the intermediate bit string is given by:

```
MIN(31,CEIL((p-q)*3.32)).
```

If (p-q) is negative or zero, the result is a null bit string.

The following examples show the intermediate strings that are generated from several fixed-point decimal values:

```
precision:      (1)             (2,1)
value    :      1               1.1
string   :      '0001'B         '0001'B
```

FLOAT BINARY

If the precision of the source is (p), the length of the intermediate bit string is given by:

```
MIN(31,p).
```

FLOAT DECIMAL

If the precision of the source is (p), the length of the intermediate bit string is given by:

```
MIN(31,CEIL(p*3.32)).
```

PICTURE (numeric character)

Data is first interpreted as decimal with scale and precision determined by the corresponding PICTURE specification. The item is then converted according to the rules given for FIXED DECIMAL or FLOAT DECIMAL to BIT.

CHARACTER

Character 0 becomes bit 0 and character 1 becomes bit 1. Any character other than 0 or 1 will raise the CONVERSION condition, if enabled. A null string becomes a null bit string. The generated bit string, which has the same length as the source character-string, is assigned to the target according to the rules given for BIT to BIT.

BIT

The source string is assigned to the target string from left to right. If the source string is longer than the target, excess bits on the right are ignored, and the STRINGSIZE condition will be raised, if enabled. If the target is longer than the source, the target is padded on the right with blanks.

Missing from orig document

Missing from orig document

Missing from orig document

Missing from orig document

Section G: Built-in Functions and Pseudovariables

All of the built-in functions that are available to the programmer are given in this section and are presented in alphabetical order. Any built-in function that can also be used as a pseudovariable has a subentry describing the action of the pseudovariable.

The general form of a built-in function reference is as follows:

function name [(x)|(x_1,x_2...,x_n)]

where x or x_1,x_2...,x_n represent the arguments required. For some functions one or more arguments are optional. For example:

SUBSTR(x_1,x_2[,x_3])

Each function in the alphabetical list is identified by the general form of the function reference (the pseudovariable reference is always identical to the equivalent function reference). In general, each function description has the following items:

1. A description of the value returned.
2. Details of the arguments.
3. Any other qualifications on the use of the function.
4. When applicable, a description of the action of the equivalent pseudovariable.

CLASSIFICATION OF BUILT-IN FUNCTIONS

The built-in functions can be classified according to the PL/I features they are intended to serve. These classes are:

String-handling
Arithmetic
Mathematical
Array-handling
Condition-handling
Storage Control
Multitasking
STREAM Input/Output
DATE and TIME functions

The first four classes are all computational built-in functions.

String-handling Built-in Functions

These functions simplify the processing of bit and character strings. They are:

BIT	REPEAT
BOOL	STRING
CHAR	SUBSTR
HIGH	TRANSLATE
INDEX	UNSPEC
LENGTH	VERIFY
LOW	

Arithmetic Built-in Functions

These functions allow the programmer to control conversion of base, scale, mode, and precision both directly and during basic arithmetic operations. Other functions in this class are used to investigate simple properties of arithmetic values, for example, the SIGN function indicates the sign of an arithmetic value. They are:

ABS	IMAG
ADD	MAX
BINARY	MIN
CEIL	MOD
COMPLEX	MULTIPLY
CONJG	PRECISION
DECIMAL	REAL
DIVIDE	ROUND
FIXED	SIGN
FLOAT	TRUNC
FLOOR	

Mathematical Built-in Functions

These functions provide standard mathematical operations. They are:

ACOS	LOG
ASIN	LOG2
ATAN	LOG10
ATAND	SIN
ATANH	SIND
COS	SINH
COSD	SQRT
COSH	TAN
ERF	TAND
ERFC	TANH
EXP	

Array-Handling Built-in Functions

These functions all operate on array arguments and return a single value property of an array. They are:

ALL	LBOUND
ANY	POLY
DIM	PROD
HBOUND	SUM

Condition-Handling Built-in Functions

These functions allow the programmer to investigate interrupts that arise from enabled conditions. Each of the functions returns a value that is defined only within the scope of an on-unit that can be entered for the condition specific to the built-in function or within the scope of an on-unit for the ERROR or FINISH condition when raised as standard system action. They are:

DATAFIELD	ONFILE
ONCHAR	ONKEY
ONCODE	ONLOC
ONCOUNT	ONSOURCE

Storage Control Built-in Functions

These functions return special values concerning based or controlled variables, or identify the location of based variables. They are:

ADDR	NULL
ALLOCATION	OFFSET
EMPTY	POINTER

Multitasking Built-in Functions

These functions allow the programmer to investigate the current state of an event variable. They are:

COMPLETION
PRIORITY
STATUS

Stream Input/Output Built-in Functions

These functions allow the programmer to investigate the current state of a file. They are:

COUNT
LINENO

CONVERSION OF ARGUMENTS

Conversion of arguments can occur for many of the built-in functions. Arguments to these built-in functions can be operational expressions. An expression argument, which can include references to built-in functions, is evaluated and converted, according to the rules for data conversion, to a form suitable for the built-in function. The data type required by each argument is given in each function description.

String-Handling Built-in Functions

Some of these functions require arithmetic as well as string arguments. The arithmetic arguments denote the length of a string and therefore should be integer or capable of being converted to integer. The string arguments can be represented by an arithmetic expression that will be converted to string either according to data conversion rules or according to the rules given in the function description. The programmer should ensure that the conversion will cause the function to operate on the string type he requires.

Arithmetic Built-in Functions

Some of these functions derive the data type of their results from one or more arguments. When the data types of the arguments differ, they are converted according to the following scheme: if scales differ, fixed-point is converted to floating-point; if bases differ, decimal is converted to binary; and if modes differ, real is converted to complex. These rules are applied after any string-type arguments have been connected to arithmetic. When a data attribute of the result cannot agree with that of the argument, for example, the FLOOR built-in function, the rules are given in the function description.

The symbol N is used to represent the maximum precision allowed for fixed-point results. The value of N is defined as:

15 for FIXED DECIMAL
31 for FIXED BINARY

Mathematical Built-in Functions

All of these functions operate on floating-point values to produce a floating-point result and therefore, if any argument is not floating-point, it will be converted.

Array-handling Built-in Functions

Any conversion of arguments required for these functions is noted in the function description.

ACCURACY OF THE MATHEMATICAL FUNCTIONS

The accuracy of a result is influenced by two factors:

1. The accuracy of the argument.
2. The performance of the algorithm.

Most arguments contain errors. An error in a given argument may have accumulated over several steps prior to the evaluation of a function. Even data fresh from input conversion may contain slight errors. The effect of argument error on the accuracy of a result depends solely on the nature of the mathematical function and not on the algorithm that computes the result. Errors of this type are not discussed further in this publication.

Performance statistics for each mathematical function are given in figures G.1 and G.2. The values are based on the assumption that the arguments are free from error.

For each function, accuracy values are given for the valid argument range or representative segments of it. In each case the particular statistics given are the most meaningful to the function and range under consideration.

For example, the root-mean-square(RMS) of the relative error and the maximum relative error of a set of results are generally useful and revealing statistics, but are useless for the range of a function where its value becomes zero; the slightest error of the argument value can cause an unbounded fluctuation in the relative magnitude of the result. Such is the case with SIN(x) for values of 'x' close to pi; in this range it is more appropriate to discuss absolute errors.

The values for short and long precision floating-point arguments are given in figure G.1. They are derived from random distribution of 5000 arguments per range, generated to be either uniform or exponential, as appropriate. The values for extended precision floating-point arguments are given in figure G.2. They are derived from 2000 randomly-distributed arguments, generated to have one of the four types of distribution noted at the foot of each part of the figure.

Note that, in both figures, each value quoted for the maximum error refers to a particular sample and should be regarded only as a guide to the true maximum error.

Maximum and RMS values are given for short, long, and extended floating-point results.

Maximum and RMS values for the relative or (where necessary) the absolute errors are given for each function range. These are defined as follows:

Let f(x) = the true value for the function

g(x) = the calculated value for the function

Then the absolute error of the result is

ABS(f(x)-g(x))

and the relative error of the result is

ABS((f(x)-g(x))/(f))

Let the number of sample results obtained be n; then the RMS of the absolute error is:

SQRT(∑((f(x)-g(x))**2)/n)

and the RMS of the relative error is

SQRT(∑(((f(x)-g(x))/f(x))**2)/n)

Function Name	Argument Mode	Range	Short Floating Point Relative Error *10**8		Long Floating Point Relative Error *10**17	
			RMS	MAX	RMS	Max
ACOS(x)	real	ABS(x)≤0.5	43	88	7.2	20
		0.5<ABS(x)≤1	16	89	6.6	21
ASIN(x)	real	ABS(x)≤0.5	10	54	4.4	21
		0.5<ABS(x)≤1	26	94	5.9	21
ATAN(x)	real	ABS(x)<1	13	90	4.1	21
		full range[2]	25	99	5.2	17
	complex	full range[2]	21	110	5.2	44
ATAN(x_1,x_2)	real	ABS(x_1)≤1, ABS(x_2)≤1[2]	29	160	6.9	36
ATAND(x)	similar to real ATANH(x)					
ATANH(x)	real	ABS(x)≤0.2	46	110	-	-
		ABS(x)<0.9	39	120	-	-
		ABS(x)≤0.25	-	-	5.8	21
		ABS(x)≤0.95	-	-	9.0	25
	complex	full range[2]	22	120	5.6	41
COS(x)	real[1]	0≤x≤pi	4.7	12	7.3	27
		-10≤x<0,pi<x≤10	4.6	12	6.9	27
		10<ABS(x)≤100	4.6	12	100	270
	complex[3]	ABS(a)≤10,ABS(b)≤1	120	320	31	380
COSD(x)	similar to real COS(x)					
COSH(x)	real	ABS(x)≤1	41	96	-	-
		1<ABS(x)<2	21	72	-	-
		ABS(x)≤170	20	82	-	-
		ABS(x)≤17	-	-	11	39
		ABS(x)≤5	-	-	11	38
	complex[3]	ABS(a)≤10,ABS(b)≤1	97	310	25	73

[1] RMS and Max values given are absolute errors.
[2] All these ranges are distributed exponentially; all other distributions are uniform.
[3] Where (a+i*b) represents x.

Figure G.1 (Part 1 of 3). Performance statistics for the mathematical built-in functions with short and long precision floating-point arguments

Function Name	Argument Mode	Range	Short Floating Point Relative Error *10**8		Long Floating Point Relative Error *10**17	
			RMS	MAX	RMS	Max
ERF(x)	real	ABS(x)≤1	11	85	2.6	19
		1<ABS(x)<2.04	3.7	11	0.95	2.9
		2.04<ABS(x)<3.9192	3.5	6.0	-	-
		2.04<ABS(x)<6.092	-	-	0.80	1.4
ERFC(x)	real	-3.8<x<0	30	94	-	-
		-6<x<0	-	-	6.5	21
		0≤x≤1	13	69	2.7	15
		1<x≤2.04	37	200	9.1	43
		2.04<x<4	37	130	8.7	33
		4≤x<13.3	820	1500	200	350
EXP(x)	real	-1<x<1	13	44	5.4	21
		full range	12	46	4.7	43
	complex	ABS(a)≤170 ABS(b)≤pi/2	65	240	-	-
		ABS(a)≤170, pi/2<ABS(b)≤20	63	230	-	-
		ABS(a)<1 ABS(b)<pi/2	-	-	19	62
		ABS(a)<20 ABS(b)<20	-	-	20	82
LOG(x)	real	excluding 0.5<x<2.0[2]	12	84	5.5	34
		0.5<x<2.0[3]	2.5	6.8	2.4	4.7
	complex	full range[2]	38	190	13	53
LOG2(x)	real	excluding 0.5<x<2.0[2]	34	98	8.8	43
		0.5<x<2.0[3]	23	48	2.9	5.8
LOG10(x)	real	excluding 0.5<x<2.0[2]	22	110	6.6	32
		0.5<x<2.0[1]	2.3	7.2	1.2	2.9

[1] RMS and Max values given are absolute errors.
[2] All these ranges are distributed exponentially; all other distributions are uniform.
[3] Where (a+i*b) represents x.

Figure G.1 (Part 2 of 3). Performance statistics for the mathematical built-in functions with short and long precision floating-point arguments

Function Name	Argument Mode	Range	Short Floating Point Relative Error *10**8		Long Floating Point Relative Error *10**17	
			RMS	MAX	RMS	Max
SIN(x)	real[1]	ABS(x)≤pi/2	4.8	12	1.8	7.7
		pi/2<ABS(x)≤10	4.6	13	32	240
		10<ABS(x)≤100	4.6	12	93	270
	complex[3]	ABS(a)≤10,ABS(b)≤1	120	340	200	11000
SIND(x)	similar to real SIN(x)					
SINH(x)	real	ABS(x)≤1	20	88	-	-
		1<ABS(x)<2	25	100	-	-
		ABS(x)≤170	20	82	-	-
		ABS(x)≤17	-	-	10	36
		ABS(x)<0.881374	-	-	3.7	20
		0.881374<ABS(x)≤5	-	-	10	35
	complex	ABS(a)≤10,ABS(b)≤1	88	270	23	64
SQRT(x)	real	full range[2]	13	48	3.1	11
	complex	full range[2]	54	220	13	49
TAN(x)	real[4]	ABS(x)≤pi/4	29	160	6.2	39
		pi/4<ABS(x)<pi/2	37	150	-	-
		pi/4<ABS(x)<1.5	-	-	47	230
		pi/2<ABS(x)≤10	32	480	-	-
		1.5<ABS(x)≤10	-	-	7800	47000
		10<ABS(x)≤100	31	140	7800	27000
	complex[3]	ABS(a)<1,ABS(b)<9	53	290	17	71
TAND(x)	similar to read TAN(x)					
TANH(x)	real	ABS(x)≤0.7	15	78	-	-
		0.7<ABS(x)≤9.011	3.9	2.3	-	-
		ABS(x)≤0.54931	-	-	3.8	19
		0.54931<ABS(x) ≤20.101	-	-	1.0	16
	complex[3]	ABS(a)<9,ABS(b)<1	52	270	17	69

[1] RMS and Max values given are absolute errors.
[2] All these ranges are distributed exponentially; all other distributions are uniform.
[3] Where (a+i*b) represents x.
[4] Each figure here depends on the particular points encountered near the singularities of the function, where no error control can be maintained.

Figure G.1 (Part 3 of 3). Performance statistics for the mathematical built-in functions with short and long precision floating-point arguments

Function Name	Argument Mode	Range	Distribution Type (see foot of table)	Relative Error *10**34	
				RMS	Max
ACOS(x)	real	ABS(x)≤1	U	9.9	32
ASIN(x)	real	ABS(x)≤1	U	8.1	32
ATAN(x)	real	ABS(x)<10**75	T	7.3	30
	complex[2]	full range	EU	12	170
ATAN(x_1,x_2)	real	full range	EU	8.5	38
ATANH(x)	real	ABS(x)<0.25	U	8.6	28
		ABS(x)≤0.95	U	18	50
	complex[2]	full range	EU	11	59
COS(x)	real	0≤x<pi[1]	U	1.5	3.3
		-10<x<0,pi≤x<10[1]	U	1.6	3.5
		10≤ABS(x)<200[1]	U	1.6	3.5
	complex[2]	ABS(a)<10 ABS(b)<1	U U	24	62
COSH(x)	real	ABS(x)<10	U	15	61
	complex[2]	ABS(a)<10 ABS(b)<1	U U	20	67
ERF(x)	real	ABS(x)<1	U	5.3	30
		1≤ABS(x)<2.8437	U	2.3	9.2
		2.8437≤ABS(x)<5	U	1.3	1.9
ERFC(x)	real	-5<x<0	U	12	31
		0≤x<1	U	5.8	33
		1≤x<2.8437	U	28	77
		2.8437≤x<5	U	180	490

[1]RMS and Max values are for absolute errors [2]Where x=a+i*b

E exponential
U uniform (linear)
T tangents of linearly-scaled angles in (-pi/2,pi/2)

EU a+i*b=r*EXP(i*k) where x=a+i*b or (ATAN only) x_1=a, x_2=b, and:
r has E distribution in (0,10**75)
k has U distribution in (-pi,pi)

Figure G.2 (Part 1 of 3). Performance statistics for the mathematical built-in functions with extended-precision floating-point arguments

Function Name	Argument Mode	Range	Distribution Type (see foot of table)	Relative Error *10**34	
				RMS	Max
EXP(x)	real	ABS(x)<1	U	4.3	15
		ABS(x)<10	U	3.8	15
		-180<x<174	U	3.7	15
	complex[2]	ABS(a)<170 ABS(b)<pi/2	U U	7.8	35
		ABS(a)<170 pi/2≤ABS(b)<100	U U	8.0	33
LOG(x)	real	0.99<x<1.01[1]	U	0.084	0.20
		0.5<x<2[1]	U	1.7	3.2
		10**-78<x<10**75	E	8.9	45
	complex[2]	full range	EU	9.8	51
LOG2(x)	real	0.99<x<1.01[1]	U	0.055	0.13
		0.5<x<2[1]	U	1.0	1.9
		10**-78<x<10**75	E	4.4	30
LOG10(x)	real	0.99<x<1.01[1]	U	0.038	0.16
		0.5<x<2[1]	U	1.5	2.9
		10**-78<x<10**75	E	12	38
SIN(x)	real	ABS(x)<pi/2[1]	U	1.2	3.0
		pi/2≤ABS(x)<10[1]	U	1.6	3.5
		10≤ABS(x)<200[1]	U	1.5	3.6
	complex[2]	ABS(a)<10 ABS(b)<1	U U	24	60
SINH(x)	real	ABS(x)<1	U	6.8	29
		1≤ABS(x)<10	U	13	54
	complex[3]	ABS(a)<10 ABS(b)<1	U U	18	53

[1]RMS and Max values are for absolute errors [2]Where x=a+i*b

E exponential
U uniform (linear)
T tangents of linearly-scaled angles in (-pi/2,pi/2)

EU a+i*b=r*EXP(i*k) where x=a+i*b or (ATAN only) x_1=a, x_2=b, and:
r has E distribution in (0,10**75)
k has U distribution in (-pi,pi)

Figure G.2 (Part 2 of 3). Performance statistics for the mathematical built-in functions with extended-precision floating-point arguments

Function Name	Argument Mode	Range	Distribution Type (see foot of table)	Relative Error *10**34 RMS	Relative Error *10**34 Max
SQRT(x)	real	10**-50<x<10**50	E	3.0	15
		10**-78<x<10**75	E	2.8	14
	complex[2]	full range	EU	7.1	21
TAN(x)	real	ABS(x)<pi/4	U	9.6	36
		pi/4≤ABS(x)<pi/2	U	8.9	39
		pi/2≤ABS(x)<10	U	12	52
		10≤ABS(x)<200	U	11	46
	complex[2]	ABS(a)<1 ABS(b)<9	U U	15	61
TANH(x)	real	ABS(x)<0.54931	U	5.0	25
		0.54931≤ABS(x)<5	U	2.6	21
	complex[2]	ABS(a)<9 ABS(b)<1	U U	15	53

[1]RMS and Max values are for absolute errors [2]Where x=a+i*b

E exponential
U uniform (linear)
T tangents of linearly-scaled angles in (-pi/2,pi/2)

EU a+i*b=r*EXP(i*k) where x=a+i*b
or (ATAN only) x_1=a, x_2=b, and:
r has E distribution in (0,10**75)
k has U distribution in (-pi,pi)

Figure G.2 (Part 3 of 3). Performance statistics for the mathematical built-in functions with extended-precision floating-point arguments

AGGREGATE ARGUMENTS

The only functions that can accept structure arguments are ADDR, ALLOCATION, and STRING.

All built-in functions that can have arguments can have array arguments. But whereas ADDR, ALLOCATION, STRING, and the Array-handling functions return single values, all other functions return an array of values. Thus for functions such as SUBSTR, any one of the arguments can be an array (if more than one is an array, the bounds must be identical). This facility is equivalent to placing the function reference in a DO-loop where one or more arguments is a subscripted array reference that is modified by the control variable.

NULL ARGUMENTS

A number of built-in functions do not require arguments. It should be noted that the functions must either be explicitly declared with the BUILTIN attribute or contextually declared by including a null argument list in the function reference, e.g., ONCHAR(). Otherwise, the name cannot be recognized by the compiler as a built-in function name.

The functions without arguments are:

DATAFIELD
DATE
EMPTY
NULL
ONCHAR
ONCODE
ONCOUNT
ONFILE
ONKEY
ONLOC
ONSOURCE
PRIORITY (when optional argument for pseudovariable omitted)
STATUS (when optional argument omitted)
TIME

PSEUDOVARIABLES

Certain built-in functions can be used to represent receiving fields. In this form they are pseudovariables. Except when noted in the description of the pseudovariable, it can appear on the left of the equal sign in an assignment or DO statement; it can appear in a data list of a GET statement; and it can appear as the string name in a KEYTO, STRING, or REPLY option.

Since all pseudovariables are also built-in functions, only a short description is given in the relevant function description.

Note that pseudovariables cannot be nested; for example, the following statement is invalid:

```
UNSPEC(SUBSTR(A,1,2)) = '00'B;
```

The pseudovariables are:

COMPLETION	REAL
COMPLEX	STATUS
IMAG	STRING
ONCHAR	SUBSTR
ONSOURCE	UNSPEC
PRIORITY	

ABS(x) — Arithmetic

ABS returns the absolute value of a given expression x. If x is real, it is the positive value of x; if x is complex, it is the positive square root of the sum of the squares of the real and imaginary parts.

If x is fixed and complex with precision (p,q), the precision of the result is given by:

(MIN(N,p+1),q)

where N is the maximum allowable number of digits.

ACOS(x) — Mathematical

ACOS returns a floating-point value that represents the inverse (arc) cosine in radians of a given value x.

x must be real, and the absolute value must be less than or equal to 1, i.e., ABS(x)<=1. The result is in the range:

0<=ACOS(x)<=pi

ADD(x_1,x_2,x_3[,x_4]) — Arithmetic

ADD returns the sum of two values x_1 and x_2 with a precision specified by x_3 and x_4.

x_1 and x_2 values to be added.

x_3 unsigned decimal integer constant specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

x_4 decimal integer constant, optionally signed, specifying the scale factor of the result. For a fixed-point result, if x_3 is given, then x_4 must also be given. For a floating-point result, only x_3 can be given.

ADDR(x) — Storage Control

ADDR returns a pointer value that identifes the location at which a given variable x has been allocated.

x a variable of any data type and organization and of any storage class except:

1. A BASED, DEFINED, parameter, subscripted, or structure-base-element variable that is an unaligned fixed-length bit string.
2. A minor structure whose first base element is an unaligned fixed-length bit string (except where it is also the first element of the containing major structure).

3. A major structure that has the DEFINED attribute or is a parameter, and that has an unaligned fixed-length bit string as its first element.

4. A variable not in connected storage.

If x is an aggregate, the returned value identifies the first element.

If x is a varying string, the returned value identifies the two-byte prefix.

If x is an area, the returned value identifies the control information.

If x is a controlled variable that has not been allocated, the null pointer value is returned.

If x is a parameter and a dummy argument has been created, the returned value identifies the dummy argument.

Note that because of condition 4 above, if x is a parameter, it must have the CONNECTED attribute.

ALL(x) — Array-Handling

ALL returns a bit string in which each bit is 1 if the corresponding bit in each element of the given array x exists and is 1. The length of the result is equal to that of the longest element.

If x is not a bit-string array, it is converted to bit string. It must not be iSUB-defined.

ALLOCATION(x) — Storage Control

Abbreviation: ALLOCN

ALLOCATION returns a default-precision fixed-point binary integer specifying the number of generations that can be accessed in the current task for a given controlled variable x.

x the name of the controlled variable. The name must be level one and unsubscripted.

If x is not allocated, the result is zero.

ANY(x) — Array-Handling

ANY returns a bit string where each bit is 1 if the corresponding bit in any element of the given array x exists and is 1. The length of the result is equal to that of the longest element.

If x is not a bit-string array, it is converted to bit string. It must not be iSUB-defined.

ASIN(x) — Mathematical

ASIN returns a floating-point value that represents the inverse (arc) sine in radians of a given value a.

x must be real, and the absolute value must be less than or equal to 1, i.e., ABS(x)<=1. The result is in the range:

-pi/2<=ASIN(x)<=pi/2

ATAN(x1[,x2]) — Mathematical

ATAN returns a floating-point value that represents the inverse (arc) tangent in radians of a given value x_1 or of a given ratio x_1/x_2.

If x_1 alone is specified and is real, the result is in the range:

-pi/2<ATAN(x_1)<pi/2

If x_1 alone is specified and is complex, it must not be +i or -i. The result is given by:

-i*ATANH(i*x_1)

If x_1 and x_2 are specified, they must both be real. It is an error if x_1 and x_2 are both zero. The results for all other values of x_1 and x_2 are given by:

arctan(x_1/x_2) for $x_2>0$

pi/2 for $x_2=0$ and $x_1>0$

-pi/2 for $x_2=0$ and $x_1<0$

pi + arctan(x_1/x_2) for $x_2<0$ and $x_1\geq 0$

-pi+arctan(x_1/x_2) for $x_2<0$ and $x_1<0$

ATAND(x_1[,x_2]) Mathematical

ATAND returns a floating-point value that represents the inverse (arc) tangent in degrees of a given value x_1 or of a given ratio x_1/x_2.

If x_1 alone is specified it must be real. The result is in the range:

-90<ATAND(x_1)<90

If x_1 and x_2 are specified, they must both be real. The result is defined in terms of the function ATAN as:

180/pi*ATAN(x_1,x_2)

ATANH(x) Mathematical

ATANH returns a floating-point value that represents the inverse (arc) hyperbolic tangent of a given value x.

If x is real, its absolute value must be less than 1, i.e., ABS(x)<1.

If x is complex, it must not be +1 or -1. The result is defined as:

LOG((1+x)/(1-x))/2

BINARY(x_1[,x_2[,x_3]]) Arithmetic

BINARY returns the binary representation of a given value x_1 with a precision specified by x_2 and x_3.

x_1 value to be converted to binary base.

x_2 unsigned decimal integer specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

x_3 decimal integer, optionally signed, specifying the scale factor of the result. For a fixed-point result, if x_2 is given, then x_3 must also be given. For a floating-point result, only x_2 can be given. If both x_2 and x_3 are omitted, the precision of the result is determined from the rules for base conversion.

BIT(x_1[,x_2]) String-Handling

Bit returns a bit string representation of a given value x_1.

x_1 expression to be converted.

x_2 an expression that can be converted to integer specifying the length of the resulting bit string. If necessary, x_2 is converted to a binary integer of precision (15,0). If x_2 is omitted, the length is determined by the rules for type conversion.

BOOL(x_1,x_2,x_3) String-Handling

BOOL returns a bit string that is the result of a Boolean operation, specified by x_3, on bit strings x_1 and x_2. The length of the result is equal to that of the longer operand, x_1 or x_2.

x_1 and x_2 bit-string expressions or expressions that may be converted to bit strings.

x_3 bit string of four bits. Each bit specifies the result when a bit from x_1 is compared with the corresponding bit from x_2 as follows:

x_1	x_2	result
0	0	bit 1 of x_3
0	1	bit 2 of x_3
1	0	bit 3 of x_3
1	1	bit 4 of x_3

If x_1 and x_2 are different lengths, the shorter is padded on the right with zeros to match the longer. If x_3 is not a bit string expression of length 4, it will be converted and padded on the right with zeros or truncated on the right, as necessary.

CEIL(x) Arithmetic

CEIL returns the smallest integer greater than or equal to a given value x. x must be real.

If x is fixed-point with precision (p,q), the precision of the result is given by:

(MIN(N,MAX(p-q+1,1)),0)

where N is the maximum number of digits allowable.

CHAR(x_1[,x_2]) String-Handling

CHAR returns a character string representation of a given value x_1.

x_1 expression to be converted.

x_2 an expression that can be converted to integer specifying the length of the resulting character string. If necessary, x_2 is converted to a binary integer of precision (15,0). If x_2 is omitted, the length is determined by the rules for type conversion.

COMPLETION(x) Multitasking

COMPLETION returns a single bit specifying the completion value of a given event x. If the event is incomplete, '0'B is returned; if complete, '1'B is returned.

COMPLETION Pseudovariable

The pseudovariable sets the completion value of the given event x. x must be inactive. No interrupt can occur during assignment to the pseudovariable. The COMPLETION pseudovariable cannot be used as the control variable in a DO-goup.

COMPLEX(x_1,x_2) Arithmetic

Abbreviation: CPLX(x_1,x_2)

COMPLEX returns a complex value formed from two given values x_1 and x_2.

x_1 real value that is to be the real part of the result.

x_2 real value that is to be the imaginary part of the result.

If x_1 and x_2 differ in base, the decimal one is converted to binary; if they differ in scale, the fixed-point is converted to floating-point. The result will have the same base and scale. Both x_1 and x_2 must be real.

The precision of the result, if fixed-point, is given by:

(MIN(N,MAX(p_1-q_1,p_2-q_2)+MAX(q_1,q_2)), MAX(q_1,q_2))

where (p_1,q_1) and (p_2,q_2) are the precisions of x_1 and x_2 respectively, and N is the maximum number of digits allowable.

If the arguments, after any necessary conversions have been performed, are floating point, and their precisions are p_1 and p_2, then the precision of the result is MAX(p_1,p_2).

COMPLEX Pseudovariable

The pseudovariable assigns the real part of a complex value to the variable x_1 and the imaginary part to the variable x_2. Only a complex value can be assigned to the pseudovariable. The COMPLEX pseudovariable cannot be used as the control variable in a DO-group.

CONJG(x) Arithmetic

CONJG returns the conjugate of a given complex value x, i.e., the same value with the sign of the imaginary part reversed. If x is real, it will be converted to complex.

COS(x) Mathematical

COS returns a floating-point value that represents the cosine of a given value x.

x an expression whose value is in radians.

If x is complex, the result is given by:

cos(a)*cosh(b)-i*sin(a)*sinh(b)

where (a+i*b) represents x.

COSD(x) Mathematical

COSD returns a floating-point value that represents the cosine of a given value x.

x an expression whose value is in degrees. x must be real.

COSH(x) Mathematical

COSH returns a floating-point value that represents the hyperbolic cosine of a given value x.

If x is complex, the result is given by:

```
cosh(a)*cos(b)+i*sinh(a)*sin(b)
```

where (a+i*b) represents x.

COUNT(x) STREAM Input/Output

COUNT returns a binary integer of default precision specifying the number of data items transmitted during the last GET or PUT operation on the specified file x.

x a file expression, the file must have the STREAM attribute.

Note that if an on-unit or procedure is entered during a GET or PUT operation and, within that on-unit or procedure, a GET or PUT operation is executed for the same file, the value of COUNT is reset for the new operation; it is restored when the original GET or PUT is continued.

DATAFIELD Condition-Handling

DATAFIELD is used in a NAME condition on-unit to return a character string whose value is the name and contents of the field that caused the condition to be raised.

It can also be used in an on-unit for an ERROR or FINISH condition raised as part of the standard system action for the NAME condition.

If DATAFIELD is used out of context, a null string is returned.

DATE

DATE returns a character string of length six, in the form yymmdd, where:

yy	the current year
mm	the current month
dd	the current day

DECIMAL(x_1[,x_2[,x_3]]) Arithmetic

Abbreviation: DEC(x_1[,x_2[,x_3]])

DECIMAL returns the decimal representation of a given value a_1 with a precision specified by x_2 and x_3.

x_1 value to be converted to decimal base.

x_2 unsigned decimal integer constant specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

x_3 decimal integer constant, optionally signed, specifying the scale factor of the result. For a fixed-point result, if x_2 is given, then x_3 must also be given. For a floating-point result, only x_2 can be given.

If both x_2 and x_3 are omitted, the precision of the result is determined from the rules for base conversion.

DIM(x_1,x_2) Array-Handling

DIM returns a default-precision fixed-point binary integer specifying the current extent of a specified dimension x_2 of a given array x_1.

x_1 the given array; it must be currently allocated.

x_2 the element expression specifying a particular dimension of x_1. If necessary, x_2 is converted to a binary integer of precision (15,0).

x_1 must not have less than (x_2) dimensions.

DIVIDE(x_1,x_2,x_3[,x_4]) Arithmetic

DIVIDE returns the quotient of two values x_1 and x_2 with a precision specified by a_3 and x_4.

x_1 dividend

x_2 divisor

x_3 unsigned decimal integer constant specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

x_4 decimal integer constant, optionally signed, specifying the scale factor of the result. For a fixed-point result,

both x_3 and x_4 must be given. For a floating-point result, only x_3 can be given.

EMPTY — Storage Control

EMPTY returns an area of zero extent. It is used to free all allocations in an area. Note that the value of this function is automatically assigned to an area variable when it is allocated.

ERF(x) — Mathematical

ERF returns a floating-point value that represents the error function of a given value x. x must be real.

The result is given by:

$$ERF(x) = \frac{2}{\sqrt{\Pi}} \int_0^x e^{-t^2} dt$$

ERFC(x) — Mathematical

ERFC returns a floating-point value that represents the complement of the error function of a given value x. x must be real.

The result is defined in terms of the function ERF as:

```
1-ERF(x)
```

EXP(x) — Mathematical

EXP returns a floating-point value that represents the base of the natural logarithm system e to a given power x.

FIXED(x_1[,x_2[,x_3]]) — Arithmetic

FIXED returns the fixed-point representation of a given value x_1 with a precision specified by x_2 and x_3.

x_1 value to be converted to fixed-point scale.

x_2 unsigned decimal integer constant specifying the total number of digits in the result.

x_3 decimal integer constant, optionally signed, specifying the scale factor of the result. If x_3 is omitted, a scale factor of zero is assumed.

If both x_2 and x_3 are omitted, the default value (15,0), for a binary result, or (5,0), for a decimal result, is assumed.

FLOAT(x_1[,x_2]) — Arithmetic

FLOAT returns the floating-point representation of a given value x_1 with a precision specified by x_2.

x_1 value to be converted to floating-point scale.

x_2 unsigned decimal integer constant specifying the total number of digits in the result. If x_2 is omitted, the default value 21, for a binary result, or 6, for a decimal result, is assumed.

FLOOR(x) — Arithmetic

FLOOR returns the largest integer less than or equal to a given value x. x must be real.

If x is fixed-point with precision (p,q), the precision of the result is given by:

```
(MIN(N,MAX(p-q+1,1)),0)
```

where N is the maximum number of digits allowable.

HBOUND(x_1,x_2) — Array-Handling

HBOUND returns a default-precision fixed-point binary integer specifying the current upper bound of a specified dimension x_2 of a given array x_1.

x_1 the given array; it must be currently allocated.

x_2 an element expression specifying a particular dimension of x_1. If necessary, x_2 is converted to a binary integer of precision (15,0).

x_1 must not have less than (x_2) dimensions.

HIGH(x) String-Handling

HIGH returns a character string of length x where each character is the highest character in the collating sequence (hexadecimal FF).

x expression specifying the length. If necessary, x is converted to a binary integer of precision (15,0).

IMAG(x) Arithmetic

IMAG returns the imaginary part of a given complex value x. If real, x is converted to complex. The mode of the result is real.

IMAG Pseudovariable

The pseudovariable assigns a real value or real part of a complex value to the imaginary part of a given complex variable x. x must be complex.

INDEX(x_1,x_2) String-Handling

INDEX returns a halfword binary integer indicating the starting position within the string x_1 of a substring identical to string x_2.

x_1 string to be searched

x_2 string to be searched for

If x_2 does not occur in x_1, the value zero is returned.

If x_2 occurs more than once in x_1, the starting position of the first occurrence is returned.

If any argument is character or decimal, conversions are performed to produce character strings. Otherwise if the arguments are bit and binary, or both binary, conversions are performed to produce bit strings.

LENGTH(x) String-Handling

LENGTH returns a default-precision fixed-point binary integer specifying the current length of a given string x. If x is binary, it is converted to bit string; otherwise any other conversion required is to character string.

LBOUND(x_1,x_2) Array-Handling

LBOUND returns a default-precision fixed-point binary integer specifying the current lower bound of a specified dimension x_2 of a given array x_1.

x_1 the given array; it must be currently allocated.

x_2 an element expression specifying the particular dimension of x_1. If necessary, x_2 is converted to a binary integer of precision (15,0).

x_1 must not have less than (x_2) dimensions.

LINENO(x) STREAM Input/Output

LINENO returns a default-precision fixed-point binary integer specifying the current line number of the specified file x.

x a file expression, the file must have the PRINT attribute.

LOG(x) Mathematical

LOG returns a floating-point value that represents the natural logarithm, i.e., base e, of a given value x. If x is real, it must be greater than zero. If x is complex, it must not be equal to 0+0I. The function is multiple-valued if x is complex; hence, only the principal value can be returned. The principal value has the form:

(a+i*b)

where b is the range:

-pi<b<=pi.

LOG2(x) Mathematical

LOG2 returns a floating-point value that represents the binary logarithm, i.e., base 2, of a given value x. x must be real and greater than zero.

LOG10(x) Mathematical

LOG10 returns a floating-point value that represents the common logarithm, i.e., base 10, of a given value x. x must be real and greater than zero.

LOW(x) String-Handling

LOW returns a character string of length x where each character is the lowest character in the collating sequence (hexadecimal 00). If necessary, x is converted to binary integer of precision (15,0).

x expression specifying the length

MAX(x_1,x_2...,x_n) Arithmetic

MAX returns, from a set of two or more arguments, the value of the argument with the largest value.

x_1,x_2...,x_n list of values from which the largest is to be returned.

The maximum number of arguments that the function will accept is 64. All the arguments must be real.

If the arguments are fixed-point with precisions:

(p_1,q_1), (p_2,q_2)...,(p_n,q_n)

the precision of the result is given by:

(MIN(N,MAX(p_1-q_1,p_2-q_2...,p_n-q_n)+
MAX(q_1,q_2...,q_n)),MAX(q_1,q_2...,q_n))

If the arguments, after any necessary conversions have been performed, are floating point, and their precisions are p_1,p_2,p_3...p_n, then the precision of the result is MAX(p_1,p_2,p_3...p_n).

MIN(x_1,x_2...,x_n) Arithmetic

MIN returns, from a set of two or more arguments, the value of the argument with the smallest value.

x_1,x_2...,x_n list of values from which the smallest is to be returned. smallest is to be returned.

The maximum number of arguments that the function will accept is 64. All the arguments must be real.

If the arguments are fixed-point with precisions:

(p_1,q_1),(p_2,q_2)...,(p_n,q_n)

the precision of the result is given by:

(MIN(N,MAX(p_1-q_1,p_2-q_2...,p_n-q_n)+
MAX(q_1,q_2...,q_n)),MAX(q_1,q_2...,q_n))

If the arguments, after any necessary conversions have been performed, are floating point, and their precisions are p_1,p_2,p_3...p_n, then the precision of the result is MAX(p_1,p_2,p_3...p_n).

MOD(x_1,x_2) Arithmetic

MOD returns the smallest positive value, R, such that:

$(x_1 - R)/x_2 = n$ where n is an integer.

Thus, the result is the smallest positive value that must be subtracted from a given value x_1 to make it exactly divisible by the given value x_2.

x_1 must be real. If x_1 is positive, the result is the remainder of the division of x_1 and x_2; if x_1 is negative, the result is the modular equivalent of this remainder.

x_2 must be real. If x_2 is zero, the ZERODIVIDE condition is raised. If the result is floating-point, the precision is the greater of those of x_1 and x_2; if the result is fixed-point, the precision is given by:

(MIN(N,p_2-q_2+MAX(p_1,q_2)),MAX(q_1,q_2))

where (p_1,q_1) and (p_2,q_2) are the precisions of x_1 and x_2 respectively.

If x_1 and x_2 are fixed-point with different scale factors, the result may be truncated on the right, and the SIZE condition is raised, if enabled.

MULTIPLY(x_1,x_2,x_3[,x_4])

MULTIPLY returns the product of two values x_1 and x_2 with a precision specified by x_3 and x_4.

x_1 and x_2 values to be multiplied.

x_3 unsigned decimal integer constant specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

x_4 decimal integer constant, optionally signed, specifying the scale factor of the result. For a fixed-point result, both x_3 and x_4 must be given. For a floating-point result, only x_3 can be given.

NULL — Storage Control

NULL returns a null pointer value, i.e., a value that cannot identify any generation of a variable. The null pointer value can be converted to OFFSET by assignment of the built-in function value to an offset variable.

OFFSET(x_1,x_2) — Storage Control

OFFSET returns an offset value derived from a given pointer x_1 and relative to a given area x_2. If x_1 is null, then the null value is returned.

x_1 a pointer expression. It must identify a generation of a based variable within area x_2.

x_2 an area variable

If x_1 is an element expression, then x_2 must be an element variable.

ONCHAR — Condition-Handling

ONCHAR returns the character that caused the CONVERSION condition to be raised. It can be used in an on-unit for the CONVERSION condition or for ERROR or FINISH condition raised as standard system action for the CONVERSION condition.

If the ONCHAR built-in function is used out of context, a blank is returned unless ONCHAR has a value given to it by an assignment to the pseudovariable out of context; in this case, the character assigned to the pseudovariable is returned by the built-in function.

ONCHAR Pseudovariable

The pseudovariable resets the current value of the ONCHAR built-in function. The value assigned to the pseudovariable is converted to a character string of length 1. The new character is used when the conversion is re-attempted.

If the pseudovariable is used out of context, and the next reference to the built-in function is also out of context, then the character assigned to the pseudovariable is returned. The out-of-context assignment is otherwise ignored.

ONCODE — Condition-Handling

ONCODE returns a default-precison fixed-point binary integer that defines the type of interrupt that caused the on-unit to become active. It can be used in any on-unit. All ON-codes are defined in section H, "On-Conditions".

If ONCODE is used out of context, zero is returned.

ONCOUNT — Condition-Handling

ONCOUNT returns a default-precision fixed-point binary integer specifying the number of interrupts that remain when an on-unit is entered. Both types of multiple interrupt are discussed in section H, "ON-Conditions".

If ONCOUNT is used out of context, zero is returned.

ONFILE — Condition-Handling

ONFILE returns a character string whose value is the name of the file for which an input/output or CONVERSION condition is raised. It can be used in an on-unit for any input/output or CONVERSION condition, or for the ERROR or FINISH condition raised as standard system action for an input/output or the CONVERSION condition.

If ONFILE is used out of context, a null string is returned.

ONKEY — Condition-Handling

ONKEY returns a character string whose value is the key of the record that caused an input/output condition to be raised. It can be used in an on-unit for any input/output condition, except ENDFILE, or for the ERROR or FINISH condition raised as standard system action for an input/output condition. Note that ONKEY is always set

for operations on a KEYED file, even if the statement that causes the condition to be raised has not specified the KEY, KEYTO, or KEYFROM options.

The result is determined by the following rules:

1. For any input/output condition (other than ENDFILE), or for the ERROR or FINISH condition raised as standard system action for these conditions, the result is the value of the recorded key from the I/O statement causing the error.

   For REGIONAL(1) data sets, the result is an eight-byte character representation of the region number. If the key was incorrectly specified, the result is the last eight bytes of the source key. If the source key is less than eight bytes, it is padded on the right with blanks to make it eight bytes. If the key was correctly specified, the eight-byte character string consists of the region number in character form padded on the left with blanks, if necessary.

2. For a REWRITE statement that attempts to write an updated record on to an indexed data set when the key of the updated record differs from that of the input record, the result is the value of the embedded key of the input record.

If ONKEY is used out of context, a null string is returned.

ONLOC — Condition-Handling

ONLOC returns a character string whose value is the name of the entry-point of the procedure in which the condition was raised. It can be used in any on-unit.

If ONLOC is used out of context, a null string is returned.

ONSOURCE — Condition-Handling

ONSOURCE returns a character string whose value is the contents of the field that was being processed when the CONVERSION condition was raised. It can be used in an on-unit for the CONVERSION condition or for the ERROR or FINISH condition raised as standard system action for the CONVERSION condition.

If ONSOURCE is used out of context, a null string is returned.

ONSOURCE Pseudovariable

The pseudovariable resets the current value of the ONSOURCE built-in function. The value assigned to the pseudovariable is converted to a character string and, if necessary, is padded on the right with blanks to match the length of the field that caused the error. The new string is used when the conversion is re-attempted.

When conversion is re-attempted, the string assigned to the pseudovariable is processed as a single data item. For this reason, the error correction process should not assign a string containing more than one data item when the conversion occurs during the execution of a GET LIST or GET DATA statement. The presence of blanks or commas in the string will cause further conversion error.

POINTER(x_1,x_2) — Storage Control

Abbreviation: PTR(x_1,x_2)

POINTER returns a pointer value derived from a given offset value x_1 and a given area x_2. If x_1 is null then the null value is returned.

x_1 an offset expression. It must identify a generation of a based variable, but not necessarily in x_2. If it is not in x_2, the generation must be equivalent to one in x_2.

x_2 an area variable.

Generations of based variables in different areas are equivalent if, up to the allocation of the latest generation, the variables have been allocated and freed the same number of times as each other.

POLY(x_1,x_2) — Array-Handling

POLY returns a floating-point value that represents a polynomial formed from two given unidimensional arrays x_1 and x_2.

x_1 an array defined as a(m:n), where (m:n) represents the lower and upper bounds.

x_2 an array defined as x(p:q), where (p:q) represents the lower and upper bounds.

If the elements of one or both of the arrays are not floating-point, they are converting to floating-point

The returned value is defined as:

$$a(m) + \sum_{j=1}^{n-m} (a(m+j) * \prod_{i=0}^{j-1} x(p+i))$$

If (q-p)<(n-m-1) then x(p+i)=x(q) for (p+i)>q.

If m=n then the result is a(m).

If x_2 is an element expression, it is interpreted as an array of one element, i.e., x(1), and the result is defined as:

$$\sum_{j=0}^{n-m} a(m+j)*x(1)**j$$

x_1 must not be iSUB-defined.

PRECISION(x_1,x_2[,x_3]) Arithmetic

Abbreviation: PREC(x_1,x_2[,x_3])

PRECISION returns a given value x_1 with a precision specified by x_2 and x_3.

x_1 value whose precision is to be changed.

x_2 unsigned decimal integer constant specifying the number of digits that the value of x_1 is to have after conversion; it must not exceed the implementation limit.

x_3 decimal integer constant, optionally signed, specifying the scale factor of the result. For a fixed-point result, x_3 must be given. For a floating-point result, x_3 must not be given.

PRIORITY(x) Multitasking

PRIORITY returns a halfword binary integer indicating the priority associated with the given task variable x. It gives the priority relative to the priority of the current task. No interrupt can occur during evaluation of PRIORITY.

PRIORITY Pseudovariable

The pseudovariable adjusts the priority of the task associated with the given task variable x. It receives a halfword binary integer, and sets the priority of a to the received value, relative to the priority held by the current task immediately prior to the assignment. x may be associated with any active or inactive task, including the current one. The task variable may be omitted, in which case the variable associated with the current task is assumed. No interrupt can occur during assignment to the PRIORITY pseudovariable. The PRIORITY pseudovariable cannot be used as the control variable in a DO group.

PROD(x) Array-Handling

PROD returns the product of all the elements in a given array x.

x an array of integers or floating-point elements.

If the elements of x are non-integer fixed-point, they are converted to floating-point.

If the elements of x are string, they are converted to integers. The precision of the result for fixed-point integers is (N,0), where N is the maximum number of digits allowable. x must not be iSUB-defined.

REAL(x) Arithmetic

REAL returns the real part of a given complex value x. x will be converted to complex if it is real.

REAL Pseudovariable

The pseudovariable assigns a real value or the real part of a complex value to the real part of a given complex variable x. x must not be real.

REPEAT(x_1,x_2) String-Handling

REPEAT returns a string consisting of the string x_1 concatenated to itself the number of times specified by x_2, i.e., there will be (x_2+1) occurrences of the string x_1.

x_1 string to be repeated

x_2 an expression that can be converted to integer indicating number of repetitions.

If x_1 is arithmetic, it will be converted to string - bit string if it is binary, character string if it is arithmetic. If x_2 is zero or negative, the string x_1 is returned.

If x_2 is an array, then x_1 should be an array with identical bounds.

ROUND(x_1,x_2) Arithmetic

ROUND returns the given value x_1 rounded at a digit specified by x_2.

x_1 the value to be rounded.

x_2 decimal integer constant, optionally signed, specifying the digit at which rounding is to occur. If x_2 is positive, it is the (x_2)th digit to the right of the point; if negative, it is the (x_2+1)th digit to the left of the point.

If x_1 is floating-point, x_2 is ignored; the rightmost bit of the mantissa is set to 1.

The precision of a fixed-point result is given by:

(MAX(1,MIN(p-q+1+x_2,N)),x_2)

where (p,q) is the precision of x_1 and N is the maximum number of digits allowable.

Note that the rounding of a negative value results in the rounding of its absolute value, then the sign is replaced.

SIGN(x) Arithmetic

SIGN returns a default-precision fixed-point binary integer that indicates whether a given value x is positive, zero, or negative. The value returned is as follows:

value of x	value returned
x > 0	+1
x = 0	0
x < 0	-1

x must be real.

SIN(x) Mathematical

SIN returns a floating-point value that represents the sine of a given value x.

x an expression whose value is in radians.

If x is complex, the result is given by:

sin(a)*cosh(b)+i*cos(a)*sinh(b)

where (a+i*b) represents x.

SIND(x) Mathematical

SIND returns a floating-point value that represents the sine of a given value x.

x an expression whose value is in degrees. x must be real.

SINH(x) Mathematical

SINH returns a floating-point value that represents the hyperbolic sine of a given value x. If x is complex, the result is given by:

sinh(a)*cos(b)+i*cosh(a)*sin(b)

where (a+i*b) represents x.

SQRT(x) Mathematical

SQRT returns a floating-point value that represents the square root of x. If x is real, it must not be less than zero. The result is the positive square root of x. If x is complex, the function is multiple-valued; hence, only the principal value can be returned. The principal value has the form:

(a+i*b)

where either a>0, or a=0 and b>=0.

STATUS[(x)] Multitasking

STATUS returns a default-precision fixed-point binary integer specifying the status value of a given event x. If the event is normal, zero is returned; if abnormal, non-zero is returned. If no argument is specified, the event associated with the current task is assumed.

STATUS Pseudovariable

The pseudovariable resets the status value of a given event x. No interrupt can occur during assignment to the pseudovariable.

STRING(x) String-Handling

STRING returns an element string that is the concatenation of all the elements of a string data aggregate.

x an array or structure expression whose elements are either all character strings and/or numeric character data, or all bit strings.

x cannot be an operational expression or a function reference.

If x is a structure that has padding caused by ALIGNED elements, the padding is not included in the result.

If any of the strings in the aggregate x are of varying length, only the current length, and not including the two-byte length prefix, is concatenated.

x must not be iSUB-defined.

If x is an element variable, the rules for aggregates apply except that there is no concatenation.

STRING Pseudovariable

The pseudovariable assigns a string, piece by piece, to the given aggregate variable x, until either all of the aggregate elements are filled or no piece of the assigned string remains. In the latter case, any remaining strings in the aggregate variable are filled with blanks or, if varying-length, are given zero length.

The STRING pseudovariable must not be used in the data specification of a GET statement, nor in an INTO or KEYTO option of a READ statement.

The STRING pseudovariable cannot be used as the control variable in a DO-group.

A varying-length string is filled to its maximum length.

SUBSTR(x_1,x_2[,x_3]) String-handling

SUBSTR returns a substring of the given string x_1.

x_1 string from which the substring is to be extracted.

x_2 an expression that can be converted to integer indicating the starting position of the substring in x_1.

x_3 an expression that can be converted to integer specifying the length of the substring in x_1. If x_3 is zero, a null string is returned. If x_3 is omitted, the substring returned is position x_2 in x_1 to the end of x_1.

If x_1 is not a string, it is converted to a bit string if binary or a character string if decimal.

The STRINGRANGE condition, if enabled, is raised if the values of x_2 and x_3 are such that the substring does not lie entirely within x_1. If STRINGRANGE is not enabled, then under the optimizing compiler the result is undefined and under the checkout compiler, standard system action is taken (even if there is STRINGRANGE on-unit established.)

SUBSTR Pseudovariable

The pseudovariable assigns a string to a substring of the given string x. The remainder of string x is unchanged.

SUM(x) Array-handling

SUM returns the sum of all the elements in a given array x.

x an array of arithmetic elements.

If the elements of x are fixed-point, the precision of the result is (N,q), where N is the maximum number of digits allowable and q is the scale factor of x. If the elements of x are strings, they are converted to integers.

x must not be iSUB-defined.

TAN(x) Mathematical

TAN returns a floating-point value that represents the tangent of a given value x.

x an expression whose value is in radians.

If x is complex, the result is defined as:

```
REAL (TAN(x)) = TAN(a)*(1-TANH(b)**2)/
                (1+(TAN(a)*TANH(b))**2)

IMAG(TAN(x)) = TANH(b)*(1+TAN(a)**2)/
               (1+(TAN(a)*TANH(b))**2)
```

where (a+i*b) represents x.

TAND(x) Mathematical

TAND returns a floating-point value that represents the tangent of a given value x.

x an expression whose value is in degrees. x must be real.

TANH(x) Mathematical

TAND returns a floating-point value that represents the hyperbolic tangent of a given value x.

x an expression whose value is in radians.

If x is complex the result is defined as:

```
-i*TAN(i*x)
```

TIME

TIME returns a character string of length nine, in the form hhmmssttt, where:

hh	the current hour
mm	number of minutes
ss	number of seconds
ttt	number of milliseconds

If no timing facility is available, TIME returns the value (9)'0'.

TRANSLATE(x_1,x_2[,x_3]) String-handling

TRANSLATE returns a string the same length as a given string x_1 where all or some of the characters may have been changed. Characters are changed according to a look-up table provided by strings x_2 and x_3.

The function operates on each character of x_1 as follows:

If a character in x_1 is found in x_3, then the character in x_2 that corresponds to the one in x_3 is copied to the result; otherwise, the character in x_1 is copied directly to the result.

x_1 character string to be searched for possible translation of all or some of its characters.

x_2 character string containing the translation values of characters.

x_3 character string containing the characters that are to be translated. If x_3 is omitted, a string of 256 characters is assumed; it contains all possible characters arranged in ascending order (hexadecimal 00 through FF).

Strings x_2 and x_3 should be the same length; otherwise x_2 is padded with blanks, or truncated, on the right to match the length of x_3.

Any non-character arguments are converted to character.

TRUNC(x) Arithmetic

TRUNC returns an integer that is the truncated form of a given value x. If x is positive or zero, the result is the largest integer less than or equal to x. If x is negative, the result is the smallest integer greater than or equal to x. x must be real.

If x is fixed-point with precision (p,q), the precision of the result is given by:

```
(MIN(N,MAX(p-q+1,1)),0)
```

where N is the maximum number of digits allowable.

UNSPEC(x) String-handling

UNSPEC returns a bit string that is the internal coded form of a given value x.

x expression of any data type.

The length of the returned bit-string depends on the attributes of x.

If x is a varying-length string, its two-byte prefix is included in the returned bit-string.

If x is complex, the length of the returned string is twice the value given in the following table.

bit-string length	attributes of x
16	FIXED BINARY (p,q) for p<16
32	FIXED BINARY (p,q) for p>15 FLOAT BINARY (p) for p<22 FLOAT DECIMAL (p) for p<7 POINTER (standard length) OFFSET FILE constant or variable POINTER (under checkout compiler with COMPATIBLE option)
64	FLOAT BINARY (p) for 21<p<54 FLOAT DECIMAL (p) for 6<p<17 LABEL constant or variable ENTRY constant or variable
128	FLOAT BINARY(p) for 53<p<110 FLOAT DECIMAL(p) for 16<p<34 TASK POINTER (under checkout compiler with NOCOMPATIBLE option)
256	EVENT
n	BIT (n)
n+16	BIT VARYING where n is the maximum length of x.
8*n	CHARACTER (n) PICTURE (with character-string length of n)
8*(n+2)	CHARACTER VARYING where n is the maximum length of x.
8*(n+16)	AREA (n)
8*FLOOR(n)	FIXED DECIMAL (p,q) where n = (p+2)/2

UNSPEC Pseudovariable

The pseudovariable assigns a bit string directly to the given variable x, i.e., no conversion to the data type of the variable is attempted. The bit string is passed, if necessary, on the right with zeros to match the length of the variable. If x is a varying length string, its two-byte prefix is included in the field to which the bit string is assigned.

VERIFY(x_1,x_2) String-handling

VERIFY returns a default-precision fixed-point binary integer indicating the position in the given string x_1 of the first character or bit that is not in the given string x_2. If all the characters or bits in x_1 do appear in x_2, a value of zero is returned. The arguments are converted to strings if they are arithmetic. If one string argument is bit and the other character, the bit is converted to character.

x_1 string to be scanned for any character not in x_2.

x_2 the verification string, consisting of a set of characters in any order.

If either argument is character or decimal, conversions are performed to produce character strings. Otherwise, if the arguments are bit and binary or both binary, conversions are performed to produce bit.

Section H: On-conditions

Introduction

The on-conditions are those exceptional conditions that can be specified in PL/I by means of an ON statement. If a condition is enabled, the occurrence of the condition will result in an interrupt. The interrupt, in turn, will result in the execution of the current action specification for that condition. If an ON statement for that condition is not in effect, the current action specification is the standard system action for that condition. If an ON statement for that condition is in effect, the current action specification is either SYSTEM, in which case the standard system action for that condition is taken, or an on-unit, in which case the programmer has supplied his own action to be taken for that condition.

Some conditions are always enabled unless they have been explicitly disabled by condition prefixes; others are always disabled unless they have been explicitly enabled by condition prefixes; and still others are always enabled and cannot be disabled.

Those conditions that are always enabled unless they have been explicitly disabled by condition prefixes are:

CONVERSION

FIXEDOVERFLOW

OVERFLOW

UNDERFLOW

ZERODIVIDE

Each of the above conditions can be disabled by a condition prefix specifying the condition name preceded by NO without intervening blanks. Thus, one of the following names in a condition prefix will disable the respective condition:

NOCONVERSION

NOFIXEDOVERFLOW

NOOVERFLOW

NOUNDERFLOW

NOZERODIVIDE

Such a condition prefix renders the corresponding condition disabled throughout the scope of the prefix; the condition remains enabled outside this scope. (Scope of a condition prefix is discussed in chapter 14, "Exceptional Condition Handling and Program Checkout".)

Conversely, those conditions that are always disabled unless they have been enabled by a condition prefix are:

SIZE

SUBSCRIPTRANGE

STRINGRANGE

STRINGSIZE

CHECK

The appearance of one of these five in a condition prefix renders the condition enabled throughout the scope of the prefix; the condition remains disabled outside this scope. Further, a condition prefix specifying NOSIZE, NOSUBSCRIPTRANGE, NO STRINGRANGE, NOSTRINGSIZE, or NOCHECK will disable the corresponding condition throughout the scope of that prefix. Since SIZE, STRINGRANGE, and SUBSCRIPTRANGE represent errors that are likely to prevent successful execution, the checkout compiler checks for these conditions, and takes standard system action, even when they are disabled, although an on-unit cannot be entered while the corresponding condition is disabled.

All other conditions are always enabled and remain so for the duration of the program. These conditions are:

AREA

ATTENTION (checkout compiler only)

CONDITION

ENDFILE

ENDPAGE

ERROR

FINISH

KEY

NAME

PENDING

RECORD

TRANSMIT

UNDEFINEDFILE

CONDITION CODES (ON-CODES)

The ONCODE built-in function may be used by the programmer in any on-unit to determine the nature of the error or condition that caused entry into that on-unit. The codes corresponding to the conditions and errors checked for are given below:

Code	Error or Exceptional Condition
0	The ONCODE function has been used outside an on-unit.
ERROR Condition Code	
3	Execution of SIGNAL ERROR statement in place of statement diagnosed as in error.
FINISH Condition Codes	
4	SIGNAL FINISH, STOP, or EXIT statement executed. or Main procedure completed normally.
ERROR Condition Code	
9	SIGNAL ERROR statement executed.

Note: For further ERROR condition codes, see code numbers 1000 onwards.

NAME Condition Codes	
10	SIGNAL NAME statement executed. or Unrecognizable identifier in GET DATA input stream.
RECORD Condition Codes	
20	SIGNAL RECORD statement executed.
21	Record variable smaller than record size.
22	Record variable larger than record size.
23	Attempt to write or locate a zero length record.
24	Zero length record has been read from a REGIONAL data set.
TRANSMIT Condition Codes	
40	SIGNAL TRANSMIT statement executed.
41	Uncorrectable transmission error in output data set.
42	Uncorrectable transmission error in input data set.
KEY Condition Codes	
50	SIGNAL KEY statement executed.
51	Key specified cannot be found.
52	Attempt to add keyed record which has same key as a record already present in data set, or, in a REGIONAL(1) data set, attempt to write into a region already containing a record.
53	Value of expression specified in KEYFROM option during sequential creation of INDEXED or REGIONAL data set is less than value of previously specified key or region number.
54	Key conversion error has occurred, possibly due to region number not being numeric character.
55	Key specification is null string or begins (8)'1'B.
56	Attempt to access a record using a key that is outside the data set limits.
57	No space available to add a keyed record.
ENDFILE Condition Code	
70	SIGNAL ENDFILE statement executed. or Attempt to read past the file delimiter.
UNDEFINEDFILE Condition Codes	
80	SIGNAL UNDEFINEDFILE statement has been executed.
81	Conflict in file attributes exists at open time between attributes in DECLARE statement and those in explicit or implicit OPEN statement.
82	Conflict between file attributes and physical organization of data set, e.g. between file organization and device type.

83 After merging ENVIRONMENT options with DD statement and data set label, data set specification is incomplete, e.g. blocksize or record format has not been specified.

84 No DD statement associating file with a data set.

85 During initialization of a DIRECT OUTPUT file associated with a REGIONAL data set, an input/output error occurred.

86 Linesize greater than implementation-defined maximum.
or
Invalid value in an ENVIRONMENT option.

87 After merging ENVIRONMENT options with DD statement and data set label, conflict exists in data set specification, e.g. record format incompatible with blocksize or file organization.

ENDPAGE Condition Code

90 SIGNAL ENDPAGE statement executed.
or
Attempt to start new line when line number is equal to current page size.

PENDING Condition Code

100 SIGNAL PENDING statement executed.
or
READ issued for TRANSIENT INPUT file when message queue empty.

STRINGSIZE Condition Code

150 SIGNAL STRINGSIZE statement executed.
or
Characters have been lost in an assignment to a character-string variable or temporary or in an input/output operation.

OVERFLOW Condition Code

300 SIGNAL OVERFLOW statement has been executed.
or
Magnitude of floating-point number exceeds permitted maximum.

FIXEDOVERFLOW Condition Code

310 SIGNAL FIXEDOVERFLOW statement executed.
or
Length of result of fixed-point arithmetic operation exceeds permitted maximum.

ZERODIVIDE Condition Code

320 SIGNAL ZERODIVIDE statement executed.
or
Attempt to divide by zero.

UNDERFLOW Condition Code

330 SIGNAL UNDERFLOW statement executed.
or
Magnitude of a floating-point number is smaller than the permitted minimum.

SIZE Condition Code

340 SIGNAL SIZE statement executed.
or
High-order non-zero digits have been lost in an assignment to a variable or temporary, or significant digits have been lost in an input/output operation.

341 High order non-zero digits have been lost in an input/output operation.

STRINGRANGE Condition Code

350 SIGNAL STRINGRANGE statement executed.
or
Length of the arguments of a SUBSTR reference failed to comply with the rules described for the SUBSTR built-in function.

AREA Condition Codes

360 Attempt to allocate a based variable within an area that contains insufficient free storage for allocation to be made.

361 Insufficient space in target area for assignment of source area.

362 SIGNAL AREA statement executed.

ATTENTION Condition Code

400 Checkout compiler only: SIGNAL ATTENTION statement executed
or
Attention signaled from terminal.

CONDITION Condition Code

500 SIGNAL CONDITION (condition) statement has been executed.

CHECK Condition Codes

510 SIGNAL CHECK statement executed.
or
Value of all or part of variable is about to change, or execution of labeled or named statement is about to take place, within scope of CHECK prefix.

SUBSCRIPTRANGE Condition Code

520 SIGNAL SUBSCRIPTRANGE statement executed.
or
Subscript has been evaluated and found to lie outside its specified

521 Subscript of iSUB-defined variable lies outside bounds of corresponding dimension of base variable.

CONVERSION Condition Codes

600 SIGNAL CONVERSION statement executed.

601 Invalid conversion attempted during input/output of a character string.

603 Error during processing of an F-format item for a GET STRING statement.

604 Error during processing of an F-format item for a GET FILE statement.

605 Error during processing of an F-format item for a GET FILE statement following a TRANSMIT condition.

606 Error during processing of an E-format item for a GET STRING statement.

607 Error during processing of an E-format item for a GET FILE statement.

608 Error during processing of an E-format item for a GET FILE statement following a TRANSMIT condition.

609 Error during processing of a B-format item for a GET STRING statement.

610 Error during processing of a B-format item for a GET FILE statement.

611 Error during processing of a B-format item for a GET FILE statement following TRANSMIT condition.

612 Error during character string to arithmetic conversion.

613 Error during character string to arithmetic conversion for a GET or PUT FILE statement.

614 Error during character string to arithmetic conversion for a GET or PUT FILE statement following a TRANSMIT condition.

615 Error during character string to bit string conversion.

616 Error during character string to bit string conversion for a GET or PUT FILE statement.

617 Error during character string to bit string conversion for a GET or PUT FILE statement following a TRANSMIT condition.

618 Error during character string to picture conversion.

619 Error during character string to picture conversion for a GET or PUT FILE statement.

620 Error during character string to picture conversion for a GET or PUT FILE statement following a TRANSMIT condition.

621 Error in decimal P-format item for a GET STRING statement.

622 Error in decimal P-format input for a GET FILE statement.

623 Error in decimal P-format input for a GET FILE statement following a TRANSMIT condition.

624 Error in character P-format input for a GET FILE statement.

625 Error exists in character P-format input for a GET FILE statement.

626 Error exists in character P-format input for a GET FILE statement following a TRANSMIT condition.

ERROR Condition Codes

Note: For other ERROR conditions, see condition codes 3 and 9.

1002 GET or PUT STRING specifies data that exceeds size of string.

1003 Further output prevented by TRANSMIT or KEY conditions having been previously raised for the data set.

1004 Attempt to use PAGE, LINE, or SKIP ≤ 0 for non-print file.

1005 In DISPLAY(element-expression) REPLY (character-variable) statement, element-expression or character-variable is of zero length.

1007 A REWRITE or a DELETE statement has not been preceded by a READ.

1008 Unrecognized identifier in a string specified in a GET STRING DATA statement.

1009 An input/output statement specifies an operation or an option which conflicts with the file attributes.

1011 Data management has detected an input/output error but is unable to provide any information about its cause.

1013 Previous input operation incomplete; REWRITE or DELETE statement specifies data which has been previously read in by a READ statement with an EVENT option, and no corresponding WAIT has been executed.

1014 Attempt to initiate further input/output operation when number of incomplete operations equals number specified by ENVIRONMENT option NCP(n) or by default.

1015 Event variable has been specified for an input/output operation when already in use.

1016 After UNDEFINEDFILE condition has been raised as a result of an unsuccessful attempt to implicitly open a file, the file was found to be unopened on normal return from the on-unit.

1018 End of file or string was encountered in data before end of data-list or (in edit-directed transmission) format list.

1019 Attempt to close file which was not opened in current task.

1020 Further input/output attempted before WAIT statement executed to ensure completion of previous READ.

1021 Attempt to access a record locked by another file in this task.

1022 Attempt to write a record onto a sequential output data set on which space is not available.

1023 Exclusive file closed while records still locked in a subtask.

1500 Computational error; short floating point argument of SQRT built-in function is negative.

1501 Computational error; long floating point argument of SQRT built-in function is negative.

1502 Computational error; extended floating point argument of SQRT built-in function is negative.

1503 Computational error in LOG, LOG2, or LOG10 built-in function; extended floating point argument is ≤ 0.

1504 Computational error in LOG, LOG2, or LOG10 built-in function; short floating point argument is ≤ 0.

1505 Computational error in LOG, LOG2 or LOG10 built-in function; long floating point argument is ≤ 0.

1506 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of short floating point argument exceeds (2**18)*pi (SIN and COS) or (2**18)*180 (SIND and COSD).

1507 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of long floating point argument exceeds (2**50)*pi (SIN and COS) or (2**50)*180 (SIND and SIND).

1508 Computational error; absolute value of short floating point argument of TAN or TAND built-in functuion exceeds, respectively, (2**18)*pi or (2**18)*180.

1509 Computational error; absolute value of long floating point argument of TAN or TAND built-in function exceeds, respectively, (2**50)*pi or (2**50)*180.

1510 Computational error; short floating point arguments of ATAN or ATAND built-in function both zero.

1511 Computational error; long floating point arguments of ATAN or ATAND built-in function both zero.

1514 Computational error; absolute value of short floating point argument of ATANH built-in function ≥ 1.

1515 Computational error; absolute value of long floating point argument of ATANH built-in function ≥ 1.

1516 Computational error; absolute value of extended floating point argument of ATANH built-in function ≥ 1.

1517 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of extended floating point argument exceeds (2**106)*pi (SIN and COS) or (2**106)*180 (SIND and COSD).

1518 Computational error; absolute value of short floating point argument of ASIN or ACOS built-in function exceeds 1.

1519 Computational error; absolute value of long floating point argument of ASIN or ACOS built-in function exceeds 1.

1520 Computational error; absolute value of extended floating point argument of ASIN, ACOS built-in function exceeds 1.

1521 Computational error; extended floating point arguments of ATAN or ATAND built-in function both zero.

1522 Computational error; absolute value of extended floating point argument of TAN or TAND built-in function ≥ (2**106)*pi or (2**106)*180, respectively.

1550 Computational error; real short floating-point base is zero and fixed-point integer exponent not positive.

1551 Computational error; real long floating-point base is zero and fixed-point integer exponent not positive.

1552 Computational error; real short floating point base is zero and the floating-point or non-integral exponent is not positive.

1553 Computational error; real long floating point base is zero and the floating-point or non-integral exponent is not positive.

1554 Computational error; complex short floating point base is zero and fixed-point integer exponent is not positive.

1555 Computational error; complex long floating point base is zero and fixed-point integer exponent is not positive.

1556 Computational error; complex short floating point base is zero and floating-point or non-integral exponent is not positive and real.

1557 Computational error; complex long floating point base is zero and floating-point or non-integral exponent is not positive and real.

1558 Computational error; complex short floating point argument of ATAN or ATANH built-in function has value, respectively, or ±1I or ±1.

1559 Computational error; complex long floating point argument of ATAN or ATANH built-in function has value, respectively, of ±1I or ±1.

1560 Computational error; real extended floating-point base is zero and fixed-point integer exponent not positive.

1561 Computational error; real extended floating point base is zero and floating-point or non-integral exponent is not positive.

1562 Computational error; complex extended floating point base is zero and integer exponent is not positive.

1563 Computational error; complex extended floating point base is zero and floating-point or non-integral exponent is not positive.

1564 Computational error; complex extended floating point argument of ATAN or ATANH built-in function has value, respectively, of ±1I or ±1.

2002 WAIT statement cannot be executed because of restricted system facility.

3000 Field width, number of fractional digits, and number of significant

digits (w,d, and s) specified for E-format item in edit-directed input/output statement do not permit transmission without loss of significant digits or sign.

3004 Checkout compiler only: A-format width unspecified in format list for GET EDIT statement.

3005 Checkout compiler only: B-format width unspecified in format list for GET EDIT statement.

3006 Picture description of target does not match non-character-string source.

3008 Checkout compiler only: remote format item specifies label not in current block.

3500 Checkout compiler only: argument to HIGH built-in function is less than zero.

3501 Checkout compiler only: argument to LOW built-in function is less then zero.

3502 Checkout compiler only: argument to BIT built-in function less than zero.

3503 Checkout compiler only: argument to CHAR built-in function less than zero.

3798 ONCHAR or ONSOURCE pseudovariable used out of context.

3799 In an on-unit entered as a result of the CONVERSION condition being raised by an invalid character in the string being converted, the character has not been corrected by use of the ONSOURCE or ONCHAR pseudovariables.

3800 Checlout compiler only: length of data aggregate exceeds system limit of 2**24 bytes.

3801 Checkout compiler only: element of an array in a structure cannot be mapped.

3802 Checkout compiler only: array bound is out of valid range.

3803 Checkout compiler only: array has lower bound greater than upper bound.

3804 Checkout compiler only: string has length greater than permitted maximum.

3805 Checkout compiler only: length of string less than zero.

3806 Checkout compiler only: size of area exceeds permitted maximum.

3807 Checkout compiler only: size of area is less than zero.

3808 Aggregate cannot be mapped in COBOL or FORTRAN.

3901 Attempt to invoke task having name variable that is already associated with an active task.

3904 Event variable specified as argument to COMPLETION pseudovariable while already in use for a DISPLAY statement.

3906 Assignment to an event variable that is already active.

3907 Attempt to associate an event variable that is already associated with an active task.

3909 Attempt to create a subtask (using CALL statement) when insufficient main storage available.

3910 Attempt to attach a task (using CALL statement) when number of active tasks was already at limit defined by ISASIZE paramter of EXEC statement.

3911 WAIT statement in on-unit specifies that an event already being waited for in task from which on-unit was entered.

3912 Attempt to execute CALL with TASK option in block invoked while executing PUT FILE(SYSPRINT) statement.

3913 CALL statement with TASK option specifies an unknown entry point.

3914 Attempt to call FORTRAN or COBOL routines in two tasks simultaneously.

4000 Checkout compiler only: use of uninitialized variable as source.

4001 Checkout compiler only: reference to CONTROLLED variable before it has been allocated.

4002 Controlled variable with bound, length, or size as * has been specified in an ALLOCATE statement when no previous allocation exists.

4003 Checkout compiler pnly: IN option of ALLOCATE statement specifies an area not the same as that declared to be associated with offset variable specified in SET option.

4050 Checkout compiler only: attempt to refer to a based variable whose pointer has the null or other initial value.

4051 Checkout compiler only: attempt to free a variable that has no valid allocation in its associated area.

4052 Checkout compiler only: pointer addresses based variable whose attributes differ from attributes of variable declared with that pointer value.

4053 Checkout compiler only: reference to based variable when pointer addresses storage that no longer contains the variable.

4054 Checkout compiler only; locator variable refers to a locate mode input/output buffer when buffer is not the latest one or when file is closed.

4055 Checkout compiler only: attempt to assign to an offset variable a locator that does not reference storage in the appropriate area.

4056 POINTER or OFFSET built-in function does not address a valid allocation of storage in the specified area.

4057 Checkout compiler only: locator qualifying a based variable refers to storage which has not been allocated in current task.

4058 Checkout compiler only: a based structure is referred to by means of a pointer that is not valid for that structure.

5000 Checkout compiler only: number of arguments being passed does not match number of parameters.

5001 Checkout compiler only: attributes of argument being passed do not match attributes of corresponding parameter.

5002 Checkout compiler only: attributes of value being returned do not match those implied by context of function reference.

5003 Checkout compiler only: attempt to return a value from a block invoked by a CALL statement.

5004 Checkout compiler only: block invoked as a function without returning a value.

5005 FORTRAN routine would pass invalid data type.

5050 Checkout compiler only: attempt to use defined variable whose storage extends beyond end of base variable.

or

POSITION attribute specifies value greater than permitted maximum.

5051 Checkout compiler only: size of simple defined area greater than that of base variable.

8091 Operation exception.

8092 Privileged operation exception.

8093 EXECUTE exception.

8094 Protection exception.

8095 Addressing exception.

8096 Specification exception.

8097 Data exception.

9002 Checkout compiler only: attempt to execute GO TO statement specifying label in an inactive block.

9003 Checkout compiler only: attempt to invoke an entry point in a procedure compiled by the optimizing compiler when that procedure's containing block is inactive.

9004 Checkout compiler only: linkage editor cannot find entry constant on specified data set.

9005 Checkout compiler only: attempt to use label variable in a GO TO statement when value not in label list.

9050 Program has been terminated by an abend.

9051 Attempt to invoke procedure compiled by the checkout compiler from one compiled by the optimizing compiler.

9101 Checkout compiler only: number of lines specified in STEPLINES compiler option has been transmitted.

9200 Program check occurred in SORT/MERGE program.

9201 Parameter or returned value specified for SORT exit does not match specification in invocation of PL/I sort entry name.

9250 Procedure to be fetched cannot be found.

9251 Permanent transmission error when fetching a procedure.

Multiple Interrupts

A multiple interrupt is the simultaneous occurrence of two or more interrupts.

With IBM System/360 and System/370 systems using Processors other than Models 91 and 195, a multiple interrupt can only occur for the conditions TRANSMIT and RECORD. The interrupt for TRANSMIT is always processed first. The interrupt for RECORD will be ignored unless there is an on-unit for TRANSMIT that causes normal return.

In systems using IBM System/360 Models 91 and 195 Processors, a second type of multiple interrupt, known as an imprecise interrupt, can occur during parallel processing. The interrupt may be due to the raising of a PL/I condition or a hardware exception which subsequently raises the ERROR condition. The conditions and exceptions that may cause an imprecise interrupt are shown below, in the order in which they are processed.

PL/I on-conditions:

1. UNDERFLOW
2. FIXEDOVERFLOW
3. SIZE
4. OVERFLOW
5. ZERODIVIDE

Hardware interrupts:

6. Data exception
7. Specification exception
8. Addressing exception
9. Protection exception

Event I/O and imprecise interrupts cannot occur as part of the same multiple interrupt.

Imprecise interrupt conditions are processed successively, until one of the following occurs, in which case no subsequent conditions are processed.

1. The processing of a condition causes termination of the task, through either standard system action, normal return from an on-unit, or abnormal termination in the on-unit.

2. Control is transferred out of an on-unit by means of a GO TO statement, so that a normal return is not allowed to take place.

List of Conditions

This section presents conditions in alphabetical order. In general, the following information is given for each condition:

1. General format -- given only when it consists of more than the condition name.

2. Description -- a discussion of the condition, including the circumstances under which the condition can be raised. Note that an enabled condition can always be raised by a SIGNAL statement; this fact is not included in the descriptions.

3. Result -- the result of the operation that caused the condition to occur. This applies when the condition is disabled as well as when it is enabled. In some cases, the result is not defined; that is, it cannot be predicted. This is stated wherever applicable.

4. Standard system action -- the action taken by the system when an interrupt occurs and the programmer has not specified an on-unit to handle that interrupt.

5. Status -- an indication of the enabled/disabled status of the condition at the start of the program, and how the condition may be disabled (if possible) or enabled.

6. Normal return -- the point to which control is returned as a result of the normal termination of the on-unit. A GO TO statement that transfers control out of an on-unit is an abnormal on-unit termination. Note that if a condition (except the ERROR condition)

Missing from orig document

Missing from orig document

1. If a name in the CHECK prefix is a statement label constant, the condition is raised and the interrupt occurs prior to the execution of the statement to which the label is prefixed. If the label is prefixed to a FORMAT statement, the condition is not raised.

2. If a name in the CHECK prefix is a variable (as specified in the general format above), the condition is raised whenever the value of the variable, or of any part of the variable, is changed by any statement within the scope of the prefix.

   Specifically, if the identifier ID represents the variable, the condition is raised in the following cases:

   a. ID appears on the left-hand side of an assignment statement. (This applies to BY NAME assignment only if the name mentioned changes its value.)

   b. ID is set as a result of a pseudovariable appearing on the left-hand side of an assignment statement.

   c. ID appears as the control variable of a DO-group or a repetitive specification in a data list (or it is set as a result of a pseudovariable appearing as the control variable of a DO-group or a repetitive specification in a data list).

   d. ID appears in the data list of an edit-directed or list-directed GET statement.

   e. ID is altered by data-directed input.

   f. ID appears in the REPLY option of a DISPLAY statement.

   g. ID appears in the STRING option of a PUT statement.

   h. ID is passed as an argument to a programmer-defined procedure, no dummy argument is created, the procedure terminates with a RETURN or END, and the procedure is not invoked with the TASK, PRIORITY, or EVENT option.

   i. ID appears in the KEYTO or INTO option of a READ statement. Note that if the READ statement has an EVENT option, the CHECK condition will not be raised.

   j. ID is a locator variable and appears in a SET option or is set implicitly.

   k. ID is a non-static variable set by the INITIAL attribute.

   In a, b, d, and e above, if ID is a data aggregate, the CHECK condition is raised and the interrupt occurs each time an element of that aggregate is given a value. If ID is an element of a data aggregate, the condition is raised for that element only, not the whole array.

   The condition is not raised under any of the following circumstances:

   a. If the value of a variable defined on ID or on part of ID changes in any of the ways described above.

   b. If the parameter that represents the argument ID changes value.

   c. If ID appears in a GO TO or RETURN statement or any statement that involves the execution of a GO TO or RETURN statement.

   Note that in all of the above contexts, ID can appear in subscripted or qualified form. Note also that ID need not appear in the name list of a CHECK prefix; it only need represent a structure or element contained by, or containing, a name in the list.

   The interrupt for a CHECK condition occurs immediately after the assignment to ID, except in case h. Then it occurs immediately after execution of the subroutine's RETURN or END statement. In a DO statement, the interrupt occurs each time control proceeds sequentially to the statement following the DO statement. If the DO specifies repetitive execution, the interrupt occurs each time the control variable changes value.

   If a statement causes a CHECK condition to be raised for several names, the conditions will be raised in the left-to-right order of appearance of the names.

3. If an identifier in the CHECK prefix name list is an entry constant, the condition is raised and the interrupt occurs prior to each invocation of the entry point corresponding to the entry constant. The condition is raised only if the entry point is invoked by the entry constant given in the prefix.

Result: When CHECK is raised, there is no effect on the statement being executed.

Standard System Action: In the absence of a CHECK on-unit, the output consists of the current statement number together with the data shown in figure H.1.

Variable or Constant	Checkout Compiler	Optimizing Compiler
Arithmetic or string variable	Name and Value	
Area, file, entry event, label, locator or task variable Entry or label constant	As for PUT DATA	Name

Figure H.1. Output for the CHECK condition

If SIGNAL CHECK without a name-list is given, in the absence of a CHECK on-unit, within the scope of a CHECK prefix that is also without a name list, all problem data identifiers within the scope of the prefix are printed, together with their values. In addition, under the checker the names and values of all internal program control variables and the names of all external program control variables within the scope of the prefix are printed.

Note: Standard system action for the CHECK condition requires access to the variable; consequently, if SIGNAL CHECK is given for an unallocated variable, an error will result, as it would if the variable were accessed by an on-unit. Under the checker, a comment will be printed and execution continued if the variable has the INTERNAL attribute; variables with the EXTERNAL attribute or any variable under the optimizer will raise ERROR.

Status: CHECK is disabled by default and within the scope of a NOCHECK condition prefix. It is enabled only within the scope of a CHECK prefix.

For other details of the enabling and disabling of the CHECK condition, see chapter 14, "Execution-Time Facilities of the Checkout Compiler".

Normal Return: Upon the normal completion of the on-unit for the CHECK condition, execution continues immediately following the point at which the interrupt occurred.

CONDITION(name) — Programmer-named

Abbreviation: COND(name)

The "name" must be specified by the programmer. The appearance of an identifier with CONDITION in an ON, SIGNAL, or REVERT statement constitutes a contextual declaration for it; the identifier is given the EXTERNAL attribute.

An identifier may also be declared explicitly as a condition name by means of the CONDITION attribute.

Description: CONDITION is raised by a SIGNAL statement that specifies the appropriate identifier. The identifier specified in the SIGNAL statement determines which CONDITION condition is to be raised.

Standard System Action: In the absence of an on-unit for this condition, the system prints a message and continues with the statement following SIGNAL.

Status: CONDITION is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit, execution continues with the statement following the SIGNAL statement that caused the interrupt.

CONVERSION — Computational

Abbreviation: CONV

Description: The CONVERSION condition occurs whenever an invalid conversion is attempted on character-string data. This attempt may be made internally or during an input/output operation. For example, the condition occurs when a character other than 0 or 1 exists in a character string being converted to a bit string; other examples are when a character string being converted to a numeric character field contains characters not permitted by the PICTURE specification, or when a string being converted to coded arithmetic data does not contain the character representation of an arithmetic constant.

All conversions of character-string data are carried out character-by-character in a left-to-right sequence and the condition occurs for each invalid character. The condition is also raised if the all characters in the string are blank. When an invalid character is encountered, an interrupt occurs (provided, of course, that CONVERSION has not been disabled) and the

current action specification for the condition is executed. If the action specification is an on-unit, the invalid character can be corrected within the unit by using the ONSOURCE or ONCHAR pseudovariables. When one of these pseudovariables has been used, the conversion is retried on return from the on-unit. If the error has not been corrected the program will loop. If these pseudovariables have not been used the ERROR condition is raised.

Result: When CONVERSION occurs, the contents of the entire result field are undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: CONVERSION is enabled throughout the program, except within the scope of a condition prefix specifying NOCONVERSION.

Normal Return: Upon the normal termination of the on-unit for this condition, control returns to the beginning of the string and the conversion is retried.

ENDFILE (element-file-expr) Input/Output

Description: The ENDFILE condition can be raised during a GET or READ operation; it is caused by an attempt to read past the file delimiter of the file named in the GET or READ statement. It applies only to SEQUENTIAL INPUT, SEQUENTIAL UPDATE and STREAM INPUT files.

In record-oriented I/O, ENDFILE is raised whenever a file delimiter is encountered during the execution of a READ statement.

In stream-oriented I/O, ENDFILE is raised during the execution of a GET statement if a file delimiter is encountered either before any items in the GET statement data list have been transmitted or between transmission of two of the data items. If a file delimiter is encountered within a data item, or if it is encountered while an X format item is being implemented, the ERROR condition is raised.

If the file is not closed after ENDFILE occurs, then any subsequent GET or READ statement for that file immediately raises the ENDFILE condition again.

If ENDFILE is raised by an input/output statement using the EVENT option, the interrupt does not take place until the execution of a subsequent WAIT statement for that event in the same procedure.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: The ENDFILE condition is always enabled; it cannot be disabled.

Normal Return: Upon the normal termination of the on-unit for the condition, execution continues with the statement immediately following the GET or READ statement that caused the ENDFILE (or, if ENDFILE was raised by a READ with the EVENT option, control passes back to the WAIT statement from which the on-unit was invoked).

Note: If a file is closed in an on-unit for this condition, the results of normal return are undefined. Exit from such an on-unit should be by means of a GO TO statement.

ENDPAGE (element-file-expr) Input/Output

Description: The ENDPAGE condition is raised when a PUT statement results in an attempt to start a new line beyond the limit specified for the current page. This limit can be specified by the PAGESIZE option in an OPEN statement; if PAGESIZE has not been specified, a default limit of 60 is applied. The attempt to exceed the limit may be made during data transmission (including associated format items, if the PUT statement is edit-directed), by the LINE option, or by the SKIP option. ENDPAGE can also be raised by a LINE option or LINE format item that specified a line number less than the current line number.

When ENDPAGE is raised, the current line number is one greater than that specified by the PAGESIZE option (or 61, if the default applies) so that it is possible to continue writing on the same page. The on-unit may start a new page by execution of a PAGE option or a PAGE format item, which sets the current line to 1.

ENDPAGE is raised only once per page. If the on-unit does not start a new page, the current line number may increase indefinitely. If a subsequent LINE option or LINE format item specifies a line number that is less than or equal to the current line number, ENDPAGE is not raised, but a new page is started with the current line set to 1. An exception is that if the current line number is equal to the specified line number, and the file is position on column 1 of the line, ENDPAGE is not raised.

If ENDPAGE is raised during data transmission, then, on return from the

on-unit, the data is written on the current line, which may have been changed by the on-unit. If ENDPAGE results from a LINE or SKIP option, then, on return from the on-unit, the action specified by LINE or SKIP is ignored.

Standard System Action: In the absence of an on-unit, the system starts a new page. If the condition is signaled, execution is unaffected and continues with the statement following the SIGNAL statement.

Status: ENDPAGE is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit for this condition, execution of the PUT statement continues in the manner described above.

ERROR — System Action

Description: The ERROR condition is raised under the following circumstances:

1. As a result of the standard system action for an ON-condition for which that action is to "print an error message and raise the ERROR condition".
2. As a result of an error (for which there is no ON-condition) occurring during program execution.
3. As a result of a SIGNAL ERROR statement.

Standard System Action: This depends on the processing mode:

Batch processing (optimizing or checkout compiler): If the condition is raised in the major task, the FINISH condition is raised and the task is terminated. If the condition is raised in any other task, the task is terminated.

Conversational processing (checkout compiler only): Control is passed to the terminal. Processing that is then initiated at the terminal takes place as if it were in an ERROR on-unit, and completion of this processing (other than by a GO TO statement out of the on-unit) constitutes a return from the on-unit.

Status: ERROR is always enabled; it cannot be disabled.

Normal Return: With certain exceptions, this depends on the processing mode:

Batch processing (optimizing or checkout compiler): The standard system action for batch processing mode is taken.

Conversational processing (checkout compiler only): The FINISH condition is raised.

The exceptional cases occur under the checkout compiler when a SIGNAL ERROR statement is executed in place of a statement in which the compiler has found an error. In these cases, normal return is to the statement following the one in which ERROR was signalled. The cases are characterized by an oncode of 3.

FINISH — System Action

Description: The FINISH condition is raised during execution of a statement which would cause the termination of the major task of a PL/I program, that is, by a STOP statement in any task, or an EXIT statement in the major task, or a RETURN or END statement in the initial procedure of the major task. The condition is also raised by SIGNAL FINISH in any task, and as part of the standard system action for the ERROR condition. The interrupt occurs in the task in which the statement is executed, and any on-unit specified for the condition is executed as part of that task. An abnormal return from the on-unit will avoid any subsequent task termination processes and permit the interrupted task to continue.

Standard System Action: This depends on the processing mode:

Batch processing (optimizing or checkout compiler): No action is taken; that is, processing is continued from the point at which the condition was raised.

Conversational processing (checkout compiler only): Control is passed to the terminal. Processing that is then initiated at the terminal takes place as if it were in a FINISH on-unit, and completion of that processing (other than by a GO TO statement out of the on-unit) constitutes a normal return from the on-unit.

Status: FINISH is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit, execution of the interrupted statement is resumed.

FIXEDOVERFLOW Computational

Abbreviation: FOFL

Description: The FIXEDOVERFLOW condition occurs when the length of the result of a fixed-point arithmetic operation exceeds the maximum length allowed by the implementation. This maximum is 15 for decimal fixed-point values and 31 for binary fixed-point values.

Result: The result of the invalid fixed-point operation is undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: FIXEDOVERFLOW is enabled throughout the program, except within the scope of a condition prefix that specifies NOFIXEDOVERFLOW.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

KEY (element-file-expr) Input/Output

Description: The KEY condition can be raised only during operations on keyed records. It is raised in any of the following cases:

1. The keyed record cannot be found.
2. An attempt is made to add a duplicate key.
3. The key is out of sequence.
4. An error occurred in the conversion of the key.
5. The key has a null string or begins with the dummy record string (8)'1'B.
6. No space is available to add the keyed record.
7. The key is outside the data set limits (regional data sets only).

If KEY is raised by an input/output statement using the EVENT option, the interrupt does not occur until the execution of a subsequent WAIT statement for that event in the same procedure.

When a LOCATE statement is used for a REGIONAL(3) data set with V-format or U-format records, and there is not enough room in the specified region, the KEY condition is not raised until transmission of the record is attempted. Neither the record that causes the condition to be raised nor the current record is transmitted.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: KEY is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit for this condition, control passes to the statement immediately following the statement that caused KEY to be raised (or, if KEY was raised by an input/output statement with the EVENT option, control passes back to the WAIT statement from which the on-unit was invoked).

Note: If a file is closed in an on-unit for this condition, the results of normal return are undefined. Exit from such an on-unit should be by means of a GO TO statement.

NAME (element-file-expr) Input/Output

Description: The NAME condition can be raised only during a data-directed GET statement with the FILE option. It is raised in any of the following situations where an unrecognizable element variable appears in the stream:

1. There is an invalid character in the variable:

   A non-blank delimiter (comma, semicolon, or end-of-file mark) on left hand side of equals sign.

   A non-blank character between the right parenthesis and the equal sign

   A subscript character is not a digit

2. There is an invalid blank in the variable:

   Within the name or a subscript value. (Note: Blanks are permitted on either side of the period in a qualified name, or between a sign and a digit in a subscript)

3. The name is missing or invalid:

   No counterpart in the data list

   If there is no data list, the name is not known in the block

   Qualified name is not fully qualified

   More than 256 characters for a fully qualified name

   The name is iSUB-defined

4. A subscript list is missing or invalid:

   A subscript is missing

   Incorrect number of subscripts

   More than five digits in a subscript (leading zeros ignored)

   A subscript is beyond the permitted range

The programmer may retrieve the incorrect data field by using the built-in function DATAFIELD in the on-unit.

Standard System Action: In the absence of an on-unit, the system ignores the incorrect data field, prints a message, and continues the execution of the GET statement.

Status: NAME is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit for this condition, the execution of the GET statement continues with the next identifier in the stream.

OVERFLOW — Computational

Abbreviation: OFL

Description: The OVERFLOW condition occurs when the magnitude of a floating-point number exceeds the permitted maximum. The magnitude of a floating-point number or intermediate result must not be greater than approximately 10^{75} or 2^{252}.

Result: The value of such an invalid floating-point number is undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: OVERFLOW is enabled throughout the program, except within the scope of a condition prefix specifying NOOVERFLOW.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

PENDING (element-file-expr) — Input/Output

Description: Except when signaled, the PENDING condition can be raised only during execution of a READ statement for a TRANSIENT INPUT file. It is raised when an attempt is made to read a record that is temporarily unavailable (i.e., when the message queue associated with the file contains no messages at the time the READ statement is executed).

Standard System Action: In the absence of an on-unit, the action is as described for normal return.

Status: PENDING is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit for this condition, control returns to the point of interrupt (unless the condition was signaled), where execution is suspended until an appropriate record becomes available. If the condition was signaled, execution continues with the statement immediately following the SIGNAL statement that caused the interrupt.

Note: The value of the ONKEY built-in function when the PENDING condition is raised is a null string.

RECORD (element-file-expr) — Input/Output

Description: The RECORD condition can be raised only during a READ, WRITE, LOCATE, or REWRITE operation. It is raised by any of the following:

1. When the record length specified for a file with fixed-length records is smaller than the variable in a READ INTO statement; the remainder of the variable is undefined. If the variable is a varying-length string, RECORD is not raised if the SCALARVARYING option is applied to the file.

2. When the record is larger than the variable in a READ INTO statement; the remainder of the record is lost.

3. When the maximum record length is smaller than the variable in a WRITE, REWRITE, or LOCATE statement. For WRITE or REWRITE, the remainder of the variable is lost; for LOCATE, the variable is not transmitted.

4. When the record length specified for a file with fixed-length records is larger than the variable in a WRITE, REWRITE, or LOCATE statement; the remainder of the record is undefined. If the variable is a varying-length string, RECORD is not raised if the SCALARVARYING option is applied to the file.

5. When the variable in a WRITE or REWRITE statement indicates a zero length; no transmission occurs. If the variable is a varying-length string, RECORD is not raised if the SCALARVARYING option is applied to the file.

If the SCALARVARYING option is applied to the file (it must be applied to a file using locate mode to transmit varying-length strings), a 2-byte length prefix is transmitteed with an element varying-length string. The length prefix is not reset if the RECORD condition is raised. If the SCALARVARYING option is not applied to the file, the length prefix is not transmitted; on input, the current length of a varying-length string is set to the shorter of the record length and the maximum length of the string.

If RECORD is raised by an input/output statement using the EVENT option, the interrupt does not occur until the execution of a subsequent WAIT statement for that event in the same procedure.

The RECORD condition is not raised for undefined-length records read from:

A CONSECUTIVE data set to a SEQUENTIAL UNBUFFERED file

A REGIONAL(3) data set to a DIRECT file

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: RECORD is always enabled; it cannot be disabled.

Normal Return: Upon normal completion of the on-unit, execution continues with the statement immediately following the one for which RECORD occurred (or if RECORD was raised by an input/output statement with an EVENT option, control returns to the WAIT statement from which the on-unit was invoked).

Note: If a file is closed in an on-unit for this condition, the results of normal return are undefined. Exit from such an on-unit should be by means of a GO TO statement.

SIZE — Computational

Description: The SIZE condition occurs only when high-order (i.e., leftmost) significant binary or decimal digits are lost in an assignment to a variable or an intermediate result or in an input/output operation. This loss may result from a conversion involving different data types, different bases, different scales, or different precisions.

The SIZE condition differs from the FIXEDOVERFLOW condition in that, whereas FIXEDOVERFLOW occurs when the size of a calculated fixed-point value exceeds the maximum allowed by the implementation (see the description of the FIXEDOVERFLOW condition), whereas SIZE occurs when the size of the value being assigned to a data item exceeds the declared (or default) size of the data item. SIZE can be raised on assignment of a value regardless of whether or not FIXEDOVERFLOW was raised in the calculation of that value.

The declared size is not necessarily the actual precision with which the item is held in storage; however, the limit for SIZE is the declared or default size, not the actual size in storage. For example, a fixed binary item of precision (20) will occupy a fullword in storage, but SIZE is raised if a value whose size exceeds FIXED BINARY(20) is assigned to it.

Result: The contents of the data item receiving the wrong-sized value are undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: SIZE is disabled within the scope of a NOSIZE condition prefix and elsewhere throughout the program, except within the scope of a condition prefix specifying SIZE. Under the checkout compiler, the standard system action takes place for SIZE under the circumstances given under "Description" above, even when the condition is disabled; no on-unit for this condition can be entered, however, while it is disabled.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

STRINGRANGE Program-checkout

Abbreviation: STRG

Definition: The STRINGRANGE condition is raised whenever the lengths of the arguments to a SUBSTR reference fail to comply with the rules described for the SUBSTR built-in function. It is raised for each such reference.

Standard System Action: A message is printed and processing continues as described for normal return.

Status: STRINGRANGE is disabled by default and within the scope of a NOSTRINGRANGE condition prefix. It is enabled only within the scope of a STRINGRANGE condition prefix. Under the checkout compiler, the standard system action takes place for STRINGRANGE under the circumstances given under "Definition" above, even when the condition is disabled; no on-unit for this condition can be entered, however, while it is disabled.

Normal Return: On normal return from the on-unit, execution continues with a revised SUBSTR reference whose value is defined as follows:

Assuming that the length of the source string (after execution of the on-unit, if specified) is k, the starting point is i, and the length of the substring is j;

1. If i is greater than k the value is the null string.

2. If i is less than or equal to k, the value is that substring beginning at the mth character or bit of the source string and extending n characters or bits, where m and n are defined by:

```
m=MAX(i,1)

n=MAX(0,MIN(j+MIN(i,1)-1,k-m+1))
    [if j is specified]

n=k-m+1
    [if j is not specified]
```

This means that the new arguments are forced within the limits.

The values of i and j are established before entry to the on-unit; they are not reevaluated on return from the on-unit.

The value of k may change in the on-unit if the first argument of SUBSTR is a varying-length string. The value n is computed on return from the on-unit using any new value of k.

STRINGSIZE Program-checkout

Abbreviation: STRZ

Definition: The STRINGSIZE condition is raised when a string is about to be assigned to a shorter string.

Result: After the interrupt, the truncated string is assigned to its target string. The right hand characters or bits of the source string are truncated so that the target string can accomodate the source string.

Standard System Action: A message is printed and processing continues.

Status: STRINGSIZE is disabled by default and within the scope of a NOSTRINGSIZE condition prefix. It is enabled only within the range of a STRINGSIZE condition prefix.

Normal Return: On normal return from the on-unit, execution continues from the point of interruption.

SUBSCRIPTRANGE Program-checkout

Abbreviation: SUBRG

Description: SUBSCRIPTRANGE can be raised whenever a subscript is evaluated and found to lie outside its specified bounds. The condition is also raised when an iSUB subscript is outside the range given in the declaration of the iSUB defined array. The order of raising SUBSCRIPTRANGE relative to evaluation of other subscripts is undefined.

Result: When SUBSCRIPTRANGE has been raised, the value of the illegal subscript is undefined, and, hence, the reference is also undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: SUBSCRIPTRANGE is disabled by default and within the scope of a NOSUBSCRIPTRANGE condition prefix. It is enabled only within the scope of a SUBSCRIPTRANGE condition prefix. Under the checkout compiler, the standard system action takes place for SUBSCRIPTRANGE under the circumstances given under "Description" above, even when the condition is disabled; no on-unit for this condition can be entered, however, while it is disabled.

Normal Return: Normal return from a SUBSCRIPTRANGE on-unit raises the ERROR condition.

TRANSMIT (element-file-expr) Input/Output

Description: The TRANSMIT condition can be raised during any input/output operation. It is raised by a permanent transmission error and therefore signifies that any data transmitted is potentially incorrect.

During input, TRANSMIT is raised after assignment of the potentially incorrect record. If records are blocked, TRANSMIT is raised for each subsequent record in the block. During output, TRANSMIT is raised after transmission of the potentially incorrect data item has been attempted.

If records are blocked, transmission will occur when the block is complete, rather than after each I/O statement

When a spanned record is being updated, the TRANSMIT condition is raised on the last segment of a record only. It is not raised for any subsequent records in the same block, although the integrity of these records cannot be assumed.

If TRANSMIT is raised by an input/output statement using the EVENT option, the interrupt does not take place until the execution of a subsequent WAIT statement for that event in the same procedure.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: TRANSMIT is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the on-unit, processing continues as though no error had occurred, allowing another condition (e.g., RECORD) to be raised by the statement or data item that raised the TRANSMIT condition. (If TRANSMIT is raised by an input/output statement with an EVENT option, control returns to the WAIT statement from which the on-unit was invoked.)

Note: If a file is closed in an on-unit for this condition, the results of normal return are undefined. Exit from such an on-unit should be by means of a GO TO statement

UNDEFINEDFILE (element-file-expr) Input/Output

Abbreviation: UNDF(element-file-expr)

Description: The UNDEFINEDFILE condition is raised whenever an attempt to open a file is unsuccessful. If the attempt is made by means of an OPEN statement that specifies more than one file name, then the condition is raised as follows:

Checkout compiler: After an attempt to open each file

Optimizing compiler: After attempts to open all the other files specified in the statement

If the condition is raised for more than one file in the same OPEN statement, on-units will be executed according to the order of appearance (taken from left to right) of the file names in that OPEN statement.

If the condition is raised by an implicit file opening in an input/output statement without the EVENT option, then, upon normal return from the on-unit, processing continues with the remainder of the interrupted input/output statement. If the file was not opened in the on-unit, then the statement cannot be continued and the ERROR condition is raised.

If the condition is raised by an implicit file opening in an input/output statement having an EVENT option, then the interrupt occurs before the event variable is initialized. In other words, the event variable retains its previous value and remains inactive. On normal return from the on-unit, the event variable is initialized, that is, it is made active and its completion value is set to '0'B (provided the file has been opened in the on-unit). Processing then continues with the remainder of the interrupted statement. However, if the file has not been opened in the on-unit, the event variable remains uninitialized, the statement cannot be continued, and the ERROR condition is raised.

Some cases for which the UNDEFINEDFILE condition is raised are as follows:

1. A conflict in attributes exists.
2. The blocksize has not been specified.
3. There is no recognizable DD statement for the file.
4. The TOTAL option of the environment attribute has been specified and either attributes have been added on

an OPEN statement or attributes implied by an I/O statement conflict with default attributes.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: UNDEFINEDFILE is always enabled; it cannot be disabled.

Normal Return: Upon the normal completion of the final on-unit, control is given to the statement immediately following the statement that caused the condition to be raised (see "Description" for action in the case of an implicit opening).

UNDERFLOW Computational

Abbreviation: UFL

Description: The UNDERFLOW condition occurs when the magnitude of a floating-point number is smaller than the permitted minimum. (For System/360 implementations, the magnitude of a non-zero floating-point value may not be less than approximately 10^{-78} or 2^{-260}.)

UNDERFLOW does not occur when equal numbers are subtracted (often called significance error).

Note that the expression X**(-Y) (where Y>0) can be evaluated by taking the reciprocal of X**Y; hence, the OVERFLOW condition may be raised instead of the UNDERFLOW condition.

Result: The invalid floating-point value is set to 0.

Standard System Action: In the absence of an on-unit, the system prints a message and continues execution from the point at which the interrupt occurred.

Status: UNDERFLOW is enabled throughout the program, except within the scope of a condition prefix specifying NOUNDERFLOW.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

ZERODIVIDE Computational

Abbreviation: ZDIV

Description: The ZERODIVIDE condition occurs when an attempt is made to divide by zero. This condition is raised for fixed-point and floating-point division.

Result: The result of a division by zero is undefined.

Standard System Action: In the absence of an on-unit, the system prints a message and raises the ERROR condition.

Status: ZERODIVIDE is enabled throughout the program, except within the scope of a condition prefix specifying NOZERODIVIDE.

Normal Return: Upon normal termination of the on-unit for this condition, control returns to the point immediately following the point of interrupt.

Section I: Attributes

This section gives detailed descriptions of all attributes in alphabetical order. Alternative attributes are discussed together.

Figure I.1 has been compiled from the individual rules for attributes and is intended to serve as a quick reference to the following:

1. The classification of attributes according to data type.

2. The valid combinations of attributes that may be applied to a data item.

   For a variable, attributes must be selected from the columns Data Attributes, Scope Attributes, Storage Attributes, and Alignment Attributes. For the types of constants shown in the table, attributes must be selected from columns Data Attributes and Scope Attributes. Note that a complete set of attributes for a data item may be obtained by explicit or contextual declaration and programmer-defined or standard defaults.

3. Those attributes that conflict.

   Attributes shown as applying to one data type conflict with those of any other data type, except for those attributes shown as applying to both types. Alternative attributes within a data type, e.g., BIT and CHARACTER, are conflicting.

The following example illustrates the function of the figure:

```
DECLARE ST BIT(10);
```

Given the above declaration, the standard default attributes, AUTOMATIC, INTERNAL, and UNALIGNED will be applied to the name ST.

Figure I.2 is an expansion of the entry for file constants in figure I.1, to include the relationships between file attributes and options of the ENVIRONMENT attribute for the different data set organizations. The figure also shows the attribute implications of each file attribute.

ALIGNED and UNALIGNED

Abbreviation: UNAL for UNALIGNED

The ALIGNED and UNALIGNED attributes specify the positioning of data elements in storage, to influence speed of access or storage economy respectively. They may be specified for element, array, or structure variables.

ALIGNED specifies that the data element is to be aligned on the storage boundary corresponding to its data type requirement.

UNALIGNED specifies that a bit string is to be mapped on the next available bit boundary, and that a halfword, a word, or doubleword item is to be mapped on the next available byte boundary.

General format:

```
ALIGNED|UNALIGNED
```

General rules:

1. Although they are essentially element data attributes, ALIGNED and UNALIGNED can be applied to any array or structure. This is equivalent to applying the attribute to all contained elements that are not explicitly declared with the ALIGNED or UNALIGNED attribute.

2. Application of either attribute to a contained array or structure overrides an ALIGNED or UNALIGNED attribute that otherwise would apply to elements of the contained aggregate by having been specified for the containing structure.

3. The LIKE attribute is expanded before the ALIGNED and UNALIGNED attributes are applied to the contained elements of the LIKE structure variable. The only ALIGNED and UNALIGNED attributes that are carried over from the LIKE structure variable are those explicitly specified for substructures and elements of the structure variable.

DATA TYPE	DATA ATTRIBUTES	SCOPE ATTRIBUTES	STORAGE ATTRIBUTES	ALIGNMENT ATTRIBUTES
Arithmetic variable[1]	REAL|COMPLEX FLOAT|FIXED BINARY|DECIMAL (precision)	{ INTERNAL } { EXTERNAL } INTERNAL is standard default and mandatory for: AUTOMATIC, BASED, DEFINED, parameter and standard default for: CONTROLLED, STATIC	Storage Class: { AUTOMATIC } { STATIC } { BASED } { CONTROLLED } AUTOMATIC is standard default for INTERNAL. STATIC is standard default for EXTERNAL. [INITIAL] Defined: DEFINED [POSITION] Simple Parameter: parameter [CONNECTED] Controlled Parameter: parameter CONTROLLED [INITIAL]	{ ALIGNED } { UNALIGNED }
String variable	BIT|CHARACTER (length) [VARYING]			{ ALIGNED } { UNALIGNED }
Picture variable	{ PICTURE } { REAL|COMPLEX PICTURE }			
Label variable	LABEL			{ ALIGNED } { UNALIGNED }
File variable	FILE VARIABLE			
Entry variable[2]	ENTRY IRREDUCIBLE|REDUCIBLE RETURNS [OPTIONS] [VARIABLE]			
Locator variable	POINTER|{OFFSET[(area-variable)]}			
Area variable	AREA(size)			ALIGNED
Event variable	EVENT			
Task variable	TASK			
File constant[3]	FILE[ENVIRONMENT] STREAM|RECORD INPUT|OUTPUT|UPDATE SEQUENTIAL|DIRECT| TRANSIENT BUFFERED|UNBUFFERED [KEYED] [BACKWARDS] [PRINT] [EXCLUSIVE]	{ INTERNAL } { EXTERNAL }	Aggregate Variables Arrays: (dimension) may be added to the declaration of any variable. Structures: the attributes that may be specified for a name in a structure depend upon the level at which the name is declared: 1. For a major structure name, exclude data type; the LIKE attribute may be specified. 2. For a minor structure name, exclude data type, scope, and storage; the LIKE attribute may be specified. 3. For a base element name, exclude scope and storage.	
Entry constant	(as for entry variables, but excluding VARIABLE)	EXTERNAL		
Built-in entry constant	BUILTIN	INTERNAL		
Generic entry constant	GENERIC			
Condition constant	CONDITION	{ INTERNAL } { EXTERNAL }		

Standard default attributes are underlined.

[1] Identifiers that are implicitly declared (or explicitly declared with only scope, storage, or alignment) are assumed to be arithmetic variables. If the initial letter of the identifier is I through N, FIXED BINARY (15,0) are standard defaults; all others are FLOAT DECIMAL (6). If BINARY, DECIMAL, REAL, or COMPLEX are specified, FLOAT is standard default; otherwise if precision is specified with a scale factor, FIXED is standard default.

[2] ENTRY is implied by IRREDUCIBLE, REDUCIBLE, RETURNS, or OPTIONS. An entry constant may have the parameter attribute.

[3] File attributes, and their relationship to options of the ENVIRONMENT attribute, are described in Figure I.2. A file constant may have the parameter attribute.

Figure I.1. Classification of attributes according to data type

	Applicable Attributes and Options	STREAM	RECORD: SEQUENTIAL: BUF: CONSECUTIVE	BUF: INDEXED	BUF: REGIONAL	BUF: Teleprocessing	UNBUF: CONSECUTIVE	UNBUF: REGIONAL	DIRECT: INDEXED	DIRECT: REGIONAL	Attributes Implied
	Types of File										
F	FILE	I	I	I	I	I	I	I	I	I	
I	STREAM	D	-	-	-	-	-	-	-	-	FILE
L	RECORD	-	I	I	I	I	I	I	I	I	FILE
E	INPUT	D	D	D	D	D	D	D	D	D	FILE
	OUTPUT	O	O	O	O	O	O	O	O	O	FILE
A	UPDATE	-	O	O	O	-	O	O	O	O	FILE RECORD
T	SEQUENTIAL	-	D	D	D	-	D	D	-	-	FILE RECORD
T	DIRECT	-	-	-	-	-	-	-	S	S	FILE RECORD KEYED
R	BUFFERED	-	D	D	D	I	-	-	-	-	FILE RECORD SEQUENTIAL
I	UNBUFFERED	-	-	-	-	-	S	S	-	-	FILE RECORD SEQUENTIAL
B	PRINT	O	-	-	-	-	-	-	-	-	FILE STREAM OUTPUT
U	BACKWARDS	-	O	-	-	-	O	-	-	-	FILE RECORD SEQUENTIAL INPUT
T	KEYED	-	-	O	O	I	-	O	I	I	FILE RECORD
E	TRANSIENT	-	-	-	-	I	-	-	-	-	FILE
	EXCLUSIVE	-	-	-	-	-	-	-	O	O	FILE RECORD
	ENVIRONMENT	I	I	S	S	S	I	S	S	S	FILE
O	F|FB|FS|FBS|V| VB|VS|VBS|U	I	S	-	-	-	S	-	-	-	
P	F|FB|D|DB|U	S	S	-	-	-	-	-	-	-	ASCII data sets only
T	F|V|VS|U	-	-	-	S	-	-	S	-	S	Only F for REGIONAL (1) and (2)
I	F|FB|V|VB	-	-	S	-	-	-	-	S	-	VS invalid with UNBUF
O	RECSIZE(n)	I	I	I	I	I	I	I	I	I	One or both must be specified for
N	BLKSIZE(n)	I	I	I	I	-	I	I	I	I	CONSECUTIVE, INDEXED, and REGIONAL files.
S	ASCII	O	O	-	-	-	-	-	-	-	
O	BUFOFF(n)	O	O	-	-	-	-	-	-	-	
F	CTLASA|CTL360	-	O	-	-	-	O	-	-	-	invalid for ASCII data sets.
	SCALARVARYING	-	O	O	O	-	O	O	O	O	
E	LEAVE	O	O	-	-	-	O	-	-	-	
N	REREAD	O	O	-	-	-	O	-	-	-	
V	COBOL	-	O	O	O	-	O	O	O	O	
I	BUFFERS(n)	I	I	I	I	I	-	-	-	-	
R	CONSECUTIVE	-	D	-	-	-	D	-	-	-	
O	INDEXED	-	-	S	-	-	-	-	S	-	
N	REGIONAL ({1|2|3})	-	-	-	S	-	-	S	-	S	
M	TP({M|R})	-	-	-	-	S	-	-	-	-	
E	KEYLENGTH(n)	-	-	S	S	-	-	S	S	S	for REGIONAL(2) and (3) OUTPUT only
N	KEYLOC(n)	-	-	O	-	-	-	-	O	-	
T	NCP(n)	-	O	O	O	-	O	O	O	O	
	TRKOFL	-	O	-	O	-	O	O	-	O	invalid for REGIONAL(3)
	INDEXAREA(n)	-	-	-	-	-	-	-	O	-	
	ADDBUFF(n)	-	-	-	-	-	-	-	O	-	
	NOWRITE	-	-	-	-	-	-	-	O	-	UPDATE files only.
	GENKEY	-	-	O	-	-	-	-	O	-	INPUT or UPDATE files only; KEYED is required.
	TOTAL	O	O	O	O	O	O	O	O	O	

Key:

- I attribute or option must be specified or implied.
- D default attribute or option.
- O optional attribute or option: specified only if required.
- S attribute or option must be specified.
- \- invalid attribute or option.

The term "specified" includes the appearance of an option in the ENVIRONMENT attribute or in the DCB subparameter of the DD card.

Additional Notes:

1. UPDATE is invalid for tape files.
2. BACKWARDS is valid only for tape files.
3. KEYED is required for INDEXED and REGIONAL output.

Figure I.2. File declarations

4. For overlay defining involving bit- and character-class data, both the defined item and the overlaid part of the base item must be UNALIGNED. For all other types of defining, equivalent items must be either both ALIGNED or both UNALIGNED.

5. The ALIGNED and UNALIGNED attributes of an argument actually passed must match the attributes of the corresponding parameter. If these attributes of the original argument do not match those of the corresponding parameter, a dummy argument is created.

6. If a based variable is used to refer to a generation of another variable, the ALIGNED and UNALIGNED attributes of both variables must agree.

7. The alignment of string data depends not only on the use of ALIGNED or UNALIGNED, but also on whether the strings are fixed-length or varying-length. A summary of string alignment is included in figures K.1 and K.2.

8. TASK, EVENT, and AREA cannot be unaligned.

9. If an unaligned fixed-length bit string is used as the argument of the ADDR function, or appears as the first element of a based structure which is used in a LOCATE or ALLOCATE statement, the locator value returned may not address the bit string at the first bit position.

Assumptions:

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

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

3. Constants take the default for ALIGNED or UNALIGNED.

AREA

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

General format:

```
AREA [(size)]
```

General rules:

1. The area size for areas that are not of static storage class is given by an expression whose integral value specifies the number of bytes to be reserved.

2. The size for areas of static storage class must be specified as a decimal integer constant. The theoretical maximum size permitted is 16,777,200 bytes; in practice the maximum depends on the amount of main storage available to the program.

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

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

5. No operators can be applied to area variables.

6. An area variable cannot be unaligned.

7. If an area has the BASED attribute, the size attribute must be a decimal integer constant unless the area is a member of a based structure and the REFER option is used (see chapter 8, "Storage Control").

8. For RECORD input/output, only the extent (rather than the declared size) and control information of an area is transmitted (except when the area is in a structure and is not the last item in it - then, the declared size is transmitted).

Assumptions:

1. If the size specification is omitted, a default value is assumed. For this implementation, it is 1000.

2. An area variable can be contextually declared by its appearance in an OFFSET attribute or an IN option.

AUTOMATIC, STATIC, CONTROLLED and BASED

Abbreviations: AUTO for AUTOMATIC
CTL for CONTROLLED

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

AUTOMATIC specifies that storage is to be allocated upon each entry to the block to which the storage declaration is internal. The storage is released upon exit from the block. If the block is a procedure that is invoked recursively, the previously allocated storage is "pushed down" upon entry; the latest allocation of storage is "popped up" upon termination of each generation of the recursive procedure (for a discussion of push-down and pop-up stacking, see chapter 6, "Program Organization").

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

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

BASED, like CONTROLLED, specifies that full control over storage allocation and freeing will be maintained by the programmer, but by various methods that are described in chapter 8, "Storage Control" multiple allocations are not stacked but are available at any time; each can be identified by the value of a pointer variable.

General format:

```
STATIC|AUTOMATIC|CONTROLLED|
BASED[(element-locator-expression)]
```

General rules:

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

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

3. Parameters can be declared explicitly with the storage class attribute CONTROLLED, but not STATIC, BASED, or AUTOMATIC.

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

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

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

   a. Whenever a locator value is needed to complete a based variable reference, and none is explicitly specified, the value of the locator expression in the relevant BASED attribute is used. It is an error if no locator has been declared.

   b. When reference is made to a based variable, the data attributes assumed are those of the based variable, while the qualifying pointer variable identifies the location of data.

   c. A based variable can be used to identify and describe existing data; to obtain storage by means of the ALLOCATE statement; or to obtain storage in an output buffer by means of the LOCATE statement.

   d. The relative locations of based variables allocated within an area can be identified by the values of offset variables.

   e. The EXTERNAL attribute cannot appear with a based variable declaration, but a based variable reference can be qualified by an external pointer variable.

f. A based structure can be declared to contain adjustable area-sizes, array-bounds, and string-length specifications, by using the REFER option. See chapter 8, "Storage Control".

g. References to based variables in a CHECK prefix list or in a data list for data directed input/output cannot be explicitly locator qualified.

h. A BASED VARYING string must have a maximum length equal to the maximum length of any string upon which it is defined. For example:

```
DECLARE A CHAR(50) VARYING
                   BASED(Q),
        B CHAR(50) VARYING;
        Q=ADDR(B);
```

i. The INITIAL attribute may be specified for a based variable. The values are used only upon explicit allocation of the based variable with an ALLOCATE or LOCATE statement.

   If both the REFER option and the INITIAL attribute are used for the same member, initialization is done after the object of the REFER has been assigned its value.

Assumptions:

1. Default storage class is AUTOMATIC for internal variables and STATIC for external variables.

2. A pointer variable can be contextually declared by its appearance:

   in the BASED attribute

   in the SET option of a LOCATE, ALLOCATE, or READ statement

   as a locator qualifier.

BACKWARDS

The BACKWARDS attribute specifies that the records of a SEQUENTIAL INPUT file associated with a data set on magnetic tape are to be accessed in reverse order, i.e., from the last record to the first record.

General format:

```
BACKWARDS
```

General rules:

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

2. The BACKWARDS attribute applies to magnetic tape files only.

BASED

See AUTOMATIC.

BINARY and DECIMAL

Abbreviations: BIN for BINARY
DEC for DECIMAL

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

General format:

```
BINARY|DECIMAL
```

General rule:

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

Assumptions:

Undeclared identifiers (or identifiers declared only with one or more of the dimensions, UNALIGNED, ALIGNED, scope, and storage class attributes) are assumed to be arithmetic variables with assigned attributes depending upon the initial letter. For identifiers beginning with any letter I through N, the standard default attributes are REAL FIXED BINARY (15,0). For identifiers beginning with any other alphabetic character, the standard default attributes are REAL FLOAT DECIMAL (6). If FIXED or FLOAT and/or REAL or COMPLEX are declared, then DECIMAL is assumed.

BIT, CHARACTER and VARYING

Abbreviations: CHAR for CHARACTER
VAR for VARYING

The BIT and CHARACTER attributes are used to specify string variables. The BIT attribute specifies a bit string. The CHARACTER attribute specifies a character string.

General format:

```
{BIT      }
{CHARACTER} [(length)] [VARYING]
```

General rules:

1. The length attribute specifies the length of a fixed-length string or the maximum length of a varying-length string. If it is not specified, a length of one is assumed. For a bit string the length is specified in bits, and for a character string, in bytes.

2. The VARYING attribute specifies that the variable is to represent varying-length strings, in which case length specifies the maximum length. The current length at any time is the length of the current value. The storage allocated for varying-length strings is two bytes longer than the declared maximum length. The initial two bytes hold the string's current length (in bytes for a character string or bits for a bit string).

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

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

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

   The asterisk notation can be used for parameters or controlled variables. The length can be taken from a previous allocation or, for CONTROLLED variables, it can be specified in a subsequent ALLOCATE statement.

   There are restrictions on the use of asterisks and expressions in the length specifications of the elements of data aggregates in parameter descriptors: expressions may be used only for controlled parameters, and asterisks must not be used if the corresponding argument is such that a dummy is created.

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

6. If a string has the BASED attribute, the length attribute must be a decimal integer constant unless the string is a member of a based structure and the REFER option is used. (See chapter 8, "Storage Control").

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

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

9. The maximum length allowed for a bit- or character-string variable is 32,767. The minimum length for any string is zero.

BUFFERED and UNBUFFERED

Abbreviations: BUF for BUFFERED
UNBUF for UNBUFFERED

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

The UNBUFFERED attribute specifies that such records need not pass through buffers. It does not, however, specify that they must not. Hidden buffers will, in fact, be used if INDEXED, REGIONAL(2), or REGIONAL(3) is specified in the ENVIRONMENT attribute or if the records are variable-length.

General format:

```
BUFFERED|UNBUFFERED
```

General rules:

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

2. The locate-mode I/O statements LOCATE and READ SET can be used only on buffered files.

Assumption:

The standard default is BUFFERED.

BUILTIN

The BUILTIN attribute specifies that any reference to the associated name within the scope of the declaration is to be interpreted as a reference to the built-in

function, a pseudovariable, or built-in subroutine of the same name.

General format:

BUILTIN

General rules:

1. BUILTIN is used to refer to a built-in function, a pseudovariable or a built-in subroutine in a block that is contained in another block in which the same identifier has been declared to have another meaning.
2. If the BUILTIN attribute is declared for a name, the attribute INTERNAL is implied. No other attributes may be given to the name.
3. The BUILTIN attribute cannot be declared for parameters. Built-in functions without arguments should be declared, either explicitly, with the BUILTIN attribute, or contextually by using a null argument list, or implicitly using a DEFAULT statement. A list of these built-in functions is given in section G, "Built-in Functions and Pseudovariables."

CHARACTER

See BIT.

COMPLEX and REAL

Abbreviation: CPLX for COMPLEX

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

General format:

REAL|COMPLEX

General rule:

If a numeric character variable is to represent complex values, the COMPLEX attribute must be specified with the PICTURE attribute. The COMPLEX attribute is the only other arithmetic or string data attribute that can be specified with the PICTURE attribute.

Assumption:

The standard default is REAL.

CONDITION

Abbreviation: COND

The CONDITION attribute specifies that the associated identifier is a condition name.

General format:

CONDITION

General rules:

1. The only other attributes that can apply to a condition name are the scope attributes, INTERNAL and EXTERNAL.
2. The only statements in which a condition name can appear are ON, SIGNAL, REVERT, DECLARE, and DEFAULT.

Assumptions:

An identifier that appears with the CONDITION condition in an ON, SIGNAL, or REVERT statement is contextually declared to be a condition name.

The default scope is EXTERNAL.

CONNECTED

Abbreviation: CONN

The CONNECTED attribute is applied only to parameters, and specifies that the parameter will be a reference to connected storage only and, hence, allows the parameter to be used as a target or source in record-oriented I/O or as a base in string overlay defining.

General format:

CONNECTED

General rules:

1. CONNECTED is an additive attribute of non-controlled aggregate parameters and may be associated only with level-one names. It may be specified in a DECLARE statement or in a

parameter descriptor of an ENTRY attribute.

2. An argument passed to a CONNECTED parameter must be a reference to connected storage. If not, a dummy argument is created in connected storage.

CONTROLLED

See AUTOMATIC.

DECIMAL

See BINARY.

DEFINED

Abbreviation: DEF

The DEFINED attribute specifies that the variable being declared is to be associated with some or all of the storage associated with the designated base variable.

General format:

```
DEFINED{base-variable|(base-variable)}
   [POSITION(element-expression)]}
```

The "base-variable" is the variable whose storage is to be associated with the variable being declared; the latter is the "defined variable".

The POSITION attribute specifies the beginning of the part of a string base variable with which the defined variable is to be associated. The position is that of the first bit or character in the required part of the base variable.

General rules:

1. The purpose of defining one variable on another is to allow the programmer to refer to internally stored data by more than one name. The name of the base variable is the name initially declared for the data. Each variable defined on this base variable has a different name. If the internally stored data is a data aggregate, a defined variable can comprise all the data or only a specified part of it. The defined variable does not inherit any attributes from the base variable.

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

   If the POSITION attribute is specified, string overlay defining is in effect; in this case the base variable must not contain iSUB references. If the subscripts specified in the base variable contain references to iSUB variables, iSUB defining is in effect. If neither iSUB variables nor the POSITION attribute is present, then simple defining is in effect if the base variable and defined variable match according to the criteria given below; otherwise string overlay defining is in effect. For a tabulated summary of these rules, see Figure I.3.

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

   For simple and iSUB defining a PICTURE attribute can only be matched by a PICTURE attribute that is identical except for repetition factors. For a reference to specify a valid base variable in string overlay defining, the reference must be to connected storage. The implementation allows the programmer to override the matching rule completely, provided he is willing to accept that this could have unwanted side-effects on his program.

POSITION attribute specified	References to iSUB variables in base item subscripts	Base and defined match[1]	Type of defining in effect
YES	-	-	string overlay
NO	YES	-	iSUB
	NO	YES	simple
		NO	string overlay

[1]A definition of matching in this context is given in General Rule 2.

Figure I.3. Guide to types of defining

3. The values specified or derived for any array bounds, string lengths, or area sizes in a defined variable need not always match those of the base variable, but must be such that the defined array, string or area can be contained in the corresponding base array, string or area.

4. Some attributes of the base variable need not or cannot match those of the defined variable. The following restriction should be noted:

   Base variable:

   May be EXTERNAL or INTERNAL
   Qualified, or subscripted, or both
   A parameter (in string overlay defining, the parameter must refer to connected storage)

   Cannot be BASED
   DEFINED

   Defined variable:

   Must be INTERNAL
   A level-one identifier

   May have Dimension attribute

   Cannot be INITIAL
   AUTOMATIC/BASED/CONTROLLED/STATIC
   A parameter

5. If the base variable is EXTERNAL, it must be known in the procedure to which the defined variable is internal. An EXTERNAL base variable may be known in several external procedures; a change to its value made in one of these causes a similar change to the value of the defined variable.

6. In references to defined data, the SUBSCRIPTRANGE and STRINGSIZE conditions are raised for the array bounds and string lengths of the defined variable, not the base variable.

7. The determination of values and the interpretation of names occurs in the following sequence:

   a. The array bounds, string lengths, and area sizes of a defined variable are evaluated on entry to the procedure in which the variable is declared.

   b. A reference to a defined variable is a reference to the current generation of the base variable. When a defined variable is passed as an argument without creation of a dummy, the corresponding parameter refers to the generation of the base variable that is current when the argument is passed. This remains true even if the base variable is reallocated within the invoked procedure.

   c. When a reference is made to the defined variable, the order of

evaluation of the subscripts of the base and defined variable is undefined.

Simple Defining

Simple defining allows an element, array or structure variable to be referred to by another name.

General rules:

1. The defined and base variables can comprise any data type; they must match, in the sense described earlier in this section. If the ALIGNED or UNALIGNED attribute is specified for an element in the defined variable, it must also be specified for the corresponding element in the base variable.

2. The defined variable may have the dimension attribute. The base variable may be subscripted; the subscripts must not be iSUB variables.

3. The POSITION attribute cannot be used in simple defining.

4. In simple defining of an array:

   a. The base variable can be a cross-section of an array.

   b. The number of dimensions in the defined variable must be equal to the number of dimensions in the base variable.

5. In simple defining of a string, the length of the defined string must be less than or equal to the length of the base string.

6. In simple defining of an area, the size of the defined area must be equal to the size of the base area.

7. A base variable may be, or may contain, a VARYING string, provided that the corresponding part of the defined variable is a VARYING string of the same maximum length.

Examples:

```
DCL A(10,10,10),
    X1(2,2,2) DEF A,
    X2(10,10) DEF A(*,*,5),
    X3 DEF A(L,M,N);
```

X1 is a three-dimensional array that consists of the first two elements of each row, column and plane of A. X2 is a two-dimensional array that consists of the fifth plane of A. X3 is an element that consists of the element identified by the subscript expressions L,M, and N.

```
DCL B CHAR(10),
    Y CHAR(5) DEF B;
```

Y is a character string that consists of the first five characters of B.

```
DCL C AREA(500),
    Z AREA(500) DEF C;
```

Z is an area defined on C.

```
DCL 1 D UNALIGNED,
        2 E,
        2 F,
            3 G CHAR(10) VAR,
            3 H,
    1 S UNALIGNED DEF D,
        2 T,
        2 U,
            3 V CHAR(10) VAR,
            3 W;
```

S is a structure defined on D; for simple defining the organization of the two structures must be identical. A reference to T is a reference to E, V to G, etc.

iSUB Defining

iSUB defining allows a programmer to create a defined array that consists of designated elements from a base array. Both defined and base arrays can be arrays of structures.

General rules:

1. The defined and base arrays can comprise any data types, and must have identical attributes (apart from the dimension attribute).

2. The defined variable must have the dimension attribute. In the declaration of the defined array, the base array must be subscripted, and the subscript positions cannot be specified as asterisks.

3. The POSITION attribute cannot be used in iSUB defining.

4. An iSUB variable is a reference, in the subscript list for the base array, to the ith dimension of the defined array. At least one subscript in the base-array subscript-list must be an

iSUB expression which, on evaluation, gives the required subscript in the base array. The value of i ranges from 1 to n, where n is the number of dimensions in the defined array. The number of subscripts for the base array must be equal to the number of dimensions for the base array.

5. As well as the general rules for evaluation, the following should be noted:

   a. If a reference to a defined array does not specify a subscript expression, subscript evaluation occurs during the evaluation of the expression or assignment in which the reference occurs.

   b. The value of i is specified as a decimal integer constant. Within an iSUB expression, an iSUB variable is treated as a fixed binary variable, with default precision.

   c. A subscript in a reference to a defined variable is evaluated even if there is no corresponding iSUB in the base-variable subscript list.

6. iSUB-defined variables may not appear in the explicit or assumed data-list of a data-directed transmission statement or a CHECK statement or prefix.

Examples:

```
DCL A(100,100) CHAR(1),
    X(10,10) CHAR(1)
     DEF A(1SUB+20,2SUB+90);
```

X is a two-dimensional array that consists of the elements of A that lie within the bounds 21 - 30 for the first dimension, and 91 - 100 for the second dimension.

```
DCL B(2,5),
    Y(5,2) DEF B(2SUB,1SUB);
```

Y is a two-dimensional array that consists of the elements of B with the bounds transposed.

```
DCL A(10,10)
    B(5,5) DEF A(1+1SUB/5,1+2SUB/5);
```

In this case there is a many-to-one mapping of certain elements of B to a single element of A. B(I,J) is defined on:

A(1,1) for I<5 and J<5
A(1,2) for I<5 and J=5
A(2,1) for I=5 and J<5
A(2,2) for I=5 and J=5

Since all the elements B(I,J) are defined on the single element A(1,1) when I<5 and J<5, assignment of a value to one of these elements causes the same value to be assigned to all of them.

String Overlay Defining

String overlay defining allows a programmer to associate a defined variable with the storage for a base variable. Both the defined and the base variable must be string or picture data.

General rules:

1. Neither the defined nor the base variable can have the ALIGNED or the VARYING attributes.

2. Both the defined and the base variables must belong to the bit class, or both must belong to the character class. The bit class consists of:

   a. Fixed-length bit strings.

   b. Aggregates of fixed-length bit strings.

   The character class consists of:

   a. Fixed-length character strings.

   b. Character string and numeric pictured data.

   c. Aggregates of a and b.

3. iSUB variables cannot be used for the base variable in string overlay defining.

4. The POSITION attribute can be used to specify the bit or character within the base variable at which the defined variable is to begin. It has the format:

   POSITION(element-expression)

   where the expression, on evaluation, provides the position of the required bit or character relative to the start of the base variable. This attribute can precede or follow the DEFINED attribute; if it is omitted, POSITION(1) is assumed. The value provided by the expression can range from 1 to n, where n is defined as

n = N(b) - N(d) + 1

where N(b) is the number of bits or characters in the base variable, and
N(d) is the number of bits or characters in the defined variable.

The expression is evaluated, and converted to an integer, at each reference to the defined item. The absolute maximum permissible value is 32767.

5. When the defined variable is a bit class aggregate:

   a. the POSITION attribute can contain only an unsigned decimal integer constant;

   b. the base variable must not be subscripted.

6. The base variable must refer to data in connected storage.

7. Under the optimizing compiler, an array overlay-defined on another array is always assumed to be in unconnected storage. Under the checkout compiler, it is treated as being in unconnected storage only when the bounds of the base and defined items differ.

Examples:

```
DCL A CHAR(100),
    V(10,10) CHAR(1) DEF A;
```

V is a two-dimensional array that consists of all the elements in the character string A.

```
DCL B(10) CHAR(1),
    W CHAR(10) DEF B;
```

W is a character string that consists of all the elements in the array B.

```
DCL C(10,10) BIT(1),
    X BIT(40) DEF C POS(20);
```

X is a bit string that consists of 40 elements of C, starting at the 20th element.

```
DCL E PIC'99V.999',
    Z1(6) CHAR(1) DEF E,
    Z2 CHAR(3) DEF E POS(4),
    Z3(4) CHAR(1) DEF E POS(2);
```

Z1 is a character-string array that consists of all the elements of the decimal numeric picture E.
Z2 is a character string that consists of the elements '999' of the picture E.
Z3 is a character-string array that consists of the elements '9.99' of the picture E.

Dimension Attribute

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

General format:

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

where "bound" is:

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

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

General rules:

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

2. The bounds specification indicates the bounds as follows:

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

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

   c. An asterisk specifies that the actual bounds are to be specified in an ALLOCATE statement, if the variable is CONTROLLED, or in a declaration of an associated argument, if the variable is a simple parameter. Thus, the asterisk notation can be used only for parameters and CONTROLLED variables.

3. Bounds that are expressions are evaluated and converted to FIXED BINARY (15,0) when storage is allocated for the array. For simple parameters, bounds can be only optionally signed decimal integer constants or asterisks.

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

5. The bounds of arrays declared BASED must be optionally signed decimal integer constants unless the array is part of a based structure and the REFER option is used. (See chapter 8, "Storage Control".)

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

7. The maximum permissible number of dimensions is 15. The minimum permissible value for a lower bound is -32768; the maximum permissible for an upper bound is 32767.

DIRECT, SEQUENTIAL, and TRANSIENT

Abbreviation: SEQL for SEQUENTIAL

The DIRECT, SEQUENTIAL, and TRANSIENT attributes specify access information for the data set associated with a file.

The DIRECT and SEQUENTIAL attributes specify the manner in which the records in a data set associated with a RECORD file are to be accessed. SEQUENTIAL implies that the records are to be accessed according to their physical or logical sequence in the data set. (The records in an INDEXED data set are processed in their logical sequence; the records in a CONSECUTIVE or REGIONAL data set are processed in their physical sequence.) DIRECT specifies that the records are to be accessed by use of a key; each record must, therefore, have a key associated with it. Either of these two attributes implies the RECORD attribute.

The TRANSIENT attribute is designed for teleprocessing applications. It indicates that the contents of the data set associated with the file are reestablished each time the data set is accessed. In effect, this means that records can be continually added to the data set by one program during the execution of another program that continually removes records from the data set. Thus the data set can be considered to be a continuous queue through which the records pass in transit between a message control program and a message processing program.

Note that DIRECT and SEQUENTIAL specify only the current usage of the file; they do not specify physical properties of the data set associated with the file. The data set associated with a SEQUENTIAL file may actually have keys recorded with the data. Most data sets accessed by DIRECT files are created by SEQUENTIAL files. However, a data set associated with a TRANSIENT file differs from those associated with DIRECT and SEQUENTIAL files in that its contents are dynamic; reading a record removes it from the data set. Such a data set can never be created or accessed by a DIRECT or SEQUENTIAL file.

The use of TRANSIENT files is almost totally dependent on the implementation; for this reason, a list of rules for the use of TRANSIENT is given below the general rules and assumptions.

General format:

```
SEQUENTIAL|DIRECT|TRANSIENT
```

General rules:

1. DIRECT files must be KEYED; this attribute is implied by DIRECT. SEQUENTIAL files may or may not have the KEYED attribute.

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

3. TRANSIENT files must have the KEYED attribute.

Assumptions:

1. Default is SEQUENTIAL for RECORD files.

2. If a file is implicitly opened by an UNLOCK statement, DIRECT is assumed.

3. The TRANSIENT attribute does not imply any file attributes other than FILE.

The following rules apply specifically to the use of the TRANSIENT attribute:

1. The TRANSIENT attribute can be specified only for RECORD KEYED BUFFERED (or UNBUFFERED) files with either the INPUT or OUTPUT attribute.

2. The ENVIRONMENT attribute with one of the two teleprocessing format options (TP(M) or TP(R)) must be declared for TRANSIENT files.

3. Input can be specified only by a READ statement with the KEYTO option and either the INTO option or the SET option.

4. Output can be specified only by a WRITE statement or a LOCATE statement, either of which must have the KEYFROM option.

5. The EVENT option is not permitted.

6. The "data set" associated with a TRANSIENT file is in fact a queue of messages maintained automatically in main storage by a separate message control program using the teleprocessing facilities of the operating system. The queue is always accessed sequentially.

7. The element expression specified in the KEYFROM option should have as its value a recognized terminal or process queue identification.

ENTRY

The ENTRY attribute specifies that the identifier being declared is either an external entry constant or an entry variable. It is also used to describe the attributes of the parameters of the entry point.

General format:

```
ENTRY[(parameter-descriptor-list)]
```

where "parameter-descriptor-list" is:

```
[parameter descriptor[,parameter
descriptor]...]
```

Rules for Parameter Descriptor lists

1. A parameter descriptor list can only be given to describe the attributes of the parameters of the associated external entry constant or entry variable.

   If no parameter descriptor list is given, the arguments are assumed to match the parameters; if a parameter descriptor list is given, it is used for argument and parameter matching and the creation of dummy arguments; the parameter descriptor list must be supplied if arguments do not match the parameters.

2. A descriptor describes the attributes of a single parameter. For example, the descriptors for the parameters in the following procedure:

```
TEST:PROCEDURE (A,B,C,D,E,F);

   DECLARE A FIXED DECIMAL (5),
           B FLOAT BINARY (15),
           C POINTER,
           1 D,
            2 P,
            2 Q,
             3 R FIXED DECIMAL,
           1 E,
            2 X,
            2 Y,
             3 Z,
           F(4) CHARACTER (10);
            .
            .
            .
   END TEST;
```

could be declared as follows:

```
DECLARE TEST ENTRY
           (DECIMAL FIXED (5),
           BINARY FLOAT (15),
           ,
           1,
            2,
            2,
             3 DECIMAL FIXED,
            ,
           (4) CHARACTER (10));
```

3. The parameter descriptors must appear in the same order as the parameters they describe. If a descriptor is absent, the argument is assumed to match the parameter.

4. If a descriptor is not required for a parameter, the absence of a descriptor must be indicated in one of the following ways:

   by a comma:
   ENTRY(CHARACTER(10),,,FIXED DECIMAL) indicates four parameters;

   by an asterisk followed by a comma or the closing parenthesis of the parameter descriptor list: ENTRY(*) indicates one parameter;

   by the closing parenthesis when it follows a comma with no intervening descriptor: ENTRY(FLOAT BINARY,) indicates two parameters.

   A declaration ENTRY( ) is equivalent to ENTRY with no parameter descriptor list and the entry name must never have any arguments.

   In the example in rule 2 above, the parameter C has no descriptor nor has the structure parameter E.

5. In general, the attributes may appear in any order in a parameter

descriptor, but for an array parameter descriptor, the dimension attribute must be the first specified. For a structure parameter descriptor, the level numbers must appear in the same order as the level numbers of the corresponding parameter, and they must precede the attributes for each level; the descriptor level numbers need not be the same as those of the parameter, but the structuring must be identical; the attributes for a particular level may appear in any order.

Note: Each descriptor level number, together with any attributes specified for the level, is delimited by a comma (see example above).

6. Defaults are not applied to a parameter descriptor unless attributes or level numbers are specified in the descriptor. If a level number and/or the dimension attribute only is specified in a descriptor, FLOAT DECIMAL(6) REAL are assumed.

7. Extents (lengths, sizes, and bounds) in parameter descriptors may only be specified by decimal integer constants or by asterisks. Extents in descriptors for controlled parameters may only be specified by asterisks.

8. Attributes given in the parameter descriptor list can be established implicitly by use of the DEFAULT statement in conjunction with the DESCRIPTORS option. However they are not applied for missing descriptors.

General rules:

1. The ENTRY attribute, without a parameter descriptor list, is implied by the attributes OPTIONS, REDUCIBLE, IRREDUCIBLE, and RETURNS.

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

3. The ENTRY attribute must be specified or implied for a parameter representing an entry constant or entry variable argument.

   The maximum permissible depth of nesting of the ENTRY attribute is two. For example:

```
DCL E ENTRY(ENTRY(FIXED));
```

   is permissible, but:

```
DCL E ENTRY(ENTRY(ENTRY(FIXED)));
```

   is not permissible.

4. Factoring of attributes is not permitted within the parameter descriptor list of an ENTRY attribute specification.

5. External entry constants must be explicitly declared.

6. The optional attribute VARIABLE is an additive attribute. When given, it specifies that the associated identifier is an entry variable. The VARIABLE attribute is declared implicitly if the identifier is declared with any one or more of the following attributes:

ALIGNED	dimension
AUTOMATIC	INITIAL
BASED	parameter
CONTROLLED	STATIC
DEFINED	UNALIGNED

7. The use of an entry variable in a CALL statement or function reference means that associated entry points cannot be known until execution time. When an entry variable declared without a parameter descriptor list appears either in a CALL statement or as a function reference that involves passing arguments, the arguments are assumed to match the parameters of the referenced entry point. However, if a parameter descriptor list is given in the declaration of an entry variable, the parameters of the referenced entry point are assumed to match the attributes given in the parameter descriptor list; dummy arguments are created if necessary.

8. When a reference to any entry expression includes an argument list (which may be a null argument list), the procedure it represents is always invoked.

9. When a reference to any entry expression does not include an argument list, the procedure it represents is not invoked in the following contexts:

   a. The righthand side of an assignment to an entry variable.

   b. Comparison with an entry expression.

   c. An argument to a generic entry name.

   d. An argument passed to an entry parameter.

   e. An argument to the UNSPEC built-in function.

f. Any context that requires a variable (applicable only to entry variables).

10. An entry variable used in a CALL statement must have as its value an entry point of a block that is active at the time the CALL statement is executed. If the variable has an invalid value, the checkout compiler will raise the ERROR condition; under the optimizing compiler, however, detection of such an error is not guaranteed.

11. The values of two entry expressions may be compared using either the = or ¬= comparison operator. It is not an error to specify, in a comparison operation, an entry variable whose value is an entry point of an inactive block.

12. Entry names on the same PROCEDURE or ENTRY statement do not compare equal.

13. The ENTRY attribute cannot be specified in a RETURNS attribute or option. ENTRY statement do not compare equal.

Assumptions:

The ENTRY attribute can be implied. The appearance of an identifier as a label prefix of either a PROCEDURE statement or an ENTRY statement constitutes an explicit declaration of that identifier as an entry constant. Its attributes are obtained from this explicit declaration and from the declarations, if any, given in an additional DECLARE statement. The attributes are obtained as follows:

Scope attribute: For an external entry constant, the scope is EXTERNAL (INTERNAL is invalid). For an entry variable, the scope is INTERNAL by default.

RETURNS Attribute: This is obtained from the RETURNS attribute in the DECLARE statement.

ENVIRONMENT

Abbreviation: ENV

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

General format:

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

Options in the "option list" are separated by blanks or commas. The option list is defined individually for each implementation of PL/I. For this implementation, it is as follows:

```
[record-format] [BUFFERS(n)]
[data-set-organization]
[magnetic tape handling]
[carriage-control]
[COBOL] [data-management-optimization]
[key-classification]
[KEYLENGTH(n)]
[KEYLOC(n)]
[SCALARVARYING]
[teleprocessing format]
[direct access device usage]
[ASCII - data interchange code]
[BUFOFF[(n)] - buffer offset]
[TOTAL]
```

The options may appear in any order. They are described in chapter 11, "Stream-Oriented Transmission" and chapter 12, "Record-Oriented Transmission".

The ENVIRONMENT attribute may be included only in a DECLARE statement. It cannot be specified as an option of an OPEN statement. It can be specified as an option of the CLOSE statement for the volume disposition options LEAVE and REREAD.

EVENT

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

General format:

```
EVENT
```

General rules:

1. An identifier may be explicitly declared with the EVENT attribute in a DECLARE statement. It may be contextually declared by its appearance in an EVENT option of a CALL statement, in a WAIT statement, in a DISPLAY statement, or in various input/output statements (see chapter 10, "Input and Output", and chapter 17, "Multitasking").

2. Event names may also have the following attributes:

   Dimension

   Scope (the default is INTERNAL)

   Storage class (the default is AUTOMATIC)

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

   INITIAL or INITIAL CALL

3. An event variable has two separate values:

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

   b. A fixed-point binary value of default precision (i.e.,(15,0)) which reflects the status value of the variable. A zero value indicates normal, nonzero indicates abnormal status.

   The values of the event variable can be separately returned by use of the COMPLETION and STATUS built-in functions. The COMPLETION function returns a bit-string value corresponding to the completion value of the variable; STATUS returns a fixed binary value corresponding to the status value.

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

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

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

   a. Use of the COMPLETION pseudovariable, to set the completion value.

   b. Use of the STATUS pseudovariable, to set the status value.

   c. Event variable assignment.

   d. By a statement with the EVENT option.

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

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

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

6. On allocation of an event variable, its status and completion values are undefined.

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

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

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

10. On execution of an input/output statement with the EVENT option, the event variable, if inactive, is set to zero status value and to incomplete.

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

11. Event variables cannot be unaligned.

12. Two event variables can be compared using a = or a ¬= comparison operator. The variables compare equal if both the status and completion values are equal, otherwise they compare not equal.

EXCLUSIVE

Abbreviation: EXCL

The EXCLUSIVE attribute specifies that records in a DIRECT UPDATE file may be locked by an accessing task to prevent other tasks from interfering with an operation. The section entitled "EXCLUSIVE Attribute" in chapter 10, "Input and Output", contains a table showing the effects of various operations on EXCLUSIVE files and the records contained in them.

General format:

```
EXCLUSIVE
```

General rules:

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

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

3. A DELETE or REWRITE statement referring to a locked record will automatically unlock the record at the end of the DELETE or REWRITE operation; if the record has been locked by another task, the task executing the DELETE or REWRITE statement will wait until the record is unlocked. While a DELETE or REWRITE operation is taking place, the record is always locked.

4. Automatic unlocking takes place at the end of the operation, on completion of any on-units entered because of the operation (that is, at the corresponding WAIT statement when the EVENT option has been specified) or by a GO TO branch out of such an on-unit.

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

6. Closing an EXCLUSIVE file unlocks all the records locked by that task in the file.

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

Assumptions:

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

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

EXTERNAL and INTERNAL

Abbreviations: EXT for EXTERNAL
INT for INTERNAL

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

General format:

```
EXTERNAL|INTERNAL
```

General rules:

1. When a major structure name is declared EXTERNAL in more than one block, the attributes of the structure members must be the same in each case, although the corresponding member names need not be identical.

2. Members of structures always have the INTERNAL attribute and cannot be declared with any scope attribute. However, a reference to a member of an external structure, using the member name known to the block containing the reference, is effectively a reference to that member in all blocks in which the external name is known, regardless of whether the corresponding member names are identical.

Assumptions:

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

FILE

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

General format:

FILE

General rules:

1. File description attributes, such as RECORD, INPUT, etc., cannot be applied to a file variable.

2. A file expression is a file constant, a file variable or a function reference that represents a file value. It may be used as:

   a. an argument to the FILE or COPY option

   b. an argument to be passed to a function or subroutine

   c. an argument to an input/output condition name for ON, SIGNAL, and REVERT statements

   d. an argument to a RETURN statement

3. On-units can be established for a file constant through a file variable that represents its value.

For example:

```
DCL F FILE,
    G FILE VARIABLE;
    G=F;
L1:  ON ENDFILE(G);
L2:  ON ENDFILE(F);
```

The statements labelled L1 and L2 are equivalent.

4. A dummy argument is created for a file constant argument to a CALL statement or function reference.

5. A file variable may be specified in a CHECK prefix list. The CHECK condition is not raised for such a file variable by its appearance as a FILE option in ON, SIGNAL, and REVERT statements.

6. The value of a file variable may be transmitted by record-oriented transmission statements. The value may not be valid after transmission.

7. The values of two file expressions may be compared using either the = or ¬= comparison operator. The expressions compare equal only if they represent file values, all of whose parts are equal.

Assumptions:

The FILE attribute can be implied for a file constant by any of the "file description attributes". Refer to chapter 10, "Input and Output", for discussion of the file attributes. In addition, an identifier can be contextually declared as a file constant through its appearance in the FILE option of any input or output statement, or in an ON statement for any input/output condition.

An identifier with the FILE attribute is assumed to be a file variable if the identifier is an element of an array or structure, or if any of the following additional attributes is specified:

Storage class attributes
dimension attributes
parameter
ALIGNED or UNALIGNED
DEFINED
INITIAL
VARIABLE

FIXED and FLOAT

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

General format:

```
FIXED|FLOAT
```

General rule:

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

Assumptions:

Undeclared identifiers (or identifiers declared only with one or more of the dimension, ALIGNED or UNALIGNED, scope, and storage class attributes) are assumed to be arithmetic variables with assigned attributes depending upon the initial letter. For identifiers beginning with any letter I through N, the standard default attributes are REAL FIXED BINARY (15,0). For identifiers beginning with any other alphabetic character, the standard default attributes are REAL FLOAT DECIMAL (6). If BINARY or DECIMAL and/or REAL or COMPLEX are specified, FLOAT is assumed.

FLOAT

See FIXED.

GENERIC

The GENERIC attribute is used to define an entry name that is generic to a specified group of entry expressions. When the generic name is referred to, one of the specified entry expressions is selected, based upon the arguments specified for the generic name in the reference.

General format:

```
GENERIC (entry-expression WHEN
        (generic-descriptor-list)
        [,entry-expression WHEN
        (generic-descriptor-list)]...);


where generic-descriptor-list is:-
        [descriptor[,descriptor]...]
```

General rules:

1. The only attribute than can be specified for the name being given the GENERIC attribute is INTERNAL.

2. Each entry expression following the GENERIC attribute corresponds to one member of the generic group. An entry-expression must be a constant or variable of type ENTRY. It must not be based, subscripted, or defined.

3. The same entry-expression may appear more than once within a single GENERIC declaration with different lists of descriptors.

4. The selection of a particular entry expression is based upon the arguments of, or absence of all arguments from, the reference to the generic name. When a generic name is referred to, the number of arguments and attributes of each argument are compared with each generic descriptor list from left to right until all the attributes in one generic descriptor list are found to be attributes of the arguments. The reference is then interpreted as a reference to the member with the matching generic descriptor list.

5. The only attributes allowed are those that affect generic selection; these are:

   ```
   ALIGNED
   AREA (No size may be specified)
   Base
   BIT (No length may be specified)
   CHARACTER (No length may be
              specified)
   ENTRY (No descriptor list may be
         specified)
   EVENT
   FILE
   LABEL (No label list may be
         specified)
   Mode
   OFFSET (No area variable may be
          specified)
   PICTURE 'picture-specification'
   POINTER
   Precision (Number of digits and
             scale factor must be
             specified)
   Scale
   TASK
   UNALIGNED
   VARYING
   ```

   A missing descriptor may be indicated by an asterisk or a comma in the generic descriptor list.

6. An entry expression used as an argument in a reference to a generic value only matches a descriptor of

type ENTRY. If there is no such description, the program is in error.

7. An argument with the GENERIC attribute matches an ENTRY attribute in a generic descriptor list.

8. Under the optimizing compiler, if a locator attribute (POINTER or OFFSET) is specified in the generic descriptor list, the corresponding parameter must have the same attribute; no conversion from one type to the other can be performed when the entry-point is invoked. Under the checkout compiler, the conversion can be performed.

9. Aggregates may not be specified in a generic descriptor list, though they may be passed as arguments to a generic entry name. Under the optimizing compiler, no dummy argument can be created for such an aggregate.

10. Generic names (as opposed to references) may be specified as arguments to non-generic entry names.

    If the non-generic entry name is an entry variable or an external entry constant it must be declared with a parameter descriptor list. The descriptor for the generic argument must be ENTRY with a parameter descriptor list. This nested list is used to select the argument to be passed. For example:

```
A:  PROC;
 DCL B GENERIC (C WHEN(FIXED),
                D WHEN(FLOAT)),
    E ENTRY (ENTRY(FIXED));
 CALL E(B);
  .
  .
 END A;
```

    When procedure E is invoked, C is selected and passed as the argument, since the descriptor specifies that the parameter specified by the entry name parameter is FIXED.

    If the non-generic entry name is an internal entry constant, the corresponding parameter must be declared ENTRY with a parameter descriptor list. This list is used to select the argument to be passed. For example:

```
A:  PROC;
 DCL B GENERIC (C WHEN(FIXED),
                D WHEN(FLOAT));
 CALL E(B);
 E:  PROC(P);
    DCL P ENTRY(FIXED);
     .
     .
    END E;
 END A;
```

    When procedure E is invoked, C is selected and passed as the argument, since the parameter of entry name parameter is declared to be FIXED.

INITIAL

Abbreviation: INIT

The INITIAL attribute has two forms. The first specifies a constant, expression, or function reference, whose value is to be assigned to a data item when storage is allocated to it. The second form specifies that, through the CALL option, a procedure is to be invoked to perform initialization at allocation. The variable is initialized by assignment during the execution of the called routine (rather than by this routine being invoked as a function that returns a value to the point of invocation).

General format:

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

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

General rule:

The INITIAL attribute cannot be given to constants, defined data, structures or parameters (except CONTROLLED parameters).

Rules for form 1:

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

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

   [+|-] arithmetic-constant

   bit-string-constant

   character-string-constant

entry-constant

file-constant

label-constant

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

The term "expression" denotes an element expression used to provide an initial value to be assigned to the initialized data item. An expression is always enclosed in parentheses when specified in the INITIAL attribute. The term "reference" denotes a reference to a variable or a function which can be used for the initial value of the data item.

3. The time at which the INITIAL attribute is applied depends on the storage class of the variable.

   STATIC: When the external procedure in which the variable is declared is entered.

   AUTOMATIC: When the block in which the variable is declared is entered.

   CONTROLLED: When the ALLOCATE statement is executed.

   BASED: When an ALLOCATE or a LOCATE statement is executed for the variable. If the variable is referenced only by setting a pointer and is never specified in an ALLOCATE or LOCATE statement, the INITIAL attribute specified in a DECLARE statement is never applied.

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

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

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

7. Only constant values can be specified in the INITIAL attribute for STATIC variables, except that the NULL built-in function may be used to initialize a STATIC pointer variable.

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

   (iteration-factor)
   reference|constant| (expression)

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

   (iteration-factor) *

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

9. The iteration factor can be an element expression, except for STATIC data, in which case it must be an unsigned decimal integer constant. When storage is allocated for the array, the expression is evaluated to give an integer that specifies the number of iterations.

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

11. The initialization of an array of strings may include both string repetition and iteration factors. Where only one of these is given it is taken to be a string repetition factor unless the string constant is placed in parentheses. Note that a string repetition factor must be an unsigned decimal integer constant. For example, consider the following:

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

12. Iterations may be nested.

13. It is an error to specify an iteration factor in an INITIAL attribute of a scalar item.

14. Names used in expressions and function references for initial values must be

known within the block in which the initialized item is declared.

15. STATIC label or entry variables cannot have the INITIAL attribute.

16. An alternate method of initialization is available for elements of arrays of non-STATIC label variables: an element of a label array can appear as a statement prefix, provided that all subscripts are optionally signed decimal integer constants. The effect of this appearance is the initialization of that array element to a value that is a constructed label constant for the statement prefixed with the subscripted reference. This statement must be immediately internal to the block containing the declaration of the array. Only one form of initialization can be used for a given label array. If CHECK is specified for such an array and the elements of the array are initialized in this way, the CHECK condition is not raised at initialization.

17. If the attributes of an item in the INITIAL attribute differ from those of the data item itself, then, provided the attributes are compatible, conversion will be performed.

18. If a STATIC EXTERNAL item is given the INITIAL attribute in more than one declaration, the value specified must be the same in every case.

Rules for form 2:

1. The "entry-expression" and "argument-list" passed must satisfy the condition stated for prologues as discussed in chapter 6, "Program Organization".

2. Form 2 cannot be used to initialize STATIC data.

Examples:

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

b.
```
DECLARE MAXVALUE INITIAL (99),
        MINVALUE INITIAL (-99);
```

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

d.
```
DECLARE TABLE (20,20) INITIAL
        CALL SET_UP (X,Y);
```

e.
```
DECLARE 1 A(8),
          2 B INITIAL (0),
          2 C INITIAL ((8)0);
```

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

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

In Example d, SET_UP is the name of a procedure that sets the initial values of elements in TABLE. X and Y are arguments passed to SET_UP.

In Example e, B and C inherit a dimension of (8) but, whereas only the first element of B is initialized, all the elements of C are initialized.

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

INPUT, OUTPUT, and UPDATE

The INPUT, OUTPUT, and UPDATE attributes indicate the function of the file. INPUT specifies that data is to be transmitted from auxiliary storage to the program. OUTPUT specifies that data is to be transmitted from the program to auxiliary storage either to create a new data set or extend an existing one. UPDATE specifies that the data can be transmitted in either direction; that is, the file is both an input and an output file.

General format:

```
INPUT|OUTPUT|UPDATE
```

General rules:

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

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

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

Assumptions:

Default is INPUT. The PRINT attribute implies OUTPUT. The EXCLUSIVE attribute implies UPDATE.

INTERNAL

See EXTERNAL.

IRREDUCIBLE and REDUCIBLE

Abbreviations: IRRED for IRREDUCIBLE
RED for REDUCIBLE

These attributes are used for optimization. The checkout compiler merely checks them for syntax errors, applies the implied attribute, and then ignores them. Their presence in a program processed by the checkout compiler is not an error.

They are specified in entry-constant declarations of function procedures. REDUCIBLE specifies that if the entry name appears with an argument list that is identical to an argument list used in an earlier invocation, the function need not necessarily be reinvoked and the result of the earlier evaluation may be used. IRREDUCIBLE specifies that this type of optimization is not permitted. Optimization within a function procedure is not affected by either attribute.

General format:

```
IRREDUCIBLE|REDUCIBLE
```

General rule:

1. These attributes can be applied only to external entry constants or entry variables, since internal entry names cannot be declared. For internal entry constants, the equivalent options can be applied to PROCEDURE or ENTRY statements.

Assumptions:

The IRREDUCIBLE and REDUCIBLE attributes imply ENTRY.

The standard default is IRREDUCIBLE.

KEYED

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

General format:

```
KEYED
```

General rules:

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

2. The KEYED attribute can be specified for RECORD files only, and must be associated with direct access devices or with a file with the TRANSIENT attribute.

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

Assumption:

The DIRECT attribute implies KEYED.

LABEL

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

General format:

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

General rules:

1. If a list of statement label constants is given, the variable must have as its value a member of the list when used in a GO TO statement or R format item. The label constants in the list must be known in the block containing the declaration. Under the optimizing compiler, the maximum permissible number of label constants in the list is 125. There is no limit under the checkout compiler.

2. The parenthesized list of statement label constants can be used in a LABEL attribute specification for a label array.

3. A label variable may not be used to identify a PROCEDURE or ENTRY statement, and an entry constant may not be assigned to a label variable.

4. A subscripted label specifying an element of a label array can appear as a statement label prefix if the label variable is not STATIC, but it cannot appear in an END statement after the keyword END. For further information, see the INITIAL attribute.

5. A label variable may have another label variable or a label constant assigned to it. When such an assignment is made, the environment of the source label is assigned to the target.

6. The INITIAL attribute cannot be specified for STATIC label variables.

7. A label variable used in a GO TO statement must have as its value a label constant that is used in a block that is active at the time the GO TO is executed. If the variable has an invalid value, the checkout compiler will raise the ERROR condition; under the optimizing compiler, however, detection of such an error is not guaranteed.

8. Labels may be compared. Comparison operators permitted for labels are = and ¬=. Labels on the same statement compare equal. It is not an error to specify, in a comparison operation, a label variable whose value is a label constant used in a block that is no longer active.

9. A label prefixed to a null statement does not compare equal to a label prefixed to the statement immediately following the null statement.

   For example:

```
A:;
B:  X=7;
```

   Label A is not equal to label B.

10. A label prefixed to a FORMAT statement does not compare equal with the label prefixed to the following statement.

11. A label prefixed to an END statement does not compare equal with the label prefixed to the following statement.

12. The label on IF statement does not compare equal with that on the succeeding THEN clause.

Length Attribute

See BIT.

LIKE

The LIKE attribute specifies that the name being declared is a structure variable with the same structuring as that for the name following the attribute keyword LIKE. Substructure names, elementary names, and attributes for substructure names and elementary names are to be identical.

General format:

```
LIKE structure-variable
```

General rules:

1. The "structure-variable" can be a major structure name or a minor structure name. It can be a qualified name, but it cannot be subscripted.

2. The "structure-variable" must be known in the block containing the LIKE attribute specification. The structure names in all LIKE attributes are associated with declared structures before any LIKE attributes are expanded. For example:

```
DECLARE 1 A, 2 C, 3 E, 3 F,
        1 D, 2 C, 3 G, 3 H;
  .
  .
  .
BEGIN;
DECLARE 1 A LIKE D, 1 B LIKE A.C;
  .
  .
  .
END;
```

These declarations result in the following:

```
1 A LIKE D is expanded to give:

  1 A, 2 C, 3 G, 3 H

1 B LIKE A.C is expanded to give:

  1 B, 3 E, 3 F
```

3. a. Neither the "structure variable" nor any of its substructures can be declared with the LIKE attribute. For example, the following is invalid:

```
DECLARE 1 A LIKE C,
        1 B,
          2 C,
            3 D,
            3 E LIKE X,
          2 F,
        1 X,
          2 Y,
          2 Z;
```

because the LIKE attribute of A specifies a structure, C, that contains an identifier, E, that has the LIKE attribute.

b. "Structure variable" must not be a substructure of a structure declared with the LIKE attribute. For example, the following is invalid:

```
DECLARE 1 A LIKE G.C,
        1 B,
          2 C,
            3 D,
            3 E,
          2 F,
        1 G LIKE B;
```

because the LIKE attribute of A specifies a substructure, G.C, of a structure, G, declared with the LIKE attribute.

c. Under the optimizing compiler, no substructure of the major structure containing "structure variable" can have the LIKE attribute. For example, the following is invalid under the optimizing compiler:

```
DECLARE 1 A LIKE C,
        1 B,
          2 C,
            3 D,
            3 D,
            3 E,
          2 F LIKE X,
        1 X,
          2 Y,
          2 Z;
```

because the LIKE attribute of A specifies a structure, C, within a structure, B, that contains a substructure, F, having the LIKE attribute.

4. Neither additional substructures nor elementary names can be added to the created structure; any level number that immediately follows the "structure variable" in the LIKE attribute specification in a DECLARE statement must be algebraically equal to or less than the level number of the name declared with the LIKE attribute.

5. Attributes of the "structure variable" itself do not carry over to the created structure. For example, storage class attributes do not carry over. If the "structure variable" following the keyword LIKE represents an array of structures, its dimension attribute is not carried over. Attributes of substructure names and elementary names, however, are carried over; contained dimension and length attributes are recomputed. An exception is that this does not apply to the INITIAL attribute for any elements of a label array that has been initialized by prefixing to a statement.

6. If a direct application of the description to the structure declared LIKE would cause an incorrect continuity of level numbers (for example, if a minor structure at level 3 were declared LIKE a major structure at level 1) the level numbers are modified by a constant before application.

7. The LIKE attribute is expanded before the ALIGNED and UNALIGNED attributes are inherited by the contained elements of a structure.

8. The LIKE attribute is expanded before the standard defaults or DEFAULT statements are applied.

OFFSET and POINTER

Abbreviation: PTR for POINTER

The OFFSET and POINTER attributes describe locator variables. A pointer variable can be used in a based variable reference to identify a particular

generation of the based variable. Offset variables identify a location relative to the start of an area; pointer variables identify any location, including those within areas.

General format:

```
POINTER|OFFSET
  [(element-area-variable)]
```

General rules:

1. A pointer variable can be explicitly declared in a DECLARE statement, or it can be contextually declared by its appearance as a pointer qualifier, by its appearance in a BASED attribute, or by its appearance in a SET option.

2. An offset variable cannot be contextually declared. If no area variable is specified the offset can only be used as a locator qualifier through use of the POINTER built-in function.

3. The value of a pointer variable can be set in any of the following ways:

   a. With the SET option of a READ statement.

   b. By a LOCATE statement.

   c. By an ALLOCATE statement.

   d. By assignment of the value of another locator variable, or a locator value returned by a user-defined function.

   e. By assignment of an ADDR or NULL built-in function value.

4. The value of an offset variable can be set in any one of the following ways:

   a. By an ALLOCATE statement.

   b. By assignment of the value of another locator variable, or a locator value returned by a user-defined function.

   c. By assignment of the NULL built-in function value.

5. Locator variables cannot be operands of any operators other than the comparison operators = and ¬=.

6. Locator data cannot be converted to any other data type, but pointer can be converted to offset, and vice versa.

   A pointer value is converted to offset by effectively deducting the pointer value for the start of the area from the pointer value to be converted. This conversion is limited to pointer values that relate to addresses within the area named in the OFFSET attribute. Except when assigning the NULL built-in function value, it is an error to attempt to convert to an offset variable that is not associated with an area.

   In conversion of offset data to pointer, the offset value is added to the pointer value of the start of the area named in the OFFSET attribute. It is an error to attempt to convert an offset variable that is not associated with an area.

   In any conversion of locator data under the optimizing compiler, if the offset variable is a member of a structure, or if it appears in a DO statement or a multiple assignment statement, then the area associated with that offset variable must be an unsubscripted, non-defined, element variable. The area may be based, but if so, its qualifier must be an unsubscripted, non-based, non-defined pointer; and this pointer must not be used to qualify the area explicitly in declaration of the offset variable. No such restrictions apply to the checkout compiler.

7. A locator value can be assigned only to a locator variable. When an offset value is assigned to an offset variable, the area variables named in the OFFSET attributes are ignored.

8. With one exception, locator data cannot be transmitted using STREAM input/output. The exception is that, for the checkout compiler, locator variables can appear in a PUT DATA or PUT LIST statement.

9. Whenever implicit conversion between pointer and offset takes place the area variable designated in the OFFSET attribute is used to establish the value.

Assumption:

The variable named in the OFFSET attribute is contextually declared to have the AREA attribute.

OPTIONS

The OPTIONS attribute specifies characteristics of entry data. The OPTIONS attribute implies the ENTRY attribute and is additive. It has no effect on argument passing and generic selection.

General Format:

```
OPTIONS(options-list)
```

It is used in the following manner:

```
DECLARE identifier

 [ENTRY[(parameter-descriptor-list)]]

 [VARIABLE] OPTIONS(option-list);
```

The options are separated by blanks. For this implementation, the options are:

```
{COBOL|FORTRAN}

[NOMAP [(argument-list)]]
[NOMAPIN [(argument-list)]]
[NOMAPOUT [(argument-list)]]

[INTER]
```

These options specify facilities used for interlanguage communication. They are described briefly below; a full account of the effect and usage is given in chapter 19, "Interlanguage Communication".

General rules:

1. The OPTIONS attribute can only be used in an entry declaration. It can only be specified for external entry constants, or entry variables, or parameters.

2. The options can be specified in any order.

3. The COBOL option specifies that the designated entry point is in a COBOL subprogram.

4. The FORTRAN option specifies that the designated entry point is in a FORTRAN subroutine or function.

5. The NOMAP, NOMAPIN and NOMAPOUT options prevent the manipulation of data aggregates at the interface between PL/I and either COBOL or FORTRAN.

   One or more of these options can appear in the same OPTIONS-attribute specification.

   The arguments to which each option applies can be specified in the optional "argument-list" that follows the option keyword. The format of the "argument-list" is:

   ```
   (ARGi[,ARGj]...)
   ```

   where i,j,... are decimal integers, and the option is to apply to the ith, jth,... items in the argument list of procedure reference.

   Only the arguments to which this option applies are specified in the argument list; they can be specified in any order.

   If there is no argument list for an option, the option is assumed to apply to all the arguments passed on invocation of the entry name.

   An OPTIONS specification should not include the same argument in more than one specified or assumed argument list.

6. The INTER option specifies that any interrupts occurring during the execution of a COBOL subprogram (or a FORTRAN routine) that are not dealt with by the COBOL (or FORTRAN) interrupt handling facilities are dealt with by the PL/I interrupt handling facilities.

Examples:

```
DCL COBOLA OPTIONS(COBOL NOMAP(ARG1)
    NOMAPOUT(ARG3));
     .
     .
     .
CALL COBOLA(X,Y,Z); /* X, Y, Z ARE
    STRUCTURES */
     .
     .
     .;

DCL FORTRNA OPTIONS(FORTRAN INTER);
     .
     .
     .
CALL FORTRNA(L,M); /* L AND M ARE
    ARRAYS */
     .
     .
     .;
```

OUTPUT

See INPUT.

Parameter Attribute

The parameter attribute specifies that a name in an invoked procedure represents an argument passed to that procedure.

General rules:

1. An identifier is explicitly declared with the parameter attribute by its appearance in a parameter list. The identifier must not be subscripted or qualified.

2. A parameter list is specified in a PROCEDURE or ENTRY statement. Parameters in a parameter list correspond, from left-to-right, with arguments in an argument list. The number of arguments and parameters must be the same.

3. Attributes other than parameter can be supplied by a DECLARE statement internal to the procedure. A parameter cannot be declared with any file attributes other than FILE, or with any of the attributes STATIC, AUTOMATIC, BASED, BUILTIN, EXTERNAL, GENERIC, or DEFINED.

4. If a parameter is to be used as a base item for string overlay defining, or is to be specified in record-oriented transmission, the CONNECTED attribute must be declared explicitly.

5. Bounds, lengths, and sizes of simple parameters must be specified either by asterisks or by constants. Only controlled parameters may have the INITIAL attribute.

6. If the attributes of an argument do not match those given for the corresponding parameter, a dummy argument is generated with attributes that agree with those of the parameter. The original argument is then converted and assigned to the dummy argument. The conversion is performed automatically for internal entry constants; but for external entry constants and entry variables, a parameter-descriptor list must be given in an appropriate entry declaration if conversion is required.

   The relationships between arguments and parameters is discussed in chapter 9, "Subroutines and Functions".

Assumptions:

If attributes are not supplied in a DECLARE statement, default attributes are applied, depending on the initial letter of the parameter identifier and on any associated DEFAULT statement. A parameter has the INTERNAL attribute by default.

PICTURE

Abbreviation: PIC

The PICTURE attribute is used to define the internal and external formats of character-string and numeric character data and to specify the editing of data. Numeric character data is data having an arithmetic value but stored internally in character form. Numeric character data must be converted to coded arithmetic before arithmetic operations can be performed.

The picture characters are described in "Picture Specification Character" in Part II.

General format:

```
PICTURE

  { 'character-picture-specification' }
  { 'numeric-picture-specification'   }
```

A "picture specification", either character or numeric, is composed of a string of picture characters enclosed in single quotation marks. An individual picture character may be preceded by a repetition factor, which is a decimal integer constant, n, enclosed in parentheses, to indicate repetition of the character n times. If n is zero, the character is ignored. Picture characters are considered to be grouped into fields, some of which contain subfields.

General rules:

1. The "character-picture-specification" is used to describe a character-string data item.

2. The "numeric-picture-specification" is used to describe a character item that represents either an arithmetic value or a character-string value, depending upon its use.

3. A numeric character data item can have only a decimal base. Its scale and precision are specified by the picture characters. The PICTURE attribute cannot be specified in combination with base, scale, or precision attributes. If the mode of the numeric character data is COMPLEX, however, the COMPLEX attribute must be explicitly stated.

4. Only coded arithmetic data or character string data representing arithmetic constants may be assigned to a numeric picture variable.

POINTER

See OFFSET.

POSITION

See DEFINED.

Precision Attribute

The precision attribute is used to specify the minimum number of significant digits to be maintained for the values of the data items, and to specify the scale factor (the assumed position of the binary or decimal point). The precision attribute applies to both binary and decimal data.

General format:

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

The "number-of-digits" is an unsigned decimal integer constant and "scale-factor" is an optionally signed decimal integer constant. The precision attribute specification is often represented, for brevity, as (p,q), where p represents the "number-of-digits" and q represents the "scale-factor".

General rules:

1. The precision attribute must follow, with no intervening keywords or names, the scale (FIXED or FLOAT), base (DECIMAL or BINARY), or mode (REAL or COMPLEX) at the same factoring level.
2. The number of digits specifies the number of digits to be maintained for data items assigned to the variable. The scale factor specifies the number of fractional digits. No point is actually present; its location is assumed.
3. The scale factor can be specified for fixed-point variables only; the number of digits is specified for both fixed-point and floating-point variables.
4. When the scale is FIXED and no scale factor is specified, it is assumed to be zero; that is, the variable is to represent integers.
5. The scale factor of the variable, or of an intermediate result must be in the range -128 through +127.
6. The scale factor can be negative, and it can be larger than the number of digits. A negative scale factor (-q) always specifies integers, with the point assumed to be located q places to the right of the rightmost actual digit. A positive scale factor (q) that is larger than the number of digits always specifies a fraction, with the point assumed to be located q places to the left of the rightmost actual digit. In either case, intervening zeros are assumed, but they are not stored; only the specified number of digits are actually stored.
7. The precision attribute cannot be specified in combination with the PICTURE attribute.
8. The maximum number of digits allowed is 15 for decimal fixed-point data, 31 for binary fixed-point data, 33 for decimal floating-point data, and 109 for binary floating-point data.

Assumptions:

The standard defaults for precision are as follows:

(5,0) for DECIMAL FIXED
(15,0) for BINARY FIXED
(6) for DECIMAL FLOAT
(21) for BINARY FLOAT

PRINT

The PRINT attribute specifies that the data of the file is ultimately to be printed. The PAGE and LINE options and format items of the PUT statement and the PAGESIZE option of the OPEN statement can be used only with files having the PRINT attribute. These options are described in section J, "Statements".

General format:

PRINT

General rules:

1. The PRINT attribute implies the OUTPUT and STREAM attributes.

2. The PRINT attribute conflicts with the RECORD attribute. (However RECORD files can be associated with the printer; see chapter 12, "Record-Oriented Transmission".)

3. The PRINT attribute causes the initial data byte within each record to be reserved for ANS printer control characters. These control characters are set by the PAGE, SKIP, or LINE format items or options.

Assumption:

If no FILE or STRING specification appears in a PUT statement, the standard output file SYSPRINT is assumed.

REAL

See COMPLEX.

RECORD and STREAM

The RECORD and STREAM attributes specify the kind of data transmission to be used for the file. STREAM indicates that the data of the file is considered to be a continuous stream of data items, in character form, to be assigned from the stream to variables, or from expressions into the stream. RECORD indicates that the file consists of a collection of physically separate records, each of which consists of one or more data items in any form. Each record is transmitted as an entity to or from a variable.

General format:

```
RECORD|STREAM
```

General rules:

1. A file with the STREAM attribute can be specified only in the OPEN, CLOSE, GET, PUT, ON, and assignment statements.

2. A file with the RECORD attribute can be specified only in the OPEN, CLOSE, READ, WRITE, REWRITE, LOCATE, UNLOCK, DELETE, ON, and assignment statements.

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

4. A file with the RECORD attribute cannot have the PRINT attribute.

Assumptions:

Default is STREAM. If a file is implicitly opened by a READ, WRITE, REWRITE, LOCATE, UNLOCK, or DELETE statement, RECORD is assumed.

REDUCIBLE

See IRREDUCIBLE.

RETURNS

The RETURNS attribute is specified in an ENTRY declaration to define the data attributes of a value returned by an entry variable or an external procedure.

General format:

```
RETURNS (attribute...)
```

It is used in the following manner:

```
DECLARE identifier
   [ENTRY (parameter descriptor list)]
   [VARIABLE] RETURNS (attribute...);
```

General rules:

1. The attributes in the parenthesized list following the keyword RETURNS must be separated by blanks (except for attributes, such as precision, that are enclosed in parentheses). They must agree with the attributes specified either explicitly in the RETURNS option of the PROCEDURE or ENTRY statement to which the entry name is prefixed, or by default.

2. The attributes specify the data characteristics of the value returned when the entry is invoked as a function.

3. The only attributes that may be specified are string or arithmetic attributes (including VARYING), or ALIGNED, UNALIGNED, POINTER, OFFSET, AREA, FILE, EVENT, TASK, and LABEL. The OFFSET attribute may include an area name; under the optimizing compiler, this must be a non-defined, unsubscripted, unqualified name, but under the checkout compiler it may be

any area expression other than a function reference. The LABEL attribute may include a list of label constants.

4. If RETURNS attributes are not specified with an explicitly declared entry constant of an external function procedure, default attributes are applied according to the entry constant identifier. Standard default assumptions are given below.

   Note: The value returned by a procedure function reference should agree with the attributes specified by RETURNS; if it does not agree, there is an error since no conversion will be performed.

5. String lengths and area sizes must be specified by decimal integer constants. The returned value has the specified length or size.

Assumptions:

If the RETURNS attribute is not specified for an external entry point, a RETURNS attribute is assumed specifying default attributes; the defaults are either as specified in a DEFAULT statement or are the standard defaults: REAL FIXED BINARY (15,0) if the entry constant begins with any of the letters I through N, otherwise, REAL FLOAT DECIMAL (6).

SEQUENTIAL

See DIRECT.

Size Attribute

See AREA.

STATIC

See AUTOMATIC.

STREAM

See RECORD.

TASK

The TASK attribute describes a variable that may be used as a task name, to test or control the relative priority of a task.

General format:

TASK

General rules:

1. An identifier can be explicitly declared with the TASK attribute in a DECLARE statement, or it can be contextually declared by its appearance in a TASK option of a CALL statement.

2. Task variables can also have the following attributes:

   a. Dimension

   b. Scope (the default is INTERNAL)

   c. Storage class (the default is AUTOMATIC)

   d. DEFINED (task variables may only be defined on other task names)

   e. INITIAL and INITIAL CALL

3. A task expression can be used in the following contexts only:

   a. In the TASK option of a CALL statement

   b. As an argument of the PRIORITY pseudovariable or built-in function

   c. As an argument in a CALL statement or function reference

   d. As a parameter in a PROCEDURE or ENTRY statement

   e. In an ALLOCATE or FREE statement

   f. In an assignment statement

   g. In a RETURN statement

   h. As the control variable of a DO-loop.

   i. In a comparison operation.

4. A task variable may be associated with the priority of a task by including the task name in the TASK option of a CALL statement. A task variable is said to be active if its associated task is active. A task variable must be in an allocated state when it is associated with a task and must not be freed while it is active. An active task variable cannot be associated with another task.

5. A task variable contains a single value, a priority value. This value is a fixed-point binary value of precision (15,0). This value can be tested and adjusted by means of the PRIORITY built-in function and pseudovariable. The built-in function returns the priority of the task argument relative to the priority of the task executing the function. Similarly, the pseudovariable permits assignment, to the named task variable, of a priority relative to the priority of the task executing the assignment.

6. Unless the priority of the task variable is set by means of either the PRIORITY pseudovariable or the PRIORITY option of the CALL statement which invokes the task, its priority will be undefined.

7. Task data cannot be converted to any other data type.

8. Assignment of task data to an inactive task variable is permitted. The value assigned must be the priority of a task derived from a task expression.

9. Two task expressions can be compared using = or a ¬= comparison operator. The variables compare equal if their priorities are equal, otherwise they compare not equal.

TRANSIENT

See DIRECT.

UNALIGNED

See ALIGNED.

UNBUFFERED

See BUFFERED.

UPDATE

See INPUT.

VARIABLE

The VARIABLE attribute can be used with the ENTRY, FILE, or LABEL attributes to establish the name as a variable.

General format:

VARIABLE

VARYING

See BIT.

Section J: Statements

This section presents the PL/I statements in alphabetical order. (The preprocessor statements are alphabetically arranged at the end of this section.) Most statements are accompanied by the following information:

1. Function -- a short description of the meaning and use of the statement
2. General format -- the syntax of the statement
3. Syntax rules -- rules of syntax that are not reflected in the general format
4. General rules -- rules governing the use of the statement and its meaning in a PL/I program

ALLOCATE

Abbreviation: ALLOC

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

General format:

```
ALLOCATE option[,option]...;
```

where "option" has one of two forms:

Option 1

```
[level] identifier[dimension]
[attribute]...
```

Option 2

```
based-variable-identifier
[SET(element-locator-variable)]
[IN(element-area-variable)]
```

Syntax rules:

Syntax rules 1 through 6 apply only to Option 1:

1. "Level" indicates a level number. The first identifier appearing after the keyword ALLOCATE must be a level 1 identifier.
2. Each identifier must represent data of the controlled storage class or be an element of a controlled major structure.
3. "Dimension" indicates a dimension attribute. "Attribute" indicates an AREA, BIT, CHARACTER, or INITIAL attribute.
4. A dimension attribute, if present, must specify the same number of dimensions as that declared for the associated identifier.
5. The attribute BIT may appear only with a BIT identifier; CHARACTER may appear only with a CHARACTER identifier; AREA may only appear with an area identifier.
6. A structure element name, other than the major structure name, may appear only if the relative structuring of the entire major structure containing the element appears as in the DECLARE statement for that structure. In this case, dimension attributes must be specified for all identifers that are declared with the dimension attribute.

Syntax rules 7 and 8 apply only to Option 2:

7. The based variable appearing in the ALLOCATE statement may be an element variable, an array, or a major structure. When it is a major structure, only the major structure name is specified.
8. The SET option, if present, may appear preceding or following the IN option.

General rules:

Rules 1 through 6 apply only to Option 1:

1. When Option 1 is used, an ALLOCATE statement for an identifier for which storage was allocated and not freed causes storage for the identifier to be "pushed down" or stacked. This pushing down creates a new generation of data for the identifier. When storage for this identifier is freed, using the FREE statement, storage is "popped up" or removed from the stack.
2. Bounds for arrays, lengths of strings, and sizes of areas are fixed at the execution of an ALLOCATE statement.
   a. If a bound, length, or size is explicitly specified in an ALLOCATE statement, it overrides

that given in the DECLARE statement.

b. If a bound, length, or size is specified by an asterisk in an ALLOCATE statement, the bound, length or size is taken from the current generation. If no generation of the variable exists, the bound, length, or size is undefined and the program is in error.

c. Either the ALLOCATE statement or a DECLARE or DEFAULT statement must specify any necessary dimension, size, or length attributes for an identifier. Any expression taken from a DECLARE or DEFAULT statement is evaluated at the point of allocation using the conditions enabled at the ALLOCATE statement, although names in the expression are interpreted in the environment of the DECLARE or DEFAULT statement.

d. If, in either an ALLOCATE or a DECLARE statement, bounds, lengths, or sizes are specified by expressions that contain references to the variable being allocated, the expressions are evaluated using the value of the most recent generation of the variable.

3. Upon allocation of an identifier, initial values are assigned to it if the identifier has an INITIAL attribute in either the ALLOCATE statement or DECLARE statement. Expressions or a CALL option in the INITIAL attribute are executed at the point of allocation, using the conditions enabled at the ALLOCATE statement, although the names are interpreted in the environment of the declaration. If an INITIAL attribute appears in both DECLARE and ALLOCATE statements, the INITIAL attribute in the ALLOCATE statement is used. If initialization involves reference to the variable being allocated, the reference will be to the new generation of the variable.

4. To determine whether or not storage has been allocated for an identifier and how many generations exist, the built-in function ALLOCATION may be used.

5. A parameter that is declared CONTROLLED may be specified in an ALLOCATE statement.

6. Any evaluations performed at the time the ALLOCATE statement is executed (e.g., evaluation of expressions in an INITIAL attribute) must not be interdependent.

Rules 7 through 12 apply only to Option 2:

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

8. The allocation of a based variable involves the based variable to be allocated, a locator variable to identify the new generation, and an area if the generation is to be allocated in an area. If no SET option is specified, a SET option is assumed to specify the locator variable given in the BASED attribute of the based variable declaration; it is an error, in such a case, if this BASED attribute does not specify a locator variable. If the SET option specifies an offset variable and no IN option is present then an IN option is assumed to specify the area given in the OFFSET attribute of the offset variable declaration; in such a case, it is an error if this OFFSET attribute does not specify an area variable.

9. If the SET option specifies an offset variable, the locator value identifying the new generation is assigned to the offset variable; the IN option must be present, or be assumed, and it must specify either the same area as that specified in the OFFSET attribute of the offset variable declaration, or an area contained in or containing that area.

10. If no IN option is present and none is assumed, the new generation is allocated in storage associated with the task which executes the ALLOCATE statement. The SET option in this case must specify a pointer variable.

11. If the IN option appears in, or is assumed for, the ALLOCATE statement, storage will be allocated in the named area, for the based variable. If sufficient storage does not exist within this area, the AREA condition will be raised.

12. The amount of storage allocated for a based variable depends on its attributes, and on its dimensions, length, or size specifications if these are applicable at the time of allocation.

These attributes are determined from the declaration of the based variable, and additional attributes may not be specified in the ALLOCATE statement. A based structure may contain adjustable array bounds or string lengths or area sizes (see "REFER Option", in "Storage Control" in Part I). Note that the asterisk notation for bounds, length, or size is not permitted for based variables.

Assignment Statement

The assignment statement is used to evaluate an expression and to assign its value to one or more target variables; the target variables may be element, array, or structure variables. The target variables can be pseudovariables.

General formats:

The assignment statement has three general format options. They are given in figure J.1.

Syntax rules:

1. In Option 2, each target variable must be an array. If the right-hand side contains arrays of structures, then all target variables must be arrays of structures. The BY NAME option may be given only when the right-hand side contains at least one structure.
2. In Option 3, each target variable must be a structure.

General rules:

1. Aggregate assignments (Options 2 and 3) are expanded into a series of element assignments according to rules 5 through 8.
2. An element assignment is performed as follows:
   a. Subscripts and locator qualifications of the target variables, and the second and third arguments of SUBSTR pseudovariable references, are evaluated first. (The order of evaluation of subscripts and qualifiers is undefined).
   b. The expression on the right-hand side is then evaluated.
   c. For each target variable (in left to right order), the expression is converted to the characteristics of the target variable according to rules for data conversion (except that whenever a conversion of arithmetic base is involved, the value is converted directly to the precision of the target variable). The converted value is then assigned to the target variable.
   d. The element variable can be a variable with the PICTURE attribute. The rules for assignments to picture targets are described in section D, "Picture Specification Characters".
3. The following rules apply to string element assignment:

```
Option 1 (Element Assignment)

  {element-variable}  [,element-variable]
  {pseudovariable }   [,pseudovariable ]   ... =  element-expression;

Option 2 (Array Assignment)

  {array-variable }  [,array-variable ]
  {pseudovariable }  [,pseudovariable ]   ... = {structure-expression [,BY NAME]}
                                                {array-expression [,BY NAME]     };
                                                {element-expression              }

Option 3 (Structure Assignment)

  {structure-variable}[,structure-variable ]
  {pseudovariable    }[,pseudovariable     ]  ... = {structure-expression [,BY NAME]}
                                                    {element-expression              };
```

Figure J.1. General formats of the assignment statement

a. The assignment is performed from left to right, starting with the leftmost position.

b. If the target variable is a fixed-length string, the expression value is truncated on the right if it is too long (raising the STRINGSIZE condition, if enabled) or padded on the right (with blanks for character string, zeros for bit strings) if the value is too short. (Note that a string pseudovariable is considered to be a fixed-length string.) The resulting value is assigned to the target.

c. If the target is a VARYING string and the value of the expression is longer than the maximum length declared for the variable, the value is truncated on the right (raising the STRINGSIZE condition, if enabled). The target string obtains a current length equal to its maximum length. If the value of the expression is not longer than the maximum length, the value is assigned; the target string obtains a current length equal to the length of the value.

4. The following rules apply to other element assignments:

   a. If the target is an area variable, the expression must be an area variable or function. The AREA condition will be raised by this assignment if the size of the target area is insufficient for the current extent of the area being assigned.

   b. If the target is a pointer variable, the expression can only be a pointer (or offset) variable or a pointer (or offset) function reference. If the expression is of offset type, its value is converted to pointer.

   c. If the target is an offset variable, the expression can only be an offset (or pointer) variable or an offset (or pointer) function reference. If the expression is of pointer type, its value is converted to offset.

   d. If the target is a label variable, the expression can only be a label variable or label constant. Environmental information (i.e., information that identifies the invocation of the block) is always assigned to the label variable.

   e. If the target is an event variable, the expression can only be an event variable. The assignment is uninterruptable, and it involves both the completion and status values. An event variable does not become active when it has an active event variable assigned to it. It is an error to assign to an active event variable.

   f. If the target is a STATUS pseudovariable, a value can be assigned whether or not the event variable is active. It is an error to assign to a COMPLETION pseudovariable if the named event variable is active.

   g. If the target is an entry variable, the expression can only be an entry expression.

   h. If the target is a file name variable, the expression can only be a file expression.

   i. If the target is a task variable, the expression can only be a task variable or a task function reference. The task variable specified must be inactive. The assignment involves the priority of the task variable or task function reference.

5. The first target variable in an aggregate assignment is known as the master variable. If the master variable is an array, then an array expansion (Rule 6) is performed; otherwise, a structure expansion (Rules 7 and 8) is performed. The CHECK condition for assignment to a target variable is raised (when suitably enabled) after assignment to each element. In the case of BY NAME assignment, the CHECK condition for the target variable is raised regardless of whether any value is assigned to an item. The label prefix of the original statement is applied to a null statement preceding the other generated statements.

6. In Option 2, all array operands must have the same number of dimensions and identical bounds. The array assignment is expanded into a loop as follows.

```
LABEL: DO j1 = LBOUND(master-variable,1) TO
           HBOUND(master-variable,1);

       DO j2 = LBOUND(master-variable,2) TO
              HBOUND(master-variable,2);
         .
         .
         .

       DO jn = LBOUND(master-variable,n) TO
              HBOUND(master-variable,n);

       generated assignment statement

       END LABEL;
```

In this expansion, n is the number of dimensions of the master variable that are to participate in the assignment. In the generated assignment statement, all array operands are fully subscripted, using (from left to right) the dummy variables j1 to jn. If an array operand appears with no subscripts, it will only have the subscripts j1 to jn; if cross-section notation is used, the asterisks are replaced by j1 to jn. If the original assignment statement (which may have been generated by Rule 7 or Rule 8) has a condition prefix, the generated assignment statement is given this condition prefix. If the original assignment statement (which may have been generated by Rule 8) has a BY NAME option, the generated assignment statement is given a BY NAME option. If the generated assignment statement is a structure assignment, it is expanded as given below.

7. In Option 3, where the BY NAME option is not specified, the following rules apply:

   a. None of the operands can be arrays, although they may be structures that contain arrays.

   b. All of the structure operands must have the same number, k, of immediately contained items.

   c. The assignment statement (which may have been generated by Rule 6) is replaced by k generated assignment statements. The ith generated assignment statement is derived from the original assignment statement by replacing each structure operand by its ith contained item; such generated assignment statements may require further expansion according to Rule 6 or Rule 7. All generated assignment statements are given the condition prefix of the original statement.

8. In Option 3, where the BY NAME option is given, the structure assignment, which may have been generated by Rule 6, is expanded according to steps a through d below. None of the operands can be arrays.

   a. The first item immediately contained in the master variable is considered.

   b. If each structure operand and target variable has an immediately contained item with the same identifier, an assignment statement is generated as follows: the statement is derived by replacing each structure operand and target variable with its immediately contained item that has this identifier. If any structure contains no such identifier, no statement is generated. If the generated assignment is a structure or array-of-structures assignment, BY NAME is appended. The first generated assignment is given the label prefix of the original assignment statement; all generated assignment statements are given the condition prefix of the original assignment statement.

   c. Step b is repeated for each of the items immediately contained in the master variable. The assignments are generated in the order of the items contained in the master variable.

   d. Steps a through c may generate further array and structure assignments. These are expanded according to Rules 6 through 8.

BEGIN

The BEGIN statement heads and identifies a begin block.

General format:

```
BEGIN[ORDER|REORDER];
```

Syntax rules:

1. A label of a BEGIN statement may be subscripted, but such a label cannot appear in an END statement.

General rules:

1. A BEGIN statement is used in conjunction with an END statement to delimit a begin block. A complete discussion of begin blocks can be found in chapter 6, "Program Organisation."

2. ORDER and REORDER are optimization options for use by the optimizing compiler. If they are included in a program processed by the checkout compiler, they are checked for syntax errors and then ignored. Their presence in such a program is not an error.

3. ORDER and REORDER specify the extent to which the block is to be optimized. In general, ORDER permits optimization to the degree such that the latest values of variables set in a block are guaranteed available in a computational on-unit entered at any point during execution of the block. REORDER permits a greater degree of optimization; with REORDER the latest values of variables set in the block are not guaranteed available in an on-unit entered during execution of the block. If neither is specified, ORDER is assumed, but REORDER is inherited by all contained blocks unless they explicitly specify ORDER.

CALL

The CALL statement invokes a procedure and causes control to be transferred to a specified entry point of the procedure.

General format:

```
CALL {entry-expression|generic-name|
                  built-in name}

     [(argument [,argument] . . .)]

     [TASK [(element-task-name)]]
     [EVENT (element-event-name)]
     [PRIORITY (expression)];
```

Syntax rules:

1. The entry expression, generic name, or built-in name represents the entry point of the subroutine invoked.

2. The TASK, EVENT, and PRIORITY options can appear in any order.

General rules:

1. The TASK, EVENT, and PRIORITY options, when used alone or in any combination, specify that the invoked and invoking procedures are to be executed asynchronously. Note that if either the EVENT option or the PRIORITY option, or both, are used without the TASK option, the created task will have no name. (See chapter 17, "Multitaksing".)

2. When the TASK option is used, the task name, if given, is associated with the task created by the CALL. Reference to this name enables the priority of the task to be controlled at some other point by the use of the PRIORITY pseudovariable and built-in function.

3. When the EVENT option is used, the event name is associated with the completion of the task created by the CALL statement. Another task can then wait for completion of this created task by specifying the event name in a WAIT statement.

   Upon execution of the CALL statement, the event variable is made active, and the completion value is set to '0'B and the status value to 0. Upon termination of the created task, the completion value is set to '1'B and, unless the task has been terminated by a RETURN or END statement, the status is set to 1 if still zero.

4. If the PRIORITY option is used, the expression in the PRIORITY option is evaluated to an integer m, of an implementation-defined precision (15,0). The priority of the named task is then made m relative to the task in which the CALL is executed.

   If a CALL statement with the EVENT or TASK option does not have the PRIORITY option, the priority of the invoked task is made equal to that of the task variable in the TASK option, if there is a task variable, or else made equal to the priority of the invoking task. The programmer must specify a priority if he uses a task variable, (by means of either a PRIORITY option on the CALL statement or the PRIORITY built-in function prior to the CALL statement), otherwise the task will be of undefined priority.

5. Expressions in these options, as well as any argument expressions, are evaluated in the task in which the call is executed. This includes execution of any on-units entered as the result of the evaluations.

6. The environment of the invoked procedure is established after evaluation of the expressions named in Rule 5, and before the procedure is invoked.

7. A CALL statement must not be used to invoke a procedure if control is returned to the invoking procedure by means of a RETURN(expression) statement.

8. See chapter 9, "Subroutines and Functions" for detailed descriptions of the interaction of arguments with the parameters that represent these arguments in the invoked procedure.

9. If the procedure invoked by the CALL statement has been specified in a FETCH or RELEASE statement, and if it is not present in main storage, the CALL statement initiates dynamic loading of the procedure from auxiliary storage. The execution of the invocation is delayed until the procedure has been loaded.

   In this case, the entry expression must be an entry constant, and it must be equivalent to both the name by which the procedure is known in external storage and a point through which the procedure may be entered; and the same constant must have appeared in a FETCH or RELEASE statement compiled at the same time as the CALL statement. A main procedure may not be dynamically loaded. A fetched procedure may not fetch a further procedure.

CHECK

The CHECK statement causes the CHECK condition to be dynamically enabled for specified or assumed names.

The PL/I checkout compiler implements the CHECK statement in this sense, but the PL/I optimizing compiler implements this statement by checking the syntax and then ignoring it.

General format:

```
CHECK[(name-list)];
```

Syntax rules:

1. The optional "name-list" is one or more names separated by commas.

2. A name must be one of the following:

   a. An unsubscripted variable representing element, an array or a structure of any data type. The variable must not be iSUB-defined or locator qualified.

   b. A label constant.

   c. An entry constant.

3. If a name-list is specified, the CHECK statement applies to those names only. The names must be known in the block in which the CHECK statement is executed.

   If no names are specified, the CHECK statement is assumed to apply to every name known in the external procedure that contains the CHECK statement, whether or not these names were known at the time the CHECK statement was executed. These names may be known in other, separately compiled, external procedures.

General Rules:

1. Execution of a CHECK statement has the effect of enabling a CHECK condition-prefix, or of modifying an existing CHECK condition-prefix, for every statement that is executed after the execution of the CHECK statement.

   The prefixes thus derived operate in the same way as ordinary prefixes. If the condition is raised, any CHECK on-unit established is executed. If there is no on-unit, the standard system action for the CHECK condition is taken. The situations in which the CHECK condition is raised are described in "CHECK Condition", in section H, "On-Conditions".

2. The variable can be of any storage class, or DEFINED, or a parameter.

3. If the name of a structure or an array of structures appears in the name list, this is expanded into a list of the names of all the elements in the structure or array of structures, in the order in which they were declared. This expanded list appears in the name list for the derived prefixes.

4. The information provided by standard system action for the CHECK condition for a particular name is:

   a. The statement number of the statement in which the references to the name occurs.

b. Information similar to that put out by a PUT DATA statement for the particular type of variable.

If the name is the name of an array, the information includes the subscripted name of the element to which a new value is being assigned.

5. If the name is an entry name, this can be specified as an entry constant or an entry variable, whether it appears in a function reference, a CALL statement, or an INITIAL CALL attribute. If the reference is to an entry variable, the information provided by standard system action includes the name of the entry constant associated with the particular invocation of the entry variable.

6. A CHECK statement remains effective until:

   a. The program terminates, or

   b. An appropriate NOCHECK statement is executed.

CLOSE

The CLOSE statement dissociates the named file from the data set with which it was associated by opening in the current task.

General format:

```
CLOSE FILE(file-expr )
    [ENVIRONMENT({LEAVE|REREAD})]
    [,FILE(file-expr )
    [ENVIRONMENT({LEAVE|REREAD})]]...;
```

General rules:

1. The FILE(file-expression) option specifies which file is to be closed. It must appear once. Several files can be closed by one CLOSE statement. There must be a FILE option for each one.

2. A closed file can be reopened.

3. Closing an unopened file, or an already closed file, has no effect.

4. The CLOSE statement cannot be used to close a file in a task different from the one that opened the file. If a file is not closed by a CLOSE statement, it is automatically closed at the completion of the task in which it was opened.

6. All input/output events associated with the file that have a status value of zero when the file is closed are set complete, with a status value of 1.

7. A CLOSE statement unlocks all records in the file previously locked in the task in which the CLOSE appears.

8. The ENVIRONMENT attribute with either the REREAD or LEAVE options can be given.

DECLARE

Abbreviation: DCL

The DECLARE statement is the principal method for explicitly declaring attributes of names.

General format:

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

Syntax rules:

1. Any number of identifiers may be declared in one DECLARE statement.

2. "Level" is a nonzero unsigned decimal integer constant. If a level number is not specified, level 1 is assumed for all element and array variables. Level 1 must be specified for all major structure names. A blank space must separate a level number from the identifier following it.

3. Attributes specified in DECLARE statements are separated by blanks. Except for the dimension, length, and precision attribute specifications, they may appear in any order. The dimension attribute specification must immediately follow the array name; the length and precision attribute specifications must follow one of their associated attributes. A comma must follow the last attribute specification for a particular name (or the name itself if no attributes are specified with it), unless it is the last name in the DECLARE statement, in which case the semicolon is used.

4. "SYSTEM" specifies that the standard default attributes are to be applied to the associated identifier;

attributes are not taken from DEFAULT statements. "SYSTEM" may appear before, after, or between the other attributes.

Factoring of Attributes

Attributes common to several names can be factored in a declaration to eliminate repeated specification of the same attribute for many identifiers. Factoring is achieved by enclosing the names in parentheses, and following this by the set of attributes which apply. All factored attributes must apply to all of the names. No factored attribute can be overridden for any of the names, but any name within the list may be given other attributes so long as there is no conflict with the factored attributes. Factoring of attributes is permitted only in the DECLARE and DEFAULT statement, but not within an ENTRY attribute declaration. The dimension attribute may be factored. The precision and length attributes can be factored only in conjunction with an associated keyword attribute. Factoring can be nested as shown in the fourth example below.

Names within the parenthesized list are separated by commas.

Note: Structure level numbers can also be factored, but a factored level number must precede the parenthesized list.

```
DECLARE (A,B,C,D) BINARY FIXED (31);

DECLARE (E DECIMAL(6,5),
         F CHARACTER(10)) STATIC;

DECLARE 1 A, 2(B,C,D) (3,2) BINARY
          FIXED (15), ...;

DECLARE ((A,B) FIXED(10), C FLOAT(5))
          EXTERNAL;
```

General rules:

1. A particular level 1 identifier can be specified in only one DECLARE statement within a particular block. All attributes given explicitly for that identifier must be declared together in that DECLARE statement. (Note, however, that identifiers having the FILE attribute may be given attributes in an OPEN statement as well. See "The OPEN Statement" in this section and chapter 10, "Input and Output" for further information.)

2. Attributes of external names, in separate blocks and compilations, must be consistent (except that an INITIAL attribute given in one declaration need not be repeated).

3. Labels may be prefixed to DECLARE statements. However, a branch to such a label is treated as a branch to a null statement. Condition prefixes cannot be attached to a DECLARE statement.

DEFAULT

Abbreviation: DFT

The DEFAULT statement allows the programmer to specify the default attributes to be applied to designated identifiers that require implicit declaration of some or all of their attributes. The DEFAULT statement can specify default attributes for:

1. Explicitly declared identifiers

2. Contextually declared identifiers

3. Attributes to be included in parameter descriptors

4. Implicitly declared identifiers and values returned from function procedures

General format: See figure J.2.

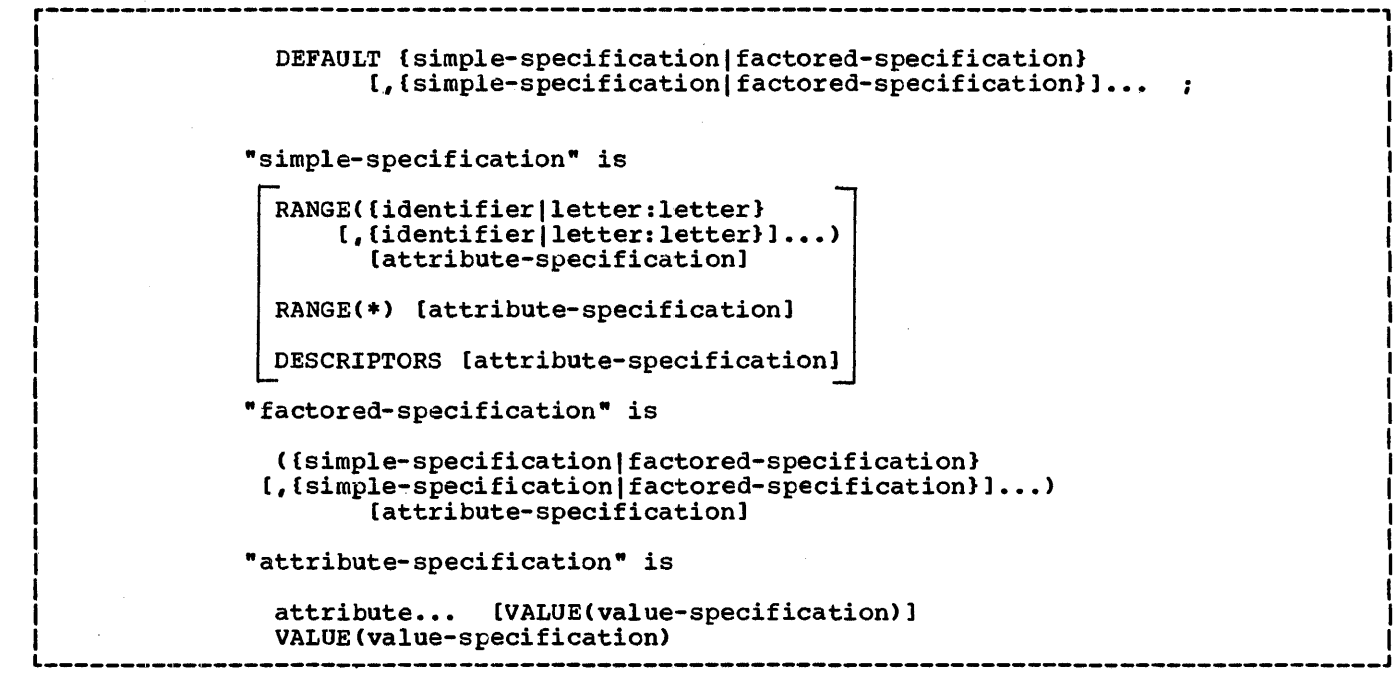


Figure J.2. General formats of the DEFAULT statement

General Rules:

1. Any attributes not applied according to DEFAULT statement rules for any partially complete explicit or contextual declarations, and for implicit declarations, are supplied according to standard default rules.

2. The scope of a DEFAULT statement is the block in which it occurs, and all blocks within that block which neither include another DEFAULT statement with the same range, nor are contained in a block having a DEFAULT statement with the same range.

   It is possible for a containing block to have a DEFAULT statement with a range that is partly covered by the range of a DEFAULT statement in a contained block. In such a case, the range of the DEFAULT statement in the containing block is reduced by the range of the DEFAULT statement in the contained block.

   For example:

```
 P:  PROCEDURE;
L1:  DEFAULT RANGE (XY) FIXED;
       .
       .
       .

 Q:  BEGIN;
L2:  DEFAULT RANGE (XYZ) FLOAT;
    END P;
```

   The range and scope of DEFAULT statement L1 is all identifiers in the procedure P beginning with the characters XY, together with all identifiers in begin block Q beginning with the characters XY, except for those beginning with the characters XYZ. The range and scope of the DEFAULT statement L2 is all the identifiers in begin block Q beginning with characters XYZ.

3. VALUE (value-specification) may appear anywhere within an attribute specification, except before an array dimension attribute.

4. VALUE establishes any default rules for a string length, area size, and precision. The base and scale attributes in the value specification must be present to identify a particular precision specification with a particular attribute.

5. A value specification is a list of one or more of the following in any order;

   a. AREA (size)

   b. BIT (length)

   c. CHARACTER (length)

   d. {base-attribute scale-attribute|
      scale-attribute base-attribute}
      (precision[,scale factor])

The base and scale attributes may be factored, if, when expanded, the above format is used.

The size of AREA data, or length of BIT or CHARACTER data, can be an expression or a decimal integer constant, or can be specified as an asterisk.

Example:

```
DEFAULT RANGE(A:C)
        VALUE (FIXED DECIMAL(10),
               FLOAT DECIMAL(14),
               AREA(2000));
DECLARE B FIXED DECIMAL, C FLOAT
          DECIMAL,
        A AREA;
```

These statements are equivalent to:

```
DECLARE B FIXED DECIMAL(10), C FLOAT
      DECIMAL(14), A AREA(2000);
```

6. RANGE designates the particular identifiers to which the attributes specified in a DEFAULT statement apply.

   a. The form of RANGE(identifier) is used when the default rules are to apply to those identifiers which contain the letters indicated in "identifier" as their first and subsequent letters. For example:

      ```
      RANGE (ABC)
      ```

      applies to these identifiers:

      ```
      ABC
      ABCD
      ABCD....etc.
      ```

      but not to:

      ```
      ABD
      ACB
      AB
      A
      ```

      hence a single letter in the RANGE specification applies to all identifiers which start with that letter.

   b. An alternative specification of RANGE is the form "letter:letter" This is used to specify that identifiers with initial letters which either correspond to the two letters specified, or to any letters between the two in alphabetic sequence, are subject to the default attributes specified for a particular range. The letters given in the specification must be in increasing alphabetic order, for example:

      ```
      RANGE(A:G,I:M,T:Z)
      ```

   c. RANGE(*) specifies all identifiers in the scope of the DEFAULT statement.

7. DESCRIPTORS specifies that the associated attributes are to be included in any parameter descriptors in a parameter descriptor list of an explicit entry declaration, provided that the inclusion of any such attributes is not prohibited by the presence of alternative attributes of the same class and provided that at least one attribute is already present. From the second provision it follows that the DESCRIPTORS default attributes are not applied to parameters having null descriptors, that is, parameters whose attributes match those of the corresponding argument.

8. Factored-default-specification: this form is used as follows:

   ```
   DEFAULT (RANGE(A)FIXED, RANGE(B)
            FLOAT)BINARY;
   ```

   This statement establishes default attributes FIXED BINARY for implicitly declared identifiers with the initial letter A, and FLOAT BINARY for those with the initial letter B.

9. Labels may be prefixed to DEFAULT statements. However, a branch to such a label is treated as a branch to a null statement. Condition prefixes cannot be attached to a DEFAULT statement.

Rules for Attributes in a DEFAULT Statement:

1. The file attributes (excluding FILE), and the attributes ENTRY, ENVIRONMENT, RETURNS, LIKE, and VARIABLE are not permitted in an attribute specification. If FILE is used, it implies a scope attribute of INTERNAL and the attribute VARIABLE.

2. It is not possible to use the DEFAULT statement to create a structure. Structure elements are given default attributes according to the identifier of the element, not the qualified structure element name.

3. The following attributes are allowed in an attribute specification only if

the restriction given below for each is observed.

AREA - without a size specification
BIT - without a string length specification
CHARACTER - without a string length specification
LABEL - without a label list
Arithmetic base and scale attributes - without precision specifications

4. The CONTROLLED attribute cannot be applied to a parameter or parameter descriptor. For any identifier that is a parameter name, a specification of CONTROLLED as a default attribue will be ignored, and the attribute will be ignored if it appears in a DESCRIPTORS attribute specification.

5. The dimensions of an array are permitted as attributes, but only as the first item in an attribute specification. The bounds may be specified as a arithmetic constant or an expression involving variables. For example:

```
DEFAULT RANGE (J) (5);
DEFAULT RANGE (J) (5,5) FIXED;
```

but not

```
DEFAULT RANGE (J) FIXED (5);
```

6. The INITIAL attribute may be specified.

DELAY

The DELAY statement causes the execution of a task to be suspended for a specified period of time.

General format:

```
DELAY (element-expression);
```

General rules:

1. Execution of the DELAY statement causes the element expression to be evaluated and converted to an integer n; execution is then suspended for n milliseconds. The value is recorded to 1/50th or 1/60th second, depending on whether the frequency of the electrical supply to the machine is 50 or 60 hertz (cycles per second).

2. If no timing facility is available, DELAY acts as a null statement.

Example:

```
DELAY (20);
```

This statement causes execution of the task to be suspended for 20 milliseconds or 17 milliseconds (approximately), depending on whether the supply is 50 or 60 hertz.

DELETE

The DELETE statement deletes a record from an UPDATE file.

General format:

```
DELETE FILE (file-expr)
       [KEY(expression)]
       [EVENT(event-variable)];
```

General rules:

1. The options may appear in any order.

2. The FILE option specifies the UPDATE file; it must be specified.

3. The KEY option must be specified if the file is a DIRECT UPDATE file. It can be specified for a SEQUENTIAL UPDATE file with INDEXED organization. The expression is converted to a character string and determines which record is to be deleted.

4. If the file is a SEQUENTIAL UPDATE file, the record to be deleted is the last record that was read; the data set organization must be INDEXED.

5. The EVENT option allows processing to continue while a record is being deleted.

   When control reaches a DELETE statement containing this option, the "event variable" is made active (that is, it cannot be associated with another event) and is given the completion value '0'B, provided that the UNDEFINEDFILE condition is not raised by an implicit file opening (see "Note" below). The event variable remains active and retains its '0'B completion value until control reaches a WAIT statement specifying that event variable. At this time, either of the following can occur:

   a. If the DELETE statement has been executed successfully and neither of the conditions TRANSMIT or KEY has been raised as a result of the DELETE, the event variable is set

complete, given the completion value '1'B, and the event variable is made inactive, that is, can be associated with another event.

b. If the DELETE statement has resulted in the raising of TRANSMIT or KEY, the interrupt for each of these conditions does not occur until the WAIT is encountered. At such time, the corresponding on-units (if any) are entered in the order in which the conditions were raised. After a return from the final on-unit, or if one of the on-units is terminated by a GO TO statement, the event variable is given the completion value '1'B and is made inactive.

Note: If the DELETE statement causes an implicit file opening that results in the raising of UNDEFINEDFILE, the on-unit associated with this condition is entered immediately and the event variable remains unchanged; that is, the event variable remains inactive and retains the same value it had when the DELETE was encountered. If the on-unit does not correct the condition, then, upon normal return from the on-unit, the ERROR condition is raised; if the condition is corrected in the on-unit, that is, if the file is opened successfully, then, upon normal return from the on-unit, the event variable is set to '0'B, it is made active, and execution of the DELETE statement continues.

6. The DELETE statement unlocks a record only if that record had been locked in the same task in which the DELETE appears.

7. The DELETE statement can cause implicit opening of a file.

Example:

```
DELETE FILE(ALPHA) KEY (DKEY);
```

This statement causes the record identified by DKEY to be deleted from the data set associated with the file ALPHA. If the record was previously locked in the same task, it is unlocked.

DISPLAY

The DISPLAY statement causes a message to be displayed to the machine operator. A response may be requested.

General format:

Option 1.

```
DISPLAY (element-expression);
```

Option 2.

```
DISPLAY (element-expression)
REPLY
(character-variable|pseudovariable)
[EVENT (event-variable)];
```

General rules:

1. Execution of the DISPLAY statement causes the element expression to be evaluated and, where necessary, converted to a varying character string of implementation-defined maximum length (72 characters). This character string is the message to be displayed.

2. In Option 2, the character variable or pseudovariable receives a string that is a message to be supplied by the operator. The STRING pseudovariable must not be used. The message cannot exceed 72 characters.

3. In Option 2, if the EVENT option is not specified, execution of the program is suspended until the operator's message is received. In option 1, execution continues uninterrupted.

4. If the EVENT (event-variable) option is given, execution will not wait for the reply to be completed before continuing with subsequent statements. The completion part of the event variable will be given the value '0'B until the reply is completed, when it will be given the value '1'B. The reply is considered complete only after the execution of a WAIT statement naming the event. Another DISPLAY statement must not be executed until the previous reply is complete.

Example:

```
DISPLAY ('END OF JOB');
```

This statement causes the message "END OF JOB" to be displayed.

DO

The DO statement heads a DO-group and can also be used to specify repetitive execution of the statements within the group.

General formats:

The three format types for the DO statement are shown in Figure J-3.

Syntax rules:

1. In all three types, the DO statement is used in conjunction with the END statement to delimit a DO-group. Only Type 1 does not provide for the repetitive execution of the statements within the group.
2. In Type 3, the variable or pseudovariable must represent a single element; "variable" may be subscripted and/or qualified. Real arithmetic variables are generally used, but all variable types are allowed, provided that the expansions given in the general rules below result in valid PL/I programs. Note that if "variable" is a program-control variable, the BY and TO options cannot be used in "specification".
3. Each expression in a specification must be an element expression.
4. If "BY expression3" is omitted from a "specification," and if "TO expression2" is included, "expression3" is assumed to be 1.
5. If "TO expression2" is omitted from a "specification," repetitive execution continues until it is terminated by the WHILE clause or some statement causes control to pass out of the group.
6. If both "TO expression2" and "BY expression3" are omitted from a specification, it implies a single execution of the group, with the control variable having the value of "expression1". If "WHILE expression4" is included, this single execution will not take place unless "expression4" is true.

The following pseudovariables may not be used under the optimizing compiler: COMPLETION, COMPLEX, PRIORITY, STRING.

General rules:

1. In Type 1, the DO statement only delimits the start of a DO-group; it does not provide for repetitive execution.
2. In Type 2, the DO statement delimits the start of a DO-group and provides for repetitive execution as defined by the following:

```
LABEL:  DO WHILE (expression);
        statement-1
        .
        .
        .
        statement-n
        END;
NEXT:   statement /*STATEMENT
          FOLLOWING THE DO GROUP*/
```

The above is exactly equivalent to the following expansion:

```
LABEL:  IF (expression) THEN; ELSE
          GO TO NEXT;
        statement-1
        .
        .
        .
        statement-n
        GO TO LABEL;
NEXT:   statement /*STATEMENT
          FOLLOWING THE DO GROUP*/
```

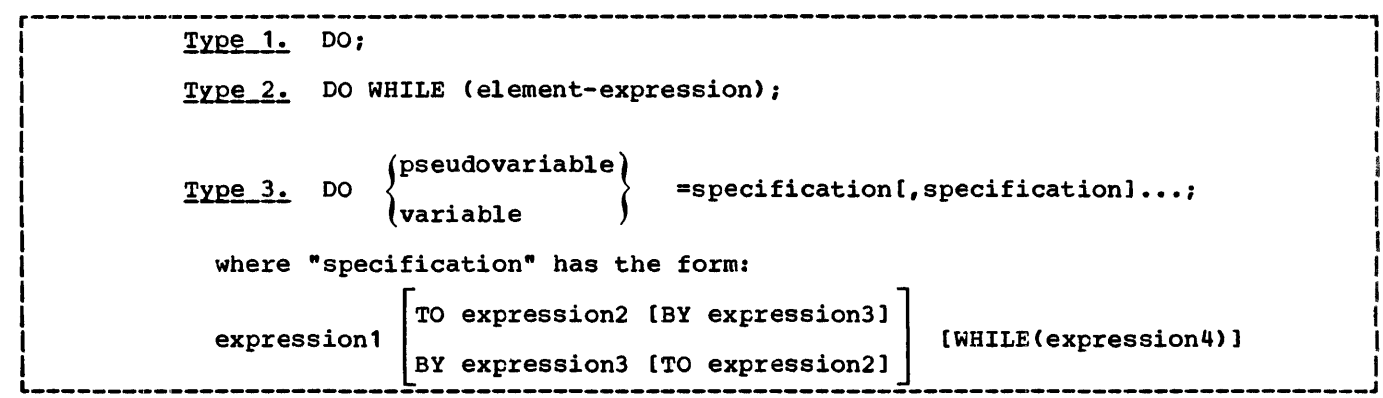


Figure J.3. General format of the DO statement

3. In Type 3, the DO statement delimits the start of a DO-group and provides for controlled repetitive execution as defined by the following:

```
LABEL:  DO-variable=
          expression1
          TO expression2
            BY expression3
          WHILE (expression4);
          statement-1
          .
          .
          .
          statement-m
LABEL1: END;
NEXT:   statement
```

For a variable that is not a pseudovariable, this is exactly equivalent to the following expansion:

```
LABEL:  p=ADDR(variable);
        e1=expression1;
        e2=expression2;
        e3=expression3;
        v=e1;
LABEL2: IF (e3>=0)&(v>e2)|
           (e3<0)&(v<e2)
           THEN GO TO NEXT;
        IF (expression4) THEN;
          ELSE GO TO NEXT;
        statement-1
        .
        .
        .
        statement-m
LABEL1: v=v+e3;
        GO TO LABEL2;
NEXT:   statement
```

In the above expansion, p is a compiler-created pointer; v is a compiler-created based-variable based on p and with the same attributes as "variable". "e1," "e2," and "e3" are compiler-created variables having the attributes of "expression1," "expression2," and "expression3," respectively. Note that the generation of the control variable is established once outside the loop, immediately before the initial value expression (expression1) is evaluated.

Additional rules for the above expansion follow:

a. The above expansion only shows the result of one "specification." If the DO statement contains more than one "specification," the statement labeled NEXT is the first statement in the expansion for the next "specification." The second expansion is analogous to the first expansion in every respect. Thus, if a second "specification" appeared in the DO statement, the second expansion would look like this:

```
NEXT    e5=expression5;
        .
        .
        .
        v=e5;
LABEL3: IF ... THEN GO TO NEXT1;
        IF (expression8) THEN;
          ELSE GO TO NEXT1;
        statement-1
        .
        .
        .
        statement-m
LABEL4: v=v+e7;
        GO TO LABEL3;
NEXT1:  statement
```

Note that statements 1 through m are not actually duplicated in the program.

b. If the WHILE clause is omitted, the IF statement immediately preceding statement-1 in the expansion is omitted.

c. If "TO expression2" is omitted, the statement "e2=expression2" and the IF statement identified by LABEL2 are omitted.

d. If both "TO expression2" and "BY expression3" are omitted, all statements involving e2 and e3, as well as the statement GO TO LABEL2, are omitted.

4. The WHILE clause in Types 2 and 3 specifies that before each repetition of statement execution, the associated element expression is evaluated, and, if necessary, converted to a bit string. If any bit in the resulting string is 1, the statements of the DO-group are executed. If all bits are 0, then, for Type 2, execution of the DO-group is terminated, while for Type 3, only the execution associated with the "specification" containing the WHILE clause is terminated; repetitive execution for the next "specification," if one exists, then begins.

5. In a "specification," "expression1" represents the initial value of the control variable (i.e., "variable" or

"pseudovariable"); "expression3" represents the increment to be added to the control variable after each execution of the statements in the group; expression2 represents the terminating value of the control variable. Execution of the statements in a DO-group terminates for a "specification" as soon as the value of the control variable, when tested at the end of the loop, is outside the range defined by "expression1" and "expression2." When execution for the last "specification" is terminated, control, in general, passes to the statement following the DO-group.

6. Control may transfer into a DO-group from outside the DO-group only if the DO-group is delimited by the DO statement in Type 1; that is, only if repetitive execution is not specified. Consequently, repetitive DO-groups cannot contain ENTRY statements.

7. The generation of a control variable that is either pointer-qualified or controlled is established outside the loop, immediately before the initial value expression (expression1) is evaluated. If the control variable generation is changed in the loop by either changing its pointer or by allocating it, the loop is continued with the control variable derived from the previous generation. However any reference to the control variable inside the loop is a reference to the subsequent generation. It is an error to free the generation.

8. Under the optimizing compiler the maximum permissible depth of nesting is 49. There is no limit under the checkout compiler.

END

The END statement terminates blocks and groups.

General format:

```
END [identifier];
```

Syntax rules:

The "identifier" is a label or entry constant; it cannot be subscripted.

General rules:

1. If a label follows END, the statement terminates the unterminated group or block headed by the nearest preceding DO, BEGIN, or PROCEDURE statement having that label. It also terminates any unterminated groups or blocks physically within that group or block.

2. If a label does not follow END, the statement terminates that group or block headed by the nearest preceding DO, BEGIN, or PROCEDURE statement for which there is no corresponding END statement.

3. If control reaches an END statement for a procedure, it is treated as a RETURN statement.

ENTRY

The ENTRY statement specifies a secondary entry point of a procedure.

General format:

```
entry-constant: [entry-constant:]...
   ENTRY [(parameter [,parameter]...)]
         [RETURNS (attribute list)]
         [IRREDUCIBLE|REDUCIBLE]
         [OPTIONS(option-list)];
```

Syntax rules:

1. The only attributes in the attribute list of the RETURNS option that may be specified with an ENTRY statement are the arithmetic, string, ALIGNED, UNALIGNED, POINTER, OFFSET, AREA, FILE, EVENT, LABEL, and TASK attributes. Strings can be given the VARYING attribute. The OFFSET attribute may include an area name; under the optimizing compiler, this must be a non-defined, unsubscripted, unqualified name. The LABEL attribute may include a list of label constants. An area size or string length must be specified by a decimal integer constant.

2. A condition prefix cannot be specified for an ENTRY statement.

3. The options RETURNS, REDUCIBLE (or IRREDUCIBLE), and OPTIONS can appear in any order.

4. The options REDUCIBLE and IRREDUCIBLE are for optimization. If they are to appear in a program processed by the checkout compiler, they are checked for syntax errors and ignored; their presence in such a program is not an error.

5. The "options-list" of the OPTIONS option specifies one or more

additional implementation-defined options. These are:

```
{COBOL|FORTRAN}

[NOMAP    [(argument-list)]]
[NOMAPIN  [(argument-list)]]
[NOMAPOUT[(argument-list)]]
```

The options are separated by blanks, and can appear in any order.

The "argument-list" is a list of the names of the parameters to which the option applies. Not more than sixty-four parameters can be specified in an argument list; they can appear in any order, and are separated by commas or blanks. If there is no argument list, the option is assumed to apply to all the parameters associated with the entry name.

NOMAP, NOMAPIN, and NOMAPOUT can all appear in the same OPTIONS specification. This specification should not include the same parameter in more than one specified or assumed argument list.

The use of these options is described in chapter 16, "Interlanguage Communication".

General rules:

1. The relationship established between the parameters of a secondary entry point and the arguments passed to that entry point is exactly the same as that established for primary entry point parameters and arguments. See chapter 9, "Subroutines and Functions", for a complete discussion of this subject.

2. As stated in syntax rule 1, the attributes specified with an ENTRY statement determine the characteristics of the value returned by the procedure when it is invoked as a function at this entry point. The value being returned by the procedure (i.e., the value of the expression in a RETURN statement) is converted, if necessary, to correspond to the specified attributes. If the attributes are not specified at the entry point, default attributes are applied, according to the first letter of the entry name used to invoke the entry point.

3. If an ENTRY statement has more than one name, each name is interpreted as though it were a single entry name for a separate ENTRY statement having the same parameter list and explicit attribute specification. For example, consider the statement:

```
A:  I:  ENTRY;
```

This statement is effectively the same as:

```
A:  ENTRY;

I:  ENTRY;
```

Since the attributes of the returned value are not explicitly stated, the characteristics of the value returned by the procedure will depend on whether the entry point has been invoked as A or I.

4. The ENTRY statement must be internal to the procedure for which it defines a secondary entry point. It may not be internal to any block contained in this procedure; nor may it be within a DO-group that specifies repetitive execution.

5. When an ENTRY statement is encountered in normal sequential flow, control passes around it.

6. IRREDUCIBLE and REDUCIBLE are optimization options that can only be specified for function procedures. REDUCIBLE specifies that if the entry name appears with an argument list that is identical to an argument list used in an earlier invocation, the function will not necessarily be reinvoked and the result of the earlier evaluation may be used. IRREDUCIBLE specifies that this type of optimization is not permitted. Optimization within a function procedure is not affected by either attribute. If neither option is specified, IRREDUCIBLE is assumed.

7. The meaning of the options in the OPTIONS option is:

   COBOL: The PL/I procedure is to be invoked at this entry point by only a COBOL subprogram.

   FORTRAN: The PL/I procedure is to be invoked at this entry point by only a FORTRAN subroutine or function.

   NOMAP, NOMAPIN, NOMAPOUT: These options prevent the automatic manipulation of data aggregates at

the interface between either COBOL or FORTRAN and PL/I.

Each option argument-list can specify the parameters to which the option applies. If there is no argument-list for an option, that option is assumed to apply to all the parameters associated with the invocation of the entry name.

EXIT

The EXIT statement causes immediate termination of the task that contains the statement and all tasks attached by this task. If the EXIT statement is executed in a major task, it is equivalent to a STOP statement.

General format:

```
EXIT;
```

General rule:

If executed in a major task, EXIT causes the FINISH condition to be raised in that task. On normal return from the FINISH on-unit, the task executing the statement, and all of its descendant tasks are terminated. The completion values of the event variables associated with these tasks are set to '1'B, and their status values to 1 (unless they are already non-zero).

FETCH

The FETCH statement indicates to the compiler that the procedures identified by the entry constants are resident on auxiliary storage and will need to be copied into main storage if they are to be executed. The FETCH statement, when executed, causes a test to be made in main storage for the named procedures. Any procedures found not to be already in main storage are loaded from auxiliary storage. A similar test and loading is performed whenever a procedure named in a FETCH statement is invoked by a CALL statement, by a CALL option of an INITIAL attribute, or by a function reference, before an attempt is made to execute that procedure. COBOL and FORTRAN routines may be fetched in the same way as PL/I procedures.

General format:

```
FETCH entry-constant
      [,entry-constant]...;
```

General rules:

1. The entry-constant must be a name by which the procedure to be fetched is known to the operating system.
2. The entry constant in the FETCH statement must be the same as the one used in the corresponding CALL statement, CALL option, or function reference.
3. A fetched procedure may not fetch any further procedures.
4. A FETCH statement will not overlap with other statements.

FLOW

The FLOW statement causes information about the transfer of control within a task to be written on the SYSPRINT file.

The PL/I checkout compiler implements the FLOW statement in this sense, but the PL/I optimizing compiler implements this statement by checking the syntax and then ignoring it.

General format:

```
FLOW;
```

General rules:

1. When a FLOW statement has been executed, the execution (in the same task) of a subsequent statement that causes a transfer of control results in a flow comment being written on the SYSPRINT file.

   A flow comment consists of:

   a. The number of the statement that causes the transfer of control

   b. The number of the statement to which control is transferred

   A flow comment is written after control is transferred, but before execution of the target statement is commenced.

2. The flow comment is written only when the transfer of control is to a point within the task that contains the FLOW statement. If control passes to a point outside this task, (because the task terminates), no further flow comments are written.

3. The statement that causes a flow comment to be written is a transfer statement; the statement to which control is transferred is a destination statement. A summary of the transfer statements and their destination statements is given below, in figure J.4.

4. The FLOW statement remains effective until:

   a. The program terminates, or

   b. The task terminates, or

   c. A NOFLOW statement is executed later in the same task.

FORMAT

The FORMAT statement specifies a format list that can be used by edit-directed transmission statements to control the format of the data being transmitted.

General format:

```
label: [label:]... FORMAT (format-list);
```

Syntax rules:

1. The "format list" must be specified according to the rules governing format list specifications with edit-directed transmission as described in chapter 10, "Input and Output".

2. At least one "label" must be specified for a FORMAT statement. One of the labels (or a label variable or a function reference representing the value of one of the labels) is the statement label designator appearing in a 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 identified by the statement label specified with R.

2. The remote format item and the FORMAT statement must be internal to the same block.

3. If a condition prefix is associated with a FORMAT statement, it must be identical to the condition prefix associated with the GET or PUT statement referring to that FORMAT statement.

4. When a FORMAT statement is encountered in normal sequential flow, control passes around it, and the CHECK condition will not be raised for a statement label attached to it.

5. It is an error to attempt to transfer control to a FORMAT statement by means of a GO TO statement.

FREE

The FREE statement causes the storage allocated for specified based or controlled variables to be freed. For controlled variables, the next most recent allocation in the task is made available, and subsequent references in the task to the identifier refer to that allocation.

Transfer Statement	Destination Statement
GO TO	Statement prefixed by GO TO label
CALL	PROCEDURE or ENTRY statement in the invoked procedure
END or RETURN statement in a procedure invoked by a CALL statement	CALL statement
END or RETURN statement that terminates a procedure invoked by the INITIAL CALL attribute	STATIC or AUTOMATIC variable: PROCEDURE or BEGIN statement of the block that contains the DECLARE statement BASED or CONTROLLED variable: ALLOCATE statement that specifies the variable
Statement that contains a function reference	PROCEDURE or ENTRY statement in the invoked procedure
RETURN statement in a procedure invoked as a function reference	Statement containing the function reference
END statement of an iterative DO group	Matching DO statement, even if there are no more iterations to be performed
Iterative DO statement, either when the statement list has been executed in full, or when the statement list is not to be executed	Statement that follows the matching END statement
END statement that terminates an on-unit, or a single statement (except GO TO or CALL) that is an on-unit	Statement to which the on-unit returns control normally
Statement (including SIGNAL) that results in an interrupt for which there is an on-unit	First, or only, statement of the on-unit

Figure J.4. Transfer and destination statements

General format:

```
FREE option[,option]...;
```

where "option" has one of two forms:

Option 1

```
identifier
```

Option 2

```
[locator-qualifier ->]
 based-variable-identifier
[IN(element-area-variable)]
```

Syntax rules:

1. In Option 1, the "identifier" is a level-one, unsubscripted variable of the controlled storage class.
2. In Option 2, the "based-variable-identifier" must be an unsubscripted, level-one based variable.
3. It is permissible to use both types of option in one statement.

General rules:

1. Controlled storage, and based storage not in an area, that has been allocated in a task cannot be freed by any other task.
2. If a specified nonbased identifier has no allocated storage at the time the FREE statement is executed, it is a no-operation.

Rules 3 through 6 apply only to Option 2.

3. If the based variable is not explicitly qualified by locator qualification, the locator declared with the based variable will be used to identify the generation of data occupying the portion of storage to be freed. If no locator has been declared the statement is in error.

4. The amount of storage freed depends upon the attributes of the based variable, including bounds and/or lengths at the time the storage is freed, if applicable. The user is responsible for determining that this amount coincides with the amount allocated. If the variable has not been allocated, the results are unpredictable.

5. A based variable can be used to free storage only if that storage has been allocated for a based variable having identical data attributes.

6. The IN option is specified or is implied, if the storage to be freed was allocated in an area. The IN option cannot appear if the based variable was not allocated in an area. Note that area assignment causes allocation of based storage in the target area; such allocations can be freed by the IN option naming the target area.

GET

The GET statement is a STREAM transmission statement that can be used in either of the following ways:

1. It can cause the assignment of data from an external source (that is, from a data set) to one or more internal receiving fields (that is, to one or more variables).

2. It can cause the assignment of data from an internal source (that is, from a character-string variable) to one or more internal receiving fields (that is, to one or more variables).

General format:

```
GET option-list;
```

Following is the format of "option list":

```
[FILE(file-expression)
|STRING(character-string-expression)]
[data-specification]
[COPY[(file-expression)]]
[SKIP[(expression)]]
```

General rules:

1. If neither the FILE option nor the STRING option appears, the file option FILE(SYSIN) is assumed.

2. One data specification must appear unless the SKIP option is specified.

3. The options may appear in any order.

4. The "file-expression" of the FILE option represents a file which has been associated, by opening, with the data set which is to provide the values. It must be a STREAM INPUT file.

   The "file-expression" of the COPY option represents a file associated with the data set which is to receive the values. It must be a STREAM OUTPUT file.

5. The "character-string-name" refers to the character string that is to provide the data to be assigned to the data list. This name may be a reference to a built-in function. Each GET operation using this option always begins at the beginning of the specified string. If the number of characters in this string is less than the total number of characters specified by the data specification, the ERROR condition is raised.

6. When the STRING option is used under data-directed transmission, the ERROR condition is raised if an identifier within the string does not have a match within the data specification.

7. The "data-specification" is as described in chapter 11, "Stream-Oriented Transmission".

8. If the FILE option refers to a file that is not open in the current task, the file is implicitly opened in the task for stream input transmission.

   If the COPY option refers to a file that is not open in the current task, the file is implicitly opened in the task for stream output transmission.

9. The COPY option, which cannot be used with the STRING option, specifies that the source data stream, as read, is to be written, without alteration, on the specified file. Each new record in the input stream starts a new record on the COPY file. If no file is specified, the default is standard print file SYSPRINT.

10. If an interrupt during the execution of a GET statement with a COPY option causes an on-unit to be entered in which another GET statement is executed for the same file, and if control is returned from the on-unit to the interrupted statement, then resumed execution of that statement will be as if no COPY option had been specified. If, in the on-unit, a PUT statement is executed for the file associated with the COPY option, the position of the data transmitted will not necessarily be immediately following the most recently-transmitted COPY data item.

11. The SKIP option causes a new current line to be defined for the data set. The expression, if present, is converted to an integer w, which must be greater than zero. If not, the compiler substitutes a value of 1. The data set is positioned at the start of the wth line relative to the current line. If the expression is omitted, SKIP(1) is assumed. The SKIP option is always executed before any data is transmitted.

12. For the effect of statement options when specified in the first GET statement following the opening of the file, see "OPEN statement" in this section.

GO TO

Abbreviation: GOTO

The GO TO statement causes control to be transferred to the statement identified by the specified label.

General format:

```
GOTO { element-label-expression; }
     { statement-number;         }
```

Syntax rules:

1. 'Element-label-expression' can be used in a GO TO statement in a source program, or in a GO TO entered in immediate mode.

2. 'Statement-number' can only be used in a GO TO immediate statement entered at the terminal when running under the checkout compiler.

General rules:

1. An element-label-expression is a label constant, a label variable, or a function reference that returns a label value. Since a label expression may have different values at each execution of the GO TO statement, control may not always pass to the same statement.

2. A GO TO statement cannot pass control to an inactive block or to another task.

3. A GO TO statement cannot transfer control from outside a DO-group to a statement inside the DO-group if the DO-group specifies repetitive execution, unless the GO TO terminates a procedure or on-unit invoked from within the DO-group.

4. If a GO TO statement transfers control from within a block to a point not contained within that block, the block is terminated. Also, if the transfer point is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are also terminated (see chapter 8, "Storage Control", for examples and details). When one or more blocks are terminated by a GO TO statement, conditions are reinstated and automatic variables are freed just as if the blocks had terminated in the usual fashion.

5. When a GO TO statement specifies a label constant contained in a block that has more than one activation, control is transferred to the activation current when the GO TO is executed.

6. When a GO TO statement transfers control out of a procedure that has been invoked as a function, the evaluation of the expression that contained the corresponding function reference is discontinued.

HALT

The HALT statement causes execution of a task being executed in conversational mode under the checkout compiler to be interrupted and control passed to the terminal.

General format:

```
HALT;
```

General rules:

1. The HALT statement is only effective in a conversational environment. In a non-conversational environment and under the optimizing compiler, the HALT statement is a null operation.

2. The HALT statement remains effective until the programmer at the terminal causes execution to be resumed.

IF

The IF statement tests the value of a specified expression and controls the flow of execution according to the result of that test.

General format:

```
IF element-expression
      THEN unit-1
      [ELSE unit-2]
```

Syntax rules:

1. Each unit is either a single statement (except DO, END, PROCEDURE, BEGIN, DECLARE, DEFAULT, FORMAT, or ENTRY), a DO-group, or a begin block.

2. The IF statement itself is not terminated by a semicolon; however, each "unit" specified must be terminated by a semicolon.

3. Each "unit" may be labeled and may have condition prefixes.

General rules:

1. The element expression is evaluated and, if necessary, converted to a bit string. When the ELSE clause (that is, ELSE and its following "unit") is specified, the following occurs:

   If any bit in the string is 1, or if the string is null, "unit-1" is executed, and control then passes to the statement following the IF statement. If all bits in the string have the value 0, "unit-1" is not executed and "unit-2" is executed, after which control passes to the next statement.

   When the ELSE clause is not specified, the following occurs:

   If any bit in the string is 1, or if the string is null, "unit-1" is executed, and control then passes to the statement following the IF statement. If all bits are 0, "unit-1" is not executed and control passes to the next statement.

   Each "unit" may contain statements that specify a transfer of control (e.g., GO TO); hence, the normal sequence of the IF statement may be overridden.

   An array or structure variable can appear in the element expression only as an argument to a function that returns an element value.

2. IF statements may be nested; that is, either "unit", or both, may itself be an IF statement. Since each ELSE clause is always associated with the innermost unmatched IF in the same block or DO-group, an ELSE with a null statement may be required to specify a desired sequence of control. Under the optimizing compiler, the maximum permissible depth of nesting is 49. There is no restriction under the checkout compiler.

LOCATE

The LOCATE statement, which applies to BUFFERED OUTPUT files, causes allocation of a based variable in a buffer; it may also cause transmission of a based variable previously allocated in a buffer.

General format:

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

Syntax rules:

1. The options may appear in any order.

2. The "variable" must be an unsubscripted level 1 based variable.

General rules:

1. The FILE option specifies the file involved. This option must appear.

2. Execution of a LOCATE statement causes the specified based variable to be allocated in the buffer. Components of the based variable that have been specified in REFER options are initialized. A pointer value is

assigned to the pointer variable named in the SET option or, if the SET option is omitted, to the pointer variable specified in the declaration of the based variable. The pointer value identifies the record in the buffer. After execution of the LOCATE statement, values may be assigned to the based variable for subsequent transmission to the data set, which will occur immediately before the next LOCATE, WRITE, or CLOSE operation on the file. The transmitted data item must not be referred after transmission.

3. If the KEYFROM option appears, the value of the expression is converted to a character string and is used as the key of the record when it is subsequently written.

4. If the FILE option refers to an unopened file, the file is opened automatically; the effect is as if the LOCATE statement were preceded by an OPEN statement referring to the file. The file is given the attributes RECORD and OUTPUT.

NOCHECK

The NOCHECK statement suppresses the action of the CHECK statement for the specified names.

The PL/I checkout compiler implements the NOCHECK statement in this sense, but the PL/I optimizing compiler implements this statement by checking the syntax and then ignoring it.

General format:

```
NOCHECK [(name-list)];
```

Syntax rules:

1. The optional "name-list" is one or more names separated by commas; a name can be qualified, but cannot be subscripted or locator-qualified.

2. A name must be one of the following:

   a. An element, an array or a structure variable of any data type.

   b. A label constant.

   c. An entry constant.

3. If a name-list is specified, the NOCHECK statement applies to those names only. These names must be known in the block in which the NOCHECK statement is executed.

   If there is no name-list, the NOCHECK statement applies to every name in the program.

General rules:

1. Execution of a NOCHECK statement has the effect of disabling the CHECK condition for specified or assumed names. The condition-prefix can be an actual prefix, written in the program, or a conceptual prefix, derived from a previous CHECK statement.

2. The NOCHECK statement remains effective until:

   a. The program terminates, or

   b. It is overidden by an appropriate CHECK statement.

NOFLOW

The NOFLOW statement suppresses the action of the FLOW statement.

The PL/I checkout compiler implements the NOFLOW statement in this sense, but the PL/I optimizing compiler implements this statement by checking the syntax and then ignoring it.

General format:

```
NOFLOW;
```

General rules:

1. The NOFLOW statement remains effective until:

   a. The program terminates, or

   b. The task terminates or

   c. It is overridden by a FLOW statement.

Null Statement

The null statement causes no action and does not modify sequential statement execution. If the label of a null statement is enabled for the CHECK condition, CHECK is raised whenever control reaches the null statement.

General format:

```
[label:]...;
```

Note that a label prefixed to a null statement does not compare equal to a label prefixed to the statement immediately following the null statement.

For example:

```
A:;
B:  X=T;
```

Label A does not compare equal to label B.

ON

The ON statement specifies what action is to be taken (programmer-defined or standard system action) when an interrupt results from the occurrence of the specified exceptional condition.

General format:

```
ON condition[SNAP]{SYSTEM;|on-unit}
```

Syntax rules:

1. The condition may be any of those described in section H, "On-Conditions".
2. The "on-unit" represents a programmer-defined action to be taken when an interrupt results from the occurrence of the specified "condition". It can be either a single unlabeled simple statement or an unlabeled begin block. If it is an unlabeled simple statement, it can be any simple statement except BEGIN, DO, END, RETURN, FORMAT, PROCEDURE, ENTRY, DECLARE, or DEFAULT. If the on-unit is an unlabeled begin block, any statement can be used freely within that block, with one exception: a RETURN statement can appear only within a procedure nested within the begin block.
3. Since the "on-unit" itself requires a semicolon, no semicolon is shown for the "on-unit" in the general format. However, the word SYSTEM must be followed by a semicolon.

General rules:

1. The ON statement determines how an interrupt occurring for the specified condition is to be handled. Whether the interrupt is handled in a standard system fashion or by a programmer-supplied method is determined by the action specification in the ON statement, as follows:

   a. If the action specification is SYSTEM, the standard system action is taken. The standard system action is not the same for every condition, although for most conditions the system simply prints a message and raises the ERROR condition. The standard system action for each condition is given in section H, "On-Conditions". (Note that the standard system action is always taken if an interrupt occurs and no ON statement for the condition is in effect.)

   b. If the action specification is an "on-unit," the programmer has supplied his own interrupt-handling action, namely, the action defined by the statement(s) in the on-unit itself. The on-unit is not executed when the ON statement is executed; it is executed only when an interrupt results from the occurrence of the specified condition (or if the interrupt results from the condition being signaled by a SIGNAL statement).

2. The action specification (i.e., "on-unit" or SYSTEM) established by executing an ON statement in a given block remains in effect throughout that block and throughout all blocks in any activation sequence initiated by that block, unless it is overridden by the execution of another ON statement or a REVERT statement, as follows:

   a. If a later ON statement specifies the same condition as a prior ON statement and this later ON statement is executed in a block that lies within the activation sequence initiated by the block containing the prior ON statement, the action specification of the prior ON statement is temporarily suspended, or stacked. It can be restored either by the execution of a REVERT statement, or by the termination of the block containing the later ON statement.

   b. If the later ON statement and the prior ON statement are internal to the same invocation of the same block, the effect of the prior ON statement is completely nullified.

3. An on-unit is always treated by the compiler as a procedure internal to the block in which it appears. (Conceptually, it is enclosed in PROCEDURE and END statements.) Any names referenced in an on-unit are those known in the environment in which the ON statement for that on-unit was executed, rather than the environment in which the interrupt occurred.

4. A condition raised during execution results in an interrupt if and only if the condition is enabled at the point where it is raised.

   a. The conditions AREA, OVERFLOW, FIXEDOVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, all of the input/output conditions, and the conditions CONDITION, FINISH, and ERROR are enabled by default.

   b. The conditions SIZE, STRINGSIZE, STRINGRANGE, SUBSCRIPTRANGE, and CHECK are disabled by default.

   c. The enabling and disabling of OVERFLOW, FIXEDOVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, SIZE, STRINGSIZE, STRINGRANGE, SUBSCRIPTRANGE, and CHECK can be controlled by condition prefixes.

5. If an on-unit is a single statement, it cannot refer to a remote format specification.

6. If SNAP is specified, then when the given condition occurs and the interrupt results, a list of all of the blocks and on-units active at the time the interrupt occurred is printed on SYSPRINT, followed by the FLOW table. This table is the same as would be produced by a PUT FLOW statement. The list of blocks and on-units, and the FLOW table, are printed by the both the checkout and the optimizing compilers.

7. Under the optimizing compiler, up to 49 on-units may be concurrently active in any one block, and up to 254 in any one compilation. There are no limits under the checkout compiler.

OPEN

The OPEN statement associates a file name with a data set. It also can complete the specification of attributes for the file, if a complete set of attributes has not been declared for the file being opened.

General format:

```
OPEN FILE(file-expr)[options-group]
  [,FILE(file-expr)[options-group]]...;
```

where "options-group" is as follows;

```
[DIRECT|SEQUENTIAL|TRANSIENT]
[BUFFERED|UNBUFFERED]
[STREAM|RECORD]
[INPUT|OUTPUT|UPDATE]
[KEYED][EXCLUSIVE]
[BACKWARDS]
[TITLE (element-expression)]
[PRINT]
[LINESIZE(element-expression)]
[PAGESIZE(element-expression)]
```

Syntax rules:

1. The INPUT, OUTPUT, UPDATE, STREAM, RECORD, DIRECT, SEQUENTIAL, TRANSIENT, BUFFERED, UNBUFFERED, KEYED, EXCLUSIVE, BACKWARDS, and PRINT options specify attributes that augment the attributes specified in the file declaration; for rules governing which of these attributes can be applied together, see chapter 11, "Input and "Output", and the corresponding attributes in section I, "Attributes".

2. The options in an "option-group" and the FILE option for a file may appear in any order.

3. The "file-expression" represents the name of the file that is to be associated with a data set. Several files can be opened by one OPEN statement.

General rules:

1. The opening of an already open file does not affect the file if the second opening takes place in the same task or an attached task. In such cases, any expressions in the "options-group" are evaluated, but they are not used.

2. If the TITLE option is specified, the "element-expression" is converted to a character string, if necessary, the first eight characters of which identify the data set (the ddname) to be associated with the file. If this option does not appear, the first eight characters of the file name (padded or truncated) are taken to be the ddname. Note that this is not the same truncation as that for external names. If the file name is a parameter, the identifier of the original argument passed to the parameter, rather than the identifier

of the parameter itself, is used as the identification.

3. The LINESIZE option can be specified only for a STREAM OUTPUT file. The expression is evaluated, converted to an integer, and used as the length of a line during subsequent operations on the file. New lines may be started by use of the printing and control format items or by options in a GET or PUT statement. If an attempt is made to position a file past the end of a line before explicit action to start a new line is taken, a new line is automatically started, and the file is positioned to the start of this new line. The following implementation-defined values apply:

   Maximum line size:

F- or U-format:	32,759
V-format	32,751

   Minimum line size:

F- or U-format	1
V-format: PRINT files	9
Non-PRINT FILES	10
Default line size	120

   The LINESIZE option cannot be specified for an INPUT file. The line size taken into consideration whenever a SKIP option appears in a GET statement is the line size, if any, that was used to create the data set; otherwise, the line size is taken to be the current length of the logical record (minus control bytes, for V-format records).

4. The PAGESIZE option can be specified only for a file having the STREAM and PRINT attributes. The element expression is evaluated and converted to an integer, which represents the maximum number of lines to a page. During subsequent transmission to the PRINT file, a new page may be started by use of the PAGE format item or by the PAGE option in the PUT statement. If a page becomes filled and more data remains to be printed before action to start a new page is taken, the ENDPAGE condition is raised. The following implementation-defined values apply:

Maximum page size	32,767
Minimum page size	1
Default page size	60

5. When a STREAM file is opened, it is conceptually positioned as if it had just completed scanning of the zeroth record - that is, it is positioned at the end of an imaginary record immediately preceding the record accessed in the first GET or PUT statement. Thus if the first GET or PUT specifies, by means of a statement option or format item, that n lines are to be skipped before the first record is accessed, the file is then positioned at the start of the nth record. For a PRINT file, "the zeroth record" means a conceptual line having a line number equal to the pagesize. The first PUT statement will cause ENDPAGE to be raised, unless it specifies only the PAGE option or format item, or only that zero lines are to be skipped; in these cases no change takes place in the file.

PROCEDURE

The PROCEDURE statement has the following functions:

- It heads a procedure.
- It defines the primary entry point to the procedure.
- It specifies the parameters, if any, for the primary entry point.
- It may specify certain special characteristics that a procedure can have.
- It may specify the attributes of the value that is returned by the procedure if it is invoked as a function at its primary entry point.

General format:

```
entry-constant: [entry-constant:]...
   PROCEDURE[(parameter[,parameter]...)]
   [OPTIONS (option-list)]
   [RECURSIVE][RETURNS(data attributes)]
   [ORDER|REORDER]
   [REDUCIBLE|IRREDUCIBLE];
```

Syntax rules:

1. The "data attributes" given in the RETURNS option represent the attributes of the value returned by the procedure when it is invoked as a function at its primary entry point. Only arithmetic, string, ALIGNED, UNALIGNED, POINTER, OFFSET, AREA, FILE, EVENT, LABEL, and TASK attributes are allowed. Strings can be given the VARYING attribute. The OFFSET attribute may include an area name; under the optimizing compiler,

this must be a non-defined, unsubscripted, unqualified, area name. The LABEL attribute may include a list of label constants. An area size or string length must be specified by a decimal integer constant.

2. OPTIONS, RECURSIVE, RETURNS, ORDER, REORDER, REDUCIBLE and IRREDUCIBLE, can appear in any order and are separated by blanks.

3. The options ORDER, REORDER, REDUCIBLE and IRREDUCIBLE are for optimization. If they are included in a program processed by the checkout compiler, they are checked for syntax errors and ignored; their presence in such a program is not an error.

4. The "options-list" of the OPTIONS option specifies one or more additional implementation-defined options. These are:

```
{MAIN|COBOL|FORTRAN}

[NOMAP[(argument-list)]]
[NOMAPIN[(argument-list)]]
[NOMAPOUT[(argument-list)]]

[REENTRANT]

[TASK]
```

The options are separated by blanks or commas, and can appear in any order.

The "argument-list" is a list of the names of the parameters to which the option applies. Not more than sixty-four parameters can be specified in an argument list; they can appear in any order and are separated by commas or blanks. If there is no argument list, the option is assumed to apply to all the parameters associated with the entry name.

NOMAP, NOMAPIN, and NOMAPOUT can all appear in the same OPTIONS-attribute specification. This specification should not include the same parameter in more than one specified or assumed argument list.

The use of COBOL, FORTRAN, NOMAP, NOMAPIN, and NOMAPOUT is described in chapter 19, "Interlanguage Communication Facilities".

Note:

a. The TASK option need not be specified for procedures to be processed by the checkout or optimizing compilers. However, it may be required if these procedures are processed by other PL/I compilers.

b. The REENTRANT option applies to code produced by a PL/I compiler; if this option is specified with either the COBOL or FORTRAN options, this has no effect on the code in the COBOL or FORTRAN program. A program that calls COBOL or FORTRAN routines is not reenterable.

c. The TASK option must not be specified with either the COBOL or the FORTRAN options.

General rules:

1. When the procedure is invoked, a relationship is established between the arguments passed to the procedure and the parameters that represent those arguments in the invoked procedure. This topic is discussed in chapter 9, "Subroutines and Functions".

2. OPTIONS may be specified only for an external procedure, and at least one external procedure must have the OPTIONS (MAIN) designation; if more than one is so designated, the operating system will invoke the one that appears first, physically.

3. RECURSIVE must be specified if the procedure might be invoked recursively; that is, if it might be reactivated while it is still active. If specified, it applies to all of the entry points (primary and secondary) that the procedure might have. It applies only to the procedure for which it is declared.

4. The "data attributes" in the RETURNS option specify the attributes of the value returned by the procedure when it is invoked as a function at its primary entry point. The value specified in the RETURN statement of the invoked procedure is converted to conform with these attributes before it is returned to the invoking procedure.

   If the RETURNS option is not specified, default attributes are supplied. In such a case, the name of the entry point (the entry constant by which the procedure has been invoked) is used to determine the default base, precision, and scale. (Since the entry point can have several entry constants, the default base, precision, and scale can differ according to the entry constant.)

5. ORDER and REORDER are optimization options. ORDER and REORDER specify the extent to which the block is to be optimized. In general, ORDER permits optimization to the degree such that the latest values of all variables set in a block are guaranteed available in a computational on-unit entered during execution of the block. REORDER permits a greater degree of optimization; with REORDER the values of variables set in the block are not guaranteed to be the most recently assigned values in an on-unit entered during execution of the block. If neither option is specified, ORDER is assumed but REORDER is inherited by all contained blocks unless they explicitly specify ORDER.

6. IRREDUCIBLE and REDUCIBLE are optimization options that can only be specified for function procedures. REDUCIBLE specifies that if the entry name appears with an argument list that is identical to an argument list used in an earlier invocation, the function will not necessarily be reinvoked and the result of the earlier evaluation may be used. IRREDUCIBLE specifies that this type of optimization is not permitted. Optimization within a function procedure is not affected by either attribute. If neither option is specified, IRREDUCIBLE is assumed.

7. If a PROCEDURE statement has more than one entry constant, the first constant can be considered as the only label of the statement; each subsequent entry constant can be considered as a separate ENTRY statement having an identical parameter list as specified in the PROCEDURE statement. For example, the statement:

```
A:  I:  PROCEDURE (X);
```

   is effectively the same as:

```
A:   PROCEDURE (X);

I:   ENTRY (X);
```

   Since the attributes of the value are not explicitly stated, the characters of the value returned by the procedure will depend on whether the procedure has been invoked as A or I.

8. The meaning of the options in the OPTIONS option is:

   COBOL: The PL/I procedure is to be invoked at its main entry point by only a COBOL subprogram.

   FORTRAN: The PL/I procedure is to be invoked at its main entry point by only a FORTRAN subroutine or function.

   MAIN: The PL/I procedure is the initial procedure of a PL/I program, and is invoked by the operating-system control program as the first step in the execution of that program.

   NOMAP, NOMAPIN, NOMAPOUT: These options prevent the automatic manipulation of data aggregates at the interface between either COBOL or FORTRAN and PL/I.

   Each option argument-list can specify the parameters to which the option applies. If there is no argument list for an option, that option is assumed to apply to all the parameters associated with the invocation of the entry name.

   REENTRANT: The code produced by the compiler is reenterable.

   TASK: The PL/I multitasking facilities are to be used.

PUT

The PUT statement is a STREAM transmission statement that can be used in either of the following ways:

1. It can cause the values in one or more internal storage locations to be transmitted to a data set on an external medium.

2. It can cause the values in one or more internal storage locations to be assigned to an internal receiving field (represented by a character-string variable).

3. Under the checkout compiler, it can cause program checkout information to be written onto the SYSPRINT file.

General format:

```
PUT [FILE (file-expression)]|
    [STRING (character-string-variable)]
    [data-specification]|
    [SNAP]|
    [FLOW[(n)]]|
    [ALL[(character-string-expression)]]
    [PAGE [LINE(element-expression)]]
    [SKIP [(element-expression)]]
    [LINE(element-expression)       ] ;
```

Syntax rules:

1. If neither the FILE nor STRING option appears, the specification FILE (SYSPRINT) is assumed. If such a PUT statement lies within the scope of a declaration of the identifier SYSPRINT, SYSPRINT must have been declared as FILE STREAM OUTPUT. If the PUT statement does not lie within the scope of a declaration of SYSPRINT, SYSPRINT is the standard system output file.

2. The FILE option specifies transmission to a data set on an external medium. The file expression in this option is the name of the file that has been associated (by implicit or explicit opening) with the data set that is to receive the values. This file must have the OUTPUT and STREAM attributes.

3. Under the checkout compiler, the SNAP option causes a list of all currently active blocks and on-units to be printed on SYSPRINT. Under the optimizing compiler, the option's syntax is checked, then it is ignored.

4. Under the checkout compiler, the FLOW option causes a comment on each of the last n transfers of control to be put into the SYSPRINT stream. The rules determining the nature of each flow comment are the same as for the FLOW statement, described earlier in this section. If n is not specified, the value specified in the appropriate compiler option is used; if no value is specified there, a default of 25 is taken. Under the optimizing compiler, the syntax of the option is checked, then it is ignored.

5. Under the checkout compiler, the ALL option causes all information provided by the SNAP and FLOW options to be put into the SYSPRINT stream, together with certain other debugging information. A description of this information is given in chapter 15, "Execution-time Facilities of the Checkout Compiler". Under the optimizing compiler, the syntax of the option is checked, then it is ignored. The value of the character-string-expression must be one or more of the option characters D,S,F,C,T,n concatenated to form a string without blanks or punctuation marks, n being one through four digits.

6. The STRING option specifies transmission from internal storage locations (represented by variables or expressions in the "data-specification") to a character string (represented by the "character-string-variable"). It cannot be used with a SNAP, FLOW, or ALL option. The "character-string-variable" can be any string pseudovariable other than STRING.

7. The "data specification" option is as described in chapter 11, "Stream-Oriented Transmission".

8. The PAGE, SKIP, and LINE options cannot appear with the STRING option.

9. The options may appear in any order; at least one must appear.

General rules:

1. If the FILE option is specified, and the "file-expression" refers to an unopened file, the file is opened implicitly as an OUTPUT file.

2. If the STRING option is specified, the PUT operation begins assigning values to the beginning of the string (that is, at the left-most character position), after appropriate conversions have been performed. Blanks and delimiters are inserted as usual. If the string is not long enough to accomodate the data, the ERROR condition is raised.

3. The PAGE and LINE options can be specified for PRINT files only. All of the options take effect before transmission of any values defined by the data specification, if given. Of the three, only PAGE and LINE may appear in the same PUT statement, in which case, the PAGE option is applied first.

4. The PAGE option causes a new current page to be defined within the data set. If a data specification is present, the transmission of values occurs after the definition of the new page. The page remains current until the execution of a PUT statement with the PAGE option, until a PAGE format item is encountered, or until the ENDPAGE condition is raised, resulting in the definition of a new page. A new current page implies line one.

   When printing at a terminal in conversational mode, the PAGE option causes three lines to be skipped.

5. The SKIP option causes a new current line to be defined for the data set. The expression, if present, is converted to an integer w, which for non-PRINT files must be greater than zero. The data set is positioned at the start of the wth line after the current line. If the expression is omitted, SKIP(1) is assumed.

   For PRINT files w may be less than or equal to zero; in this case, the effect is that of a carriage return with the same current line. If less than w lines remain on the current page when a SKIP(w) is issued, ENDPAGE is raised.

   When a SKIP option is specified on the first PUT statement of a file, the data set is positioned at the start of the wth line on the first page. If w is zero or one, it is positioned at the start of the first line.

   When printing at a terminal in conversational mode, no more than three lines may be skipped; SKIP(w) with w greater than 3 is equivalent to SKIP(3).

6. The LINE option causes a new current line to be defined for the data set. The expression is converted to an integer w. The LINE option specifies that blank lines are to be inserted so that the next line will be the wth line of the current page. If at least w lines have already been written on the current page or if w exceeds the limits set by the PAGESIZE option of the OPEN statement, the ENDPAGE condition is raised. If w is less than or equal to zero, it is assumed to be 1. If w specifies the current line, ENDPAGE is raised except when the file is positioned on column 1; in this case, the effect is as for a SKIP(0) option.

   If the LINE option is specified in the same statement as a PAGE option, the PAGE option is executed first.

   When printing at a terminal in conversational mode, the LINE option causes three lines to be skipped.

7. For the effects of statement options when specified in the first PUT statement following the opening of the file, see "OPEN statement" in this section.

READ

The READ statement causes a record to be transmitted from a RECORD INPUT or RECORD UPDATE file to a variable or buffer.

General format:

```
READ option-list;
```

The format of the option list is shown in figure J.5.

General rules:

1. The options may appear in any order.
2. The FILE option specifies the file from which the record is to be read. This option must appear. If the file specified is not open in the current task, it is opened.
3. The INTO(variable) option specifies the variable into which the record is to be read. If the variable is an aggregate, it must be in connected storage; certain uses of unaligned fixed-length bit strings are disallowed (for details, see "Data Transmitted" in chapter 12, "Record-Oriented Transmission"). String pseudovariables other than STRING may be specified.

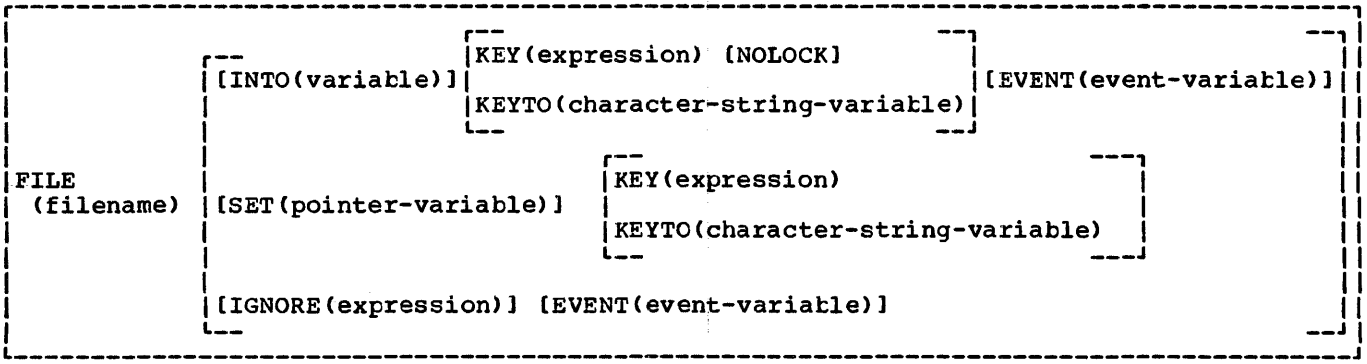


Figure J.5. Format of option list for READ statement

4. The KEY and KEYTO options can be specified for KEYED files only.

5. The KEY option must appear if the file has the DIRECT attribute. The "element-expression" is converted to a character string that represents a key. It is this key that determines which record will be read.

   The KEY option may also appear for a file having INDEXED organization and 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.

6. The KEYTO option can be given only if the file has the SEQUENTIAL and KEYED attributes. It specifies that the key of the record being read is to be assigned to the "character-string variable" according to the rules for character-string assignment. The KEYTO option can specify any string pseudovariable other than STRING. It cannot specify a variable declared with a numeric picture specification. The maximum permissible length for the character string is 256.

   Assignment to the KEYTO variable always follows assignment to the INTO variable. If an incorrect key specification is detected, the KEY condition is raised. For this implementation, the value assigned is as follows:

   a. For REGIONAL(1), the eight character region number, padded or truncated on the left to the declared length of the character-string variable. If the character-string variable is of varying length, any leading zeros in the region number are truncated and the string length is set to the number of significant digits. An all-zero region number is truncated to a single zero.

   b. For REGIONAL(2) and REGIONAL(3), the recorded key without the region number, padded or truncated on the right to the declared length of the character-string variable.

   c. For INDEXED, the recorded key, padded or truncated on the right to the declared length of the character-string variable.

   The KEY condition will not be raised for such padding or truncation.

7. The EVENT option allows processing to continue while a record is being read or ignored. This option cannot be specified for a SEQUENTIAL BUFFERED file.

   When control reaches a READ statement containing this option, the "event variable" is made active (that is, it cannot be associated with another event) and is given the completion value '0'B, provided that the UNDEFINEDFILE condition is not raised by an implicit file opening (see "Note" below). The event variable remains active and retains its '0'B completion value until control reaches a WAIT statement specifying that event variable. At this time, either of the following can occur:

   a. If the READ statement has been executed successfully and none of the conditions ENDFILE, TRANSMIT, KEY or RECORD has been raised as a result of the READ, the event variable is set complete (given the completion value '1'B), and the event variable is made inactive, that is, it can be associated with another event.

   b. If the READ statement has resulted in the raising of ENDFILE, TRANSMIT, KEY, or RECORD, the interrupt for each of these conditions does not occur until the WAIT is encountered. At such a time, the corresponding on-units (if any) are entered in the order in which the conditions were raised. After a return from the final on-unit, or if one of the on-units is terminated by a GO TO statement, the event variable is given the completion value '1'B and is made inactive.

   Note: If the READ statement causes an implicit file opening that results in the raising of UNDEFINEDFILE, the on-unit associated with this condition is entered immediately and the event variable remains unchanged; that is, the event variable remains inactive and retains the same value it had when the READ was encountered. If the on-unit does not correct the condition, then, upon normal return from the on-unit, the ERROR condition is raised; if the condition is corrected in the on-unit, that is, if the file is opened successfully, then, upon normal return from the on-unit, the event variable is set to

'0'B, it is made active, and execution of the READ statement continues.

8. Any READ statement referring to an EXCLUSIVE file will cause the record to be locked unless the NOLOCK option is specified. A locked record cannot be read, deleted, or rewritten by any other task until it is unlocked. Any attempt to read, delete, rewrite, or unlock a record locked by another task results in a wait. Subsequent unlocking can be accomplished by the locking task through the execution of an UNLOCK, REWRITE, or DELETE statement that specifies the same key, by a CLOSE statement, or by completion of task in which the record was locked.

   Note that a record is considered locked only for tasks other than the task that actually locks it; in other words, a locked record can always be read by the task that locked it and still remain locked as far as other tasks are concerned (unless, of course, the record has been explicitly unlocked by one of the above methods).

9. The SET option specifies that the record is to be read into a buffer and that a pointer value is to be assigned to the named locator variable. The pointer value identifies the record in the buffer.

10. The IGNORE option may be specified for SEQUENTIAL INPUT and SEQUENTIAL UPDATE files. The expression in the IGNORE option is evaluated and converted to an integer. If the value, n, is greater than zero, n records are ignored; a subsequent READ statement for the file will access the (n+1)th record. If n is less than 1, the option has no effect. A READ statement without an INTO, SET, or IGNORE option is equivalent to a READ with an IGNORE(1).

11. A file with INDEXED organization that is being accessed sequentially may be positioned by issuing a READ statement with the KEY option. The specified key will be used to identify the record required. Thereafter, records may be read sequentially from that point by use of READ statements without the KEY option. This applies to INPUT and UPDATE files.

    For INDEXED SEQUENTIAL files, two positioning statements can be used, with the following formats:

```
READ  FILE (file-expression)
      INTO (variable)
      KEY (expression);

READ  FILE (file-expression)
      SET (pointer-variable)
      KEY (expression);
```

12. The EVENT, IGNORE, KEY and NOLOCK options cannot be used with a TRANSIENT file.

RELEASE

The RELEASE statement frees for other purposes main storage occupied by procedures identified by the specified entry constants. Also, whenever a procedure named in a RELEASE statement is invoked by a CALL statement, a CALL option of an INITIAL attribute or a function reference, and is found not to be resident in main storage, a search is made for the procedure on auxiliary storage. If it is found, it is copied into main storage before any attempt is made to execute it.

General format:

```
RELEASE entry-constant
        [,entry-constant]...;
```

General rules:

1. At execution time, the only effect of the RELEASE statement is to free the necessary storage. It has no effect on the meaning or scope of the entry-constant.

2. The entry-constant must be the same as the one used in any corresponding CALL statements or options, or function references, and FETCH statements.

RETURN

The RETURN statement terminates execution of the procedure that contains the RETURN statement. If the procedure has not been invoked as a task, the RETURN statement returns control to the invoking procedure. The RETURN statement may also return a value.

General format:

Option 1.

```
RETURN;
```

Option 2.

```
RETURN (element-expression);
```

General rules:

1. Only the RETURN statement in Option 1 can be used to terminate procedures not invoked as function procedures; control is returned to the point logically following the invocation.

   Option 1 represents the only form of the RETURN statement that can be used to terminate a procedure initiated as a task. If the RETURN statement terminates the major task, the FINISH condition is raised prior to the execution of any termination processes. If the RETURN statement terminates any other task, the completion value of the associated event variable (if any) is set to '1'B, and the status value is left unchanged.

2. The RETURN statement in Option 2 is used to terminate a procedure invoked as a function procedure only. Control is returned to the point of invocation, and the value returned to the function reference is the value of the expression specified converted to conform to the attributes declared for the invoked entry point. These attributes may be explicitly specified at the entry point; they are otherwise implied by the initial letter of the entry name through which the procedure is invoked.

3. If control reaches an END statement corresponding to the end of a procedure, this END statement is treated as a RETURN statement (of the Option 1 form) for the procedure.

REVERT

The REVERT statement is used to cancel the effect of the latest relevant ON statement. It can affect only ON statements that are internal to the block in which the REVERT statement occurs and which have been executed in the same invocation of that block. Execution of the REVERT statement in a given block cancels the action specification of any ON statement for the named condition that has been executed in that block; it then re-establishes the action specification that was in force at the time of activation of the block.

General format:

```
REVERT condition;
```

Syntax rule:

The "condition" is any of those described in section H, "On-Conditions".

General rule:

The execution of a REVERT statement has the effect described above only if (1) an ON statement, specifying the same condition and internal to the same block, was executed after the block was activated and (2) the execution of no other similar REVERT statement has intervened. If either of these two conditions is not met, the REVERT statement is treated as a null statement.

REWRITE

The REWRITE statement can be used only for update files. It replaces an existing record in a data set.

General format:

```
REWRITE FILE (file-expression)
   [FROM(variable)]
   [KEY (element-expression)]
   [EVENT (event-variable)];
```

Syntax rules:

1. The options may appear in any order.

2. The "file-expression" represents the name of the file containing the record to be rewritten. The file must have the UPDATE attribute.

3. The FROM option specifies a variable that represents the record that will replace the existing record in the specified file. If the variable is an aggregate, it must be in connected storage; certain uses of unaligned fixed-length bit strings are disallowed (for details, see "Data Transmitted" in chapter 12, "Record-Oriented Transmission").

General rules:

1. If the file referred to by "file-expression" has not been opened, it is opened implicitly with the attributes RECORD and UPDATE.

2. The KEY option must appear if the file has the DIRECT attribute; it cannot appear otherwise. The element-expression is converted to a character string. This character string is the source key that determines which record is to be rewritten.

3. For SEQUENTIAL files with INDEXED organization, if the key is an embedded key, the user must take care that the rewritten key is the same as the key in the replaced record.

4. The FROM option must be specified for UPDATE files having either the DIRECT attribute or both the SEQUENTIAL and UNBUFFERED attributes. A REWRITE statement in which the FROM option has not been specified has the following effect:

   a. If the last record was read by a READ statement with the INTO option, REWRITE without FROM has no effect on the record in the data set.

   b. If the last record was read by a READ statement with the SET option, the record will be updated by whatever assignments were made in the buffer identified by the pointer variable in the set option. When the records are blocked, a REWRITE statement issued for any record in the block causes the complete block to be rewritten even if no REWRITE statements are issued for other records in the block.

5. The EVENT option allows processing to continue while a record is being rewritten. This option must not be specified for a SEQUENTIAL BUFFERED file.

   When control reaches a REWRITE statement containing this option, the event variable is made active (that is, it cannot be associated with another event) and is given the completion value '0'B, provided that the UNDEFINEDFILE condition is not raised by an implicit file opening (see "Note" below). The event variable remains active and retains its '0'B completion value until control reaches a WAIT statement specifying that event variable. At this time, either of the following can occur:

   a. If the REWRITE statement has been executed successfully and none of the conditions TRANSMIT, KEY, or RECORD has been raised as a result of the REWRITE, the event variable is set complete (given the completion value '1'B), and the event variable is made inactive (that is, it can be associated with another event).

   b. If the REWRITE statement has resulted in the raising of TRANSMIT, KEY, or RECORD, the interrupt for each of these conditions does not occur until the WAIT is encountered. At such time, the corresponding on-units (if any) are entered in the order in which the conditions were raised. After a return from the final on-unit, or if one of the on-units is terminated by a GO TO statement, the event variable is given the completion value '1'B and is made inactive.

   Note: If the REWRITE statement causes an implicit file opening that results in the raising of UNDEFINEDFILE, the on-unit associated with this condition is entered immediately and the event variable remains unchanged, that is, the event variable remains inactive and retains the same value it had when the REWRITE was encountered. If the on-unit does not correct the condition, then, upon normal return from the on-unit, the ERROR condition is raised; if the condition is corrected in the on-unit, that is, if the file is opened successfully, then, upon normal return from the on-unit, the event variable is set to '0'B, it is made active, and execution of the REWRITE statement continues.

6. If the record rewritten is one that was locked in the same task, it becomes unlocked.

SIGNAL

The SIGNAL statement simulates the occurrence of an interrupt. It may be used to test the current action specification for the associated condition.

General format:

```
SIGNAL condition;
```

Syntax rule:

The "condition" is any one of those described in section H, "On-Conditions".

General rules:

1. When a SIGNAL statement is executed, it is as if the specified condition has actually occurred. Sequential execution is interrupted and control is transferred to the current on-unit for the specified condition. After the on-unit has been executed, control returns to the statement immediately following the SIGNAL statement.
2. The on-condition CONDITION can cause an interrupt only as a result of its specification in a SIGNAL statement.
3. If the specified condition is disabled, no interrupt occurs, and the SIGNAL statement becomes equivalent to a null statement.
4. If there is no current on-unit for the specified condition, then the standard system action for the condition is performed.

STOP

The STOP statement causes immediate termination of the major task and all sub-tasks

General format:

```
STOP;
```

General rule:

Prior to any termination activity the FINISH condition is raised in the task in which the STOP is executed. On normal return from the FINISH on-unit, all tasks in the program are terminated.

UNLOCK

The UNLOCK statement makes the specified locked record available to other tasks for operations on the record.

General format:

```
UNLOCK option-list;
```

Following is the format of "option list":

```
FILE(file-expression) KEY(expression)
```

General rules:

1. The options may appear in either order.
2. The FILE option specifies the file involved, which must have the attributes UPDATE, DIRECT, and EXCLUSIVE.
3. In the KEY option, the "expression" is converted to a character string and determines which record is unlocked.
4. A record can be unlocked only by the task which locked it.

WAIT

The execution of a WAIT statement within an activation of a block retains control for that activation of that block within the WAIT statement until certain specified events have completed.

General format:

```
WAIT (event [,event]...)
  [(element-expression)];
```

Syntax rules:

Each event is an event variable, or an array or (for the checkout compiler only) a structure consisting only of event variables.

General rules:

1. Control for a given block activation remains within this statement until, at possibly separate times during the execution of the statement, the condition

   ```
   COMPLETION(event) = '1'B
   ```

   has been satisfied, for some or all of the event names in the list.
2. If the expression does not appear, all the event names in the list must satisfy the above condition before control is passed to the next statement in this task following the WAIT.

3. If the optional expression appears, the expression is evaluated when the WAIT statement is executed and converted to an integer. This integer specifies the number of events in the list that must satisfy the above condition before control for the block passes to the statement following the WAIT. Of course, if an on-unit entered due to the WAIT is terminated abnormally, control might not pass to the statement following the WAIT.

   If the value of the expression is zero or negative, the WAIT statement is treated as a null statement. If the value of the expression is greater than the number, n, of event names in the list, the value is taken to be n. If the statement refers to an array event name, then each of the array elements contributes to the count.

4. If the event variable named in the list has been associated with a task in its attaching CALL statement, then the condition in Rule 1 will be satisfied on termination of that task.

5. If the event variable named in the list is associated with an input/output operation initiated in the same task as the WAIT, the condition in Rule 1 will be satisfied when the input/output operation is completed. The execution of the WAIT is a necessary part of the completion of an input/output operation. If prior to, or during, the WAIT all transmission associated with the input/output operation is terminated, then the WAIT performs the following action. If the transmission has finished without requiring any input/output conditions to be raised, the event variable is set complete (i.e., COMPLETION(event name) = '1'B). If the transmission has been terminated but has required conditions to be raised, the event variable is set abnormal (i.e., STATUS(event name) = 1) and all the required on-conditions are raised. On return from the last on-unit, the event variable is set complete.

6. The order in which on-conditions for different input/output events are raised is not dependent on the order of appearance of the event names in the list. If an on-condition for one event is raised, then all other conditions for that event are raised before the WAIT is terminated or before any other input/output conditions are raised unless an abnormal return is made from one of the on-units thus entered. The raising of ON conditions for one event implies nothing about the completion or termination of transmission of other events in the list.

7. If an abnormal return is made from any on-unit entered from a WAIT, the associated event variable is set complete, the execution of the WAIT is terminated, and control passes to the point specified by the abnormal return.

8. If some of the event names in the WAIT list are associated with input/output operations and have not been set complete before the WAIT is terminated (either because enough events have been completed or due to an abnormal return), then these incomplete events will not be set complete until the execution of another WAIT referring to these events in this same task.

WRITE

The WRITE statement is a RECORD transmission statement that transfers a record from a variable in internal storage to an OUTPUT or UPDATE file.

General format:

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

Syntax rules:

1. The options may appear in any order.

2. The "file expression" specifies the file in which the record is to be written. This file must be a RECORD file that has either the OUTPUT attribute or the DIRECT and UPDATE attributes.

3. The FROM option specifies a variable that represents the record to be written. If the variable is an aggregate, it must be in connected

storage; certain uses of unaligned fixed-length bit strings are disallowed (for details see "Data Transmitted" in chapter 12, "Record-Oriented Transmission").

General rules:

1. If the file is not open in a task, it is opened for that task implicitly with the attributes RECORD and OUTPUT (unless UPDATE has been declared).

2. If the KEYFROM option is specified, the "element expression" is converted to a character string. This character string is the source key that specifies the relative location in the data set where the record is written. For REGIONAL(2), REGIONAL(3), and INDEXED, KEYFROM also specifies a recorded key whose length is determined by the KEYLEN subparameter or the KEYLENGTH option.

3. The EVENT option allows processing to continue while a record is being written. This option cannot be specified for a SEQUENTIAL BUFFERED file; record transmisson and processing are automatically overlapped in such a file.

   When control reaches a WRITE statement containing this option, the "event variable" is made active (that is, it cannot be associated with another event) and is given the completion value '0'B, provided that the UNDEFINEDFILE condition is not raised by an implicit file opening (see "Note" below). The event variable remains active and retains its '0'B completion value until control reaches a WAIT statement specifying that event variable. At this time, either of the following can occur:

   a. If the WRITE statement has been executed successfully and none of the conditions TRANSMIT, KEY, or RECORD has been raised as a result of the WRITE, the event variable is set complete (given the completion value '1'B), and the event variable is made inactive, that is, it can be associated with another event.

   b. If the WRITE statement has resulted in the raising of TRANSMIT, KEY, or RECORD, the interrupt for each of these conditions does not occur until the WAIT is encountered. At such time, the corresponding on-units (if any) are entered in the order in which the conditions were raised. After a return from the final on-unit, or if one of the on-units is terminated by a GO TO statement, the event variable is given the completion value ('1'B) and is made inactive.

   Note: If the WRITE statement causes an implicit file opening that results in the raising of UNDEFINEDFILE, the on-unit associated with this condition is entered immediately and the event variable remains unchanged; that is, the event variable remains inactive and retains the same value it had when the WRITE was encountered. If the on-unit does not correct the condition, then, upon normal return from the on-unit, the ERROR condition is raised; if the condition is corrected in the on-unit, that is, if the file is opened successfully, then upon normal return from the on-unit, the event variable is set to '0'B, it is made active, and execution of the WRITE statement continues.

4. The EVENT option cannot be used with a TRANSIENT file.

Preprocessor Statements

All of the statements that can be executed at the preprocessor stage are presented alphabetically in this section.

%ACTIVATE

Abbreviation: %ACT

The appearance of an identifier in a %ACTIVATE statement makes it active and eligible for replacement; that is, any subsequent encounter of that identifier in a nonpreprocessor statement, while the identifier is active, will initiate replacement activity.

General format:

```
%[label:]...  ACTIVATE identifier
  [RESCAN|NORESCAN][,identifier
  [RESCAN|NORESCAN]]...;
```

Syntax rules:

1. Each identifier must be a preprocessor variable, a preprocessor procedure name, or any of the built-in functions INDEX, LENGTH or SUBSTR.

2. A %ACTIVATE statement cannot appear within a preprocessor procedure.

General rules:

1. An identifier cannot be activated initially by a %ACTIVATE statement; the appearance of that identifier in a %DECLARE statement serves that purpose. An identifier can be activated by a %ACTIVATE statement only after it has been deactivated by a %DEACTIVATE statement.

2. When an identifier is active (and has been given a value -- if it is a preprocessor variable) any encounter of that identifier within a nonpreprocessor statement will initiate replacement activity in all cases except when the identifier appears within a comment or within single quotes. For example, if the source program contains the following sequence of statements:

```
% DECLARE I FIXED, T CHARACTER;

% DEACTIVATE I;

% I = 15;

% T = 'A(I)';

S = I*T*3;

% I = I+5;

% ACTIVATE I;

% DEACTIVATE T;

R = I*T*2;
```

   then the preprocessed text generated by the above would be as follows (replacement blanks are not shown):

```
S = I*A(I)*3;

R = 20*T*2;
```

3. If the identifier to which RESCAN or NORESCAN refers is the name of a preprocessor variable of type FIXED or of a preprocessor procedure which returns a FIXED value, replacement in the output stream occurs irrespective of which option is specified. If the identifier to which RESCAN or NORESCAN refers is the name of a preprocessor variable of type CHARACTER or of a procedure which returns a CHARACTER value then:

   a. RESCAN specifies that when the identifier is scanned by the preprocessor, replacement in the output stream takes place as usual.

   b. NORESCAN specifies:

      (1) That when the identifier is scanned by the preprocessor, it is replaced in the output stream by that text which is the current value of the variable named by the identifier, or by that text which is the result of invoking the procedure named by the identifier.

      (2) That this text is not to be rescanned for further replacement.

   RESCAN is the default.

4. The execution of a %ACTIVATE statement to activate a preprocessor identifier that is already activated has no effect.

%Assignment Statement

The %assignment statement is used to evaluate preprocessor expressions and to assign the result to a preprocessor variable.

General format:

```
%[label:]...  preprocessor-variable =
                  preprocessor-expression;
```

General rule:

When the value assigned to a preprocessor variable is a character string, this character string should not contain a preprocessor statement.

%DEACTIVATE

Abbreviation: %DEACT

The appearance of an identifier in a %DEACTIVATE statement makes it inactive and ineligible for replacement; that is,

any subsequent encounter of that identifier in a nonpreprocessor statement will not initiate any replacement activity (unless, of course, the identifier has been reactivated in the interim).

General format:

```
%[label:]...  DEACTIVATE identifier
        [,identifier]...;
```

Syntax rules:

1. Each "identifier" must be either a preprocessor variable, the SUBSTR built-in function, or a preprocessor procedure name.

2. A %DEACTIVATE statement cannot appear within a preprocessor procedure.

General rule:

The deactivation of an identifier does not strip it of its value, nor does it prevent it from receiving new values in subsequent preprocessor statements. Deactivation simply prevents any replacement for a particular identifier from taking place. Deactivation of a deactivated preprocessor identifier has no effect.

%DECLARE

Abbreviation: %DCL

The %DECLARE statement establishes an identifier as a preprocessor variable or a preprocessor procedure name and also serves to activate that identifier. An identifier must appear in a %DECLARE statement before it can be used as a variable or a procedure name in any other preprocessor statement.

General format:

```
%([label:]...
  DECLARE identifier
   {FIXED|CHARACTER|ENTRY|BUILTIN}
  [,identifier
   {FIXED|CHARACTER|ENTRY|BUILTIN}]...;
```

Syntax rules:

1. CHARACTER or FIXED must be specified if the "identifier" is a preprocessor variable; an entry declaration may be optionally specified if the "identifier" is a preprocessor procedure name. The declaration of a preprocessor procedure entry name can be performed explicitly by its appearance as the label of a %PROCEDURE statement. This explicit declaration however, does not cause the activation of the preprocessor procedure name.

2. Only the attributes shown in the above format can be specified in a %DECLARE statement.

3. Factoring of attributes is allowed as for nonpreprocessor DECLARE statements.

4. Any label attached to a %DECLARE statement is ignored by the scan.

General rules:

1. No length can be specified with the CHARACTER attribute. If CHARACTER is specified, it is assumed that the associated identifier represents a varying-length character string that has no maximum length.

2. A preprocessor variable declared with the attribute FIXED is also given the attributes DECIMAL and (5,0) by default.

3. A preprocessor declaration is not known until it has been encountered by the scan. If a reference to a preprocessor variable or procedure is encountered in a preprocessor statement before the declaration for that variable or procedure has been scanned, then the reference is in error.

4. The scope of all preprocessor variables, procedure names, and labels is the entire source program scanned by the preprocessor, not including any preprocessor procedures that redeclare such identifiers. The scope of a declaration in a preprocessor procedure is limited to that procedure.

5. An entry declaration may be specified for each preprocessor procedure in the source program. It is used to activate the entry name. Each time a preprocessor function is invoked, its arguments are converted if necessary to the attributes of the corresponding parameters.

   See "Preprocessor Procedures" in "Compile-Time Facilities" in Part I, for a discussion of the association of arguments and parameters at the time of invocation.

6. A preprocessor %DECLARE statement behaves as a %ACTIVATE statement when it is encountered, and activates,

with the RESCAN option, all preprocessor variables identified in the statement.

7. The BUILTIN attribute may only be specified for SUBSTR, LENGTH, or INDEX. It indicates that the associated identifier is the built-in function of the same name.

%DO

The %DO statement is used in conjunction with a %END statement to delimit a preprocessor DO-group. It cannot be used in any other way.

General format:

```
%[label:]...DO [ i=m1[TO m2 [BY m3]]
               [    [BY m3 [TO m2]]];
```

Syntax rule:

The "i" represents a preprocessor variable, and "m1," "m2," and "m3" are preprocessor expressions.

General rule:

The expansion of a preprocessor DO-group is the same as the expansion for a corresponding nonpreprocessor DO-group and "i," "m1," "m2," and "m3" have the same meaning that the corresponding expressions in a nonpreprocessor DO-group have.

See "Preprocessor DO-Groups" in chapter 16, "Compile-Time Facilities", for a discussion and an example of its use.

%END

The %END statement is used in conjunction with %DO or %PROCEDURE statements to delimit preprocessor DO-groups or preprocessor procedures.

General format:

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

Syntax rule:

The label following END must be a label of a %PROCEDURE or %DO statement. Multiple closure is permitted.

%GO TO

Abbreviation: GOTO

The %GO TO statement causes the preprocessor to continue its scan at the specified label.

General format:

```
% [label:]...  GO TO label;
```

General rules:

1. The label following the keyword GO TO determines the point to which the scan will be transferred. It must be a label of a preprocessor statement, although it cannot be the label of a preprocessor procedure.

2. A preprocessor GO TO statement appearing within a preprocessor procedure cannot transfer control to a point outside of that procedure. In other words, the label following GO TO must be contained within the procedure.

3. See "%INCLUDE Statement" for a restriction regarding the use of %GO TO with included text.

%IF

The %IF statement can control the flow of the scan according to the value of a preprocessor expression.

General format:

```
%[label:]...IF
preprocessor-expression
        %THEN preprocessor-clause-1
        [%ELSE preprocessor-clause-2]
```

Syntax rule:

A preprocessor clause is any single preprocessor statement other than %DECLARE, %PROCEDURE, %END, or %DO (percent symbol included) or a preprocessor DO-group (percent symbols included). Otherwise, the syntax is the same as that for non-preprocessor IF statements.

General rules:

1. The preprocessor expression is evaluated and converted to a bit string (if the conversion cannot be made, it is an error). If any bit in the string has the value 1, clause-1

is executed and clause-2, if present, is ignored; if all bits are 0, clause-1 is ignored and clause-2, if present, is executed. In either case, the scan resumes immediately following the IF statement, unless, of course, a %GO TO in one of the clauses causes the scan to resume elsewhere.

2. %IF statements can be nested according to the rules for nesting nonpreprocessor IF statements.

%INCLUDE

The %INCLUDE statement is used to include (incorporate) strings of external text into the source program being scanned. This included text can contribute to the preprocessed text being formed.

General format:

The %INCLUDE statement is defined as follows for these compilers:

```
%[label:]... INCLUDE

    {ddname-1 (member-name-1)}
    {member-name-1           }

    [,ddname-2 (member-name-2)]
    [,member-name-2           ]  ...;
```

Syntax rules:

1. Each "ddname" and "member name" pair identifies the external text to be incorporated into the source program. This external text must be a member of a partitioned data set.

2. A "ddname" specifies the ddname occurring in the name field of the appropriate DD statement. Its associated "member name" specifies the name of the data set member to be incorporated. If "ddname" is omitted, SYSLIB is assumed, and the SYSLIB DD statement is required.

3. A %INCLUDE statement cannot be used in a preprocessor procedure.

General rules:

1. Included text can contain nonpreprocessor and/or preprocessor statements. Its maximum permissible length is 100 characters.

2. The included text is scanned, in sequence, in the same manner as the source program; that is, preprocessor statements are executed and replacements are made where required.

3. %INCLUDE statements can be nested. In other words, included text also can contain %INCLUDE statements. A %GO TO statement in included text can transfer control to a point in the source program or in any included text at an outer level of nesting, but the reverse is not permitted. An analogous situation exists for nested DO-groups that specify iterative execution: control can be transferred from an inner group to an outer, containing group, but not from an outer group into an inner, contained group. The maximum permissible depth of nesting is 49.

4. Preprocessor statements in included text must be complete. It is not permissible, for example, to have half of a %IF statement in included text and half in the other part of the source program.

If the source program contained the following sequence of statements:

```
%DECLARE (FILENAME1, FILENAME2)
         CHARACTER;

% FILENAME1 = 'MASTER';

% FILENAME2 = 'NEWFILE';

% INCLUDE DCLS;
```

and if the SYSLIB member name DCLS contained:

```
DECLARE (FILENAME1, FILENAME2)
        FILE RECORD INPUT
        DIRECT KEYED ENVIRONMENT
        (REGIONAL(3) KEYLENGTH(8) F
        RECSIZE(80));
```

then the following would be inserted into the preprocessed text:

```
DECLARE (MASTER, NEWFILE)
        FILE RECORD INPUT
        DIRECT KEYED ENVIRONMENT
        (REGIONAL(3) KEYLENGTH(8) F
        RECSIZE(80));
```

Note that this is a way in which a central library of file declarations can be used, with each user supplying his own names for the files being declared.

%Null Statement

The %null statement can be used to provide transfer targets for %GO TO statements. It is also useful for balancing ELSE clauses in nested %IF statements.

General format:

```
% [label:]...;
```

%PROCEDURE

Abbreviation: %PROC

The %PROCEDURE statement is used in conjunction with a %END statement to delimit a preprocessor procedure. Such a preprocessor procedure is an internal function procedure that can be executed only at the preprocessor stage.

General format:

```
% label: [label:]... PROCEDURE
     [(identifier [, identifier]...)]
     RETURNS{CHARACTER|FIXED};
```

Syntax rules:

1. Each "identifier" is a parameter of the function procedure; a maximum of 15 may be specified.
2. One of the attributes CHARACTER or FIXED must be specified in the RETURNS attribute list to indicate the type of value returned by the function procedure. There can be no default.

General rules:

1. The only statements and groups that can be used within a preprocessor procedure are:
   a. the preprocessor assignment statement
   b. the preprocessor DECLARE statement
   c. the preprocessor DO-group
   d. the preprocessor GO TO statement
   e. the preprocessor IF statement
   f. the preprocessor null statement
   g. the preprocessor RETURN statement

   All of these statements and the DO-group must adhere to the syntax and general rules given for them in this section, with one exception; all percent symbols must be omitted.
2. A GO TO statement appearing in a preprocessor procedure cannot transfer control to a point outside of that procedure.
3. As implied by general rule 1, preprocessor procedures cannot be nested.
4. A preprocessor procedure can be invoked by a function reference in a preprocessor statement, or, if the function procedure name is active, by the encounter of that name in a nonpreprocessor statement.

Preprocessor RETURN

The preprocessor RETURN statement can be used only in a preprocessor procedure and, therefore, can have no leading %. It returns a value as well as control back to the point from which the preprocessor procedure was invoked.

General format:

```
[label:]... RETURN
    (preprocessor-expression);
```

General rule:

The value of the preprocessor expression is converted to the RETURNS list of attribute specified in the %PROCEDURE statement before it is passed back to the point of invocation. If the point of invocation is in a nonpreprocessor statement, replacement activity can be performed on the returned value after that value has replaced the procedure reference.

Note that the rules for preprocessor expressions do not permit the value returned by a preprocessor procedure to contain preprocessor statements.

Listing Control Statements

%CONTROL

The checkout compiler FORMAT option, when specified, may be activated and deactivated by the %CONTROL statement. Under the optimizing compiler, the syntax of the statement is checked, then it is ignored.

General format:

```
%CONTROL(FORMAT|NOFORMAT);
```

Syntax rules:

1. To influence formatting of a listing, the statement must be on a line with no other statements.
2. The statement will have no effect on the format if it appears within a comment or another statement.

General rules:

1. The %CONTROL statement has no effect if the FORMAT compiler option has not been specified.
2. The FORMAT compiler option is nullified if more %CONTROL statements have been executed with the NOFORMAT option than with the FORMAT option: the result is as if the FORMAT option had not been specified. In all other cases, the %CONTROL statement has no effect on the format.
3. The statement may be used with or without the preprocessor.
4. The %CONTROL statement is printed in the formatted listing. It is also retained in the text passed to the compiler, but is ignored by the compiler.
5. If the preprocessor is used, and a %CONTROL statement is written on the same line as one or more other statements, the preprocessor moves the %CONTROL so that it is on a line of its own in the text passed to the compiler.

%PAGE

The statement following a %PAGE statement in the program listing is printed on the first line of the next page.

General format:

```
%PAGE;
```

Syntax rules:

1. To cause formatting to take place, the statement must be on a line with no other statements.
2. The statement will have no formatting effect if it appears within a comment or another statement.

General rules:

1. The statement may be used with or without the preprocessor. It will control both the insource and the source listing.
2. After being put into effect, the %PAGE; is not printed by the preprocessor and is deleted from the text by the compiler; it does not appear in the formatted listing.
3. If the preprocessor is used, and a %PAGE statement is written on the same line as one or more other statements, the preprocessor moves the %PAGE so that it is on a line of its own in the text passed to the compiler. The insource listing is therefore not formatted, but the source listing is.
4. When the preprocessor is used, an identifier that is split across the end of a line that contains a %PAGE statement is concatenated to form one word. The second part of the word is moved onto the same line as the first part if there is sufficient space on that line, otherwise the concatenated word is printed at the start of a new line.

%SKIP

The specified number of lines following a %SKIP statement in the program listing are left blank.

General Format

```
%SKIP[(n)];
```

Syntax rules:

1. To cause formatting to take place, the statement must be on a line with no other statements.

2. The statement will have no formatting effect if it appears within a comment or another statement.

3. n must be a decimal integer constant in the range 1 through 999. Omission of the option is equivalent to specifying the value 1 for n.

General rules:

1. The statement may be used with or without the preprocessor. It will control both the insource and the source listing.

2. After being put into effect, the %SKIP statement is not printed by the preprocessor and is deleted from the text by the compiler; it does not appear in the formatted listings.

3. If the preprocessor is used, and a %SKIP statement is written on the same line as one or more other statements, the preprocessor moves the %SKIP so that it is on a line of its own in the text passed to the compiler. The insource listing is therefore not formatted, but the source listing is.

4. When the preprocessor is used, an identifier that is split across the end of a line that contains a %SKIP statement is concatenated to form one word. The second part of the word is moved onto the same line as the first part if there is sufficient space on that line, otherwise the concatenated word is printed at the start of a new line.

5. If n is greater than the number of lines remaining on the page, the equivalent of a %PAGE statement is executed in place of the %SKIP statement.

Section K: Data Mapping

This section describes structure mapping and alignment of records in buffers. The information is included because, under certain circumstances, it should be borne in mind when a program is being written. However, the information is not essential to programmers using stream-oriented transmission or unaligned data (other than bit strings); it is intended for those using record-oriented transmission (particularly locate mode) with aligned structures.

Structure Mapping

For any structure (major or minor), the length, alignment requirement, and position relative to a doubleword boundary will depend on the lengths, alignment requirements, and relative positions of its members. The process of determining these requirements for each level in turn and finally for the complete structure, is known as structure mapping.

During the structure mapping process, the compiler minimizes the amount of unused storage (padding) between members of the structure. It completes the entire process before the structure is allocated, according (in effect) to the rules discussed in the following paragraphs. It is necessary for the user to understand these rules for such purposes as determining the record length required for a structure when record-oriented input/output is used, and for determining the amount of padding or rearrangement required to ensure correct alignment of a structure for locate-mode input/output (see "Record Alignment", in this section).

Structure mapping is not a physical process. Although during this discussion such terms as "shifted" and "offset" are used, these terms are used purely for ease of discussion, and do not imply actual movement in storage; when the structure is allocated, the relative locations are already known as a result of the mapping process.

RULES

The mapping for a complete structure reduces to successively combining pairs of items (elements, or minor structures whose individual mappings have already been determined). Once a pair has been combined, it becomes a unit to be paired with another unit, and so on until the complete structure has been mapped. The rules for the process are therefore categorized as:

Rules for determining the order of pairing

Rules for mapping one pair

These rules are described below, and the example at the end of this section shows an application of the rules in detail.

Note: To follow these rules, it is necessary to appreciate the difference between logical level and level number. The item with the greatest level number is not necessarily the item with the deepest logical level. If the structure declaration is written with consistent level numbers or suitable indentation (as in the detailed example given after the rules), the logical levels are immediately apparent. In any case, the logical level of each item in the structure can be determined by applying the following rule to each item in turn, starting at the beginning of the structure declaration:

The logical level of a given item is always one unit deeper than that of the most immediate of its containing structures.

For example:

```
DCL 1 A, 4 B, 5 C, 5 D, 3 E, 8 F, 7 G;

    1   2   3   3   2   3   3
```

The lower line shows the logical level for each item in the declaration.

Rules for Order of Pairing

The steps in determining the order of pairing are as follows:

1. Find the minor structure with the deepest logical level (which we will call logical level n).
2. If the number of minor structures at logical level n exceeds one, take the

first one of them as it appears in the declaration.

3. Using the rules for mapping one pair (see below), pair the first two elements appearing in this minor structure, thus forming a unit.

4. Pair this unit with the next element (if any) appearing in the declaration for the minor structure, thus forming a larger unit.

5. Repeat rule 4 until all the elements in the minor structure have been combined into one unit. This completes the mapping for this minor structure; its alignment requirement and length, including any padding, are now determined and will not change (unless the programmer changes the structure declaration). Its offset from a doubleword boundary will also have been determined; note that this offset will be significant during mapping of any containing structure, and it may change as a result of such mapping.

6. Repeat rules 3 through 5 for the next minor structure (if any) appearing at logical level n in the declaration.

7. Repeat rule 6 until all minor structures at logical level n have been mapped. Each of these minor structures can now be thought of as an element for structure mapping purposes.

8. Repeat the process for minor structures at the next higher logical level; that is, make n equal to (n - 1) and repeat rules 2 through 7.

9. Repeat rule 8 until n = 1; then repeat rules 3 through 5 for the major structure.

Rules for Mapping One Pair

(As stated earlier, terms apparently implying physical storage are used here only for ease of discussion; the storage thus implied may be thought of as an imaginary model consisting of a number of contiguous doublewords. Each doubleword has eight bytes numbered zero through 7, so that the offset from a doubleword boundary can be given; in addition, the bytes in the model may be numbered continuously from zero onwards, starting at any byte, so that lengths and offsets from the start of a structure can be given.)

1. Begin the first item of the pair on a doubleword boundary; or, if the item is a minor structure that has already been mapped, offset it from the doubleword boundary by the amount indicated.

2. Begin the other item of the pair at the first valid position following the end of the first item. This position will depend on the alignment requirement of the second item. Alignment and length requirements for elements are given in figure K.1 and K.2. (If the item is a minor structure, its alignment requirement will have been determined already.)

3. Shift the first item towards the second item as far as the alignment requirement of the first item will allow. The amount of shift determines the offset of this pair from a doubleword boundary.

After this process has been completed, any padding between the two items will have been minimized and will remain unchanged throughout the rest of the operation. The pair can now be considered to be a unit of fixed length and alignment requirement; its length is the sum of the two lengths plus padding, and its alignment requirement is the higher of the two alignment requirements (if they differ).

Effect of UNALIGNED Attribute

The example of structure mapping given below shows the rules applied to a structure declared ALIGNED, because mapping of aligned structures is more complex owing to the number of different alignment requirements. The general effect of the UNALIGNED attribute is to reduce fullword and doubleword alignment requirements down to byte, and to reduce the alignment requirement for bit strings from byte down to bit. The same structure mapping rules apply, but the reduced alignment requirements are used. This means that unused storage between items can only be bit padding within a byte, and never a complete byte; bit padding may occur when the structure contains bit strings.

Variable Type	Stored Internally as	Storage Requirements (in Bytes)	Alignment Requirements	Explanation
BIT (n)	One byte for each group of 8 bits (or part thereof)	CEIL(n/8)	Byte	Data may begin on any byte 0 through 7
CHARACTER (n)	One byte per character	n		
PICTURE	One byte for each PICTURE character (except V, K, and the F scaling factor specification)	Number of PICTURE characters other than V, K, and F specification		
DECIMAL FIXED (p,q)	Packed decimal format (1/2 byte per digit, plus 1/2 byte for sign)	CEIL((p+1)/2)		
BIT(n) VARYING	Two-byte prefix plus one byte for each group of 8 bits (or part thereof)	2+CEIL(n/8)	Halfword	
CHARACTER(n) VARYING	Two-byte prefix plus one byte per character	2+n		
BINARY FIXED (p,q) p < =15	Halfword binary integer	2		Data may begin on byte 0,2, 4 or 6
p > 15	Fullword binary integer	4	Full word	Data may begin on byte 0 or 4 only
BINARY FLOAT (p) p < 22	Short floating-point			
DECIMAL FLOAT (p) p < 7				
POINTER[1]	-			
OFFSET[1]	-			
FILE	-			
ENTRY		8		
LABEL	-			
TASK	-	16		
EVENT	-	32		

Figure K.1 (Part 1 of 2). Summary of alignment requirements for ALIGNED data

Variable Type	Stored Internally as	Storage Requirements (in Bytes)	Alignment Requirements	Explanation
BINARY FLOAT (p) 21 < p < 54 DECIMAL FLOAT (p) 6 < p < 17	Long floating-point	8	Double word	Data may begin on byte 0 only
BINARY FLOAT(p) 53 < p < 110 DECIMAL FLOAT(p) 16 < p < 34	Extended floating-point	16		
AREA	-	16+size		

*Locators (pointers and offsets) used in programs processed by the checkout compiler can be 4 or 16 bytes long. The mapping of four-byte locators is described here; the mapping of 16-byte locators is identical except for the extra storage requirement.

Figure K.1 (Part 2 of 2). Summary of alignment requirements for ALIGNED data

TASK, EVENT and AREA data cannot be unaligned. If a structure has the UNALIGNED attribute and it contains an element that cannot be unaligned, then UNALIGNED is ignored for that element; the element is aligned by the compiler and an error message is put out. For example, in a program with the declaration

```
DECLARE 1 A UNALIGNED,
          2 B,
          2 C AREA(100);
```

C is given the attribute ALIGNED, as the inherited attribute UNALIGNED conflicts with AREA.

Variable Type	Stored Internally as	Storage Requirements (in Bytes)	Alignment Requirements	Explanation
BIT (n)	As many bits as are required, regardless of byte boundaries	n bits	Bit	Data may begin on any bit in any byte 0 through 7
CHARACTER (n)	One byte per character	n	Byte	Data may begin on any byte 0 through 7
PICTURE	One byte for each PICTURE character (except V or K)	Number of PICTURE characters other than V or K		
BIT(n) VARYING	Two-byte prefix plus one byte for each group of 8 bits (or part thereof)	2 bytes + n bits		
CHARACTER(n) VARYING	Two-byte prefix plus one byte per character	2+n		
DECIMAL FIXED(p,q)	Packed decimal format (1/2 byte per digit, plus 1/2 byte for sign)	CEIL((p+1)/2)		
BINARY FIXED (p,q) p < = 15	Halfword binary integer	2		
p > 15	Fullword binary integer	4		
BINARY FLOAT (p) p < 22	Short floating-point			
DECIMAL FLOAT (p) p < 7				
POINTER	-			
OFFSET	-			
FILE	-			

Figure K.2 (Part 1 of 2). Summary of alignment requirements for UNALIGNED data

ENTRY	-			
LABEL	-			
BINARY FLOAT (p) 21 < p < 54 DECIMAL FLOAT (p) 6 < p < 17	Long floating-point	8		
BINARY FLOAT(p) 53 < p <110 DECIMAL FLOAT(p) 16 < p < 34	Extended floating-point	16		

Note: TASK, EVENT, and AREA data cannot be UNALIGNED. A pointer or offset can be 4 or 16 bytes long (see figure K.1).

Figure K.2 (Part 2 of 2). Summary of alignment requirements for UNALIGNED data

Example of Structure Mapping

This example shows the application of the structure mapping rules for a structure declared as follows:

```
DECLARE 1 A ALIGNED,
          2 B POINTER,
          2 C,
            3 D FLOAT DECIMAL(14),
            3 E,
              4 F LABEL,
              4 G,
                5 H CHARACTER(2),
                5 I FLOAT DECIMAL(13),
              4 J FIXED BINARY(31,0),
            3 K CHARACTER(2),
            3 L FIXED BINARY(20,0),
          2 M,
            3 N,
              4 P FIXED BINARY(5),
              4 Q CHARACTER(5),
              4 R FLOAT DECIMAL(2),
            3 S,
              4 T FLOAT DECIMAL(15),
              4 U BIT(3),
              4 V CHAR(1),
            3 W POINTER,
          2 X PICTURE '$9V99';
```

The minor structure at the deepest logical level is G, so that this is mapped first. Then E is mapped, followed by N, S, C, and M, in that order. Finally, the major structure A is mapped. For each structure, a table is given showing the steps in the process, accompanied by a diagram giving a visual interpretation of the process. At the end of the example, the structure map for A is set out in the form of a table showing the offset of each member from the start of A.

	Name of Item	Alignment Requirement	Length	Offset from Doubleword		Length of Padding	Offset from G
				Begin	End		
Step 1	H	Byte	2	0	1		
	I	Double	8	0	7		
Step 2	*H	Byte	2	6	7		0
	I	Double	8	0	7	0	2
	G	Double	10	6	7		

*First item shifted right

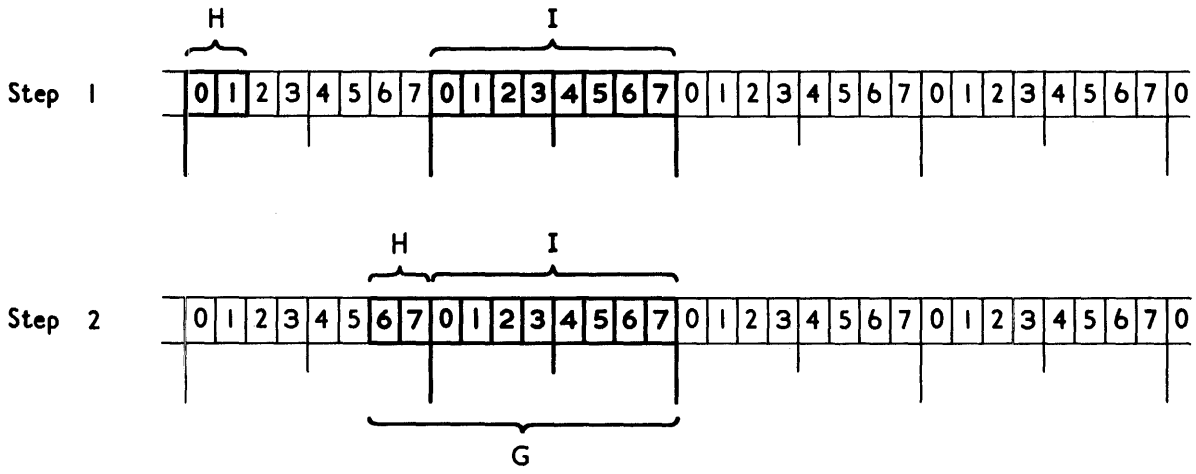


Figure K.3. Mapping of minor structure G

	Name of Item	Alignment Requirement	Length	Offset from Doubleword		Length of Padding	Offset from E
				Begin	End		
Step 1	F	Word	8	0	7		
	G	Double	10	6	7		
Step 2	*F	Word	8	4	3		0
	G	Double	10	6	7	2	10
Step 3	F through G	Double	20	4	7		
	J	Word	4	0	3	0	20
	E	Double	24	4	3		

*First item shifted right

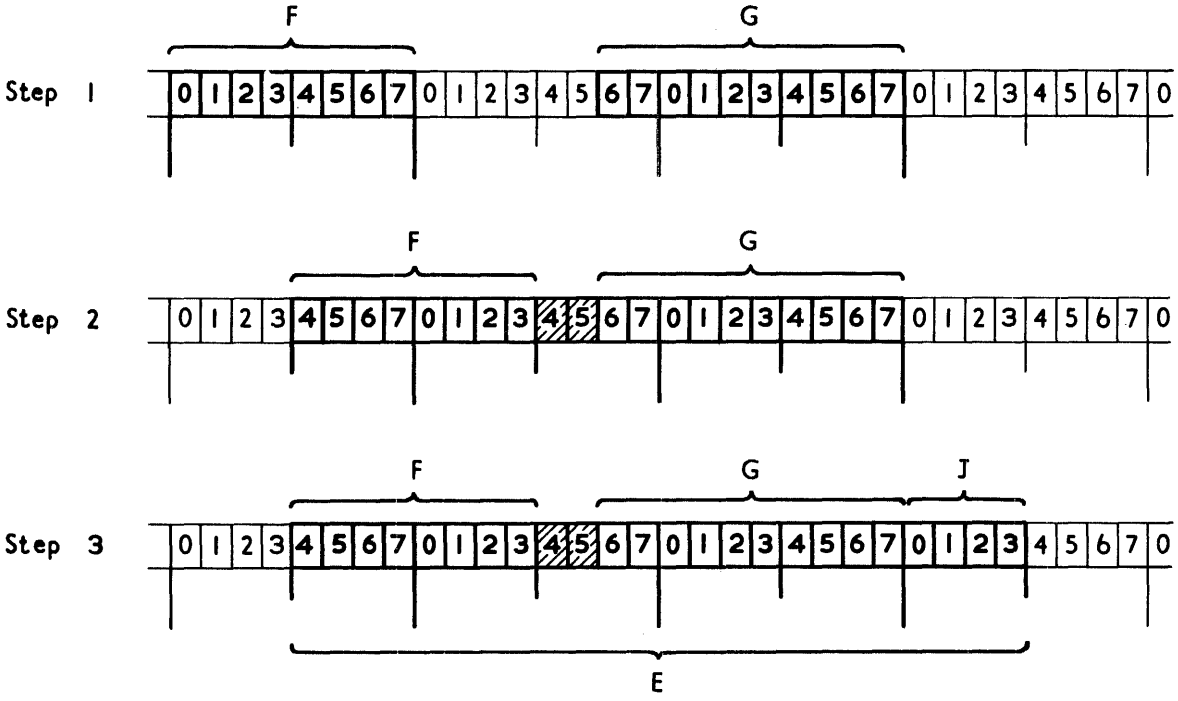


Figure K.4. Mapping of minor structure E

	Name of Item	Alignment Requirement	Length	Offset from Doubleword		Length of Padding	Offset from N
				Begin	End		
Step 1	P	Halfword	2	0	1		0
	Q	Byte	5	2	6		2
Step 2	P through Q	Halfword	7	0	6		
	R	Word	4	0	3	1	8
	N	Word	12	0	3		

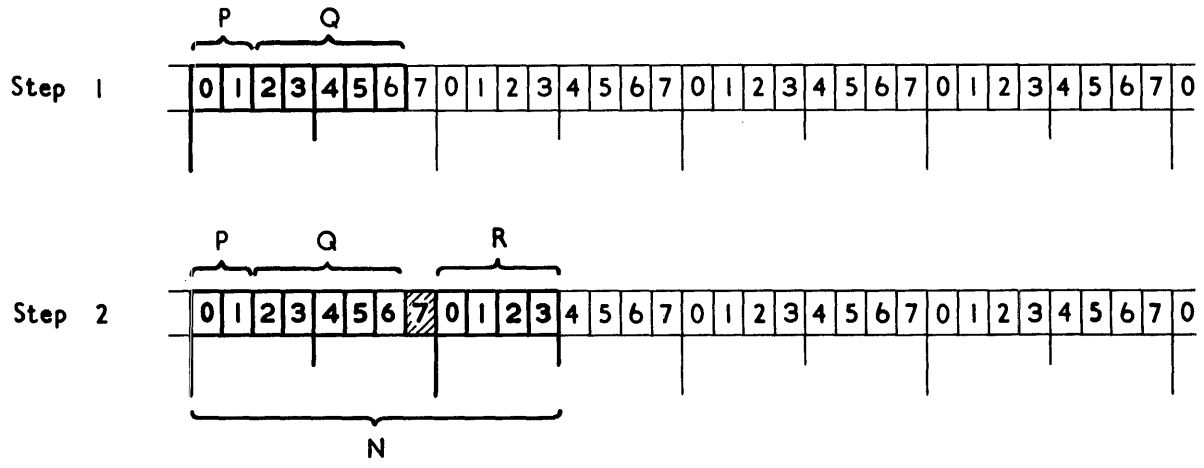
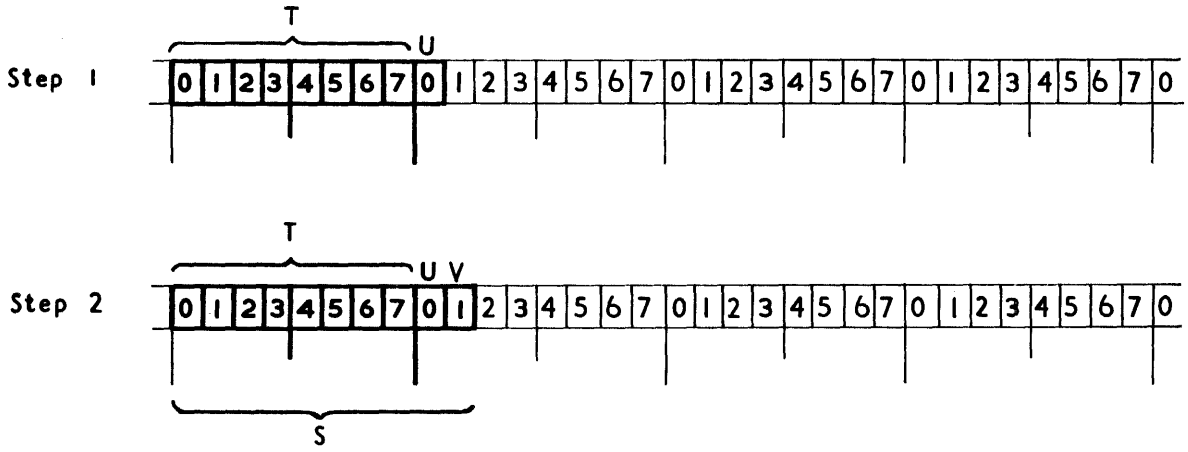


Figure K.5. Mapping of minor structure N

	Name of Item	Alignment Requirement	Length	Offset from Doubleword		Length of Padding	Offset from S
				Begin	End		
Step 1	T	Double	8	0	7		0
	U	Byte	1	0	0	0	8
Step 2	T through U	Double	9	0	0		
	V	Byte	1	1	1	0	9
	S	Double	10	0	1		



• Figure K-6. Mapping of Minor Structure S

	Name of Item	Alignment Requirement	Length	Offset from Doubleword		Length of Padding	Offset from C
				Begin	End		
Step 1	D	Double	8	0	7		0
	E	Double	24	4	3	4	12
Step 2	D through E	Double	36	0	3		
	K	Byte	2	4	5	0	36
Step 3	D through K	Double	38	0	5		
	L	Word	4	0	3	2	40
	C	Double	44	0	3		

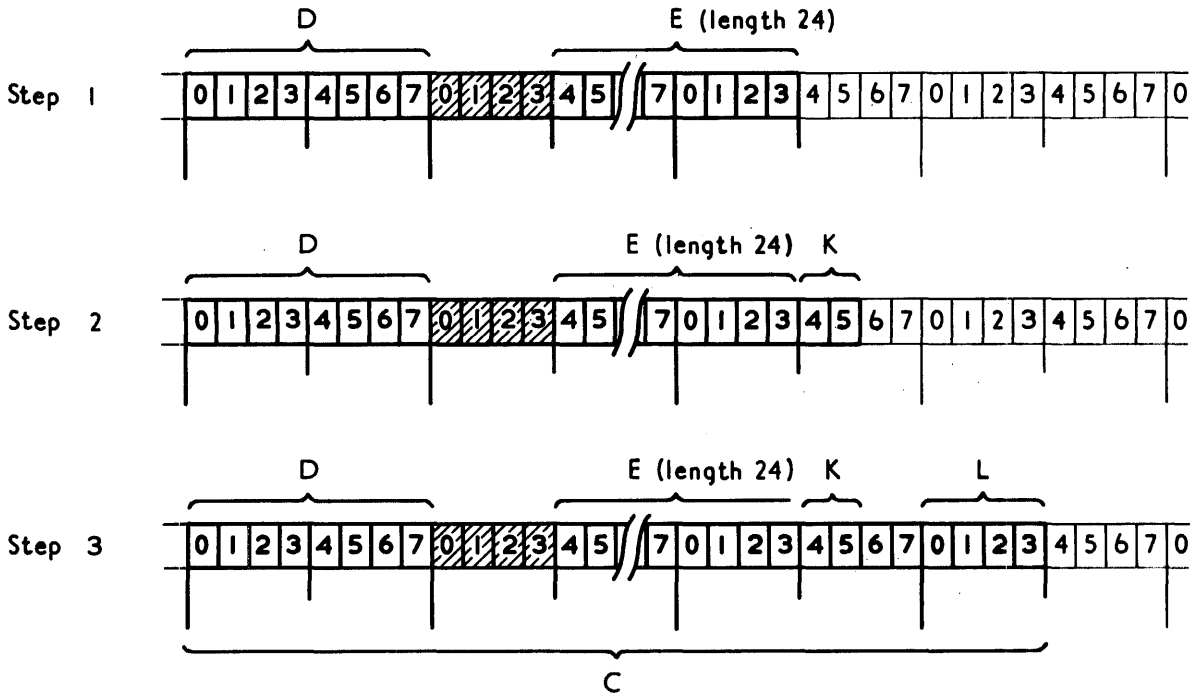


Figure K.7. Mapping of minor structure C

	Name of Item	Alignment Requirement	Length	Offset from Doubleword		Length of Padding	Offset from M
				Begin	End		
Step 1	N	Word	12	0	3		
	S	Double	10	0	1		
Step 2	*N	Word	12	4	7		0
	S	Double	10	0	1	0	12
Step 3	N through S	Double	22	4	1		
	W	Word	4	4	7	2	24
	M	Double	28	4	7		

*First item shifted right

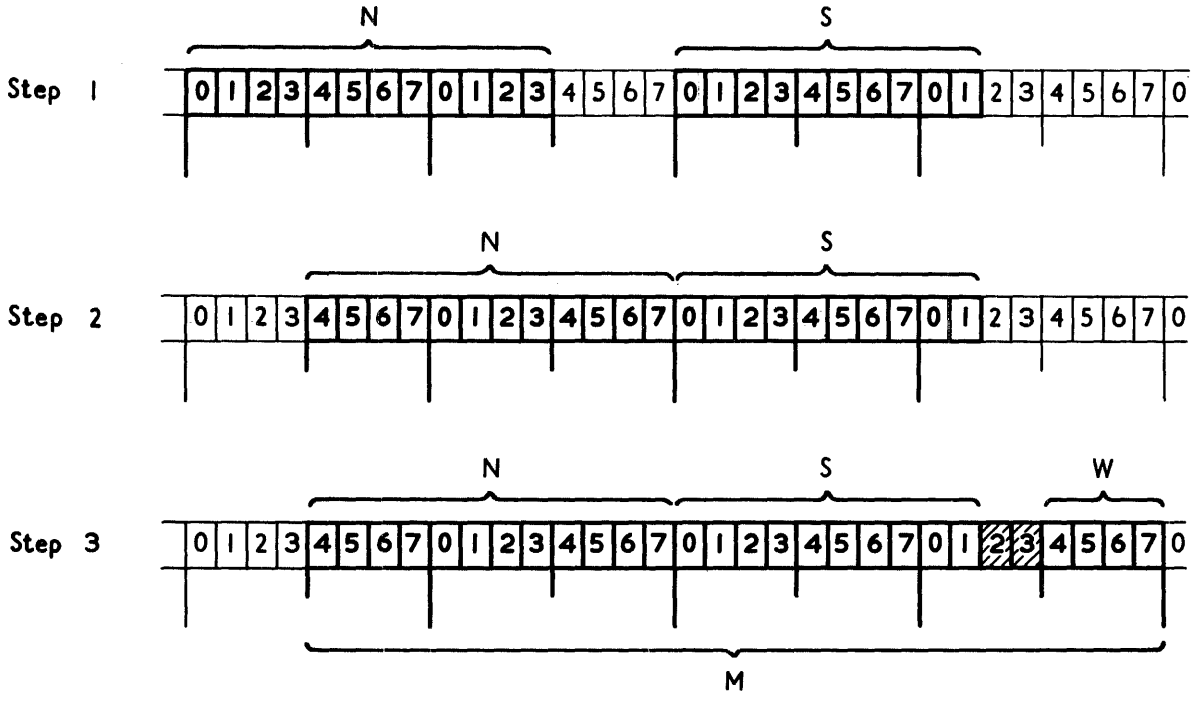


Figure K.8. Mapping of minor structure M

	Name of Item	Alignment Requirement	Length	Offset from Doubleword		Length of Padding	Offset from A
				Begin	End		
Step 1	B	Word	4	0	3		
	C	Double	44	0	3		
Step 2	*B	Word	4	4	7		0
	C	Double	44	0	3	0	4
Step 3	B through C	Double	48	4	3		
	M	Double	28	4	7	0	48
Step 4	B through M	Double	76	4	7		
	X	Byte	4	0	3	0	76
	A	Double	80	4	3		

*First item shifted right

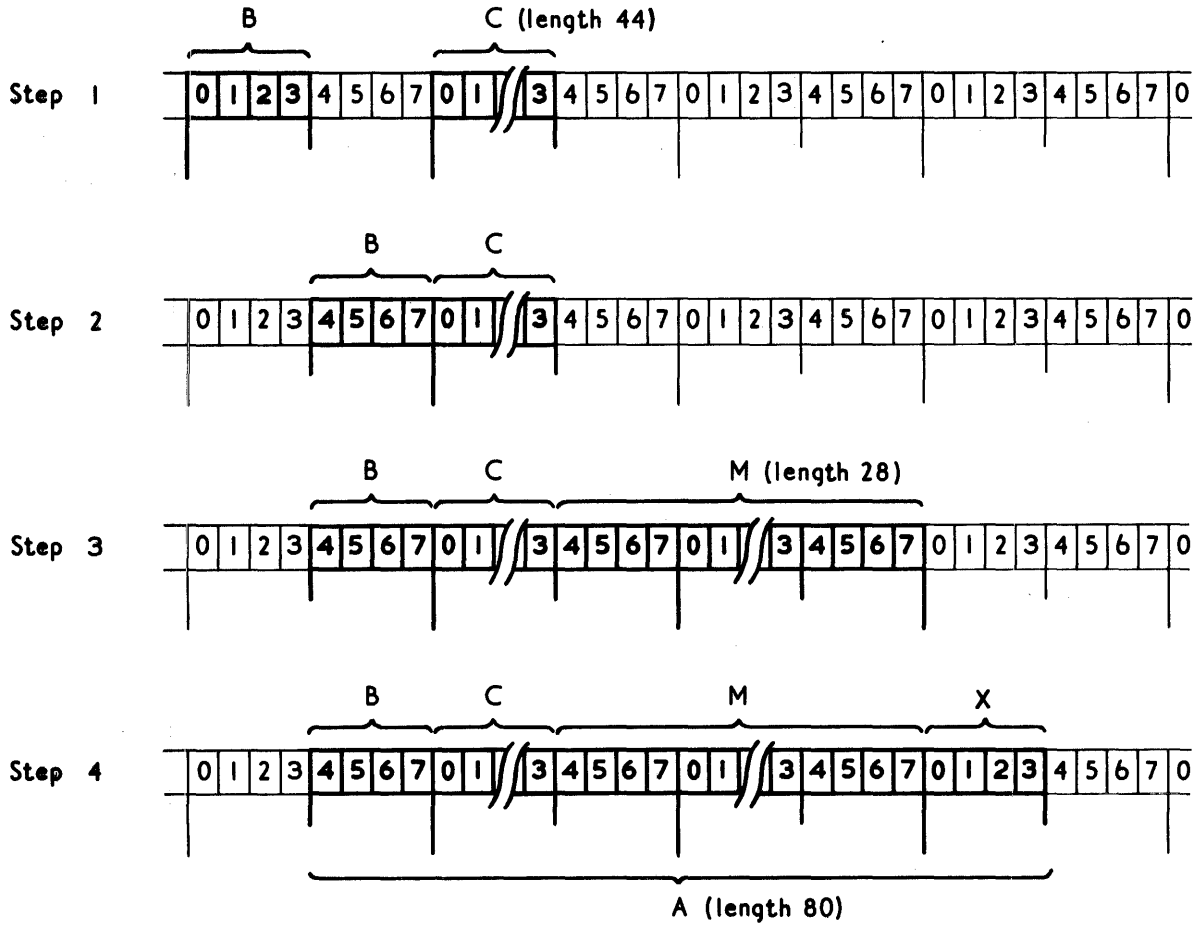


Figure K.9. Mapping of major structure A

A				From A
B				0
C			From C	4
D			0	4
padding (4)			8	12
E		From E	12	16
F		0	12	16
padding (2)		8	20	24
G	From G	10	22	26
H	0	10	22	26
I	2	12	24	28
J		20	32	36
K			36	40
padding (2)			38	42
L			40	44
M			From M	48
N		From N	0	48
P		0	0	48
Q		2	2	50
padding(1)		7	7	55
R		8	8	56
S		From S	12	60
T		0	12	60
U		8	20	68
V		9	21	69
padding (2)			22	70
W			24	72
X				76

Figure K.10. Offsets in final mapping of structure A

Record Alignment

The user must pay attention to record alignment within the buffer when using locate mode input/output. The first data byte of the first record in a block is generally aligned in a buffer on a doubleword boundary (see figure K.14); the next record begins at the next available byte in the buffer. The user must ensure that the alignment of this byte matches the alignment requirements of the based variable with which the record is to be associated.

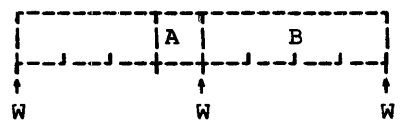
For blocked records, doubleword alignment of the first byte of data in each record in the block is ensured if the record length (RECSIZE) is a multiple of eight. For spanned records, the block size (BLKSIZE) must be a multiple of eight if this alignment is required. For data read from ASCII data sets, the first byte of the block prefix is doubleword-aligned; to ensure similar alignment of the first byte of the first record, the prefix length must be a multiple of eight bytes, less four to allow for the four record length bytes.

Most of the alignment problems described here occur in ALIGNED based or non-based variables. If these variables were UNALIGNED, the preservation of the record alignment in the buffer would be considerably easier.

If a VB-format record is to be constructed with logical records defined by the structure:

```
1 S,
  2 A CHAR(1),
  2 B FIXED BINARY(31,0);
```

this structure is mapped as in figure K.11.



W = Word boundary

Figure K.11. Format of structure S

If the block was created using a sequence of WRITE FROM(S) statements, the format of the block would be as in figure K.12, and it can be seen that the alignment in the buffer differs from the alignment of S.

There is no problem if the file is then read using move mode READ statements, e.g., READ INTO(S), because information is moved from the buffer to correctly aligned storage.

If, however, a structure is defined as:

```
1 SBASED BASED(P) LIKE S;
```

and READ SET(P) statements are used, reference to SBASED.B will, for the first record in the block, be to data aligned at a doubleword plus one byte, and will probably result in a specification interrupt.

The same problem would have arisen had the file originally been created by using the statement:

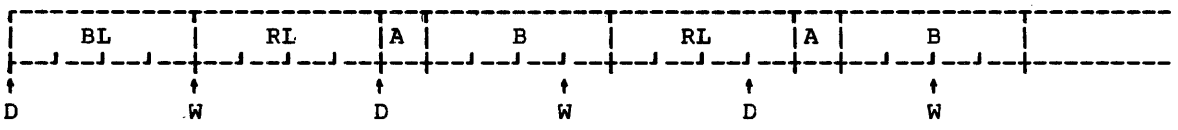
```
LOCATE SBASED SET(P);
```

Again, for the first record in the block, P would be set to address a doubleword and references to SBASED.B would be invalid.

In both cases the problem is avoided if the structure is padded in such a way that B is always correctly aligned:

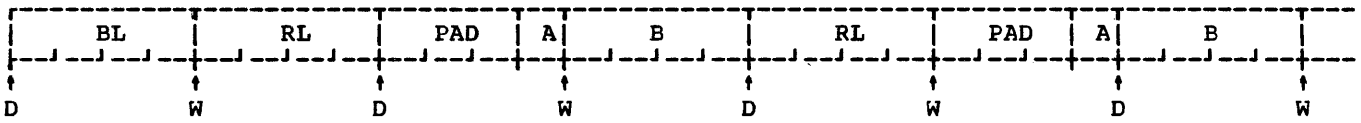
```
1 S,
  2 PAD CHAR(3),
  2 A CHAR(1),
  2 B FIXED BINARY;
```

The block format would now be as in Figure K-13; B is always on a word boundary. Padding may be required at the beginning and end of a structure to preserve alignment.



BL = Block length D = Doubleword boundary
RL = Record length W = Word boundary

Figure K.12. Block created from structure S



BL = Block length D = Doubleword boundary
RL = Record length W = Word boundary

Figure K.13. Block created by structure S with correct alignment

The alignment of different types of record within a buffer is shown in figure K.14. For all organizations and record types, except FB, V and VB records in INDEXED data sets with KEYLOC = 0 or unspecified, the first data byte in a block (or hidden buffer) is always on a doubleword boundary. The position of any successive records in the buffer depends on the record format.

For INDEXED data sets with unblocked F-format records, the LOCATE statement will use a hidden buffer if the data set key length is not a multiple of eight and the KEYLOC value is 1, 0 or is not specified (that is, RKP = 0). The pointer variable will point at this hidden buffer.

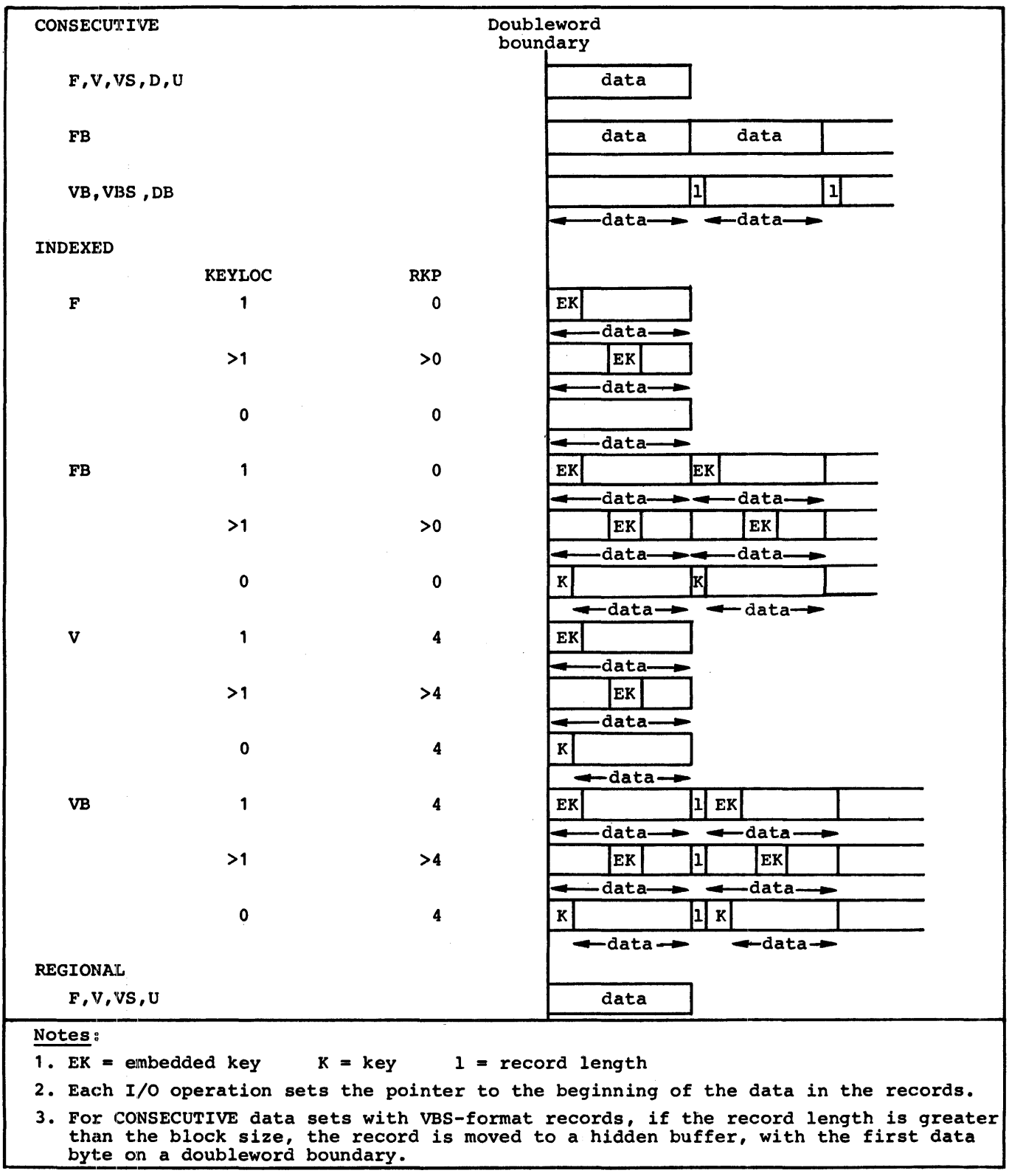
A special problem arises when using locate mode input/output in conjunction with a based variable containing adjustable extents, i.e., containing a REFER option. Consider the following structure:

```
1 S BASED(P),
  2 N,
  2 C CHAR (L REFER (N));
```

If it is desired to create blocked V-format records of this type, using locate mode input/output, record alignment must be such that N is half-word aligned. If L is not a multiple of 2 then, if the alignment of the current record is correct, that of the following record will be incorrect. Correct alignment can be obtained by the following sequence:

```
LENGTH = L;
      /* SAVE DESIRED LENGTH I */
L = 2* CEIL(L/2);
      /* ROUND UP TO MULTIPLE OF 2
*/
LOCATE S FILE (F);
N = LENGTH;
      /* SET REFER VARIABLE */
```

This technique can be adapted to other uses of the REFER option.



Notes:

1. EK = embedded key K = key l = record length
2. Each I/O operation sets the pointer to the beginning of the data in the records.
3. For CONSECUTIVE data sets with VBS-format records, if the record length is greater than the block size, the record is moved to a hidden buffer, with the first data byte on a doubleword boundary.

Figure K.14. Alignment of data in a buffer in locate mode input/output, for different formats and data set organizations

Section L: Compiler Differences

The tables in this section list the principal differences between the optimizing and checkout compilers.

Figure L.1 gives the differences that arise from the differing functions of the two compilers. There are, for instance, keywords concerned with the checkout and conversational facilities of the checkout compiler that are not implemented by the optimizing compiler, and optimization keywords that are not implemented by the checkout compiler.

Figures L.2 and L.3 show differences that do not arise directly from differing compiler functions. Figure L.2 contains general syntactic and semantic differences, and figure L.3 shows differing quantitative restrictions on the use of various facilities of the language.

The section is applicable only to error-free programs processed in batch mode.

Language feature	Optimizing compiler implementation	Checkout compiler implementation
Statements: CHECK NOCHECK FLOW NOFLOW PUT SNAP PUT FLOW PUT ALL HALT %CONTROL	Syntax-check only	Implemented
ATTENTION condition	Syntax-check only	Implemented
Options: ORDER REORDER TOTAL	Implemented	Syntax-check only
Built-in subroutines: PLICKPT PLIREST PLICANC	Implemented	Syntax-check only
PUT DATA statement and CHECK prefix specifying program control data	Names of variables only transmitted	Names and values of variables transmitted
PUT LIST statement specifying program control data	Invalid	Values of variables transmitted
Lengths of pointer and offset variables	4 bytes	With COMPATIBLE compiler option: 4 bytes With NOCOMPATIBLE compiler option: 16 bytes
Oncodes	Certain codes not implemented (See list in section H, "On-conditions").	All oncodes implemented

Figure L.1. Differences resulting from differing compiler functions

Language feature	Optimizing compiler implementation	Checkout compiler implementation
Based variable in data-directed I/O and CHECK name-list	1 Must not be based on an offset variable. 2 Must not be a member of a structure containing the REFER option. 3. Must not be based on a pointer that is based, defined, or a parameter, or a member of an aggregate.	No corresponding rules
Defined variable in data-directed I/O and CHECK name-list	Must not be defined: 1 on a controlled variable. 2 on an array with one or more adjustable bounds. 3 with a POSITION attribute specifying other than a constant.	No corresponding rules
CHECK prefix specifying label of statement to which prefix is attached	CHECK raised for the label	CHECK not raised for the label
LIKE attribute specifying a minor structure that is contained in a major structure of which some other minor structure is declared with LIKE attribute	Not allowed	No corresponding rule
Area variable in an OFFSET attribute either in DECLARE statement or RETURNS attribute or option	Must be non-defined, unsubscripted, unqualified area name	No corresponding rule
Area variable in OFFSET attribute in parameter descriptor	Not allowed	No corresponding rule
Locator conversion (offset to pointer and vice versa)	1 If offset is a structure member, or if it appears in a DO statement or multiple assignment, the associated area must be a non-based, non-defined element variable. If the area is based, its locator must be an unsubscripted, non-based, non-defined pointer, and it must not be used to explicitly qualify the area in the offset declaration. 2 Locator conversion cannot be performed between argument and parameter: both must be either offset or pointer.	No corresponding rules

Figure L.2 (Part 1 of 2). Differing qualitative restrictions

Language feature	Optimizing compiler implementation	Checkout compiler implementation
Aggregate argument to generic entry name	Dummy argument cannot be be created	No corresponding rule
Parameter string length or area size specified as other than decimal integer constant	Length or size attribute assumed to match argument: dummy never created	Dummy created if length or size differs from argument
Attributes of entry argument and parameter differ in alignment only	No dummy argument	Dummy argument created
Event names in WAIT statement	Structure of event names not allowed	No corresponding rule
Pseudovariables: COMPLETION COMPLEX PRIORITY STRING	Not allowed as control variables for DO-groups	No corresponding rule
UNDEFINEDFILE condition in OPEN statement specifying more than one file name	Raised once, after attempting to open every file	Raised at each attempt to open a file that is undefined
Standard default files SYSIN and SYSPRINT	No corresponding rule	Used by compiler. Must not be declared with attributes conflicting with compiler requirements. SYSPRINT always open, therefore no new page started for program's output

Figure L.2 (Part 2 of 2). Differing qualitative restrictions

Language feature	Optimizing compiler implementation	Checkout compiler implementation
Maximum number of blocks in one compilation	255	No corresponding rule
Maximum level of nesting of blocks	50	No corresponding rule
Maximum number of active on-units	49 in any block 254 in any compilation	No corresponding rule
Maximum level of nesting of DO and IF statements	49	No corresponding rule
Maximum level of dependency in DECLARE statement	1	No corresponding rule
Maximum number of entries in list of constants in declaration of COBOL variable	125	No corresponding rule
Maximum level of locator qualification	Depends on storage available, but never less than 10	No corresponding rule
Maximum number of names in CHECK list	255	No corresponding rule
Maximum length of character-string picture data	Depends on storage available, but never less than 1023	32767

Figure L.3. Differing quantitative restrictions

access: the act that encompasses the references to and retrieval of data.

action specification: in an ON statement, the on-unit or the single keyword SYSTEM, either of which specifies the action to be taken whenever an interrupt results from raising of the named on-condition. The action specification can also include the keyword SNAP.

activate (a block): to initiate the execution of a block. A procedure block is activated when it is invoked at any of its entry points; a begin block is activated when it is encountered in normal flow of control, including a branch.

activation (of a block):

1. The process of activating a block.
2. The execution of a block.

activation (of a preprocessor variable or entry name): the establishment of the validity for replacement of the value of a variable or the returned value of an entry name. The first activation must be the result of the appearance of the name in a %DECLARE statement. If an active variable or entry name is made inactive by a %DEACTIVATE statement it may be activated again by a %ACTIVATE statement.

active:

1. The state of a block after activation and before termination.
2. The state in which a preprocessor variable or preprocessor entry name is said to be when its value can replace the corresponding identifier in source program text.
3. The state in which an event variable is said to be during the time it is associated with an asynchronous operation. An event variable remains active and, hence, cannot be associated with another operation until a WAIT statement specifying that event variable has been executed or, in the case of an event variable associated with a task, until an EXIT, RETURN, or END statement has caused termination of the task.
4. The state in which a task variable is said to be when its associated task is attached.
5. The state in which a task is said to be before it has been terminated.

additive attributes: attributes for which there are no defaults and which, if required, must always be added to the list of specified attributes or be implied (i.e., they have to be added to the set of attributes, if they are required).

address: a specific storage location at which a data item can be stored.

adjustable extent: bound (of an array), length (of a string), or size (of an area) that may be different for different generations of the associated variable. Adjustable bounds, lengths, and sizes are specified as expressions or asterisks (or by REFER options for based variables), which are evaluated separately for each generation. They cannot be used for static variables.

aggregate: see data aggregate.

aggregate expressions: an array expression or a structure expression.

alignment: the storing of data items in relation to certain machine-dependent boundaries.

allocated variable: a variable with which main storage has been associated and not freed.

allocation:

1. The reservation of main storage for a variable.
2. A generation of an allocated variable.

alphabetic character: any of the characters A through Z of the English alphabet and the alphabetic extenders #, $, and @ (which may have different graphic representation in different countries).

alphameric character: an alphabetic character or a digit.

alternative attribute: an attribute that may be chosen from a group of two or more alternatives. If none is specified, a default is assumed.

ambiguous reference: a reference that is not sufficiently qualified to identify one and only one name known at the point of reference.

ancestral task: the attaching task or any of the tasks in a direct line from the given task to, and including, the major task.

area: a declared portion of contiguous main storage identified by an area variable and reserved, on allocation, for the allocation of based variables.

area variable: a variable with the AREA attribute; its values may only be areas.

argument: an expression in an argument list as part of a procedure reference.

argument list: a parenthesized list of one or more arguments, separated by commas, following an entry-name constant, an entry-name variable, a generic name, or a built-in function name. The list is passed to the parameters of the entry point.

arithmetic constant: a fixed-point constant or a floating-point constant. Although most arithmetic constants can be signed, the sign is not part of the constant.

arithmetic conversion: the transformation of a value from one arithmetic representation to another.

arithmetic data: data that has the characteristics of base, scale, mode, and precision. It includes coded arithmetic data and pictured numeric character data.

arithmetic operators: either of the prefix operators + and -, or any of the following infix operators: + - * / **

arithmetic picture data: decimal picture data or binary picture data containing the following types of picture specification characters.

1. Decimal digit characters.
2. Zero-suppression characters.
3. Sign and currency symbol characters.
4. Insertion characters.
5. Commercial characters.
6. Exponent characters.

array: a named, ordered collection of data elements, all of which have identical attributes. An array has dimensions specified by the dimension attribute, and its individual elements are referred to by subscripts. An array can also be an ordered collection of identical structures.

array expression: an expression whose evaluation yields an array of values.

array of structures: an ordered collection of identical structures specified by giving the dimension attribute to a structure name.

assignment: the process of giving a value to a variable.

asynchronous operation: the overlap of an input/output operation with the execution of statements or the concurrent execution of procedures using multiple flows of control for different tasks.

attachment of a task: the invocation of a procedure and the establishment of a separate flow of control to execute the invoked procedure (and procedures it invokes) asynchronously with execution of the invoking procedure.

attention: an occurence, external to a task, that could cause an interrupt to the task.

attribute:

1. A descriptive property associated with a name to describe a characteristic of items that the name may represent.
2. A descriptive property used to describe a characteristic of the result of evaluation of an expression.

automatic storage allocation: the allocation of storage for automatic variables.

automatic variable: a variable that is allocated automatically at the activation of a block and released automatically at the termination of that block.

base: the number system in terms of which an arithmetic value is represented.

base element: the name of a structure member that is not a minor structure.

base item: the automatic, controlled, or static variable or the parameter upon which a defined variable is defined. The name may be qualified and/or subscripted.

based storage allocation: the allocation of storage for based variables.

based variable: a variable whose generations are identified by locator variables. A based variable can be used to refer to values of a variable of any storage class; it can also be allocated and freed explicitly by use of the ALLOCATE and FREE statements.

begin block: a collection of statements headed by a BEGIN statement and ended by an END statement that is a part of a program that delimits the scope of names and that is activated by normal sequential flow of control, including any branch resulting from a GO TO statement.

binary: the number system based on the number 2.

bit: a binary digit (0 or 1).

bit string: a string composed of zero or more bits.

bit-string operators: the logical operators ¬ (not), & (and), and | (or).

block: a begin block or procedure block.

block heading statement: the PROCEDURE or BEGIN statement that heads a block of statements.

bounds: the upper and lower limits of an array dimension.

buffer: intermediate storage, 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: a function that is supplied by the language.

call: (verb) to invoke a subroutine by means of the CALL statement or CALL option; (noun) such an invocation.

character set: a defined collection of characters. See language character set and data character set.

character string: a string composed of zero or more characters.

character-string picture data: data described by a picture specification which must have at least one A or X picture specification character.

closing (of a file): the dissociation of a file from a data set.

coded arithmetic data: arithmetic data that is stored in a form that is acceptable, without conversion, for arithmetic calculations.

comment: a string of zero or more characters used for documentation, that is preceded by /* and terminated by */ and which is a separator.

commercial character: the following picture specification characters;

1. CR (credit).
2. DB (debit).
3. T, I, and R, the overpunched-sign characters, which indicate that the associated position in the data item contains or may contain a digit with an overpunched sign and that this overpunched sign is to be considered in the character string value of the data item.

comparison operators: infix operators used in comparison expressions. They are ¬< (not less than), < (less than), <= (less than or equal to), ¬= (not equal to), = (equal to), >= (greater than or equal to), > (greater than), and ¬> (not greater than).

compile time: in general, the time during which a source program is translated into an object module. In PL/I, it is the time during which a source program can be altered (preprocessed), if desired, and then translated into an object program.

compile-time statements: see preprocessor statements.

complex data: arithmetic data, each item of which consists of a real part and an imaginary part.

composite operators: an 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 joins two strings in the order specified, 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: see on-conditions.

condition list: a list of one or more condition prefixes.

condition name: a language keyword (or CONDITION followed by a parenthesized programmer-defined name) that denotes an on-condition that might arise within a task.

condition prefix: a parenthesized list of one or more language condition names, prefixed to a statement. It specifies whether the named on-conditions are to be enabled.

connected reference: a reference to connected storage; it must be apparent, prior to execution of the program, that the storage is connected.

connected storage: main storage of an uninterrupted linear sequence of items that can be referred to by a single name.

constant: an arithmetic or string data item that does not have a name and whose value cannot change; an unsubscripted label prefix or a file name or an entry name.

contained text: all text in a procedure (including nested procedures) except its entry names and condition prefixes of the PROCEDURE statement; all text in a begin block except labels and condition prefixes of the BEGIN statement that heads the block. Internal blocks are contained in the external procedure.

contextual declaration: the appearance of an identifier that has not been explicitly declared, in a context that allows the association of specific attributes with the identifier.

control format item: a specification used in edit-directed transmission to specify positioning of a data item within the stream or printed page.

control variable: a variable used to control the iterative execution of a group. See iterative do-group.

controlled parameter: a parameter for which the CONTROLLED attribute is specified in a declare statement; it can be associated only with arguments that have the CONTROLLED attribute.

controlled storage allocation: the allocation of storage for controlled variables.

controlled variable: a variable whose allocation and release are controlled by the ALLOCATE and FREE statements, with access to the current generation only.

conversion: the transformation of a value from one representation to another to conform to a given set of attributes.

cross section of an array: the elements represented by the extent of at least one dimension (but not all dimensions) of an array. An asterisk in the place of a subscript in an array reference indicates the entire extent of that dimension.

current generation: that generation (of an automatic or controlled variable) currently available by reference to the name of the variable.

data: representation of information or of value in a form suitable for processing.

data aggregate: a logical collection of two or more data items that can be referred to either collectively or individually; an array or structure.

data character set: all of those characters whose representation is recognized by the computer in use.

data-directed transmission: the type of stream-oriented transmission in which data is transmitted as a group, comprising one or more items separated by commas or blanks, terminated by a semicolon, where each item is of the form

```
name = value
```

The name can be qualified and/or subscripted.

data format item: a specification used in edit-directed transmission to describe the representation of a data item in the stream.

data item: a single unit of data; it is synonymous with element.

data list: a parenthesized list of expressions or repetitive specifications, separated by commas, used in a stream-oriented input or output specification that represents storage locations to which data items are to be assigned during input or values which are to be obtained for output.

data set: a collection of data external to the program that can be accessed by the program by reference to a single file name.

data specification: the portion of a stream-oriented data transmission statement that specifies the mode of transmission (DATA, LIST, or EDIT) and includes the data list (or lists) and, for edit-directed mode, the format list (or lists).

data stream: data being transferred from or to a data set by stream-oriented transmission, as a continuous stream of data elements in character form.

data transmission: the transfer of data from a data set to the program or vice versa.

deactivated: the state in which a preprocessor variable or entry name is said to be when its value cannot replace the corresponding identifier in source program text.

decimal: the number system based on the number 10.

decimal digit character: the picture specification character 9.

decimal picture data: arithmetic picture data specified by picture specification characters containing the following types of picture specification characters:

1. Decimal digit characters.
2. The virtual point picture character.
3. Zero-suppression characters.
4. Sign and currency symbol characters.
5. Insertion characters.
6. Commercial characters.
7. Exponent characters.

declaration:

1. The establishment of an identifier as a name and the construction of a set of attributes (partial or complete) for it.
2. A source of attributes of a particular name.

default: the alternative attribute or option assumed, or specified for assumption by the DEFAULT statement, when no such attribute or option has been specified.

defined item: a variable declared to represent part or all of the same storage as that assigned to another variable known as the base item.

delimiter: all operators, comments, and the following characters: percent, parentheses, comma, period, semicolon, colon, assignment symbol, and blank; they define the limits of identifiers, constants, picture specifications, iSUBs, and keywords.

descriptor: see parameter descriptor.

digit: one of the characters 0 through 9.

dimensionality: the number of bounds specifications in an array declaration.

disabled: the state in which a particular on-condition will not result in an interrupt that would cause an on-unit for that condition to be entered.

do-group: a sequence of statements headed by a DO statement and ended by its corresponding END statement, used for control purposes.

do loop: see iterative do-group.

drifting-characters: see sign and currency symbol characters.

dummy argument: temporary storage that is created automatically to hold the value of an argument that cannot be passed by reference.

edit-directed transmission: the type of stream-oriented transmission in which data appears as a continuous stream of characters and for which a format list is required to specify the editing desired for the associated data list.

element: a single item of data as opposed to a collection of data items such as an array; a scalar item.

element expression: an expression whose evaluation yields an element value.

elementary name: see base element.

element variable: a variable that represents an element; a scalar variable.

enabled: that state in which a particular on-condition will result in a program interrupt that would cause an on-unit for that condition to be entered.

entry constant: an entry name.

entry expression: an expression whose evaluation yields an entry value.

entry name: an identifier that is explicitly or contextually declared to have the ENTRY attribute (unless the VARIABLE attribute is given) or has an implied ENTRY attribute; the value of an entry variable.

entry point: a point in a procedure at which it may be invoked. (See primary entry point and secondary entry point.)

entry variable: a variable that can represent entry values. It must have both the ENTRY and VARIABLE attributes.

entry value: the entry point represented by an entry constant; the value includes the environment of the activation that is associated with the entry constant.

environment (of an activation): information associated with the invocation of a block that is used in the interpretation of references, within the invoked block, to data declared outside the block. This information includes generations of automatic variables, extents of defined variables, and generations of parameters.

environment (of a label constant): identity of the particular activation of a block to which a reference to a statement-label constant applies. This information is determined at the time a statement-label constant is passed as an argument or is assigned to a statement-label variable, and it is passed or assigned along with the constant.

epiloque: those processes that occur automatically at the termination of a block or task.

evaluation: reduction of an expression to a single value, an array of values, or a structured set of values.

event: an activity in a program whose status and completion can be determined from an associated event variable.

event variable: a variable with the EVENT attribute, which may be associated with an event; its value indicates whether the action has been completed and the status of the completion.

explicit declaration: the appearance of an identifier in a DECLARE statement, as a label prefix, or in a parameter list.

exponent characters: the following picture specification characters:

1. K and E, which are used in floating-point picture specifications to indicate the beginning of the exponent field.

2. F, the scaling factor character, specified with an integer constant which indicates the number of decimal positions the decimal point is to be moved from its assumed position to the right (if the constant is positive) or to the left (if the constant is negative).

expression: a notation, within a program, that represents a value, an array of values, or a structured set of values; a constant or a reference appearing alone, or combinations of constants and/or references with operators.

extent:

1. The range indicated by the bounds of an array dimension, the range indicated by the length of a string, or the range indicated by the size of an area.

2. The significant allocations in an area.

external name: a name (with the EXTERNAL attribute) whose scope is not necessarily confined only to one block and its contained blocks.

external procedure: a procedure that is not contained in any other procedure.

factoring: the application of one or more attributes or of a level number to a parenthesized list of names.

field (in the data stream): that portion of the data stream whose width, in number of characters, is defined by a single data or spacing format item.

field (of a picture specification): any character-string picture specification or that portion (or all) of a numeric character picture specification that describes a fixed-point number.

file: a named representation, within a program, of a data set or data sets. A file is associated with the data set or data sets for each opening.

file attribute: any of the attributes that describe the characteristics of a file.

file constant: a name declared for a file and for which a complete set of file attributes exists during the time that the file is open.

file expression: an expression whose evaluation yields a file name.

file name: a name declared for a file.

file variable: a variable to which file constants can be assigned; it must have both the attributes FILE and VARIABLE. No file-name attributes, other than FIIE, can be specified for a file-name variable.

fixed-point constant: see arithmetic constant.

floating-point constant: see arithmetic constant.

flow of control: sequence of execution.

format item: a specification used in edit-directed transmission to describe the representation of a data item in the stream (data format item) or to specify positioning of a data item within the stream (control format item).

format list: a parenthesized list of format items required for an edit-directed data specification.

fully-qualified name: a qualified name that is complete, i.e., that includes all names in the hierarchical sequence above the structure member to which the name refers, as well as the name of the member itself.

function: a function procedure (programmer-specified or built-in); a procedure that is invoked by the appearance of one of its entry names in a function reference and which returns a value to the point of reference.

function reference: the appearance of an entry-name or built-in function name (or an entry variable) in an expression.

generation (of a variable): the allocation of a static variable, a particular allocation of a controlled or automatic variable or the storage indicated by a particular locator qualification of a based variable, or by a defined variable or a parameter.

generic key: a character string that identifies a class of keys: all keys that begin with the string are members of that class. For example, the recorded keys 'ABCD', 'ABCE', and 'ABDF', are all members of the classes identified by the generic keys 'A' and 'AB', and the first two are also members of the class 'ABC'; and the three recorded keys can be considered to be unique members of the classes 'ABCD', 'ABCE', 'ABDF', respectively.

generic name: the name of a family of entry names. A reference to the name is replaced by the particular entry name whose parameter descriptors match the attributes of the arguments in the argument list at the point of invocation.

group: a do-group; it can be used wherever a single statement can appear, except as an on-unit.

identifier: a string of alphameric and, possibly, break characters, not contained in a comment or constant and which is preceded and followed by a separator; the initial character must be alphabetic.

implicit declaration: the establishment of an identifier, which has no explicit or contextual declaration, as a name. A default set of attributes is assumed for the identifier.

implicit opening: the opening of a file as the result of an input or output statement other than the OPEN statement.

infix operator: an operator that appears between two operands.

initial procedure: an external procedure whose PROCEDURE statement has the OPTIONS (MAIN) attribute. Every PL/I program must have an initial procedure. It is invoked automatically as the first step in the execution of a progam.

input/output: the transfer of data between an auxiliary medium and main storage.

insertion picture character: a picture specification character that is, on assignment of the associated data to a character string, inserted in the indicated position. When used in a P-format item for input, an insertion character serves as a checking picture character.

interleaved array: an array whose name refers to non-connected storage.

interleaved subscripts: a subscript notation, used with subscripted qualified names, in which not all of the necessary subscripts immediately follow the same component name.

internal block: a block that is contained in another block.

internal name: a name that is not known outside the block in which it is declared.

internal procedure: a procedure that is contained within a block.

internal text: all of the text contained in a block except that text that is contained in another block. Thus the text of an internal block (except its entry names) is not internal to the containing block.

interrupt: the redirection of flow of control of the program (possibly temporary) as the result of an on-condition or attention.

invocation: the activation of a procedure.

invoke: to activate a procedure at one of its entry points.

invoked procedure: a procedure that has been activated at one of its entry points.

invoking block: a block containing a statement that activates a procedure.

iteration factor: an expression that specifies:

1. In an INITIAL attribute specification, the number of consecutive elements of an array that are to be initialized with a given constant.

2. In a format list, the number of times a given format item or list of items is to be used in succession.

iterative do-group: a do-group whose DO statement specifies a control variable and/or a WHILE option.

key: data that identifies a record within a direct-access data set. 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: (applied to a name) recognized with its declared meaning; a name is known throughout its scope.

label: a name used to identify a statement other than a PROCEDURE or ENTRY statement; a statement label.

label constant: an unsubscripted name that appears prefixed to any statement other than a PROCEDURE or ENTRY statement.

label list (of a statement): all of the label prefixes of a statement.

label list (of a label variable declaration): a parenthesized list of one or more statement-label constants immediately following the keyword LABEL to specify the range of values that the declared variable may have; names in the list are separated by commas. When specified for a label array, it indicates that each element of the array may assume any of the values listed but no other.

label prefix: a label prefixed to a statement.

label variable: a variable declared with the LABEL attribute and thus able to assume as its value a label constant.

language character set: a character set which has been defined to represent program elements in the source language (in this context, character-string constants and comments are not considered as program elements).

leading zeros: zeros that have no significance in the value of an arithmetic integer; all zeros to the left of the first significant integer digit of a number.

level number: an unsigned decimal integer constant in a DECLARE or ALLOCATE statement that specifies the position of a name in the hierarchy of a structure. It precedes the name to which it refers and is separated from that name by the name's delimiter. Level numbers appear without the names in a parameter descriptor of an ENTRY attribute specification.

level-one variable: a major structure name; any unsubscripted variable not contained within a structure.

list-directed transmission: the type of stream-oriented transmission in which data in the stream appears as constants separated by blanks or commas and for which formatting is provided automatically.

locator qualification: in a reference to a based variable, either a locator variable or function reference connected by an arrow to the left of a based variable to specify the generation of the based variable to which the reference refers, or the implicit connection of a locator variable with the based reference.

locator variable: a variable whose value identifies the location in main storage of a variable or a buffer.

locked record: a record in an EXCLUSIVE DIRECT UPDATE file that is available to only one task at a time.

logical level (of a structure member): the depth indicated by a level number when all level numbers are in direct sequence, that is, when the increment between successive level numbers is one.

logical operators: the bit-string operators ¬ (not), & (and), and | (or).

lower bound: the lower limit of an array dimension.

major structure: a structure whose name is declared with level number 1.

major task: the task that has control at the outset of execution of a program. It exists throughout the execution of the program.

minor structure: a structure that is contained within another structure. The name of a minor structure is declared with a level number greater than one.

mode (of arithmetic data): a characteristic of arithmetic data; real or complex.

multiple declaration: two or more declarations of the same identifier internal to the same block without different qualifications, or two or more external declarations of the same identifier with different attributes in the same program.

multiprocessing: the use of a computing system with two or more processing units to execute two or more programs simultaneously.

multiprogramming: the use of a computing system to execute more than one program concurrently, using a single processing unit.

multitasking: a facility that allows a programmer to execute more than one PL/I procedure simultaneously.

name: an identifier appearing in a context where it is not a keyword.

nesting: the occurrence of:

1. A block within another block.
2. A group within another group.
3. An IF statement in a THEN clause or an ELSE clause.
4. A function reference as an argument of a function reference.
5. A remote format item in the format list of a FORMAT statement.
6. A parameter descriptor list in another parameter descriptor list.
7. An attribute specification within a parenthesized name list for which one or more attributes are being factored.

non-connected storage: separate locations in storage that contain related items of data that can be referred to by a single name but that are separated by other data items not referred to by that name. Examples are the storage referred to by an unsubscripted elementary name in an array of structures or by a subscripted name referring to an array cross section in which the subscript list contains an asterisk to the left of any element expression.

null locator value: a special locator value that cannot identify any location in internal storage; it gives a positive indication that a locator variable does not currently identify any generation of data.

null string: a string data item of zero length.

numeric character data: see decimal picture data.

offset variable: a locator variable with the OFFSET attribute, whose value identifies a location in storage, relative to the beginning of an area.

on-condition: an occurrence, within a PL/I task, that could cause a program interrupt. It may be the detection of an unexpected error or of an occurrence that is expected, but at an unpredictable time.

on-unit: the specified action to be executed upon detection of the on-condition named in the containing ON statement. This excludes SYSTEM and SNAP.

opening (of a file): the association of a file with a data set and the completion of a full set of attributes for the file name.

operand: an expression to whose value an operator is applied.

operational expression: an expression containing one or more operators.

operator: a symbol specifying an operation to be performed. See arithmetic operators, bit-string operators, comparison operators and concatenation.

option: a specification in a statement that may be used to influence the execution or interpretation of the statement.

packed decimal: the internal representation of a fixed-point decimal data item.

padding:

1. one or more characters or bits concatenated to the right of a string to extend the string to a required length. For character strings, padding is with blanks; for bit string, with zeros.

2. one or more characters or bits inserted in a structure so that the structure elements have the required alignment.

parameter: a name in a procedure that is used to refer to an argument passed to that procedure.

parameter descriptor: the set of attributes specified for a single parameter in an ENTRY attribute specification.

parameter descriptor list: the list of all parameter descriptors in an ENTRY attribute specification.

parameter list: a parenthesized list of one or more parameters, separated by commas following either the keyword PROCEDURE in a PROCEDURE statement, or the keyword ENTRY in an ENTRY statement. The list corresponds to a list of arguments passed at invocation.

partially-qualified name: a qualified name that is incomplete, i.e., that includes one or more, but not all, names in the hierarchical sequence above the structure member to which the partially-qualified name refers, as well as the name of the member itself.

picture specification: a character-by-character description of the composition and characteristics of decimal picture data and character-string picture data.

picture specification character: any of the characters that can be used in a picture specification. See decimal picture data and character-string picture data.

point of invocation: the point in the invoking block at which the procedure reference to the invoked procedure appears.

pointer variable: a locator variable with the POINTER attribute, whose value identifies an absolute location in main storage.

precision: the value range of an arithmetic variable expressed as a total number of digits and, for fixed-point variables, the number of those digits assumed to appear to the right of the decimal or binary point.

prefix: a label or a parenthesized list of one or more condition names connected by a colon to the beginning of a statement.

prefix operator: an operator that precedes an operand and applies only to that operand. The prefix operators are + (plus), - (minus), and ¬ (not).

preprocessor: a program that examines the source program for preprocessor statements which are then executed, resulting in the alteration of the source program.

preprocessor statement: a special statement appearing in the source program that specifies how the source program text is to be altered; it is executed as it is enccuntered by the preprocessor.

primary entry point: the entry point identified by any of the names in the label list of the PROCEDURE statement.

priority: a value associated with a task, that specifies the precedence of the task relative to other tasks.

problem data: string or arithmetic data that is processed by a PL/I program.

procedure: a collection of statements, headed by a PROCEDURE statement and ended by an END statement, that is a part of a program, that delimits the scope of names, and that is activated by a reference to one of its entry names.

procedure reference: an entry constant or variable or a built-in function name. The name may be followed by one or more argument lists. It may appear in a CALL statement or CALL option or as a function reference.

processor: a program that prepares source program text (possible preprocessed text) for execution.

program: a set of one or more external procedures, one of which must have the OPTIONS(MAIN) option in its PROCEDURE statement.

program control data: data used in a PL/I program to effect the execution of the program. Program control data consists of the following types: entry, task, file, label, event, pointer, offset, and area.

prologue: the processes that occur automatically on block activation.

pseudovariable: any of the built-in function names that can be used to specify a target variable.

qualified name: a hierarchical sequence of names of structure members, connected by periods, used to identify a component of a structure. Any of the names may be

subscripted. See also locator qualification.

range (of a default specification): a set of identifiers and/or parameter descriptors to which the attributes in a default specification of a DEFAULT statement apply.

record: the logical unit of transmission in a record-oriented input or output operation.

recorded key: a key recorded in a direct-access volume to identify an associated data record.

recursive procedure: a procedure that may be reactivated while still active in the same task.

reentrant procedure: a procedure that may be reactivated while active in another task.

REFER expression: the expression preceding the keyword REFER, from which an original bound, length, or size is taken when a based variable containing a REFER option is allocated, either by an ALLOCATE or LOCATE statement.

REFER object: the unsubscripted element variable appearing in a REFER option that specifies a current bound, length, or size for a member of a based structure. It must be a member of the structure, and it must precede the member declared with the REFER option.

reference: the appearance of a name, except in a context that causes explicit declaration.

remote format item: the letter R specified in a format list together with the label of a separate FORMAT statement.

repetition factor: a parenthesized unsigned decimal integer constant that specifies:

1. The number of occurrences of a string configuration that make up a string constant.
2. The number of occurrences of a picture specification character in a picture specification.

repetitive specification: an element of a data list that specifies controlled iteration to transmit one or more data items, generally used in conjunction with arrays.

returned value: the value returned by a function procedure to the point of invocation.

scalar item: a single item of data; an element.

scalar variable: a variable that can represent only a single data item; an element variable.

scale: a system of mathematical notation: fixed-point or floating-point scale of an arithmetic value.

scale factor: a specification of the number of fractional digits in a fixed-point number.

scope (of a condition prefix): the portion of a program throughout which a particular condition prefix applies.

scope (of a declaration): the portion of a program throughout which a particular declaration is a source of attributes for a particular name.

scope (of a name): the portion of a program throughout which the meaning of a particular name does not change.

secondary entry point: an entry point identified by any of the names in the label list of an ENTRY statement.

self-defining data: a data item, or an aggregate of data items, that includes descriptive information about attributes of the data, such as values for adjustable bounds or lengths.

separator: see delimiter.

sign and currency symbol characters: the picture specification characters, S, +, -, and $. These can be used

1. As static characters in which case they are specified only once in a picture specification and appear in the associated data item in the position in which they have been specified.
2. As drifting characters, in which case they are specified more than once (as a string in a picture specification) but appear in the associated data item at most once, immediately to the left of the significant portion of the data item.

significant allocation: any unfreed allocation in an area and any freed allocation that lies between the start of the area and the end of the unfreed allocation that is farthest from the start

of the area. If a subsequent allocation of the same size is made in the same location the original allocation ceases to be significant.

simple parameter: a parameter for which no storage-class attribute is specified; it may represent an argument of any storage class, but only the current generation of a controlled argument.

source key: a key referred to in a record-oriented transmission statement that identifies a particular record within a direct-access data set.

source program: the program that serves as input to the compiler. The source program may contain preprocessor statements.

source variable: a variable whose value is to be assigned or to take part in some other operation.

standard default: the alternative attribute or option assumed when none has been specified and there is no applicable DEFAULT statement.

standard file: a file assumed by the processor in the absence of a FILE or STRING option in a GET or PUT statement; SYSIN is the standard input file and SYSPRINT is the standard output file.

standard system action: action specified by the language to be taken in the absence of an on-unit for an on-condition.

statement: a basic element of a PL/I program that is used to delimit a portion of the program, to describe names used in the program, or to specify action to be taken. A statement can consist of a condition list, a label list, a statement identifier, and a statement body that is terminated by a semicolon.

statement body: that part of a statement that follows the statement identifier, if any, and is terminated by the semicolon; it includes the statement options.

statement identifier: the PL/I keyword that indicates the purpose of the statement.

statement-label constant: see label constant.

statement-label expression: see label expression.

statement-label variable: see label variable.

static storage allocation: the allocation of storage for static variables.

static variable: a variable that is allocated before execution of the program begins and that remains allocated for the duration of execution of the program.

stream: see data stream.

string: a connected sequence of characters or bits that is treated as a single data item.

string variable: a variable declared with the BIT or CHARACTER attribute, whose values can be either bit strings or character strings.

structure: a hierarchical set of names that refers to an aggregate of data items that may have different attributes.

structure expression: an expression whose evaluation yields a structure set of values.

structure of arrays: a structure containing arrays specified by declaring individual members names with the dimension attribute.

structure member: any of the minor structures or elementary names in a structure.

structuring: the makeup of a structure, in terms of the number of members, the order in which they appear, their attributes, and their logical level (but not necessarily their names or declared level numbers).

subfield (of a picture specification): that portion of a picture specification field that appears before or after a V picture specification character.

subroutine: a procedure that is invoked by a CALL statement or CALL option. A subroutine cannot return a value to the invoking block, but it can alter the value of variables.

subscript: an element expression that specifies a position within a dimension of an array. A subscript can also be an asterisk, in which case it specifies the entire extent of the dimension.

subscript list: a parenthesized list of one or more subscripts, one for each dimension of an array, which together uniquely identify either a single element or cross section of the array.

subtask: a task that is attached by the given task or any of the tasks in a direct line from the given task to the last attached task.

synchronous: using a single flow of control for serial execution of a program.

target variable: a variable to which a value is assigned.

task: the execution of one or more procedures by a single flow of control.

task name: an identifier used to refer to a task variable.

task variable: a variable with the TASK attribute whose value gives the relative priority of a task.

termination (of a block): cessation of execution of a block, and the return of control to the activating block by means of a RETURN or END statement, or the transfer of control to the activating block or to some other active block by means of a GO TO statement.

termination (of a task): cessation of the flow of control for a task.

truncation: the removal of one or more digits, characters, or bits from one end of an item of data when a string length or precision of a target variable has been exceeded.

upper bound: the upper limit of an array dimension.

variable: a named entity that is used to refer to data and to which values can be assigned. Its attributes remain constant, but it can refer to different values at different times. Variables fall into three categories, applicable to any data type: element, array, and structure. Variables may be subscripted and/or qualified, or locator qualified.

virtual point picture character: the picture specification character, V, which is used in picture specifications to indicate the position of an assumed decimal or binary point.

zero-suppression characters: the picture specification characters Z, Y, and *, which are used to suppress zeros in the corresponding digit positions.

Index

Indexes to systems reference library manuals are consolidated in the publication IBM System/360 Operating System: Systems Reference Library Master Index, Form GC28-6644. For additional information about any subject listed below, refer to other publications listed for the same subject in the Master Index.

Where more than one page reference is given the major reference is first.

OS PL/I Checkout and Optimizing Compilers:
Language Reference Manual

Order No. SC33-0009-2

READER'S COMMENT FORM

Your views about this publication may help improve its usefulness; this form will be sent to the author's department for appropriate action. Using this form to request system assistance or additional publications will delay response, however. *For more direct handling of such requests, please contact your IBM representative or the IBM Branch Office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Index Figures Examples Legibility

Cut or Fold Along Line

What is your occupation? _

Number of latest Technical Newsletter (if any) concerning this publication: _ _ _ _ _ _ _ _ _ _ _.

Please indicate in the space below if you wish a reply.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

Your comments, please . . .

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. Your comments on the other side of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Cut or Fold Along Line

Fold Fold

First Class
Permit 40
Armonk
New York

Business Reply Mail
No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department 813L
1133 Westchester Avenue
White Plains, New York 10604

Fold Fold

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

OS PL/I Checkout and Optimizing Compilers: Language Reference Manual Printed in U.S.A. SC33-0009-2

OS PL/I Checkout and Optimizing Compilers: Language Reference Manual Printed in U.S.A. SC33-0009-2

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
[U.S.A. only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]