60388100

CONTROL DATA CORPORATION

PL/I VERSION 1 REFERENCE MANUAL

CDC® OPERATING SYSTEMS:
NOS 1
NOS/BE 1

REVISION RECORD

REVISION	DESCRIPTION
A	Original release at PSR level 472.
(03-31-78)	
B	This revision incorporates various technical corrections and editorial improvements.
(11-01-79)	
Publication No. 60388100	

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Cover	—
Inside Cover	—
Title Page	—
ii	B
iii/iv	B
v/vi	B
vii thru xiv	B
xv	A
1-1 thru 1-7	A
2-1 thru 2-9	A
3-1 thru 3-13	A
3-14	B
4-1 thru 4-10	A
4-11	B
4-12	B
4-12.1/4-12.2	B
4-13	A
4-14	B
4-15 thru 4-22	A
4-23	B
4-24	A
4-25	B
4-26 thru 4-28	A
4-29	B
4-30 thru 4-32	A
5-1	B
5-2	B
5-3 thru 5-6	A
6-1	A
6-2	B
6-3	B
6-4 thru 6-6	A
6-7 thru 6-13	B
7-1 thru 7-4	A
7-5 thru 7-7	B
7-8	A
7-9	A

Page	Revision
7-10	B
7-11 thru 7-16	A
7-17	B
7-18	A
7-19	A
8-1	A
8-2	B
8-3	A
8-4	B
8-5 thru 8-8	A
8-9 thru 8-11	B
8-12	A
8-13	B
8-14	B
9-1 thru 9-3	B
10-1 thru 10-5	A
10-6 thru 10-11	B
11-1	B
11-2	A
11-3	A
11-4	B
11-5 thru 11-13	A
11-14	B
11-15 thru 11-25	A
11-26	B
11-27	A
12-1 thru 12-4	A
12-5	B
12-6 thru 12-11	A
12-12	B
12-13	A
12-14	A
12-15	B
12-16	B
12-17	A
12-18 thru 12-20	B
12-21 thru 12-24	A

Page	Revision
12-25	B
12-26 thru 12-28	A
12-29 thru 12-31	B
12-32 thru 12-37	A
13-1 thru 13-3	B
13-4 thru 13-7	A
14-1 thru 14-4	A
14-5	B
14-6	A
A-1 thru A-4	A
B-1	A
B-2 thru B-29	B
C-1 thru C-10	A
D-1 thru D-4	A
D-5	B
E-1 thru E-5	A
E-6	B
E-7	A
E-8	A
E-9	B
E-10 thru E-12	A
E-13	B
E-14 thru E-17	A
E-18	B
E-19	A
F-1	A
F-2	A
F-3	B
F-4	B
Index-1 thru -10	B
Comment Sheet	B
Return Env.	—
Inside Cover	—
Back Cover	—

PREFACE

PL/I is a block-structured programming language designed for use in scientific and commercial applications. PL/I Version 1 is essentially a subset of the language defined by the American National Standard Programming Language PL/I, X3.53-1976, document.

PL/I Version 1 operates under control of the following operating systems:

- NOS 1 for the CONTROL DATA® CYBER 170 Series; CYBER 70 Models 71, 72, 73, 74; and 6000 Series Computer Systems
- NOS/BE 1 for the CDC® CYBER 170 Series; CYBER 70 Models 71, 72, 73, 74; and 6000 Series Computer Systems

This manual is designed for programmers familiar with the PL/I language and the operating system under which the PL/I compiler is operating.

Related material is contained in the publications listed below.

Publication	Publication Number
CYBER Record Manager Advanced Access Methods Version 2 Reference Manual	60499300
CYBER Record Manager Basic Access Methods Version 1.5 Reference Manual	60495700
FORTRAN Common Library Mathematical Routines Reference Manual	60498200
NOS Version 1 Reference Manual, Volume 1 of 2	60435400
NOS Version 1 Reference Manual, Volume 2 of 2	60445300
NOS/BE Version 1 Reference Manual	60493800

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

CONTENTS

APPENDIXES

INDEX

FIGURES

TABLES

NOTATIONS USED IN THIS MANUAL

NOTATION USED IN SYNTAX

UPPER CASE	words are PL/I keywords.
Lower case	words are generic terms that represent the words or symbols supplied by the programmer.
[ ] Brackets	enclose optional portions of syntax. All of the syntax within the brackets can be omitted or included at programmer option. If items are stacked vertically within brackets, only one of the stacked items can be used.
{ } Braces	enclose required portions of syntax. If items are stacked vertically within braces, only one of the stacked items can be used.
[| |] Vertical Bars Within Brackets	enclose two or more vertically stacked items when each of the stacked items can be either used once, or omitted. Any items used can be written in any order.
. . . Ellipses	immediately follow an item, a pair of brackets, or a pair of braces to indicate that the item or enclosed syntax can be repeated at programmer option.
, , , Commas	immediately follow an item, a pair of brackets, or a pair of braces to indicate that the item or enclosed syntax can be repeated at programmer option. A comma is required between repeated entries.
• Bullet	separates adjacent items to indicate they can be written in any order. Two or more bullets separating items at the same bracket or brace level indicate all separated items can be permuted in any order.

Punctuation symbols shown within syntax are required unless enclosed in brackets.

NOTATION USED IN EXAMPLES

Δ	indicates the blank character.

SHADING USED IN MANUAL

Control Data restrictions and extensions to the language described in American National Standard Programming Language PL/I, X3.53-1976, are indicated by shading. Shading is also used to indicate processing that is different from that specified in the standard. Language and processing that are implementation-defined but within the standard are not shaded.

PL/I SOURCE PROGRAM 1

A PL/I program is one or more external procedures, compiled separately or together, which can interact during execution. Each procedure is composed of a series of statements that conform to the structure and syntax of the PL/I language. The source program is processed by the PL/I compiler and changed into an executable program that can be loaded and executed by the operating system.

When the compiler executes in response to a control statement entered through a batch job or interactive terminal, it performs the following functions:

Reads the file containing the source program. INPUT is the default source file.

Checks the source program for errors in program structure and language syntax, and writes diagnostic messages to a file to be printed on the line printer or displayed at a terminal.

Depending on parameters specified on the PLI control statement, writes a copy of the source program to an output file; and writes a cross reference list to this file as a summary of data item descriptions and references within the program. OUTPUT is the default output file.

Unless binary code generation is suppressed by a compiler call parameter, compiles the source lines into executable code and writes the executable instructions to a file in a format suitable for loading and executing. LGO is the default file for the compiled program.

Once the source program is compiled, it can be loaded and executed by a control statement that names the file of executable instructions. Section 13 describes other functions that can be performed during compilation.

During execution, the compiled program makes use of execution-time routines that are part of the PL/I run-time library. Files referenced in the program are opened, read, written, rewritten, and closed through the CYBER Record Manager facility common to several products running under the NOS and NOS/BE operating systems.

The job in which the program executes is responsible for making data files available before the program begins execution and for properly disposing of output files after execution ends. If the program uses information on a library, the job is also responsible for making the library file available before execution.

PROGRAM STRUCTURE

A PL/I program is composed of a sequence of structured components. The basic language elements such as names, keywords, and constants represent the lowest level structure; statements represent the middle level; and blocks represent the highest level. The basic language elements are grouped to form statements, and the statements in turn are grouped to form organized collections called blocks. The hierarchical structure of the program blocks determines how much or how little interaction between various components is to take place during the execution of a program.

This organization of the source code represents the static structure of a program.

CONTAINMENT

The terms contain and immediately contain are used throughout this manual. The terms are essential to an understanding of the way PL/I programs are organized.

A program component contains a second component if the second component appears inside of the first.

A program component of a certain kind immediately contains a second component if both of the following statements are true:

- The first component contains the second component.
- There is no other component, of the same kind as the first component, that contains the second and is contained in the first.

For instance, one can describe the room that immediately contains the table and the house that immediately contains the table.

A program component is always either completely contained or not contained at all in another program component.

BLOCKS

A source program consists of delimited areas called blocks. Each block is a sequence of PL/I statements and constitutes a section of the program. Blocks affect the program in the following ways:

- Delimit areas within which declared names are known.
- Determine the allocation and freeing of certain kinds of data storage.
- Influence the flow of control during program execution.

There are two types of blocks: procedure blocks and begin blocks. The structuring of the two types of blocks is similar. Each is headed by an identifying statement; followed by groups of statements, which in turn can be nested procedure or begin blocks; and terminated by an identifying statement. The structure of a block is as follows:

PROCEDURE or BEGIN statement

Zero or more block units

END statement

where each block unit can be any of the following components:

procedure block

begin block

do group

compound statement

DECLARE statement

ENTRY statement (procedure block only)

FORMAT statement

single statement

The significant difference between procedure blocks and begin blocks is the manner in which they are activated (entered for processing) during program execution. A procedure block is activated when it is invoked. A begin block is activated when its BEGIN statement is encountered in the normal sequential flow of control. If the BEGIN statement is labeled, the block can also be activated by an unconditional transfer (GOTO statement).

Procedure Block

A procedure block can be external or internal. An external procedure block is one that is not contained in another block. An internal procedure block is one that is contained within another block.

Every program must contain one, and only one, main procedure. The main procedure is an external procedure whose PROCEDURE statement denotes the main entry point of the program; execution is initiated at this point. The main procedure is distinguishable from all other procedures by the appearance of the OPTIONS(MAIN) clause in the PROCEDURE statement that heads the block.

A procedure is invoked at one of its entry points. Entry points are denoted by the PROCEDURE statement and all ENTRY statements that are immediately contained in the procedure. The PROCEDURE statement denotes the primary entry point of the procedure. Each ENTRY statement denotes a secondary entry point. Any entry point except the primary entry point of a main procedure can have a set of parameters to which arguments are passed when the procedure is invoked.

Each entry point is either a subroutine entry point or a function entry point. A procedure is invoked in one of two ways:

CALL statement execution	A subroutine entry name appears in a CALL statement; control passes to the specified procedure entry point.
Function reference	A function entry name appears in an expression; control passes to the function, which performs computation and returns a value to the point of invocation.

Begin Block

A begin block is always internal (contained within another block). Begin blocks, unlike procedure blocks, are not invoked. A begin block is executed when control reaches the block at the BEGIN statement either through normal sequential flow of control or transfer of control to the BEGIN statement.

Block Containment

A procedure or begin block can contain other blocks. Every block except an external procedure is completely enclosed in some other block. Blocks can be nested up to a depth of 50.

In general, each component of a PL/I program is immediately contained in exactly one block. For example, the block that immediately contains the declaration of an identifier is either a procedure block or a begin block. The declaration is considered to be immediately contained in the block even if the declaration is inside another kind of program component (such as a do group) that is contained in the block. All text in a block – except text in contained blocks – is immediately contained in the block.

There are three special cases in which elements of the program are not contained in the block one might expect:

Entry names on a PROCEDURE or ENTRY statement of an internal procedure block are not immediately contained in the procedure whose entry point is denoted by the statement; they are immediately contained in the block that immediately contains that procedure.

Labels on a BEGIN statement are not immediately contained in the begin block headed by the statement; they are immediately contained in the block that immediately contains that begin block.

Entry names on a PROCEDURE or ENTRY statement of an external procedure block are not contained in any block.

These exceptions are made because these entry names and labels are not the names of the statements on which they appear, but the names of the blocks. The name of a block must be outside the block so that the block can be accessed.

The concepts of containment and immediate containment are illustrated in figure 1-1.

DO GROUPS

A do group is a delimited area of a block. Each group is a sequence of PL/I statements that can be executed conditionally or iteratively.

A do group is headed by a DO statement, followed by some sequence of statements and blocks, and terminated by an END statement. A do group is entered when it is encountered in the normal sequential flow of control. If the DO statement is labeled, the group can be entered at the DO statement by an unconditional transfer (GOTO statement). The structure of a do group is as follows:

DO statement

Zero or more block units

END statement

where each block unit can be any of the following components:

procedure block

begin block

do group

compound statement

DECLARE statement

```
P1:
PROCEDURE;
     statement-1;
     statement-2;
B1:
     BEGIN;
          statement-3;
P2:
          PROCEDURE;
               statement-4;
E2:
          ENTRY;
               statement-5;
          END P2;
          statement-6;
B2:
          BEGIN;
               statement-7;
EXB2:
          END B2;
     END B1;
     statement-8;
EXP1:
END P1;
```

The immediately contained text for each block is:

Block P1

```
PROCEDURE;
statement-1
statement-2
B1 label
statement-8
EXP1:  END ;
```

Block B1

```
BEGIN;
statement-3
P2 entry name
E2 entry name
statement-6
B2 label
END ;
```

Block P2

```
PROCEDURE;
statement-4
ENTRY;
statement-5
END ;
```

Block B2

```
BEGIN;
statement-7
EXB2:  END ;
```

The contained text for each block is indicated by the lines to the left of the example. The labels of BEGIN statements and the entry names of PROCEDURE and ENTRY statements are not contained in the block that immediately contains those statements, but in the next outer block. Label B2 on the BEGIN statement, for example, is not contained in block B2, but in block B1. Entry name P1 on the PROCEDURE statement is not contained in block P1 since P1 is an external procedure.

Figure 1-1. Block Containment

ENTRY statement (noniterative do group only)

FORMAT statement

single statement

Do groups can be iterative or noniterative. A noniterative do group is executed once each time the DO statement is encountered. An iterative do group can be executed repetitively according to controls specified in the DO statement.

COMPOUND STATEMENTS

A compound statement is a statement that can contain embedded statements. The IF and ON statements are the statements in this structural category.

IF Statement

The IF statement allows the flow of control to take alternate paths, depending on the value of a specified expression. The structure of an IF statement is as follows:

```
IF expression
     THEN executable-unit
     ELSE executable-unit
```

or

```
IF expression
     THEN executable-unit
```

The statement is classified as compound since each executable unit consists of one or more embedded statements. An executable unit is executed or bypassed depending on the value of the expression. Each executable unit can be any of the following components:

begin block

do group

compound statement

single statement

ON Statement

The ON statement establishes an on-unit for a condition. When the ON statement is executed, it sets up the action to be taken when a specific condition occurs at some future time. The on-unit is executable code that specifies the action taken when the specific condition occurs. The structure of an ON statement is as follows:

```
ON condition on-unit
```

The statement is classified as compound since the on-unit consists of one or more embedded statements. The on-unit can be either of the following components:

begin block

single statement

The begin block or single statement representing the on-unit is executed only when an unusual condition arises during execution and results in a program interrupt. The on-unit designates the action that is to be taken as a result of the interrupt.

An on-unit acts in a manner similar to a procedure block. Two significant differences between a procedure block and an on-unit are

- The on-unit is activated automatically and only on interrupt.
- The on-unit cannot receive arguments.

SINGLE STATEMENTS

The single statement is the basic element from which all groups and blocks are formed. Single statements are listed in table 1-1.

TABLE 1-1. SINGLE STATEMENTS

Category	Statement
Program Control	CALL GOTO Null RETURN STOP
Storage Control	ALLOCATE FREE
Interrupt Handling	REVERT SIGNAL
Input/Output	CLOSE DELETE GET LOCATE OPEN PUT READ REWRITE WRITE
Data Manipulation	Assignment

BLOCK AND DO GROUP CLOSURE

The physical end of a block or do group is denoted by an END statement. The structure of an END statement is as follows:

END closure-name;

The closure name in the END statement is optional. If the closure name is included, it must match one of the following:

An entry name on a PROCEDURE statement

A label on a BEGIN statement

A label on a DO statement

An END statement with a closure name closes its associated block or do group, and also closes any contained blocks or do groups that do not have END statements. This convention is termed multiple closure. An END statement with no closure name closes only one unclosed block or do group; the block or do group closed is the innermost one not closed by any previous END statement.

Examples of block and do group closure are shown in figure 1-2.

Example 1

```
P:
PROCEDURE;
B1:
    BEGIN;
D2:
        DO;
        END D2;
    END B1;
E:
END P;
```

Example 2 (equivalent to example 1)

```
P:
PROCEDURE;
B1:
    BEGIN;
D2:
        DO;
        END;
    END;
E:
END;
```

Example 3 (equivalent to examples 1 and 2)

```
P:
PROCEDURE;
B1:
    BEGIN;
D2:
        DO;
E:
END P;
```

Label on END statement used for multiple closure is equivalent to label on outermost END statement in examples 1 and 2

Example 4 (incorrect structure)

```
P:
PROCEDURE;
B1:
    BEGIN;
B2:
        BEGIN;
    END B1;     /*ENDS B2 AND B1     */

        END B2; /*ERROR - B2 ALREADY*/
                /*CLOSED             */
END P;
```

Figure 1-2. Block and Do Group Closure

STATEMENT STRUCTURE

The basic structure of a statement includes one or more optional prefixes, a statement body, and a terminating semicolon.

A prefix can be a label, entry, format, or condition prefix. Each prefix is terminated by a colon. A label prefix identifies a statement. An entry prefix identifies an entry point at which a procedure can be invoked. A format prefix identifies a FORMAT statement. A condition prefix enables or disables program interrupts caused by one or more abnormal computational conditions.

The statement body generally consists of an identifying keyword followed by a sequence of options that are written as a combination of keywords, expressions, and references.

STATEMENT ELEMENTS

The basic elements that are used to form the prefixes and the components of the statement body are

words

literal constants

iSUBs

delimiters

Words

A word is a sequence of characters. Each word is used as a keyword or an identifier. If a word listed in the keyword vocabulary in appendix D is used in an appropriate context, it is interpreted by the compiler as a keyword. Words that fail to satisfy this condition are interpreted as identifiers.

Words are formed from the letters A through Z, digits 0 through 9, and the following special characters:

@ (CDC equivalent is ≤)

(CDC equivalent is ≡)

$

_ (CDC equivalent is ↛)

A word can contain up to 40 characters and must begin with a letter or one of the following special characters:

@

#

$

Examples of words are

X

P3

PROCEDURE

B555

BINARY

@#ABC

RATE_OF_PAY

Keywords

A keyword has special significance to the compiler and run-time routines when it is used in the context for which it was designed. Keywords are not reserved words. A particular word can be used both as a keyword and as an identifier throughout a program. A word is recognized as a keyword only when it appears in an appropriate context; in any other context, the same word is treated as an identifier.

Examples of keywords are

X	used as a format item.
PROCEDURE	used as a statement identifier.
BINARY	used as an attribute.
ENVIRONMENT	used as an OPEN or CLOSE statement option or as an attribute.

Many keywords have abbreviations as noted in appendix D. The abbreviations are recognized in the same contexts as the keywords.

Identifiers

An identifier is a name that is used to identify and refer to data, statements, and blocks. With the exception of special identifiers SYSIN and SYSPRINT, all identifiers are programmer-defined.

SYSIN is the default input file for program execution. Every stream input operation (GET statement) that does not reference a file or a string as the data source is assumed to reference SYSIN.

SYSPRINT is the default output file for program execution. Every stream output operation (PUT statement) that does not reference a file or a string as the data target is assumed to reference SYSPRINT. Every stream input operation that includes a COPY option and does not reference an output copy file is assumed to reference SYSPRINT for copy purposes. SYSPRINT is also used at run time as the system error message file.

Literal Constants

A literal constant is an unnamed data item that represents a constant arithmetic or string value. The following types of literal constants are basic statement elements:

arithmetic

simple bit string

simple character string

An arithmetic constant is a numeric data item with no embedded blanks. For example:

35	decimal constants	1010B	binary constants
.726		1010.0B	
6.623E-27		10.1E8B	

A simple bit string constant is a sequence of zero and one characters enclosed in apostrophe characters and followed by the radix factor B. For example:

'1110'B

'101'B

'111111'B

A simple character string constant is a combination of characters enclosed in apostrophe characters. For example:

```
'ABC'       represents ABC

'A B C'     represents A B C
```

When an apostrophe character is to appear in the character string value, it must be represented by a pair of adjacent apostrophes. For example:

```
'AB''C'     represents AB'C
```

iSUBs

The iSUB elements are used only in subscripts in the DEFINED attribute. An iSUB is an integer followed by the letters SUB with no intervening blanks. For example:

```
DECLARE B(5,5) ;
DECLARE D(5)
  DEFINED (B(1SUB,1SUB)) ;
```

Delimiters

Delimiters are special characters that define the limits of words, literal constants, and iSUBs. When a syntax rule does not indicate a specific delimiter, at least one blank or comment must be used to separate any two nondelimiter elements. In the example

```
PUT LIST(AMT) ;
```

the left parenthesis is required in the syntax and acts as a delimiter between LIST and AMT; therefore, a blank is not required. At least one blank is required as a delimiter between PUT and LIST.

Some delimiters act as operators in expressions. Other delimiters delimit or separate specific program components; the semicolon, for example, delimits statements.

It should be noted that the decimal point, plus, and minus characters do not act as delimiters when they appear within an arithmetic constant. Delimiter characters do not act as delimiters when they appear within a simple character string constant or a comment.

The complete set of delimiters is listed in table 1-2.

TABLE 1-2. DELIMITERS

blank
comment
:
;
,
.
(
)
->
+
-
*
/
**
=
<
>
¬ = (ASCII equivalent is ~ =)
¬ < (ASCII equivalent is ~ <)
¬ > (ASCII equivalent is ~ >)
< =
> =
¬ (ASCII equivalent is ~)
& (CDC equivalent is ∧)
! (CDC equivalent is ∨ and ASCII equivalent is !)
!! (CDC equivalent is ∨∨ and ASCII equivalent is !!)

Blanks

Any two basic elements can be separated by one or more blanks. When one or both of the elements are delimiters, the blank is optional. The following examples are equivalent:

```
A+B(X)

A + B (X)

A+ B( X)
```

When neither element is a delimiter, at least one blank is required. The following examples are equivalent:

```
END P;

END     P;
```

The following examples are not equivalent:

```
END P;

ENDP;
```

Blanks cannot appear within any basic element except a simple character string constant or a comment.

Comments

A comment consists of a sequence of characters preceded by the character pair /* and followed by the character pair */. The character pair */ must not appear within the comment itself. For example:

```
/* THIS IS A LEGAL COMMENT */

/* THIS IS AN ILLEGAL */ COMMENT */
```

Any two basic elements can be separated by a comment. A comment between any two basic elements has the same effect as a blank. For example:

```
A+/*B-FUNCTION OF X*/B (X)

END/*CLOSE BLOCK P*/P;
```

If any part of a character string constant conforms to the syntax rule for a comment, it is not interpreted as a comment, but simply as part of the character string.

Conversely, if any part of a comment conforms to the syntax rule for a character string constant, it is not interpreted as a character string constant, but simply as part of the comment.

Comments appear on the output listing of the source program and have no effect on program execution. They are used for documentation purposes only.

CHARACTER SET

The set of language characters recognized by the PL/I compiler can be combined according to the specified rules to form names and values in the source program. The PL/I character set is shown in table 1-3. Extralingual characters can only be used in comments and character strings.

CODING CONVENTIONS

PL/I source statements are assumed to be in columns 1 through 72 of each source line. The actual columns read by the compiler can be altered by the COL parameter on the PLI control statement. The maximum range for source statements is column 1 through column 100.

PL/I programming is free-form within the chosen columns. The program can be arranged on the page for maximum readability and maintainability. The source text within the columns read by the compiler is treated as a continuous stream of characters.

On a line containing an END statement that closes an external procedure, all text following the END statement is ignored by the compiler.

If several external procedures are compiled together, complete lines of text (containing blanks and comments) between two external procedures are treated as belonging to the procedure they precede. Such lines are not permitted to follow the last external procedure being compiled.

Coding conventions are illustrated in figure 1-3.

TABLE 1-3. CHARACTER SET

PL/I Language Characters

Letters:

A-Z

Digits:

0-9

Special Characters:

+	plus
-	minus
*	asterisk
/	slash
>	greater than
<	less than
=	equals
¬	not (ASCII equivalent is ~)
&	and (CDC equivalent is ∧)
|	or (CDC equivalent is ∨ and ASCII equivalent is !)
.	period
,	comma
;	semicolon
:	colon
	blank
'	apostrophe (CDC equivalent is ↑)
(	left parenthesis
)	right parenthesis
_	underline (CDC equivalent is ←)
$	dollar sign
@	commercial at (CDC equivalent is ≤)
#	number (CDC equivalent is ≡)

Extralingual Characters:

%	percent
[	left bracket
]	right bracket
"	quote (CDC equivalent is ≠)
?	question mark (CDC equivalent is ↓)
\	back slash (CDC equivalent is ≥)

```
/* THIS COMMENT
      CONTINUES OVER
         FOUR LINES
            OF SOURCE TEXT */

/* BUT THIS                  */
/*    IS A MUCH SAFER        */
/*       WAY TO WRITE        */
/*          SUCH COMMENTS    */

DECLARE C CHARACTER(100) VARYING INITIAL(↑THIS SIMPLE-CHARACTER-STRING-C
ONSTANT RUNS ACROSS THE END OF A LINE↑);

DECLARE D CHARACTER(100) VARYING INITIAL
(↑BUT THIS IS A SAFER WAY TO WRITE A VERY LONG CHARACTER STRING↑);
```

Figure 1-3. Coding Conventions

As described in section 1, the static structure of a program is the organization of source code; that is, the blocks, do groups, and statements and their relationships. This section describes the dynamic structure of a program. The dynamic structure is the flow of control during program execution, the activation and termination of blocks, and the methods by which statements in one block reference other blocks and their associated generations of storage.

FLOW OF CONTROL

The collection of external procedures that constitute a PL/I program must include exactly one main procedure. The main procedure is the external procedure block with the OPTIONS(MAIN) clause designated in its PROCEDURE statement.

When PL/I program execution is initiated by an operating system control statement, the main procedure is activated at its primary entry point. The primary entry point is the PROCEDURE statement of the main procedure. Beginning with the first executable statement immediately contained in the main procedure, control passes sequentially from statement to statement.

The following are not executed when they are encountered in the sequential flow of control:

- Procedure block – The block is bypassed, and control passes to the statement following the END statement of the procedure. The block is executed only when it is invoked by execution of a CALL statement or by evaluation of a function reference.
- On-unit – The on-unit is established when the ON statement in which it appears is executed, but the on-unit is not executed at that time. The on-unit is executed only when the associated condition is raised.
- DECLARE, ENTRY, and FORMAT statements – These statements are passed through as if they were null statements.

Sequential flow of control is illustrated in figure 2-1.

When the following are executed, they cause diversion from sequential program flow:

- Procedure invocation – The procedure is activated and control is transferred to an entry point of the invoked procedure. Actions taken by the procedure determine whether or not control returns to the point of invocation.
- Condition raising – If the condition is enabled, the current established on-unit for that condition is activated and control is transferred to the first statement of the on-unit. Actions taken by the on-unit determine whether or not control returns to the block in which the condition was raised.
- GOTO statement – Control is unconditionally transferred to a labeled statement. One or more block activations can be terminated.

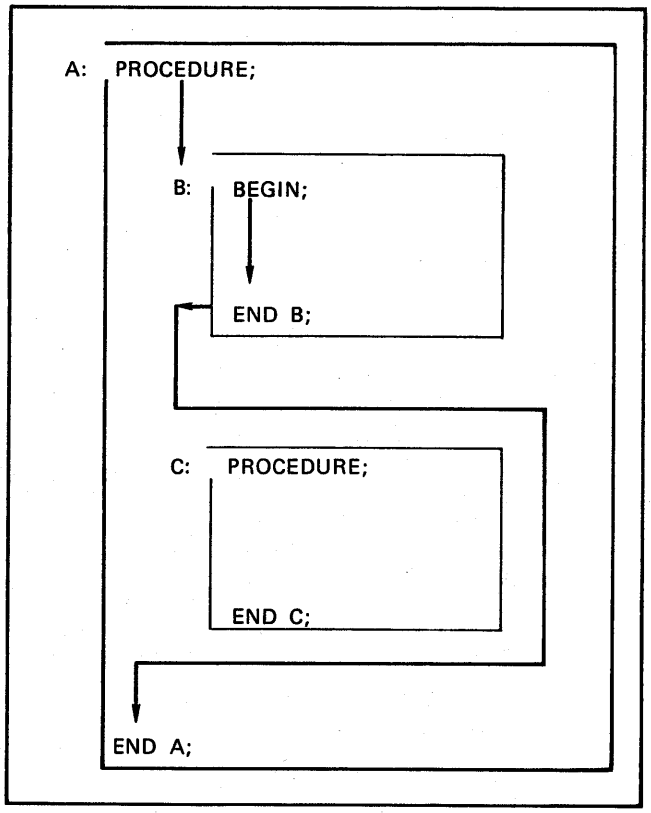


Figure 2-1. Sequential Flow of Control

- RETURN statement – Control is returned to the point at which the procedure was invoked or the block in which the condition was raised.
- IF statement – Flow of control is influenced by the evaluation of the expression in the IF statement. The flow of control takes one of two paths, depending on the value of the expression.
- Iterative do group – Control is influenced by the evaluation of expression values in the DO statement. On each iteration, control is passed to the statement following the DO statement or to the statement following the group END statement, depending upon the values of expressions in the DO statement.
- END statement of a procedure – The procedure activation is terminated. Control is returned to the point at which the procedure was invoked.
- END statement or only statement of an on-unit – When the END statement is executed, or after the only statement is executed, the on-unit activation is terminated. Control is returned to the block in which the condition was raised.

Diverted flow of control is illustrated in figure 2-2.

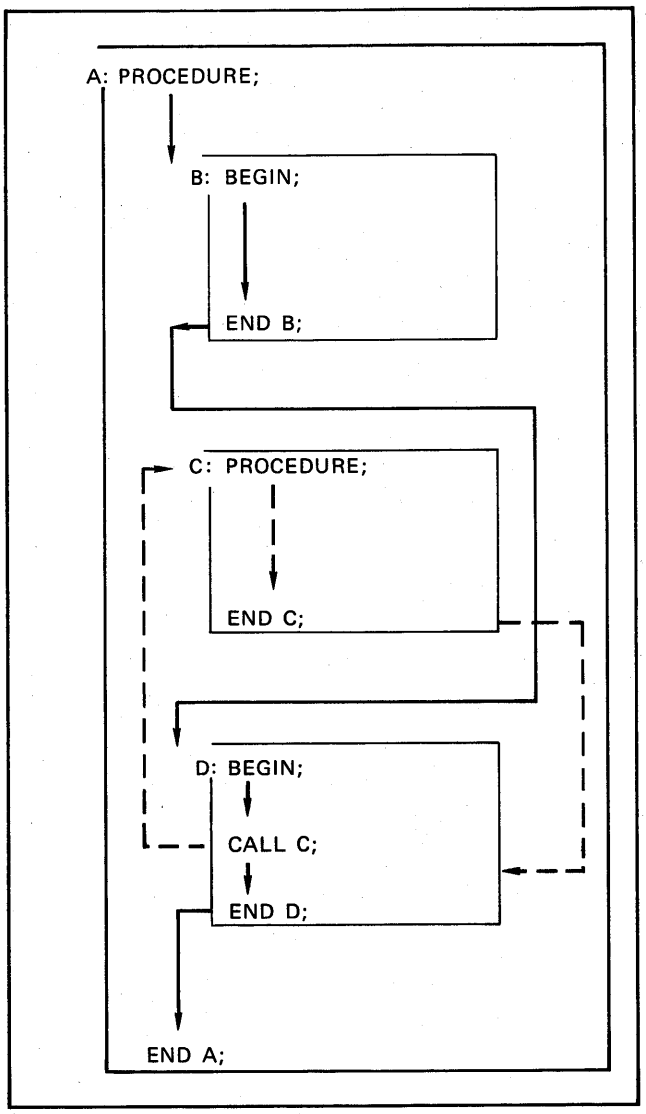


Figure 2-2. Diverted Flow of Control

BLOCK ACTIVATION

A block activation is a specific execution of a block. As stated in section 1, an on-unit acts like a block even when it is written as a single statement. All discussions of the properties and behavior of blocks in this section apply to on-units; the single statement on-unit always acts as if it had been written with surrounding BEGIN and END statements.

A block is activated under one of the following circumstances:

- The main procedure is initially activated when program execution is initiated by an operating system control statement.
- A procedure block is activated when it is invoked either by execution of a CALL statement or by evaluation of a function reference.
- A begin block is activated when control reaches the BEGIN statement through normal sequential flow or when control is transferred to the BEGIN statement by a GOTO statement.
- An on-unit is activated when the associated condition is raised and the condition is enabled.

Any block that has an unterminated activation is called an active block. Several blocks can be active at the same time; activation of one block does not cause termination of another.

An activation of a block that is already active is called a recursive block activation. A procedure block can be activated recursively only if the PROCEDURE statement of the block has the RECURSIVE option. A recursive call to a procedure can cause reactivation of already active begin blocks contained in the procedure.

When a block is activated, the following actions are taken in the order shown:

1. If a procedure block is being activated at an entry point that has parameters, extents (array bounds, string lengths, and area sizes) for the parameters are evaluated. Each parameter is associated with the corresponding argument.
2. Declarations of automatic and defined variables declared in the block are processed as follows: extents are evaluated, storage is allocated for each automatic variable, and initial values are assigned to automatic variables. Refer to section 4, Attributes, for additional details.
3. If an on-unit is being activated, appropriate condition builtin function values are established.

This process is called the prologue of the block activation.

BLOCK ACTIVATION RELATIONSHIPS

Block activations are maintained in a stack. The stack, known as the dynamic block activation stack, holds a set of system-maintained values for each activation. When a block is activated, the stack is pushed down and the new block activation is added at the top of the stack; thus, the values associated with unterminated activations are preserved.

The bottom activation in the stack is the first activation of the main procedure; the top activation in the stack is the current block activation. Each block activation is the immediate dynamic predecessor of the next (newer) one, and is the immediate dynamic successor of the previous (older) one. When the current block activation is terminated, it is removed from the stack and its immediate dynamic predecessor becomes the current block activation. The maximum number of activations in the stack at a given time is limited only by available storage.

The dynamic relationship between block activations is determined by the order of block activations and terminations, and not by the way blocks are statically nested in the source program. An activation of a begin block, for example, can be the dynamic predecessor of an activation of a procedure that contains the begin block; an activation of a begin block can also be the dynamic predecessor of an activation of an external procedure. An activation of a block can be the dynamic predecessor of another activation of the same block.

ENVIRONMENT OF A BLOCK ACTIVATION

A statement can reference an identifier (variable or named constant) that is declared in the same block as the reference or in a containing block. A reference to an identifier that is

declared in the same block as the reference is called a local reference. A reference to an identifier that is declared in a containing block is called a nonlocal reference.

If the dynamic block activation stack holds two or more activations of the same block simultaneously, multiple copies of certain variables exist. Each copy of a variable is called a generation as described in section 3, Data Elements. Multiple generations exist for each automatic variable, defined variable, and parameter variable declared in the block. In the same circumstances, multiple values exist for each label constant, entry constant, and format constant declared in the block.

When the program executes a reference to such variables or named constants, the system uses the following rules to determine which generation or value to use:

A local reference accesses the generation or value associated with the current block activation.

A nonlocal reference accesses the generation or value associated with an activation that is part of the environment of the current block activation.†

Each activation of an internal block has an environment that consists of precisely one activation of each containing block. The environment of a block activation is a subset of the dynamic block activation stack.

An activation of an external procedure has no environment. Nonlocal references can appear only in internal blocks.

As explained in section 1, each block except an external procedure is immediately contained in another block. An activation of an internal block has an immediate environment that is an activation of its immediate containing block. Since the immediate environment of a block activation can itself have an immediate environment, a list is formed; the environment of a block activation is this list of block activations.

When a block is activated or reactivated, there might be several activations of the immediate containing block in the block activation stack. Assume block B is immediately contained in block A. When B is activated to produce an activation B_k, a particular activation A_k of block A is selected as the immediate environment of B_k according to specific rules. Figure 2-3 gives six examples of environment selection. Each example is explained as follows:

Example 1

```
A: PROC OPTIONS(MAIN);
   B: BEGIN;
      .
      .
      .
      END B;
   END A;
```

Example 2

```
A: PROC OPTIONS(MAIN);
   DCL P ENTRY;
   ON COND(RED) BEGIN  /*ON-UNIT B*/;
                .
                .
                .
                END;
   CALL P;
   END A;
P: PROC;
   SIGNAL COND(RED);
   END P;
```

Example 3

```
A: PROC OPTIONS(MAIN);
   B: PROC;
      .
      .
      .
      END B;

   CALL B;  /*LOCAL REFERENCE TO B*/

   END A;
```

Example 4

```
A: PROC OPTIONS(MAIN);
   B: PROC;
      .
      .
      .
      END B;
   D: BEGIN;
      CALL B;  /*NONLOCAL REFERENCE TO B*/
      END D;
   END A;
```

Example 5

```
A: PROC OPTIONS(MAIN);
   B: PROC;
      .
      .
      .
      END B;
   C: PROC(X);
      DCL X ENTRY;
      CALL X;
      END C;

   CALL C(B);     /*LOCAL REFERENCE TO B*/
   END A;
```

Example 6

```
A: PROC OPTIONS(MAIN);
   B: PROC;
      .
      .
      .
      END B;
   C: PROC(X);
      DCL X ENTRY;
      CALL X;
      END C;
   D: BEGIN;
      CALL C(B); /*NONLOCAL REFERENCE TO B*/
      END D;
   END A;
```

Figure 2-3. Environment Selection

† Nonlocal references to a format constant are not permitted.

B is a begin block.

A_i is the immediate dynamic predecessor of B_k (example 1).

B is an on-unit.

A_i is the block activation in which B was established as an on-unit by execution of the ON statement that contains B (example 2).

B is a procedure block; the invocation of B directly references an entry name of B.

A_i is the activation of A that is either the immediate dynamic predecessor of B_k (example 3), or part of the environment of the immediate dynamic predecessor of B_k (example 4).

B is a procedure block; the invocation of B references an entry parameter whose value is an entry name of B.

A_i is the activation of A that was either the currently active block (example 5), or part of the environment of the currently active block (example 6) at the time when the constant entry name was passed as an argument to a procedure.

The environment of B_k is the list of immediate environments leading from B_k to an activation of the external procedure that contains B. It is possible for two activations of a block to have the same environment or different environments. Dynamic block activation environment concepts are illustrated in figure 2-4.

The rules for block activation and block termination ensure that A is active whenever B is active, and that A_i is in the dynamic block activation stack while B_k is active.

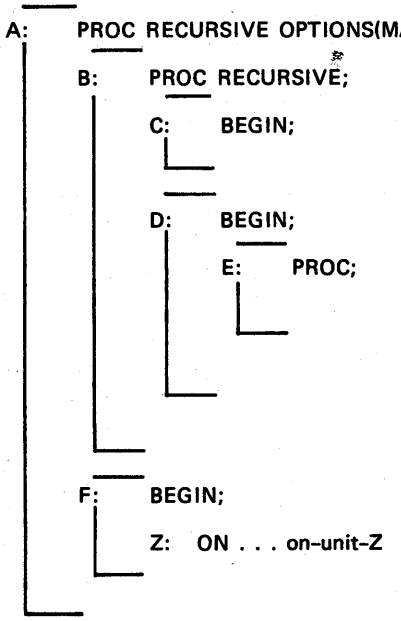
BLOCK TERMINATION

When a block activation is terminated, it is removed from the block activation stack and its dynamic predecessor becomes the current block activation.

Normal termination of the current block activation occurs under any of the following circumstances:

- A begin block or procedure block is currently active and control has reached its END statement. The current block activation is terminated.
- An on-unit that consists of a single statement is currently active and that statement has been processed without terminating the on-unit activation. The current on-unit activation is terminated.
- An on-unit that consists of a begin block is currently active and control has reached its END statement. The current on-unit activation is terminated.
- A procedure block or on-unit is currently active and control has reached a RETURN statement. The current procedure block activation or on-unit activation is terminated.

Normal procedure block, begin block, and on-unit termination is illustrated in figures 2-5, 2-6, and 2-7.



Block activations are denoted by the name of the block followed by a subscript; B_2, for example, represents the second activation in the program at block B. A possible situation might be:

	Dynamic Stack	Environment
top →	E_1	D_1 , B_2 , A_2
	D_1	B_2 , A_2
	B_2	A_2
on-unit-	Z_1	F_1 , A_2
	F_1	A_2
	A_2	none
	C_1	B_1 , A_1
	B_1	A_1
first activation →	A_1	none

The environment might also be displayed by arrows. Each arrow points to the immediate environment.

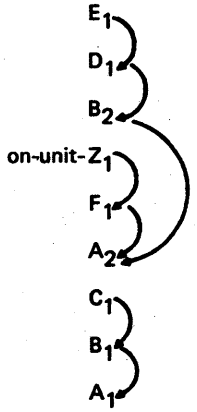
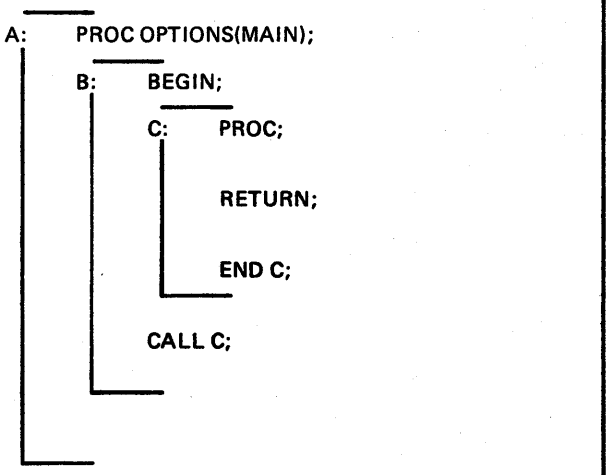


Figure 2-4. Dynamic Block Activation Environment



Block activations are denoted by the name of the block followed by a subscript.

Sample Dynamic Stack	Dynamic Stack After END C;or RETURN; Executed
top→C_1	B_1
B_1	A_1
first activation→A_1	Control passes to statement after CALL C;

Figure 2-5. Normal Procedure Block Termination

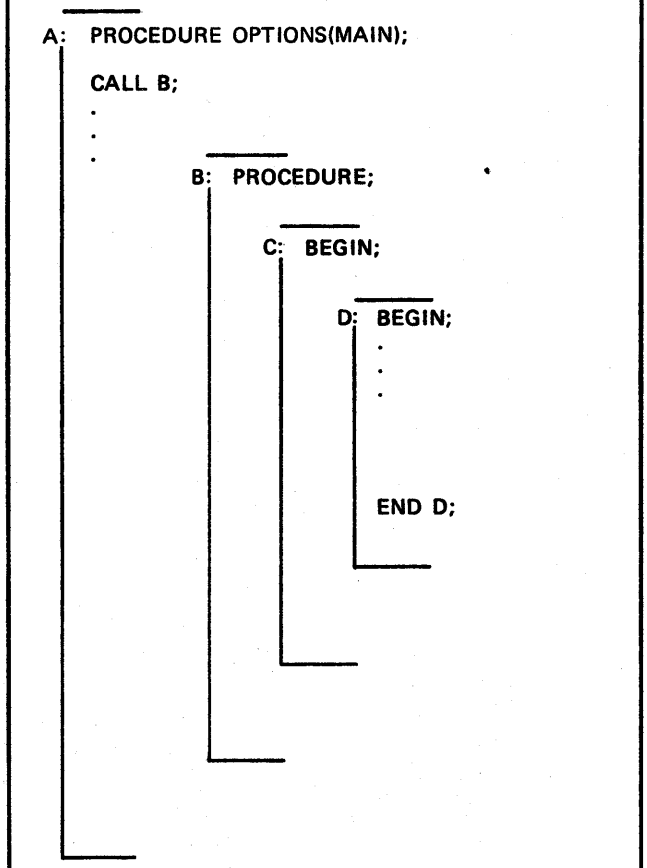
Abnormal termination of the current block activation occurs under any of the following circumstances:

- A begin block is currently active and control has reached a RETURN statement. The current activation of the begin block is terminated; and one or more of its dynamic predecessors are also terminated, up to and including the most recent procedure block or on-unit activation. The begin block activations are terminated abnormally; the procedure or on-unit activation is terminated normally. Control returns to the dynamic predecessor of the most recently activated procedure block or on-unit.

- A begin block, a procedure block, or an on-unit is currently active and control has reached a GOTO statement that transfers control to a statement not immediately contained in the same block activation as the GOTO statement. This is referred to as a nonlocal GOTO. The current block activation is terminated, as well as all of its dynamic predecessors up to (but not including) the block activation that immediately contains the statement referenced by the GOTO statement.

Abnormal block and on-unit termination is illustrated in Figures 2-8 and 2-9.

A GOTO statement that transfers control to a multiple-closure END statement can cause normal termination of one block activation and abnormal termination of its dynamic



Block activations are denoted by the name of the block followed by a subscript.

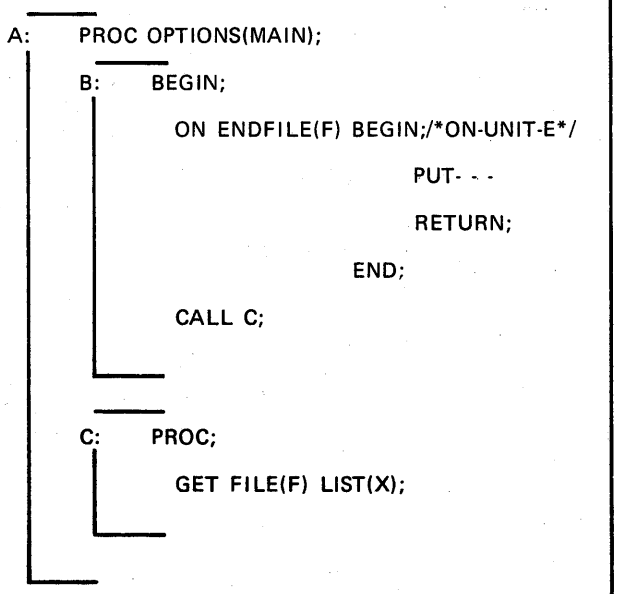
Sample Dynamic Stack	Dynamic Stack After END D; Executed
top→D_1	C_1
C_1	B_1
B_1	A_1
first activation→A_1	Control passes to statement after END D;

Figure 2-6. Normal Begin Block Termination

successors. Block termination and multiple closure concepts are illustrated in figure 2-10. Additional information regarding multiple closure appears under Block and Do Group Closure in section 1, PL/I Source Program.

When a block activation is terminated normally or abnormally, the following must be considered:

- Generations for automatic variables allocated for the block activation are freed.
- All values associated with the activation are discarded.



Block activations are denoted by the name of the block followed by a subscript.

Sample Dynamic Stack	Dynamic Stack After RETURN; Executed
top→ on-unit-E_1	C_1
C_1	B_1
B_1	A_1
first activation→ A_1	Control passes to statement after GET.

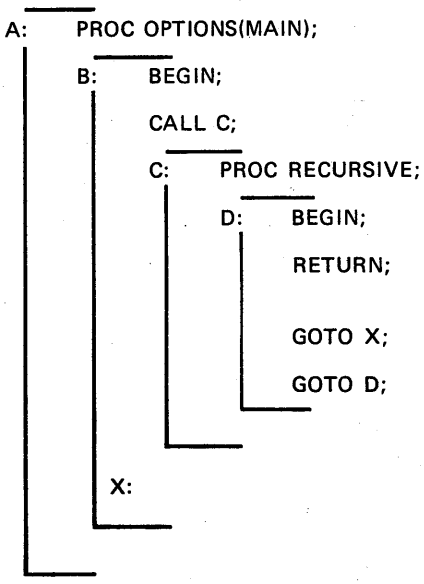
NOTE: If the ON statement were ON ENDFILE(F) RETURN; the RETURN statement would still pass control to the statement after GET.

Figure 2-7. Normal On-Unit Termination

- The termination can cause termination of one or more other block activations; specifically, a RETURN statement in a begin block, or a nonlocal GOTO in any block, can cause termination of one or more dynamic predecessors of the current block activation.
- When a procedure block is activated, execution of the statement that invoked the procedure is partially complete. The results of procedure block termination are as follows:

  After normal block termination, processing of the invoking statement continues.

  After abnormal block termination, processing of the invoking statement is never completed.



Block activations are denoted by the name of the block followed by a subscript.

Sample Dynamic Stack	Dynamic Stack After GOTO D; and D: BEGIN; Executed
top→ D_2	D_3
C_2	C_2
D_1	D_1
C_1	C_1
B_1	B_1
first activation→ A_1	A_1

GOTO D; terminates the current block activation (D_2) and causes D to be activated again (as D_3). All values associated with D_2 are discarded.

Dynamic Stack After RETURN; Executed	Dynamic Stack After GOTO X; Executed
B_1	B_1
A_1	A_1
Control passes to statement after CALL C;	

Figure 2-8. Abnormal Block Termination

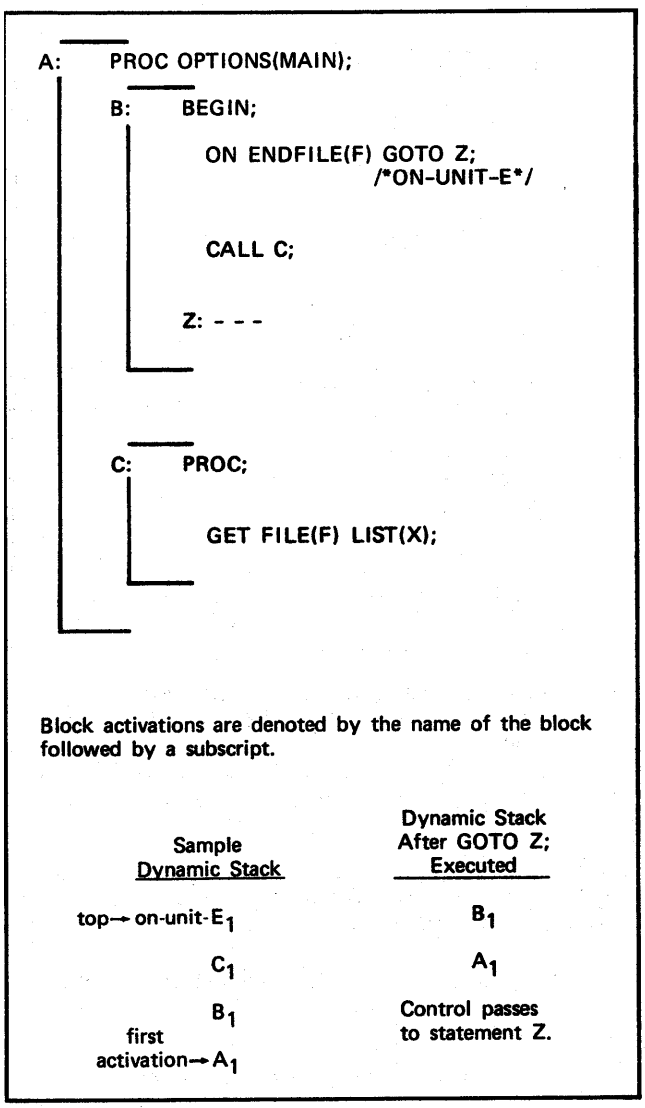


Figure 2-9. Abnormal On-Unit Termination

- When an on-unit is activated, execution of the statement in which the condition was raised is partially complete. The results of on-unit termination are as follows:

  After normal on-unit termination, control passes to the point at which the condition was raised or to the following statement, depending upon the specific condition. Processing of the statement in which the condition was raised, therefore, can continue or remain incomplete. Normal termination of the following on-units is prohibited: ERROR, FIXEDOVERFLOW, OVERFLOW, SIZE, STRINGRANGE, SUBSCRIPTRANGE, and ZERODIVIDE. Refer to section 10, Conditions, for further details.

  After abnormal on-unit termination, processing of the statement in which the condition was raised is never completed.

- Abnormal block termination can result in incomplete I/O activity.

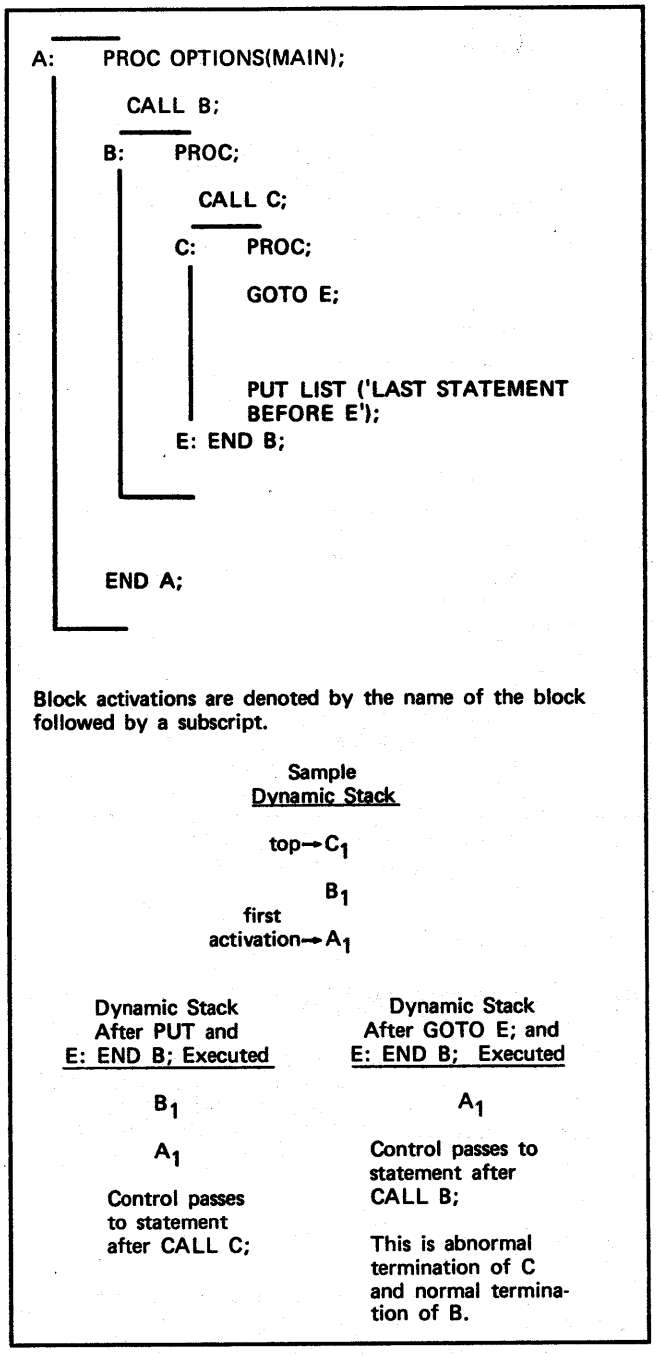


Figure 2-10. Block Termination and Multiple Closure

- A nonlocal GOTO must not cause the termination of a block activation if the immediate dynamic predecessor of that block activation is currently performing any of the following operations:

  Evaluation of extents of automatic or defined variables

  Allocation of storage for automatic variables

  Initialization of automatic variables

  Execution of an ALLOCATE statement

  Allocation of the based variable in a LOCATE statement

Figure 2-11 illustrates the effects of a nonlocal GOTO during storage allocation.

```
P: PROC(I);
   DCL A(F(I))FLOAT AUTOMATIC;
   /*FUNCTION REFERENCE INVOKES F*/
   DCL I FIXED BIN;
   F: PROC(N) RETURNS(FIXED BIN);
      DCL N FIXED BIN;
      .
      .
      .
      GOTO LAB;
      /*ILLEGAL-WILL RAISE ERROR*/
      /*WHEN EXECUTED            */
      .
      .
      .
      RETURN(...);
      END F;
   .
   .
   .
   LAB:...;
   END P;
```

Note: F must not contain a reference to any automatic, parameter, or defined variable declared in P because the prologue of P has not been completed when F is invoked.

Figure 2-11. Attempted Nonlocal GOTO During Storage Allocation

```
P: PROC OPTIONS(MAIN);
   ON FINISH BEGIN; /*ON-UNIT-X*/
             .
             .
             .
             END;

   B: BEGIN;
      ON FINISH BEGIN;  /*ON-UNIT-Y*/
                .
                .
                .
                END;
      RETURN;
      .
      .
      .
      STOP;
      END B;
   END P;
```

If the RETURN statement is executed, block B is terminated and on-unit-X is activated; the dynamic stack holds on-unit-X_1 and P_1.

If the STOP statement is executed, on-unit-Y is activated; the dynamic stack holds on-unit-Y_1, B_1, and P_1.

Figure 2-12. Program Termination

PROGRAM TERMINATION

Program termination occurs when the first or only activation of the main procedure is terminated or when a STOP statement is executed. Each type of termination is illustrated in figure 2-12 and described in the following paragraphs.

MAIN PROCEDURE TERMINATION

When control reaches the END statement of the first or only activation of the main procedure, or when a RETURN statement is executed that would cause the termination of the first or only activation of the main procedure, the following steps are taken:

1. If the statement executed was a RETURN statement contained in one or more nested begin blocks, the begin block activations are terminated.

2. The FINISH condition is raised. The first activation of the main procedure block is still active at this point.

3. If a programmed FINISH on-unit is established in the first activation of the main procedure, the on-unit is activated. The on-unit can be written to execute special termination processes. If the on-unit activation is terminated by execution of a nonlocal GOTO, program execution continues.

4. On normal termination of or in the absence of an established programmed FINISH on-unit, the system performs the following:

   Terminates the main procedure block activation.

   Closes all open files.

   Returns control to the operating system.

STOP STATEMENT EXECUTION

When a STOP statement is executed, the following steps are taken:

1. The FINISH condition is raised. More than one block can still be active at this point. In addition, incomplete storage allocations or incomplete I/O activity can exist.

2. If there is an established programmed on-unit for the FINISH condition, the on-unit is activated. The on-unit can be written to execute special termination processes. If the on-unit activation is terminated by execution of a nonlocal GOTO, program execution continues, but incomplete storage allocations or I/O activity can exist.

3. On normal termination of or in the absence of an established programmed FINISH on-unit, the system performs the following:

   Terminates all block activations.

   Closes all open files.

   Returns control to the operating system.

PROGRAM ABORT

Program execution is aborted under the following circumstances:

- The system ERROR on-unit is activated. When this occurs, the system closes all files and aborts the program (transfers control to the next appropriate EXIT control statement).
- The STORAGE condition is raised, and the system cannot obtain enough storage to print the snap output and activate the STORAGE on-unit. When this occurs, the system aborts the program. Files might not be closed, and error messages describing the abort might be absent or incomplete.
- Time limit is reached. When this occurs, the system writes a message on the dayfile, attempts to close all files, and aborts the program.
- A missing external procedure (unsatisfied external) is invoked. When this occurs, the system writes a message on the dayfile, attempts to close all files, and aborts the program.

The following circumstances will cause incorrect program execution and might result in mode errors:

- A disabled computational condition is raised.
- The source program has undiagnosed logic errors; for example, the use of a bad locator value.

When a mode error occurs, the system writes a message on the dayfile, attempts to close all files, and aborts the program.

When program abort is unable to close files, some information on those files might be lost. Program abort does not raise the FINISH condition.

This section describes the data elements that specify or represent values in the PL/I program. Each program includes a variety of data elements used for different purposes. The data elements are various types of literal constants, named constants, and variables.

Each data element has a data type. The data type indicates the way in which the data element can be used. Data types are divided into two groups, computational and noncomputational.

Each data element has an aggregate type. The aggregate type indicates whether the data element is a scalar that represents a single element or an aggregate that represents an organization of data elements.

Each data element that is a variable has a storage type. The storage type determines the way in which the variable is associated with storage.

Data type, aggregate type, and storage type are some of the characteristics associated with data elements. Characteristics of each data element are dependent on the attributes of the data element. Attributes are described in section 4, Attributes.

LITERAL CONSTANTS

A literal constant specifies a value that is constant throughout program execution. A literal constant is unnamed and has a scalar computational value. The three types of literal constants are

- Arithmetic constant
- Character string constant
- Bit string constant

ARITHMETIC CONSTANT

An arithmetic constant specifies an arithmetic value. A set of arithmetic attributes is implied by the actual form of the literal constant. Attributes indicate the following properties of the value:

- REAL mode
- DECIMAL or BINARY base
- FIXED or FLOAT scale
- precision (p,q) or (p)

Fixed Point Decimal Constant

A fixed decimal constant is an unsigned decimal number containing an optional decimal point at any position. Blanks are not permitted in the constant. The syntax of a fixed decimal constant is shown in figure 3-1.

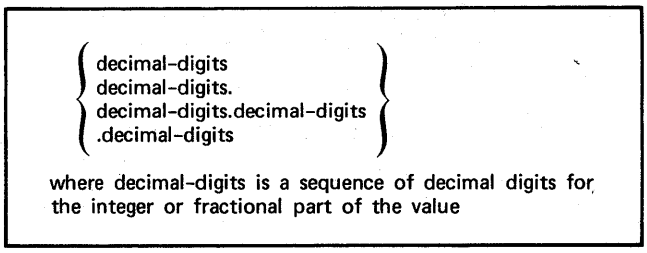


Figure 3-1. Fixed Decimal Arithmetic Constant Syntax

The implied attributes are REAL, FIXED, DECIMAL, and precision (p,q). The precision p is the number of digits in the constant, including leading and trailing zeros. No more than 14 digits can be specified. If the constant has a decimal point, q is the number of digits to the right of the decimal point, including trailing zeros. If the constant has a decimal point but no digits are to the right, q is 0. If the constant has no decimal point, the decimal point is assumed to be to the right of the decimal number and q is 0. The following are examples of fixed decimal arithmetic constants, with the implied attributes shown:

0	REAL FIXED DECIMAL (1,0)
2156.	REAL FIXED DECIMAL (4,0)
21.2000	REAL FIXED DECIMAL (6,4)
.400	REAL FIXED DECIMAL (3,3)

The maximum value for a fixed decimal constant is 99999999999999 (14 digits). The minimum value is zero, and the minimum nonzero value is .00000000000001 (14 digits). A specified value such as -3.1 is a type of expression.

Floating Point Decimal Constant

A float decimal constant is an unsigned decimal number containing an optional decimal point at any position and followed by an exponent. The exponent is the letter E immediately followed by an optionally signed decimal integer. Blanks are not permitted in the constant. The syntax of a float decimal constant is shown in figure 3-2.

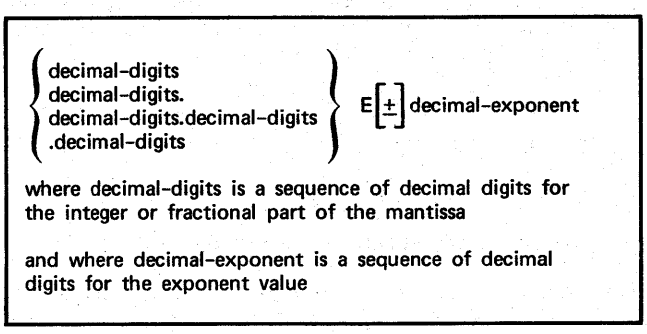


Figure 3-2. Float Decimal Arithmetic Constant Syntax

The implied attributes are REAL, FLOAT, DECIMAL, and precision (p). The precision p is the number of digits in the mantissa, including leading and trailing zeros. No more than 14 digits can be specified in the mantissa. If there is no decimal point, the decimal point is assumed to be to the right of the mantissa. The value is the mantissa multiplied by the base (10) raised to the power of the specified exponent. The exponent is specified in decimal digits and can be signed. The following are examples of float decimal arithmetic constants, with the implied attributes shown:

0E0	REAL FLOAT DECIMAL (1)
10.E2	REAL FLOAT DECIMAL (2)
326.4300E-4	REAL FLOAT DECIMAL (7)
.7E+48	REAL FLOAT DECIMAL (1)

The maximum value for a float decimal constant is approximately 1.265E+322. The minimum value is zero, and the minimum nonzero value is approximately 3.132E-294. A specified value such as -4E+20 is a type of expression.

Fixed Point Binary Constant

A fixed binary constant is an unsigned binary number containing an optional binary point at any position and followed by the letter B. The B is a radix factor that signals a binary value. Blanks are not permitted in the constant. Each digit in the binary number must be a 0 or a 1 digit. The syntax of a fixed binary constant is shown in figure 3-3.

{ binary-digits B
binary-digits. B
binary-digits.binary-digits B
.binary-digits B }

where binary-digits is a sequence of binary digits for the integer or fractional part of the value

Figure 3-3. Fixed Binary Arithmetic Constant Syntax

The implied attributes are REAL, FIXED, BINARY, and precision (p,q). The precision p is the total number of binary digits, including leading and trailing zeros. The total number of digits specified cannot exceed 48. If the constant has a binary point, q is the number of digits to the right of the binary point, including trailing zeros. If the constant has a binary point but no digits are to the right, q is 0. If the constant has no binary point, the binary point is assumed to be to the right of the binary number and q is 0. The following are examples of fixed binary arithmetic constants, with the implied attributes shown:

0B	REAL FIXED BINARY (1,0)
1111.B	REAL FIXED BINARY (4,0)
1.111B	REAL FIXED BINARY (4,3)
.010B	REAL FIXED BINARY (3,3)

The maximum value of a fixed binary constant is 1111...1111B (48 digits). The minimum value is zero, and the minimum nonzero value is .0000...0001B (48 digits). A specified value such as -.101B is a type of expression.

Floating Point Binary Constant

A float binary constant is an unsigned binary number containing an optional binary point at any position, followed by an exponent, and then followed by the letter B. The B is a radix factor that signals a binary value. Blanks are not permitted in the constant. Each digit in the binary number must be a 0 or a 1 digit. The exponent consists of the letter E immediately followed by an optionally signed decimal integer. The syntax of a float binary constant is shown in figure 3-4.

{ binary-digits
binary-digits.
binary-digits.binary-digits
.binary-digits } E[±]decimal-exponent B

where binary-digits is a sequence of binary digits for the integer or fractional part of the mantissa

where decimal-exponent is a sequence of decimal digits for the exponent value

Figure 3-4. Float Binary Arithmetic Constant Syntax

The implied attributes are REAL, FLOAT, BINARY, and precision (p). The precision p is the number of binary digits in the mantissa, including leading and trailing zeros. The number of binary digits specified in the mantissa cannot exceed 48. If there is no binary point, the binary point is assumed to be to the right of the binary mantissa. The value is the mantissa multiplied by the base (2) raised to the power of the specified exponent. The exponent is specified in decimal digits and can be signed. The following are examples of float binary arithmetic constants, with the implied attributes shown:

0000E0B	REAL FLOAT BINARY (4)
11.E2B	REAL FLOAT BINARY (2)
101.11E4B	REAL FLOAT BINARY (5)
.001E-15B	REAL FLOAT BINARY (3)

The maximum value of a float binary constant is 1.111...1111E+1069B (48 digits in the mantissa). The minimum value is zero, and the minimum nonzero value is 1.E-975B. A specified value such as -11E-2B is a type of expression.

CHARACTER STRING CONSTANT

A character string constant (also called a character constant) specifies a character value. A simple character constant is a sequence of zero or more characters enclosed in apostrophes. An apostrophe character in the character value is represented by two consecutive apostrophes. An unpaired apostrophe cannot appear inside the character constant, since an unpaired apostrophe would end the character constant.

A replicated character constant consists of a repetition factor and a simple character constant. The character value consists of repetitions of the same sequence of characters. If the repetition factor is 1, a single copy is used. If the repetition factor is 0, the character constant is a null string. If the repetition factor is greater than 1, multiple copies are

concatenated to form the character value. For example, a string of forty asterisks can be written as (40)'*'. The syntax of a character string constant is shown in figure 3-5.

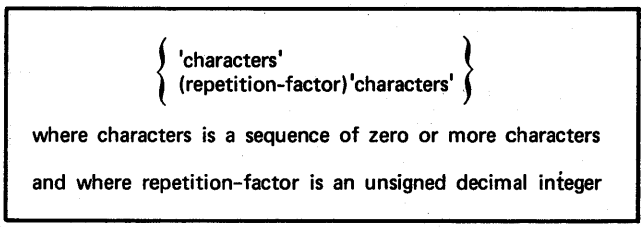


Figure 3-5. Character Constant Syntax

The implied attributes are CHARACTER and nonvarying. The length is the number of characters in the character value, after any repetition factor has been applied. Maximum length of a character constant is 16383 characters. The following are examples of character constants, with the implied attributes shown:

''	CHARACTER(0) nonvarying; null string
'ABC'	CHARACTER(3) nonvarying; value=ABC
'IT''S'	CHARACTER(4) nonvarying; value=IT'S
(3)'OK'	CHARACTER(6) nonvarying; value=OKOKOK

BIT STRING CONSTANT

A bit string constant (also called a bit constant) specifies a bit value. A simple bit constant is a sequence of zero or more bits (each bit 0 or 1) enclosed in apostrophes and followed by the letter B. The B is a radix factor that distinguishes a bit constant from a character constant. A bit constant does not represent a binary arithmetic value.

A replicated bit constant consists of a repetition factor and a simple bit constant. The bit value consists of repetitions of the same sequence of bits. If the repetition factor is 1, a single copy is used. If the repetition factor is 0, the bit constant is a null string. If the repetition factor is greater than 1, multiple copies are concatenated to form the bit value. For example, a string of sixty zero bits can be specified as (60)'0'B. The syntax of a bit string constant is shown in figure 3-6.

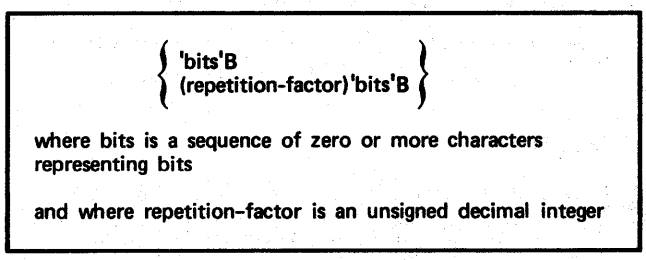


Figure 3-6. Bit Constant Syntax

The implied attributes are BIT and nonvarying. The length is number of bits in the bit value, after any repetition factor has been applied. Maximum length of a bit constant is 16383 bits. The following are examples of bit constants, with the implied attributes shown:

''B	BIT(0) nonvarying; null string
'1101'B	BIT(4) nonvarying; value=1101
(2)'101'B	BIT(6) nonvarying; value=101101

NAMED CONSTANTS

A named constant is an identifier that represents constant noncomputational values. Named constants have attributes that are not inherent in the form of the constant. Named constants have values that are not complete until the constants are referenced. A named constant is either an entry constant, a file constant, a format constant, or a label constant.

ENTRY CONSTANT

An entry constant identifies an entry point of a procedure. An entry point is a point at which the procedure can be invoked, as well as a complete description of all parameters and the returned value, if any. An entry constant is part of an entry value. An entry value identifies both an entry point and a block activation. When an entry value is used to invoke a procedure, the entry point part identifies the PROCEDURE or ENTRY statement at which execution begins, and the block activation part specifies the immediate environment of the new procedure activation.

An entry constant that identifies an entry point of an internal procedure is an internal entry constant. An internal entry constant is declared in the block that immediately contains the procedure. An entry constant represents a name of a procedure, and in a sense is prefixed to the procedure. Each time an internal entry constant is referenced, an entry value is constructed by associating the entry constant with a block activation. The block activation used is the most recent activation of the block that immediately contains the referenced procedure.

An entry constant that identifies an entry point of an external procedure is an external entry constant. The declaration is not contained in any block. No block activation is needed to complete the entry value, because an activation of an external procedure has no environment.

Each entry prefix on a PROCEDURE or ENTRY statement is declared as an entry constant. An entry constant can be internal or external. All entry constants have the ENTRY and constant attributes. DECLARE statement declaration of an external entry constant is necessary only to set up communication between external procedures.

An entry constant can be used in the following ways:

- As a subroutine reference in a CALL statement, if the entry constant represents a subroutine entry point
- As a function reference, if the entry constant represents a function entry point
- As an argument passed to an invoked procedure

Entry values cannot be returned by a function. Entry values cannot be assigned or compared. Arrays of entry constants are not supported.

Examples of entry constants are shown in figure 3-7.

FILE CONSTANT

A file constant serves as a linkage that enables a program to communicate with external data storage. External data storage is an organized collection of data. A program can access CYBER Record Manager (CRM) files, using them as input to the program, updating them, or creating them as output.

```
P:
PROCEDURE OPTIONS(MAIN);
/*P IS EXTERNAL ENTRY CONSTANT          */
   DCL STR CHAR(5);
   •
   •
   •
   CALL ZC(STR);
ZC:
   PROCEDURE (ASTRING);
   /*ZC IS INTERNAL ENTRY CONSTANT      */
      DCL ASTRING CHAR(5);
      •
      •
      •
   END ZC;
END P;
```

Figure 3-7. Entry Constant Examples

The declaration of a file constant does not create a complete linkage. All declarations except those for file constants are completed during compilation. A file constant has a special set of attributes, called file description attributes, that are not necessarily complete until program execution. A file constant is not associated with a CRM file until the file constant is opened.

A file constant must be associated with a CRM file before data can be transmitted. A file constant can be repeatedly opened and closed during program execution. When a file constant is opened, the set of file description attributes is completed. When the file is closed, the file constant is restored to the same status as before opening.

File constants can be declared by DECLARE statement, and file description attributes can be specified. Each file constant can be internal or external. File constants are external by default. All file constants have the FILE and constant attributes. In some cases, file constants are recognized by usage and are therefore declared contextually.

A file constant can be used in the following ways:

- As the file reference in a FILE option in an I/O statement
- As the file reference in a COPY option in a GET statement
- As the file reference in an I/O condition in an ON, REVERT, or SIGNAL statement
- As an argument passed to the PAGENO builtin function or pseudovariable
- As an argument passed to an invoked procedure

File values cannot be returned by a function. File values cannot be assigned or compared. Arrays of file constants are not supported.

An example of a file constant is shown in figure 3-8.

FORMAT CONSTANT

A format constant identifies a FORMAT statement. A format constant is only part of a format value. A format value identifies both a FORMAT statement and a block activation. When a format value is referenced by an R (remote) format item, the statement part identifies the format to be used, and the block activation part identifies the activation in which references in the FORMAT statement are to be evaluated.

```
DECLARE WORK4 FILE OUTPUT;
/*WORK4 IS FILE CONSTANT                */
•
•
•
OPEN FILE(WORK4);
DO WHILE(COUNT < 62);
   •
   •
   •
   PUT FILE(WORK4) LIST(TVAL,JVAL,DIFF);
END;
CLOSE FILE(WORK4);
```

Figure 3-8. File Constant Example

Each format prefix on a FORMAT statement is declared as a format constant in the block that immediately contains the statement. All format constants are internal. All format constants have the format and constant attributes.

A format constant can be referenced only in an R format item in a format specification in a GET, PUT, or FORMAT statement. A reference to a format constant must be a local reference.

Format values cannot be passed as arguments or returned by functions. Format values cannot be assigned or compared. Arrays of format constants are not supported.

An example of a format constant is shown in figure 3-9.

```
GET FILE(F) EDIT(MODELNAME,MODELNO,OPT)
                (R(FORMAT6));
•
•
•
FORMAT6:
   FORMAT (SKIP,A(12),X,F(15,3),X,B(5));
/*FORMAT6 IS FORMAT CONSTANT            */
```

Figure 3-9. Format Constant Example

LABEL CONSTANT

A label constant identifies a statement of any type other than PROCEDURE, ENTRY, or FORMAT. A label constant is only part of a label value. A label value identifies both a statement and a block activation. When a label value is used to transfer control to a statement, the statement part identifies the statement to which control is transferred, and the block activation part specifies the new current block activation after the transfer of control is completed.

Each label prefix on a statement other than PROCEDURE, ENTRY, or FORMAT is declared as a label constant in the block immediately containing the statement. All label constants are internal. All label constants have the LABEL and constant attributes. Each label prefix is either a scalar label constant or an element of a label constant array. A scalar label constant must be unique within the immediately containing block.

An array of label constants has the identifier used for all elements of the array and a set of subscripts. The subscripts of each element of the array must be unique within the array. All elements of an array of label constants must be immediately contained within the same block. The number of subscripts must be the same for every element of the array. Each subscripted label prefix declares one element of the array. In addition to the declaration of individual elements used as label prefixes, the array of label constants must be declared by DECLARE statement in the same block.

A label constant can be used in the following ways:

- As the label reference in a GOTO statement
- As the source expression in an assignment statement
- As the start expression in a DO statement, or in an embedded-do in a GET or PUT statement
- As an argument passed to an invoked procedure

Label values cannot be returned by a function. Label values cannot be compared.

Examples of label constants are shown in figure 3-10.

```
DECLARE LAB(3) LABEL;
.
.
.
LOOP3:
/*LOOP3 IS LABEL CONSTANT                    */
.
.
.
LAB(3):
/*LAB(3) IS ELEMENT                          */
/*IN ARRAY OF LABEL CONSTANTS                */
```

Figure 3-10. Label Constant Examples

VARIABLES

A variable is an identifier that represents values that can change during program execution. A variable can be referenced for its current value at any time during program execution. A variable has an undefined value until a value is assigned.

Each variable has generations that are described later in this section. Each generation associated with a variable can hold any value appropriate for the variable. Generations are allocated (that is, created and reserved) at different times, depending on the type of storage for each variable. The storage types for variables are described later in this section.

Variables are either computational or noncomputational. The computational variables are used in computational expressions and in assignment operations. The noncomputational variables are used for other purposes. Variables can be scalars or aggregates. A variable that is an aggregate can be an array, a structure, or a combination of array and structure organizations. Aggregates are described later in this section.

COMPUTATIONAL VARIABLE

A computational variable can have an arithmetic value, a string value, or a pictured value. Each computational variable has attributes that specify the computational data type of the variable. The attributes are supplied by DECLARE statement declaration or by default.

A computational variable can acquire values in the following ways:

- In assignment statement execution
- In GET statement execution
- In READ statement execution
- During initialization according to the INITIAL attribute
- By assignment to the STRING option variable during PUT statement execution
- In argument passing during procedure invocation
- By assignment of a value to the index variable during DO statement execution, or execution of an embedded-do in a GET or PUT statement
- By assignment to the KEYTO option variable during READ statement execution

Arithmetic Variable

An arithmetic variable has values that are numbers. The specific arithmetic attributes of the variable indicate the mode, the scale, the base, and the precision of the arithmetic values. The attributes specify the characteristics of the values for arithmetic calculations and for possible conversion to another arithmetic data type or another computational data type. All arithmetic variables have the variable attribute.

The mode of an arithmetic value is indicated by the REAL attribute.

The scale of an arithmetic value is either fixed point or floating point, as indicated by the FIXED or FLOAT attribute. A fixed point value is one for which a constant number of digits is maintained. The position of those digits with respect to the decimal or binary point of the number is fixed. A fixed point value has a constant scale, but the number of significant digits is variable because the value can contain leading zeros. A floating point value is one for which a constant number of significant digits is maintained. The position of the decimal or binary point can float with respect to those digits. A floating point value never contains a leading zero. A floating point value is often represented by a mantissa and an exponent.

The base of an arithmetic value is either decimal or binary, as indicated by the DECIMAL or BINARY attribute. The base affects the results of conversions from one data type to another. The base also affects the accuracy maintained for fixed point values. For example, the value 0.1 can be expressed precisely as a fixed point decimal value. The same value cannot be represented precisely as a binary value because it requires an infinitely repeating fraction. FLOAT DECIMAL values are represented internally as FLOAT BINARY. Values that are repeating binary fractions cannot be maintained precisely as FLOAT DECIMAL.

The precision of an arithmetic value indicates the number of decimal or binary digits to be maintained for the value. For floating point, the precision is simply the number of significant digits maintained. The representation of a floating point value never contains leading zeros. For fixed point, the precision has two parts. One part is the number of maintained digits, usually referred to as the precision p. The other part is a scale factor, usually referred to as scale factor q. The number of significant digits is not constant because a fixed point value can contain leading zeros. The scale factor specifies the position of the least significant digit maintained with respect to the decimal or binary point. A variable declared as FIXED DECIMAL (p,q) can maintain precisely q decimal digits to the right of the decimal point, if q is greater than zero. If q is less than zero, then p digits counting from the left are maintained, and ABS(q) decimal digits immediately to the left of the decimal point are discarded.

The maximum and minimum values for arithmetic variables are shown in table 3-1. Examples of arithmetic variables are shown in figure 3-11.

TABLE 3-1. MAXIMUM AND MINIMUM VALUES FOR ARITHMETIC VARIABLES

Arithmetic Attributes	Maximum and Minimum Value	Minimum Positive Value
FIXED DEC(14,0)	$\pm(10^{14}-1)$	1
FIXED DEC(14,255)	$\pm(10^{14}-1)*10^{-255}$	10^{-255}
FIXED DEC(14,-255)	$\pm(10^{14}-1)*10^{255}$	10^{255}
FLOAT DEC	$\sim\pm 1.265*10^{322}$	$\sim 3.132*10^{-294}$
FIXED BIN(48,0)	$\pm(2^{48}-1)$	1
FIXED BIN(48,255)	$\pm(2^{48}-1)*2^{-255}$	2^{-255}
FIXED BIN(48,-255)	$\pm(2^{48}-1)*2^{255}$	2^{255}
FLOAT BIN	$\sim\pm 1.265*10^{322}$	$\sim 3.132*10^{-294}$

```
DECLARE J FIXED DECIMAL;
DECLARE VALUEI FIXED DECIMAL;
DECLARE DF FLOAT BINARY;
•
•
•
GET LIST(J);
VALUEI = 100.00 * (J - 15);
•
•
•
GET LIST(DF);
DF = DF * 5.127864;
```

Figure 3-11. Arithmetic Variable Examples

String Variable

A string variable has values that are either character strings or bit strings. A string variable has attributes that specify the type of string, the length, and the variability of the length. All string variables have the variable attribute.

The type of string is indicated by the CHARACTER attribute or the BIT attribute. The length is associated with the CHARACTER or BIT attribute to specify the length of the string value in characters or bits. The variability of the length is determined by the VARYING or nonvarying attribute. If the string variable is VARYING, the specified length is the maximum length of the string rather than the current length of the string as set by assignment.

The maximum length of a string variable is 131071 characters or bits. Some examples of string variables are shown in figure 3-12.

```
DECLARE TSWITCH BIT(1);
DECLARE STR1 CHAR(4);
DECLARE STR2 CHAR(10);
STR1 = 'TEST';
•
•
•
IF (TSWITCH)
   THEN STR2 = STR1 || 'A12';
```

Figure 3-12. String Variable Examples

Pictured Variable

A pictured variable has values that are character values. The picture specification for the pictured variable controls the characters or digits that can be present at any position in the character string value.

A pictured variable is declared as a pictured numeric item or a pictured character item. A pictured character item has only a character string value. A pictured numeric item has a character string value and can be interpreted for an arithmetic value. Each pictured variable has the PICTURE attribute. Picture codes for pictured variables are described in section 7, Data Manipulation. All pictured variables have the variable attribute.

A pictured character item can contain any characters that are valid according to the picture specification. A pictured character item differs from a character string variable in the way values are assigned. A character value assigned to a pictured character item is validated to ensure that all characters conform to the picture specification. The length of the character string value is nonvarying and is implied by the picture specification.

A pictured numeric item has a value that is an edited character string representation of an arithmetic value. An arithmetic value assigned to the pictured numeric variable is edited through the picture and stored as a character string value. Unlike an arithmetic variable, a pictured numeric item cannot be used directly in arithmetic computations and must first be interpreted for an arithmetic value. An interpreted arithmetic value has the same arithmetic properties as arithmetic variables, with the exception that binary values cannot be represented. The mode is always REAL, and the base is always DECIMAL. The scale and precision are implied by the picture specification. The length of the character string value is nonvarying and is also implied by the picture specification. Some examples of pictured variables are shown in figure 3-13.

```
DECLARE ACCOUNT PICTURE'AA9999';
/*PICTURED CHARACTER                    */
DECLARE PAYMENT PICTURE'$$$9V.99';
/*PICTURED NUMERIC FIXED POINT          */
.
.
.
PUT SKIP EDIT(PAYMENT,ACCOUNT)
            (A(7),X(4),A(6));
```

Figure 3-13. Pictured Variable Examples

NONCOMPUTATIONAL VARIABLE

Noncomputational variables represent values that are used for purposes other than computation. Noncomputational values have a noncomputational data type and cannot be converted to computational values. Area variables, locator variables (pointer and offset), entry variables, file variables, and label variables are noncomputational variables.

Area Variable

An area variable represents storage that is reserved, upon allocation, for the subsequent allocation of based generations. Storage reserved for the allocation of based generations is called an area. All area variables have the AREA and the variable attributes.

An area variable holds an area value. An area value is an ordered list of significant allocations, including the allocated based generations and their values, and including the information about certain allocated based generations that have been freed but are still significant. An area value contains no significant allocations when the area is allocated. The area value is identical to the value returned by the EMPTY builtin function described in section 11, Builtin Functions. An allocation is significant if the allocation is made within the area and is not yet freed. An allocation is also significant if the allocation is freed but the freeing was done after another significant allocation was made within the area.

The size of an area is the number of 60-bit words utilized when the area is allocated. An area variable can have any storage type, including BASED. An area variable can be included in an aggregate.

An area value can be modified in the following ways:

- By allocating and freeing based generations within the area
- By assigning values to any based generations allocated within the area

An area variable can acquire values in the following ways:

- By assignment of an area value to the area variable during assignment statement execution
- By assignment of an area value to the index variable during DO statement execution, or execution of an embedded-do in a GET or PUT statement
- In argument passing during procedure invocation
- During initialization of the area with an area value according to the INITIAL attribute
- By assignment to the INTO option variable during READ statement execution
- By being allocated and not initialized, in which case the area is set to EMPTY( )

An area variable can be used in the following ways:

- In assignment to another area during assignment statement execution
- As the start expression in a DO statement, or embedded-do in a GET or PUT statement
- As an initial value in the INITIAL attribute
- As an argument passed to an invoked procedure
- As the value returned from a function
- As the FROM option variable in a WRITE or REWRITE statement

Area values cannot be compared. Examples of area variables are shown in figure 3-14.

```
DECLARE XY12(20) BASED CHAR(5);
DECLARE OFF71 OFFSET;
DECLARE AREA7 AREA(300);
DECLARE AREA8 AREA(300);
.
.
.
ALLOCATE XY12 SET(OFF71) IN(AREA7);
.
.
.
AREA8 = AREA7;
```

Figure 3-14. Area Variable Examples

Locator Variable

A locator variable represents a locator (pointer or offset) value. A locator value identifies the storage associated with a generation. A pointer value can identify storage allocated for a variable of any storage type. An offset value can only identify storage allocated for a based generation within an area. The offset value specifies the position of the storage relative to the beginning of the area. All locator variables have the variable attribute and either the POINTER or the OFFSET attribute.

A locator variable can acquire values in the following ways:

- By assignment of a locator value to a locator variable during assignment statement execution
- By assignment of a locator value to the index variable during DO statement execution, or execution of an embedded-do in a GET or PUT statement
- During initialization of the locator variable with a locator value according to the INITIAL attribute
- In argument passing during procedure invocation
- By assignment to the SET option variable during ALLOCATE statement execution

- By assignment (pointer variable only) to the SET option variable during READ or LOCATE statement execution
- By usage of an assumed SET option, constructed from the declaration of the BASED variable, during ALLOCATE or LOCATE statement execution
- By assignment (offset variable only) to the INTO option variable during READ statement execution

A locator value can be used in the following ways:

- In assignment to another locator variable during assignment statement execution
- As the start expression in a DO statement, or embedded-do in a GET or PUT statement
- As an initial value in the INITIAL attribute
- In referencing a based variable, as a locator value in a locator-qualifier or as an implicit locator specified in the BASED attribute
- In freeing a based variable, as a locator value in a locator-qualifier or as an implicit locator specified in the BASED attribute
- As an argument passed to an invoked procedure
- As the value returned from a function
- As an operand in a comparison operation to compare two locator values for equality
- As the FROM option variable (offset value only) in a WRITE or REWRITE statement

Conversion of locator values is performed as required for assignment, argument passing, or returning the result of a function.

Assignment and record I/O operations maintain the relative positions of based generations allocated within an area. When an area variable acquires an area value, all offset values that identify based generations in the original area value can also be used to identify the corresponding based generations in the new area value.

An example of a locator variable is shown in figure 3-15.

```
DECLARE Q POINTER;
DECLARE BVARTEST BASED(Q) FIXED DECIMAL;
•
•
•
ALLOCATE BVARTEST;
•
•
•
Q->BVARTEST = 15.0;
```

Figure 3-15. Locator Variable Example

Entry Variable

An entry variable can only be used as an entry parameter. An entry variable has values that are entry values. An entry value identifies an entry point of a procedure and a block activation. When the entry value is used to invoke a procedure, the entry point part identifies the statement at which execution begins, including a complete description of all parameters and the returned value, if any. The block activation part specifies the immediate environment of the new procedure activation. Every entry value is ultimately derived from an entry constant. The block activation identified by an entry value is determined at the time when the entry constant is passed as an argument to a procedure. All entry variables have the ENTRY and variable attributes.

An entry parameter can acquire a value only by being associated with an argument. Within any particular activation of a procedure that has an entry parameter, the entry parameter has a constant value. Values cannot be assigned to an entry parameter.

An entry value can be used in the following ways:

- As a subroutine reference in a CALL statement, if the entry value represents a subroutine entry point
- As a function reference, if the entry value represents a function entry point
- As an argument passed to an invoked procedure

Entry values cannot be returned by a function, and cannot be assigned or compared. Entry parameters are not supported as arrays. An entry parameter cannot be a member of a structure.

The declaration of an entry variable specifies the data types, alignment, and aggregate types of the parameters and returned value, if any, associated with the entry point. The entry parameter can only hold entry values consistent with the specification.

An example of an entry variable is shown in figure 3-16.

```
MAINP:
PROCEDURE OPTIONS(MAIN);
   •
   •
   •
   CALL CHOICE(ROUTINE2);
   /*ENTRY CONSTANT PASSED AS ARGUMENT*/
   •
   •
   •
CHOICE:
   PROCEDURE (ROUTINE);
      DECLARE ROUTINE ENTRY;
      /*ROUTINE IS ENTRY PARAMETER    */
      •
      •
      •
      CALL ROUTINE;
      /*VALUE OF ROUTINE IS ROUTINE2  */
      /*ROUTINE2 IS CALLED            */
   END CHOICE;
ROUTINE1:
   PROCEDURE;
   •
   •
   •
   END;
ROUTINE2:
   PROCEDURE;
   •
   •
   •
   END;
END MAINP;
```

Figure 3-16. Entry Variable Example

File Variable

A file variable can be used only as a file parameter. A file variable has values that are file constants. A reference to a file variable always has the same effect as a reference to the file constant that is the current value of the file variable.

The declaration of a file variable cannot contain any file description attributes. File description attributes can only be specified in the declaration of a file constant, and as options in an OPEN statement.

A file parameter can acquire a value only by being associated with an argument. Within any particular activation of a procedure that has a file parameter, the file parameter has a constant value. Values cannot be assigned to a file parameter.

A file value can be used in the following ways:

- As the file reference in a FILE option in an I/O statement
- As the file reference in a COPY option in a GET statement
- As the file reference in an I/O condition in an ON, REVERT, or SIGNAL statement
- As an argument passed to the PAGENO builtin function or pseudovariable
- As an argument passed to an invoked procedure

File values cannot be returned by a function. File values cannot be assigned or compared. File parameters are not supported as arrays. A file parameter cannot be a member of a structure.

An example of a file variable is shown in figure 3-17.

```
   DECLARE FOUT FILE OUTPUT;
   .
   .
   .
   CALL PROD(FOUT);
   /*FILE CONSTANT PASSED AS ARGUMENT */
   .
   .
   .
PROD:
   PROCEDURE(FILEPASS);
      DECLARE FILEPASS FILE;
      /*FILEPASS IS FILE PARAMETER     */
      OPEN FILE(FILEPASS);
      .
      .
      .
   END PROD;
```

Figure 3-17. File Variable Example

Label Variable

A label variable has values that are label values. A label value identifies a statement and a block activation. When a label value is used to transfer control to a statement, the statement part identifies the statement to which control is transferred, and the block activation part specifies the new current block activation after the transfer of control is completed.

A label variable can acquire values in the following ways:

- By assignment of a label value to a label variable during assignment statement execution
- By assignment of a label value to the index variable in a DO statement, or in an embedded-do in a GET or PUT statement
- During initialization of the label variable with a label value according to the INITIAL attribute
- In argument passing during procedure invocation

A label value can also be used in the following ways:

- As the label reference in a GOTO statement
- As the source expression in an assignment statement
- As the start expression in a DO statement, or in an embedded-do in a GET or PUT statement
- As an argument passed to an invoked procedure

Label values cannot be returned by a function. Label values cannot be compared.

An example of a label variable is shown in figure 3-18.

```
 YY:
 PROCEDURE OPTIONS(MAIN);
    DCL LAB(3) LABEL;
    /*LAB IS ARRAY OF LABEL CONSTANTS  */
    DCL LV LABEL INITIAL(LAB(2));
    /*LV IS LABEL VARIABLE             */
 LAB(1):
    .
    .
    .
    CALL ZZ(LV);
    /*LABEL VARIABLE PASSED AS ARGUMENT*/
 LAB(2):
    .
    .
    .
    LV = LAB(3);
    CALL ZZ(LV);
    /*LABEL VARIABLE PASSED AS ARGUMENT*/
 LAB(3):
    .
    .
    .
 ZZ:
    PROCEDURE (L);
       DCL L LABEL;
       .
       .
       .
       GOTO L;
    END ZZ;
 END YY;
```

Figure 3-18. Label Variable Example

GENERATIONS AND STORAGE

A generation is a description of a variable. The generation is the basic unit for referencing or assigning a value to a variable. Each generation has two components:

- The data description that describes the characteristics of the variable, including the completed attribute set and all fully evaluated extents

- The storage description that describes and locates the storage used to hold the values of the variable

A reference to a variable accesses a generation of the variable. During program execution, the number of generations of a variable is dynamic. If more than one generation of a variable exists, only one is usually accessible at any particular time. For instance, if a procedure that contains an automatic variable has been activated recursively several times, several generations exist for the automatic variable. A reference to the variable is a reference to the generation associated with the most recent activation. In the same activation, however, there might be three accessible generations of a based variable and no accessible generations of a controlled variable. The storage type of a variable determines the number of generations and the time at which each generation is created.

Each PL/I program has a pool of storage from which allocation units can be formed. The allocation unit is the basic unit for allocating and freeing storage associated with generations. The allocation unit contains the storage that holds the values of the variable. Allocation of a variable creates a generation and an associated allocation unit. The storage description of the generation fully describes the allocation unit, and the generation is called an allocated generation. Allocation of a variable is considered an indivisible operation for a variable. The variable to be allocated must be unsubscripted and cannot be a member. An array element cannot be allocated except in allocation of the entire array. A member cannot be allocated except in allocation of the entire containing structure.

Freeing of a variable is the process of releasing an allocation unit and returning the storage to the available pool of storage from which the allocation unit was formed. When an allocated generation is freed, the allocation unit is undefined and the generation can no longer be referenced. Like allocation, freeing is considered an indivisible operation. The variable to be freed must be unsubscripted and cannot be a member. An individual array element or structure member cannot be freed. In addition, the data description and storage description of the variable to be freed must precisely match those of the allocated generation.

Generations can be allocated and freed in the ways described for each storage type. The storage description part of a generation is different from an allocation unit for the following reasons:

- A generation of certain types of variables can share the storage associated with another variable. In this case, the storage description part of a generation describes an allocation unit belonging to another variable.

- Certain variables can have multiple allocation units available and can move a generation by means of locator values. The storage description part of the generation then describes a different allocation unit.

- A particular generation can describe less than a complete allocation unit. Each reference in the program effectively creates a generation. For example, a generation created by a reference to a single element of an array has a storage description part that does not describe the complete allocated generation for the entire array.

Each variable (except any member of a structure) has a storage type. Just as individual structure members and array elements cannot be allocated and freed, structure members and array elements do not have individual storage types. The available storage types are STATIC, AUTOMATIC, CONTROLLED, BASED, DEFINED, and parameter. The STATIC, AUTOMATIC, CONTROLLED, and BASED storage types are described in this section. A variable with the DEFINED attribute is defined on another variable and shares storage with that variable, as described under the DEFINED attribute in section 4. A variable with the parameter attribute is used as a received value in an invoked procedure, as described under the Parameter attribute in section 4 and described under Procedure Reference in section 6, References.

Static Storage

A static variable has the STATIC storage attribute. Each STATIC variable has one allocated generation. The generation is allocated before program execution begins. Static generations cannot be allocated or freed during program execution. Since static storage is allocated before program execution begins, any array bounds, string lengths, or area sizes must be specified as literal constants. Initial values must be literal constants or expressions containing only literal constants. An example of static storage is shown in figure 3-19.

```
DECLARE STAT STATIC CHAR(10);
/*GENERATION IS ALLOCATED BEFORE      */
/*PROGRAM EXECUTION BEGINS            */
.
.
.
STAT = ↑TEST↑;
```

Figure 3-19. Static Variable Example

Automatic Storage

An automatic variable has the AUTOMATIC storage attribute. A generation for each automatic variable is allocated automatically each time the block in which the variable is declared is activated. If the block is already active, then the previous generations allocated for the previous activations of the block are automatically saved. At any given time, multiple allocated generations for an automatic variable can exist, with one for each activation of the block in which the automatic variable is declared. When a block activation is terminated, each automatic generation allocated for that activation is freed. Automatic variables cannot be allocated or freed in any other way. Block activation and termination and the stacking of generations associated with block activations are described in section 2, Dynamic Program Structure.

Each allocation of a generation for an automatic variable can involve evaluation of extent expressions and expressions in an INITIAL attribute. Extent expressions can be used for array bounds, string lengths, and area sizes. Extent expressions and expressions in an INITIAL attribute can contain references to other variables. An example of automatic storage is shown in figure 3-20.

A local reference to an automatic variable accesses the generation associated with the current block activation. A nonlocal reference to an automatic variable accesses the generation associated with an activation that is part of the environment of the current block activation.

When a generation is allocated for an automatic variable, the STORAGE condition is raised if sufficient storage space is not available.

```
T:
   BEGIN;
      DECLARE AUTO AUTOMATIC CHAR(10);
      /*GENERATION IS ALLOCATED WHEN  */
      /*BLOCK T IS ACTIVATED          */
      •
      •
      •
      AUTO = ↑TEST↑;
   END T;
```

Figure 3-20. Automatic Variable Example

Controlled Storage

A controlled variable has the CONTROLLED storage attribute. Each controlled variable has a pushdown stack of allocated generations. A generation for a controlled variable is allocated by execution of an ALLOCATE statement that names the variable. A generation is freed by execution of a FREE statement that names the variable. ALLOCATE statement execution causes the stack of allocated generations to be pushed down and the new generation to be added at the top. Each generation in the stack is maintained intact. FREE statement execution causes the top generation to be freed and causes the stack to be popped up. Block activation and termination have no effect on the stack of generations. A reference to a controlled variable at any given time in any block can only access the top generation in the stack of generations.

Allocation and freeing of a controlled variable is completely under the control of the programmer. No value can be assigned to a controlled variable unless storage has been allocated. Any number of allocations can be made. If insufficient storage space is available for allocation of a generation, the STORAGE condition is raised. A controlled variable passed as an argument to a procedure cannot be allocated or freed within the procedure.

Each allocation of a generation for a controlled variable can involve evaluation of extent expressions and expressions in an INITIAL attribute. Extent expressions can be used for array bounds, string lengths, or area sizes. Extent expressions and expressions in an INITIAL attribute cannot contain references to the controlled variable itself. An example of controlled storage is shown in figure 3-21.

```
DECLARE CONT CONTROLLED CHAR(10);
•
•
•

ALLOCATE CONT;
/*GENERATION IS ALLOCATED WHEN        */
/*ALLOCATE STATEMENT IS EXECUTED      */
•
•
•
CONT = ↑TEST↑;
```

Figure 3-21. Controlled Variable Example

Based Storage

A based variable has the BASED storage attribute. A based variable can be used for referencing generations of any storage type and for allocating and freeing based generations.

A based reference always requires use of a based variable and a locator value. The based variable is used as the data description of the referenced generation; that is, the based variable is effectively a template. The locator value is used to supply the storage description of the referenced generation. The data description of the described storage, as allocated, must match the data description from the based variable.

A based generation can be allocated by execution of an ALLOCATE statement that names the variable. When a based generation is allocated, a storage description is assigned to a locator variable. Each allocated generation is independent of any other allocated generations for the based variable. A based generation allocated in this way can be freed by execution of a FREE statement that uses the locator value created during allocation and uses a based variable whose data description matches the data description of the allocated generation.

When a based variable is allocated, any extent expressions and expressions in an INITIAL attribute are evaluated. Extent expressions can be array bounds, string lengths, or area sizes. Extent expressions in the declaration of a based variable are evaluated again each time the based variable is referenced.

An extent for array bounds, string length, or area size of a member in a based structure can be saved as the value of another member of the same structure. See the description of REFER Option for Based Structures in section 4.

Allocation of Based Generations

A based generation can be allocated by execution of an ALLOCATE statement that names the based variable. The locator value that identifies the new generation is assigned to the variable specified by the SET option, or to the implicit locator variable specified in the declaration of the based variable. If the ALLOCATE statement has no SET option, a locator variable must have been supplied in the BASED attribute in the declaration of the based variable. If insufficient space is available for allocation, the STORAGE condition is raised.

A based generation is allocated within an area if the ALLOCATE statement has an IN option, or if an assumed IN option can be constructed from the declaration of the locator variable. If insufficient space is available for the based generation in the area, the AREA condition rather than the STORAGE condition is raised.

A based generation allocated by ALLOCATE statement can be freed by execution of a FREE statement that specifies the allocated based generation. During execution of a FREE statement, extents specified in the declaration of the based variable are evaluated. If the based generation is allocated in an area, the FREE statement can contain an IN option specifying the area, or an assumed IN option can be constructed from the declaration of the locator variable. The IN option must identify the area in which the based variable was allocated. Based generations allocated in an area are freed if the area generation is freed, or if the entire area is reassigned.

Examples of based storage are shown in figure 3-22.

```
DECLARE BASE BASED(P) CHAR(10);
DECLARE P POINTER;
.
.
.
ALLOCATE BASE;
/*GENERATION IS ALLOCATED WHEN          */
/*ALLOCATE STATEMENT IS EXECUTED        */
P->BASE = ↑TEST↑;
.
.
.
DECLARE BASE2 BASED CHAR(100);
DECLARE OFF OFFSET;
DECLARE AREA23 AREA(300);
.
.
.
ALLOCATE BASE2 SET(OFF) IN(AREA23);
/*GENERATION IS ALLOCATED WHEN          */
/*ALLOCATE STATEMENT IS EXECUTED        */
```

Figure 3-22. Based Variable Examples

Based generations can be implicitly allocated for use as record I/O buffers in one situation involving output and another involving input. A based generation allocated for use as an I/O buffer is called an allocated buffer. No implicit allocation can take place within an area. The locators used must be pointer variables.

A based generation for use as an output buffer is implicitly allocated by execution of a LOCATE statement. The LOCATE statement specifies the record output file and names a based variable. When the based generation has been allocated with the LOCATE statement, values can be assigned to the based variable. Subsequent WRITE statement execution, LOCATE statement execution, or closing of the file causes the based generation to be written as an output record and then freed. When an output buffer is allocated by LOCATE statement execution, the SET option of the LOCATE statement can specify the pointer variable to which the pointer value is assigned. If the LOCATE statement has no SET option, the pointer value is assigned to the implicit locator specified in the declaration of the based variable. If no SET option is used, an implicit locator must be available.

A based generation for use as an input buffer is implicitly allocated by execution of a READ statement with the SET option. The based generation acts as an input buffer for the record input or record update file specified by the statement, and the pointer value identifying the based generation is assigned to the pointer variable specified in the SET option. The READ statement does not specify a based variable, but the only method of access to the input record is by reference to a based variable. The based variable must correctly describe the input record. Subsequent READ statement execution (with or without a SET option), WRITE statement execution, REWRITE statement execution, DELETE statement execution, or closing of the file causes the based generation to be implicitly freed. An example of based storage using implicit allocation is shown in figure 3-23.

```
DCL BASE3(10) BASED(PTR3) CHAR(10);
DECLARE PTR3 POINTER;
.
.
.
READ FILE(TEMP) SET(PTR3);
/*GENERATION IS ALLOCATED WHEN          */
/*READ STATEMENT IS EXECUTED            */
```

Figure 3-23. Based Variable with Implicit Allocation Example

Access to Based Generations

A based generation is accessible if a locator value that points to the generation is accessible. When a locator variable is assigned a new value, the previous value of the locator variable is lost. If all locator values that point to a based generation are lost, access to the based generation is not possible. In this case, the based generation remains allocated for the duration of program execution.

Because of the properties of locator variables, based generations can be chained together for list processing. If one generation of a based structure contains locator variables that point to other based structures, chains of based generations can be constructed dynamically. When based generations are chained together with locator variables, a null locator value can be used to set the last pointer in a forward chain and the first pointer in a backward chain. A null locator value does not address a based generation and must not be used in a locator-qualified reference. A null locator value can be assigned to a locator variable, and locator variables can be tested for the null locator value. A null locator value is returned by the NULL builtin function described in section 11, Builtin Functions.

AGGREGATES

An aggregate is an organized collection of scalar data elements. A scalar is a single data item. Scalars can be collected in one way to form arrays or in another way to form structures. Scalars, arrays, or structures can be repeatedly collected to form larger arrays or structures. The particular organization that is used to collect the elements or members of an aggregate is called the aggregate type.

An array is an organized collection of array elements. All elements of an array have the same name and the same set of attributes. Elements are distinguished from one another by subscripts that specify the relative positions of elements within the array.

A structure is an organized collection of structure members. Each member has a different name and can have a different set of attributes. Each member is referenced by its name.

The following data elements can exist only as scalars and cannot be arrays or structure members:

- Literal constants
- Entry constants
- Format constants
- File constants
- Entry variables
- File variables

The following data elements can exist as scalars or as arrays but cannot be structure members:

- Label constants

The following data elements can exist as scalars, arrays, or structure members:

- Variables (except entry or file)
- Structures

SCALARS

A scalar variable or named constant has a name and a data type. A scalar does not have the dimension attribute or the structure attribute. A reference to a scalar variable is a scalar reference. All scalars have the aggregate type scalar; that is, they have an aggregate type which indicates that they are not aggregates. Each scalar named constant has a single value. Each generation of a scalar variable has a single value. Note that a scalar area variable has a single area value; the area value is scalar even if arrays are allocated within the area.

ARRAYS

An array variable or named constant has a name and the dimension attribute. Each element of the array has the same name as the array. The number of dimensions is called the dimensionality of the array. An array can have as many as 32 dimensions. To identify a particular element of an array, a reference must specify one subscript for each dimension of the array; each subscript specifies the relative position of the element, measured along one of the dimensions.

Each dimension has an extent that specifies the range of subscript values permitted for that dimension. The extent for each dimension consists of a lower bound and an upper bound. The lower bound and upper bound together are called the array bounds. If the lower bound is not specified, the lower bound is assumed to be 1. The span of a dimension is the number of possible subscript values for the dimension. The array A(5) has five possible subscripts for the dimension, and B(4:6) has three possible subscripts.

The number of elements in the entire array is the product of the spans of all dimensions. The array C(3,4) is an array of 12 elements, D(10,5,20) is an array of 1000 elements, and E(2:5,-16:-9) is an array of 32 elements.

The aggregate type of the array is defined by the dimension attribute specified for the array and the aggregate type of the array elements. Two arrays have the same aggregate type only if their dimensionality and all array bounds match precisely. The array bounds must match; it is not sufficient for the spans to be identical.

Every array has the variable attribute or the constant attribute. Each data type attribute specified for an array is applied uniformly to all elements of the array. Dimension applies to the array itself. Storage type and the scope attribute INTERNAL or EXTERNAL can apply to the entire array. The INITIAL attribute can apply to the entire array but is used to specify initial values for individual elements.

The physical organization of an array in storage is linear. The elements of an array with one dimension are stored in order of increasing subscripts. In the array F(6), the array element F(3) immediately precedes the array element F(4). Multidimensional arrays are stored in order, with the rightmost subscript varying most rapidly. In the array G(3,5), the array element G(1,1) immediately precedes the array element G(1,2), and G(1,5) immediately precedes the array element G(2,1). The array elements G(2,1) and G(3,1) are not contiguous in storage.

References to array elements are subscripted references, as described in section 6, References. Examples of arrays are shown in figure 3-24.

```
DECLARE F(6) CHAR(12) INIT((6)(1)↑ ↑);
DECLARE G(3,5) FIXED DEC INIT((15)0);
•
•
•
F(2) = ↑SPARE PARTS↑;
•
•
•
G(1,2) = 125.43;
```

Figure 3-24. Array Examples

STRUCTURES

A structure variable has a name and the structure attribute. Each member of the structure is either a variable, with a name and a data type, or a structure. A structure that is a member of a structure is called a substructure. The arrangement of substructures and members within the structure is called the configuration of the structure.

Each structure has level numbers that indicate, in outline form, the hierarchical organization of the structure. The structure is declared with the level number one. Members of the structure are declared with a greater level number. If a member is a substructure, the members of that substructure are declared with a greater level number, and so on. The specified level number for a structure must be level one, and the specified level numbers for substructures and members must be greater than one. If a structure called ID has a substructure NAME and members FIRST, MIDDLE, and LAST, the level numbers can be:

ID	structure	level 1
NAME	substructure	level 2
FIRST	member	level 3
MIDDLE	member	level 3
LAST	member	level 3

The aggregate type of a structure is defined by the structure attribute and the ordered list of aggregate types for the structure members. Two structures have the same aggregate type only if the members are arranged in the same order and the aggregate type of each member corresponds. The names of the members are not relevant.

The attributes of a level 1 structure are variable, a storage type attribute, a scope attribute INTERNAL or EXTERNAL, and an optional alignment attribute. The attributes of a substructure are variable, member, structure, INTERNAL, and an optional alignment attribute. Any structure can be declared with the LIKE attribute that expands a structure to the form of another structure. A based variable that is a structure can contain members with REFER options that hold the values of extents for other members. The attributes of a member are variable, member, INTERNAL, and the specified data type.

The physical organization of a structure in storage is linear. Members are stored in the order in which they appear in the declaration.

References to members of the structure can be simple references or structure-qualified references, as described in section 6. An example of a structure is shown in figure 3-25.

```
DECLARE 1 ID,
          2 NAME,
            3 FIRST CHAR(12),
            3 MIDDLE CHAR(1),
            3 LAST CHAR(22);
.
.
.
READ FILE(F) INTO(ID)
IF (LAST = 'SMITH')
   THEN CALL WHICHONE(ID);
```

Figure 3-25. Structure Example

ARRAYS AND STRUCTURES

Arrays and structures represent different organizational approaches for data elements. Arrays and structures are considered aggregates because the two different approaches can be combined in various ways.

A structure containing arrays is an undimensioned structure that has one or more members that are arrays. An example is shown in figure 3-26.

An array of structures is a dimensioned structure. An example is shown in figure 3-27. Note that the members inherit the dimensionality of the structure.

An array of structures containing arrays is a dimensioned structure that has one or more members that are arrays. An example is shown in figure 3-28. Note that the members inherit the dimensionality of the structure, and the additional dimensions are effectively added to the left of any specified dimensions for the members.

```
DECLARE 1 A,
          2 B(4) CHARACTER(5),
          2 C(4) FIXED DECIMAL;
```

The storage order is

B(1)	B(2)	B(3)	B(4)	C(1)	C(2)	C(3)	C(4)

Figure 3-26. Structure Containing Arrays Example

```
DECLARE 1 D(4),
          2 E FIXED DECIMAL,
          2 F BIT(12);
```

The storage order is

E(1)	F(1)	E(2)	F(2)	E(3)	F(3)	E(4)	F(4)

Figure 3-27. Array of Structures Example

```
DECLARE 1 G(2),
          2 H(3) FLOAT BINARY,
          2 J FIXED BINARY;
```

The storage order is

H(1,1)	H(1,2)	H(1,3)	J(1)	H(2,1)	H(2,2)	H(2,3)	J(2)

Figure 3-28. Array of Structures Containing Arrays Example

Attributes are descriptive and defining characteristics of data used in a PL/I program. This section discusses the categories of attributes and describes individual attributes. Attribute information is directly associated with declarations and defaults as described in section 5, Declarations. Other sections of this manual contain descriptions that directly or indirectly involve attributes.

Each identifier used in a PL/I program has attributes describing the characteristics of the identifier. The attributes for an identifier used in the program indicate whether the identifier is a variable, a named constant, a programmer-named condition, or a builtin function name. Additional attributes are used to indicate the specific use of the identifier and the type of value represented. Since identifiers are named, the attributes for an identifier are associated with the name. Attributes for identifiers can be specified in a DECLARE statement. Attributes can also be specified explicitly or contextually by usage of the identifier. The compiler adds default attributes as necessary to create a completed set of attributes for the identifier.

A descriptor is a list of attributes not directly associated with an identifier. Descriptors are used in declarations of entry constants and entry variables. A parameter descriptor specifies the attributes of a parameter received by an invoked procedure. A returns descriptor specifies attributes of the value returned from an invoked function procedure. Attributes for descriptors can be specified in the ENTRY or RETURNS attribute in a DECLARE statement or in the RETURNS option in a PROCEDURE or ENTRY statement. Descriptors are completed by the compiler with additional default attributes.

Each literal constant has attributes implied by the actual form of the constant. Attributes of the literal constant affect data manipulation and assignment operations.

The attributes that can be used for variables, named constants, conditions, builtin functions, parameter descriptors, returns descriptors, and literal constants are described separately. By convention, full upper case is used for attributes that can be specified by keyword. Lower case is used for attributes recognized in some other way. Descriptions of individual attributes are provided in alphabetic order, and a summary of attributes is provided at the end of this section.

ATTRIBUTES FOR VARIABLES

Each identifier used as a variable has the variable attribute. Each variable can have attributes for scope, storage type, aggregation, alignment, data type, and initialization. The variable attribute is implied and cannot be declared explicitly by DECLARE statement. An illustration of the attributes for variables is shown in figure 4-1.

SCOPE ATTRIBUTES

The scope attributes are

- INTERNAL
- EXTERNAL

The scope attribute indicates the scope within which the declaration of the variable is applicable. Scope applies to a scalar or an aggregate. Scope applies to the entire array rather than an individual array element and to the entire structure rather than an individual structure member.

STORAGE TYPE ATTRIBUTES

The storage type attribute specifies the way in which the variable is associated with storage. Storage type applies to a scalar or an aggregate. Storage type applies to the entire array rather than an individual array element and to the entire structure rather than an individual structure member. The storage type of a variable can be

- AUTOMATIC
- BASED
- CONTROLLED
- STATIC
- Parameter
- DEFINED

The AUTOMATIC, BASED, CONTROLLED, and STATIC attributes describe variables for which storage can be allocated, as described in section 3, Data Elements. The DEFINED and parameter attributes describe variables that share storage belonging to other variables.

AGGREGATION AND ALIGNMENT ATTRIBUTES

The attributes that describe the aggregate type and alignment of a variable are

- Dimension
- Structure
- Member
- ALIGNED
- UNALIGNED

The dimension attribute indicates that the variable is an array. The structure attribute indicates that the variable is the name of a structure. The member attribute indicates that the variable is contained in a structure. A substructure has both the structure and the member attributes. A scalar does not have the dimension or the structure attribute.

The ALIGNED and UNALIGNED attributes indicate the alignment of the variable. Any variable not declared as a structure has one of the alignment attributes.

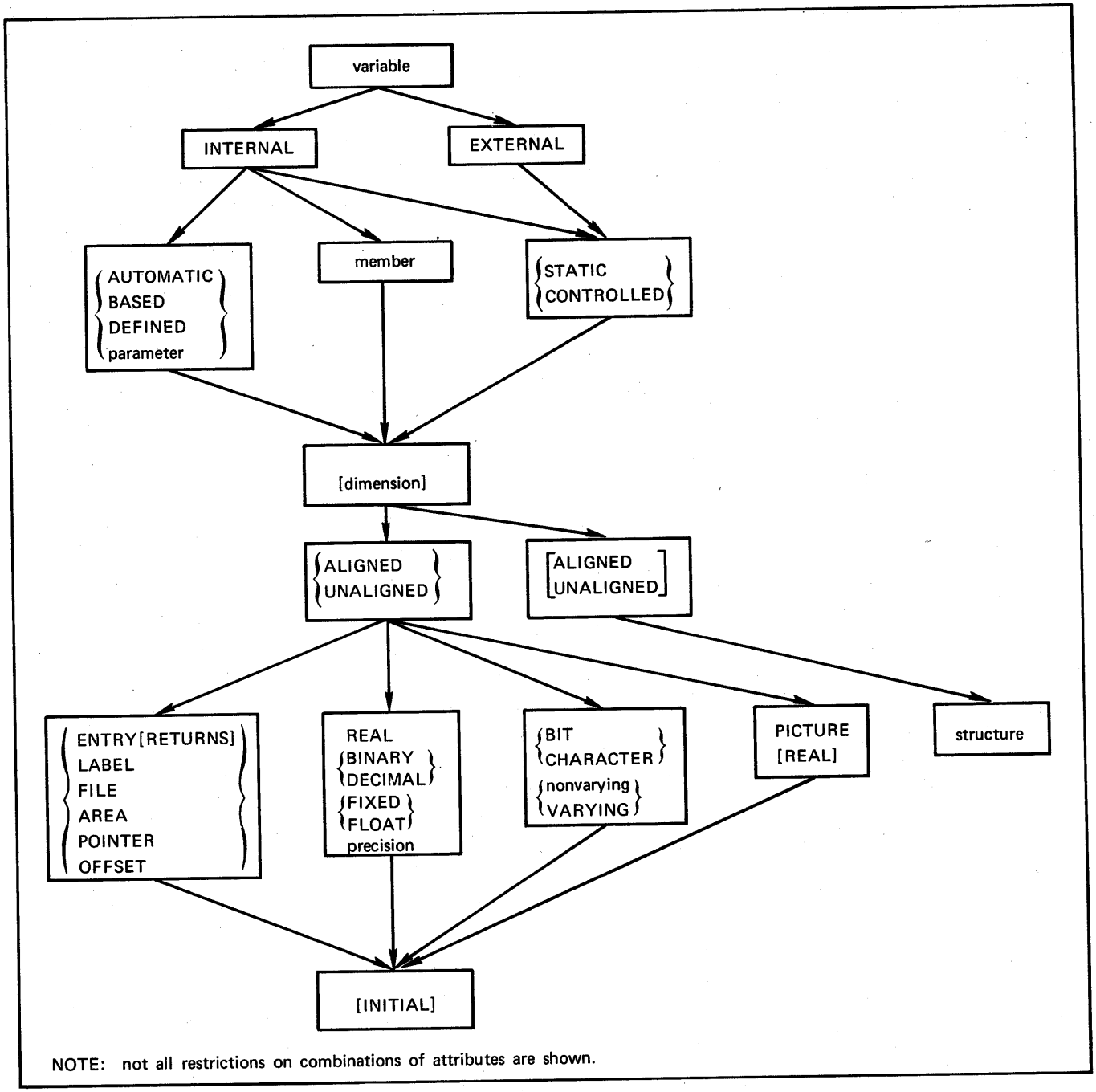


Figure 4-1. Attributes for Variables

DATA TYPE ATTRIBUTES

The data type attributes describe the type of values represented by the variable. Data type attributes cannot be specified for a structure. If specified for an array, any data type attribute applies to all elements of the array. The data type attributes can be categorized as noncomputational or computational. The computational data type attributes are further categorized as arithmetic, string, or pictured.

Noncomputational Data Type Attributes

A noncomputational data type attribute indicates the use of the variable. The noncomputational data type attributes are

- ENTRY [RETURNS]
- LABEL

- FILE
- AREA(size)
- POINTER
- OFFSET

The ENTRY attribute indicates an entry variable. Entry variables can only be used as entry parameters. The RETURNS attribute, which implies the ENTRY attribute, indicates that the entry constant values of the entry variable represent function entry points. The LABEL attribute indicates a label variable. The FILE attribute indicates a file variable. File variables can only be used as file parameters. The AREA attribute indicates an area variable and specifies the size of the area. The POINTER attribute or the OFFSET attribute indicates a locator variable.

Arithmetic Data Type Attributes

The arithmetic data type attributes indicate the type of arithmetic values represented by the variable. The arithmetic attributes are

- REAL
- BINARY
- DECIMAL
- FIXED
- FLOAT
- Precision

The REAL attribute indicates the mode of the arithmetic values. The BINARY or DECIMAL attribute indicates the base of the values. The FIXED or FLOAT attribute indicates the scale of the values. The precision attribute indicates the precision of the values.

String Data Type Attributes

The string data type attributes indicate the type of string value represented by the variable. The string attributes are

- BIT(length)
- CHARACTER(length)
- Nonvarying
- VARYING

The BIT or CHARACTER attribute indicates the type of string and the maximum length of the string. The nonvarying or VARYING attribute indicates whether the string can ever be treated as being shorter than the specified maximum length.

Pictured Data Type Attribute

The pictured data type attribute indicates that the variable represents pictured values, either pictured numeric or pictured character values. The pictured attribute is

- PICTURE

The PICTURE attribute is associated with a pictured value. The REAL attribute can accompany the PICTURE attribute for a pictured numeric item. Other arithmetic attributes cannot be declared but are implied by the picture specification.

Initialization Attribute

The INITIAL attribute specifies initial values to be assigned when storage is allocated for the variable. The initialization attribute is

- INITIAL

Some restrictions on the use of INITIAL are described later in this section under INITIAL.

ATTRIBUTES FOR NAMED CONSTANTS

Each identifier used as a named constant has the constant attribute. Each named constant has a noncomputational data type attribute. An aggregation attribute is possible. Each named constant that is a file constant has a number of possible file description attributes. The constant attribute is implied and cannot be declared explicitly in a DECLARE statement. An illustration of the attributes for named constants is shown in figure 4-2.

SCOPE ATTRIBUTES

The scope attributes are

- INTERNAL
- EXTERNAL

The scope attribute indicates the scope within which the declaration of the named constant is applicable.

AGGREGATION ATTRIBUTE

The attribute that can describe the aggregate type of a named constant is

- Dimension

The dimension attribute indicates an array of named constants. The dimension attribute can be used for an array of label constants. Dimension cannot be used for other types of constants.

NONCOMPUTATIONAL DATA TYPE ATTRIBUTES

A noncomputational data type attribute indicates the use of the named constant. The noncomputational data type attributes are

- ENTRY [RETURNS]
- LABEL
- Format
- FILE

The ENTRY attribute indicates an entry point for a procedure. The RETURNS attribute, which implies the ENTRY attribute, indicates an entry point of a function. The LABEL attribute indicates a label constant. The format attribute indicates a format constant. The FILE attribute indicates a file constant.

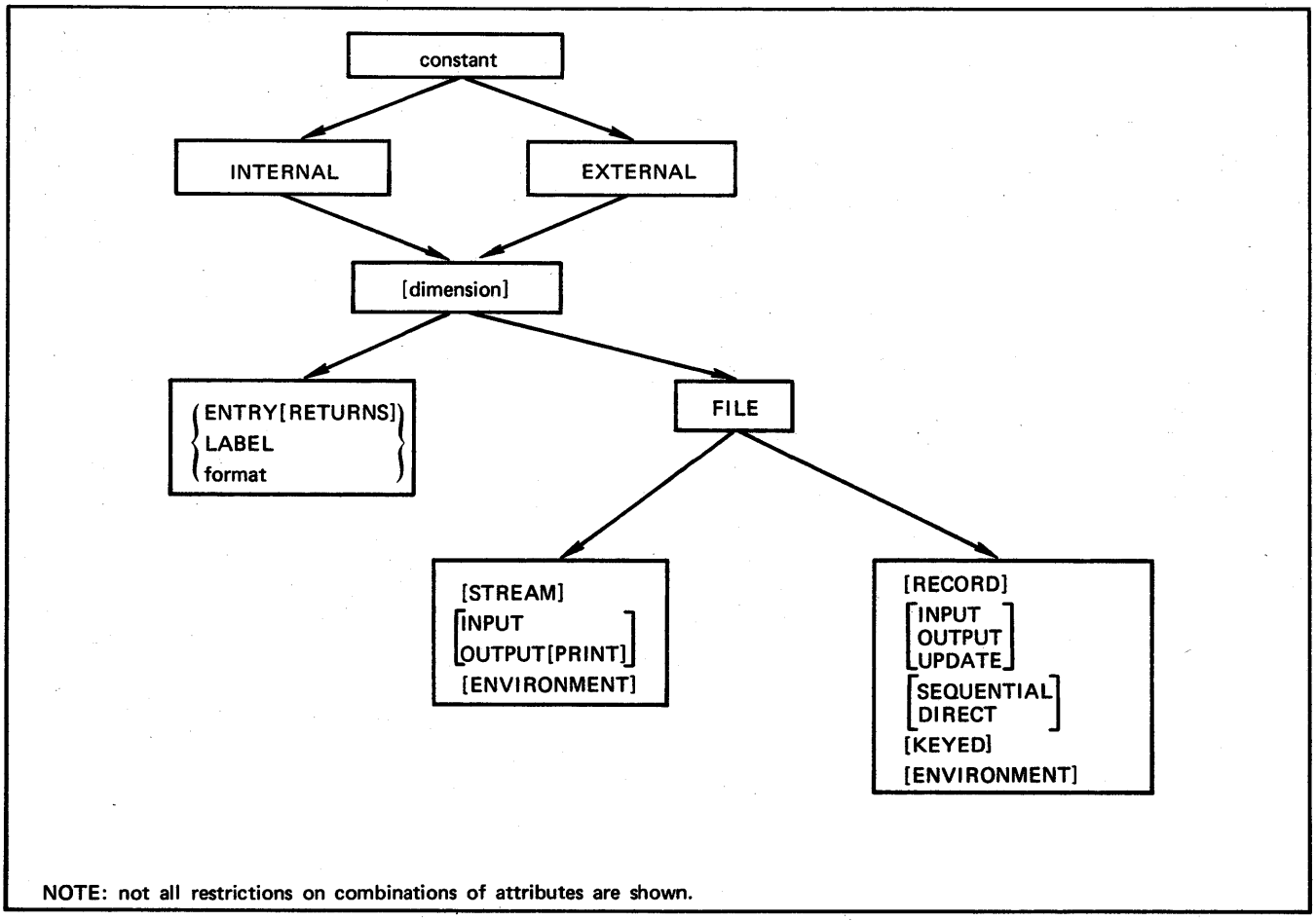


Figure 4-2. Attributes for Named Constants

FILE DESCRIPTION ATTRIBUTES

The file description attributes apply only to named constants with the FILE attribute. File description attributes are not necessarily complete during compilation. Application of file description attribute defaults occurs at run time when the file constant is opened. The file description attributes are

- STREAM
- RECORD
- INPUT
- OUTPUT [PRINT]
- UPDATE
- SEQUENTIAL
- DIRECT
- KEYED
- ENVIRONMENT(options)

All file description attributes except ENVIRONMENT imply the FILE attribute. The STREAM or RECORD attribute indicates the type of file I/O operations (stream I/O or record I/O). The INPUT, OUTPUT, or UPDATE attribute indicates the usage of the file (UPDATE is only appropriate with RECORD). The PRINT attribute, which implies the OUTPUT attribute, indicates that a file is intended for printing (PRINT is only appropriate with STREAM). The SEQUENTIAL or DIRECT attribute applies to record I/O, indicating the type of access to records of the file. The KEYED attribute can be used with DIRECT or SEQUENTIAL to indicate that keyed operations can be performed on the file. The ENVIRONMENT attribute specifies CYBER Record Manager file processing options.

ATTRIBUTES FOR CONDITIONS

Each identifier used as an identifier in a programmer-named condition has the condition attribute and the following scope attribute:

- EXTERNAL

No other attributes apply to a programmer-named condition. The condition attribute is implied and cannot be explicitly declared in a DECLARE statement. An illustration of attributes for programmer-named conditions is shown in figure 4-3.

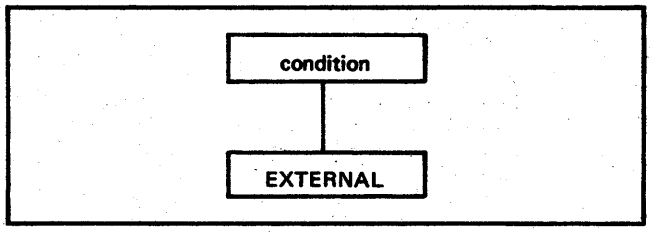


Figure 4-3. Attributes for Programmer-Named Conditions

ATTRIBUTES FOR BUILTIN FUNCTIONS

Each identifier used as the name of a builtin function has the BUILTIN attribute and the following scope attribute:

- INTERNAL

No other attributes apply to the name of a builtin function. The BUILTIN attribute can be explicitly declared in a DECLARE statement. An illustration of attributes for builtin functions is shown in figure 4-4.

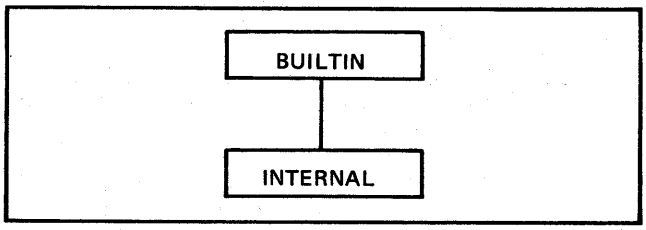


Figure 4-4. Attributes for Builtin Functions

ATTRIBUTES FOR PARAMETER DESCRIPTORS

Each parameter descriptor specifies aggregation, alignment, and data type attributes of values received as parameters in an invoked procedure. The attributes for a parameter descriptor are similar to the attributes for a parameter. An illustration of attributes for parameter descriptors is shown in figure 4-5.

Parameter descriptors have no scope or storage type. The parameter descriptor can have the same attributes as a parameter, except that the INITIAL attribute is not allowed. All noncomputational, arithmetic, string, or pictured data type attributes are available.

ATTRIBUTES FOR RETURNS DESCRIPTORS

Each returns descriptor specifies alignment and data type attributes of the value returned from a function. An illustration of attributes for returns descriptors is shown in figure 4-6.

Returns descriptors have no scope or storage type. Returns descriptors cannot have any aggregation attributes. Returns descriptors can have alignment attributes and the following noncomputational data type attributes:

- AREA(size)
- POINTER
- OFFSET

All arithmetic, string, or pictured data type attributes are available. The INITIAL attribute cannot be used.

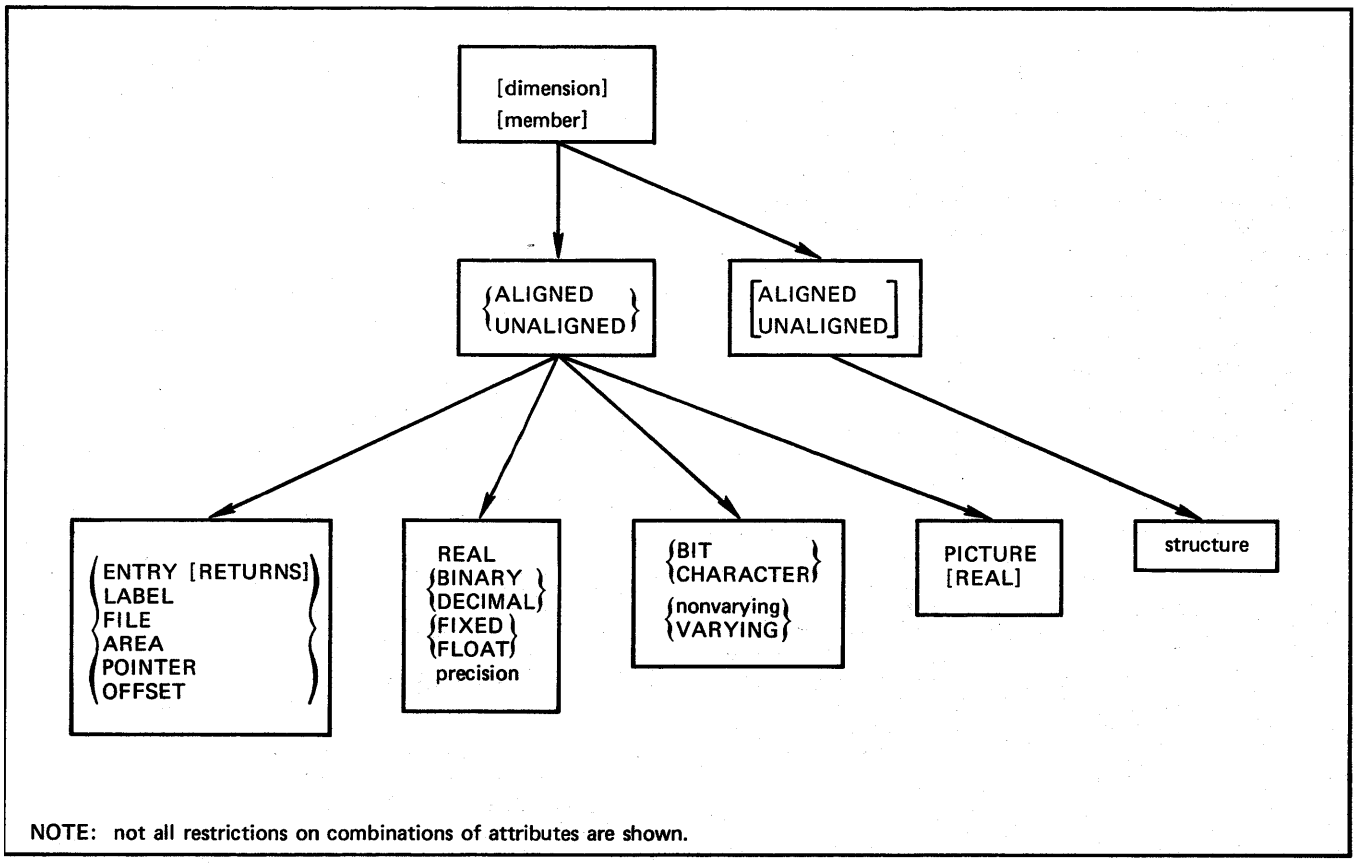


Figure 4-5. Attributes for Parameter Descriptors

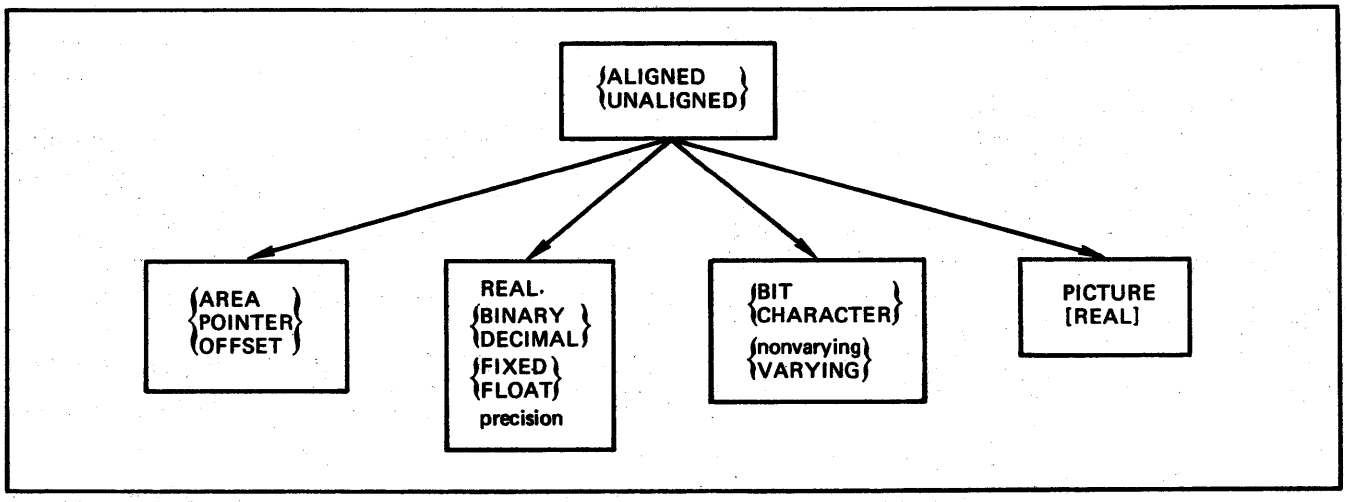


Figure 4-6. Attributes for Returns Descriptors

ATTRIBUTES FOR LITERAL CONSTANTS

Each literal constant has the constant attribute and attributes implied by the actual form of the literal constant. A literal constant can be arithmetic, bit, or character, as described in section 3, Data Elements. An arithmetic constant has arithmetic attributes identical to those available for variables. A string constant can be a bit constant with the BIT attribute and a nonvarying length, or can be a character constant with the CHARACTER attribute and a nonvarying length.

ATTRIBUTE DESCRIPTIONS

The description of each attribute includes the purpose of the attribute and the syntax used when the attribute is specified. Information about conflicts and implications is provided, as well as other information concerning the use of the attribute. Attribute descriptions are in alphabetic order, except that certain attribute descriptions have been merged in the following way:

LIKE	Listed under Structure
Member	Listed under Structure
Nonvarying	Listed under VARYING
POSITION	Listed under DEFINED
UNALIGNED	Listed under ALIGNED

ALIGNED AND UNALIGNED ATTRIBUTES

The ALIGNED and UNALIGNED attributes specify the alignment of values in storage. An aligned value is aligned on a word boundary, and an unaligned value can be packed into storage at the next available location. The alignment of values can affect processing time, I/O time, and storage requirements. In general, ALIGNED leads to decreased processing time, increased I/O time for record I/O, and increased storage requirements. In general, UNALIGNED leads to the opposite.

ALIGNED and UNALIGNED are alignment attributes used for variables, parameter descriptors, and returns descriptors. An alignment attribute can be specified for a variable that is a scalar, array, or structure. When an alignment attribute is specified for an array, all elements of the array have the same alignment attribute. When an alignment attribute is specified for a structure, the alignment is propagated through the structure. The ALIGNED attribute can be specified by keyword. The UNALIGNED attribute can be specified by keyword and can also be abbreviated. The syntax of ALIGNED is shown in figure 4-7, and the syntax of UNALIGNED is shown in figure 4-8. Examples of ALIGNED and UNALIGNED are shown in figure 4-9.

Figure 4-7. ALIGNED Attribute Syntax

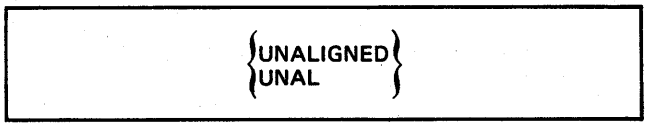


Figure 4-8. UNALIGNED Attribute Syntax

```
DECLARE A FIXED BINARY(8,2);
/*DEFAULT ALIGNED                          */

DCL B CHARACTER(10);
/*DEFAULT UNALIGNED                        */

DCL C(4) CHARACTER(5);
/*DEFAULT UNALIGNED- ARRAY PACKED          */

DCL 1 P UNALIGNED,
      2 Q,
        3 R(15) BIT(5),
        3 S CHARACTER(2) ALIGNED,
      2 T ALIGNED,
        3 V CHARACTER(3);
/*R UNALIGNED FROM Q FROM P                */
/*S ALIGNED AS DECLARED                    */
/*V ALIGNED FROM T                         */
```

Figure 4-9. ALIGNED and UNALIGNED Attribute Examples

The default alignment for BIT, CHARACTER, and PICTURE data is UNALIGNED. The default is ALIGNED for all other computational and noncomputational data. When UNALIGNED is used for BIT, CHARACTER, or PICTURE data, the specified packing occurs only for array elements or structure members with nonvarying length.

Alignment is propagated through a structure declared as ALIGNED or UNALIGNED. Each substructure or member can also be declared with an alignment. Alignment propagates from the structure to the members, level by level. The member receives the propagated alignment attribute unless the member is declared with an alignment attribute. Alignment attributes are not meaningful for structures or substructures, but alignment is propagated to each member not explicitly declared as ALIGNED or UNALIGNED. If LIKE is used in the declaration of a structure, the alignment propagates after the expansion caused by the LIKE attribute is effective.

If a variable shares storage with another variable, alignment attributes must correspond for correct access. The alignment must correspond when a defined variable is defined on the host variable; if string overlay defining is used, both the defined variable and the host variable must be completely unaligned. The alignment must correspond when a based variable is used to access a generation allocated for another variable.

Alignment of an argument used in a procedure or function reference and alignment of the corresponding parameter in the invoked procedure are significant, as described in section 6, References. If the alignment does not agree, a dummy argument is created even if alignment is not meaningful for the argument being passed. Since the use of dummy arguments that are aggregates is not supported, the alignment of an argument that is an array or structure must match the alignment of the corresponding parameter in the invoked procedure.

AREA ATTRIBUTE

The AREA attribute specifies that the variable or descriptor represents an area that can contain area values. An area is allocated storage in which based generations can be dynamically allocated and freed, as described in section 3, Data Elements. The area has a size that is specified or supplied by default.

The AREA attribute is a noncomputational data type attribute used for a variable, parameter descriptor, or returns descriptor. The AREA attribute conflicts with any other data type. AREA can be specified for a variable that is a scalar or an array. AREA conflicts with the structure attribute. The AREA attribute can be specified by keyword. The syntax of AREA is shown in figure 4-10, and examples of the AREA attribute are shown in figure 4-11.

```
AREA [(size)]

where size is an extent for the
number of 60-bit words in the area
```

Figure 4-10. AREA Attribute Syntax

AREA can be explicitly declared by DECLARE statement for a variable or explicitly specified for a descriptor. Contextual declaration of AREA occurs for an identifier used in an IN option of an ALLOCATE or FREE statement and for an identifier used in the OFFSET attribute. Contextual declaration involves use of the default area size.

The specification of size is optional. The default size is 150 words. The specification of size is an extent. The rules for extents are described later in this section under Extents.

```
DCL AREAA AREA;
/*DEFAULT SIZE 150 WORDS                    */

DCL AREAB AREA(750);
/*SIZE AS DECLARED                          */

ALLOCATE B SET(XYZ) IN(AREAC);
/*CONTEXTUAL DECLARATION OF AREAC           */

FREE B IN(AREAD);
/*CONTEXTUAL DECLARATION OF AREAD           */

DCL EFG OFFSET(AREAE);
/*CONTEXTUAL DECLARATION OF AREAE           */
```

Figure 4-11. AREA Attribute Examples

AUTOMATIC ATTRIBUTE

The AUTOMATIC attribute specifies that a variable is an automatic variable. A generation for the variable is allocated automatically whenever the block that contains the declaration of the variable is activated, as described in section 3, Data Elements. One generation of the automatic variable exists for each unterminated activation of the block.

The AUTOMATIC attribute is a storage type attribute for variables. AUTOMATIC conflicts with any other storage type and cannot be used with the EXTERNAL attribute. The scope of an automatic variable is always INTERNAL. AUTOMATIC can be specified for a variable of any aggregate type; AUTOMATIC cannot be specified for any individual array element or structure member. The AUTOMATIC attribute can be specified by keyword and can be abbreviated. The syntax of AUTOMATIC is shown in figure 4-12, and examples of the AUTOMATIC attribute are shown in figure 4-13.

```
{AUTOMATIC}
{AUTO     }
```

Figure 4-12. AUTOMATIC Attribute Syntax

```
DECLARE DEF AUTOMATIC;

DCL ABC;
/*DEFAULTS TO AUTOMATIC                     */

DECLARE GHJ INTERNAL;
/*DEFAULTS TO AUTOMATIC                     */
```

Figure 4-13. AUTOMATIC Attribute Examples

AUTOMATIC is the default storage type for any internal variable. The AUTOMATIC storage type can be declared by DECLARE statement or can be supplied by default.

BASED ATTRIBUTE

The BASED attribute specifies that a variable is a based variable. A based variable can be used to reference generations of any storage type and to allocate and free based generations, as described in section 3, Data Elements. A based reference always requires use of a based variable and a locator value. The based variable is used as the data description of the referenced generation. The locator value is used to supply the storage description of the referenced generation. The data description of the described storage, as allocated, must match the data description from the based variable in one of the following ways:

- Overlaid strings: The referenced generation and the based variable each represent either a string or an aggregate of strings, and every string in both is the same type, either all bit strings or an arbitrary mixture of character strings and pictured variables. No string can be VARYING or ALIGNED, and there can be no REFER options in the based variable. The referenced generation and the based variable can have any aggregate type. The total length of each is the sum of the string lengths. The total length of the variable must be less than or equal to the total length of the referenced generation.

- Complete correspondence: The referenced generation and the based variable must have the same aggregate type. The components (except structures) of the referenced generation and the based variable must correspond completely in data type and alignment, except that corresponding OFFSET attributes need not have identical area references. Wherever the based variable has a REFER option, the value of the referenced structure member is used to test correspondence of a string length, area size, or array bounds.

- Left to right equivalence: The referenced generation and the based variable are both undimensioned structures. If the based variable has n immediately contained members, those members must completely correspond (as described earlier) to the first n immediately contained members of the referenced generation. The referenced generation can have additional members.

The BASED attribute is a storage type attribute for variables. BASED conflicts with any other storage type and with the EXTERNAL attribute. The scope of a based variable is always INTERNAL. BASED can be specified for a variable of any aggregate type; BASED cannot be specified for any individual array element or structure member. The BASED attribute is specified by keyword. The syntax of BASED is shown in figure 4-14, and examples of the BASED attribute are shown in figure 4-15.

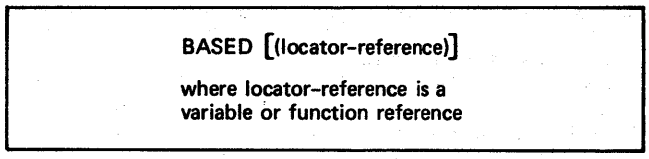


Figure 4-14. BASED Attribute Syntax

The BASED attribute must be explicitly declared by DECLARE statement. The locator-reference is an optional reference to a locator value (pointer or offset). The locator value can be a variable or a value returned from a function. For allocation of a based variable, the locator-reference must be a variable. Whenever a reference to a based variable with no explicit locator-qualifier is encountered in program execution, the locator-reference is evaluated. The locator-reference serves as an implicit locator for references if an explicit locator is not supplied. The locator-reference can be omitted if all references to the based variable supply an explicit locator and the SET option is present on all ALLOCATE and LOCATE statements used to allocate the based variable. The locator used in ALLOCATE and LOCATE statements cannot be a function reference and must be a scalar locator variable.

```
DCL B BASED;
/*LOCATOR NOT SUPPLIED                    */

DECLARE C BASED(RST);
/*LOCATOR SUPPLIED                        */
```

Figure 4-15. BASED Attribute Examples

BINARY ATTRIBUTE

The BINARY attribute specifies that an arithmetic value has the base binary. BINARY and DECIMAL are the arithmetic bases.

The BINARY attribute is an arithmetic data type attribute for variables, descriptors, and literal constants. BINARY conflicts directly with DECIMAL and cannot be used as an attribute of any nonarithmetic quantity. BINARY can be specified for a variable that is a scalar or an array. BINARY conflicts with the structure attribute. BINARY is compatible with REAL and with an arithmetic scale FIXED or FLOAT. The BINARY attribute can be specified by keyword and can be abbreviated. The syntax of BINARY is shown in figure 4-16, and examples of the BINARY attribute are shown in figure 4-17.

Figure 4-16. BINARY Attribute Syntax

```
DECLARE F FIXED BINARY(14,3);
DCL GG FLOAT BIN(12);
```

Figure 4-17. BINARY Attribute Examples

BINARY can be declared explicitly on the DECLARE statement or can be supplied by default. BINARY is supplied as the standard arithmetic default. BINARY can also be supplied in some cases when INRULE arithmetic defaults are applied. A precision attribute suffix of the form (p) or (p,q) can follow the BINARY attribute in an explicit declaration.

BIT ATTRIBUTE

The BIT attribute specifies that a value is a bit string value. BIT and CHARACTER are possible types of string values.

The BIT attribute is a string data type attribute for variables, descriptors, and literal constants. BIT conflicts with CHARACTER and any other data type attribute except VARYING and nonvarying. BIT can be specified for a variable that is a scalar or an array. BIT conflicts with the structure attribute. The BIT attribute is specified by keyword. The syntax of BIT is shown in figure 4-18, and examples of the BIT attribute are shown in figure 4-19.

```
BIT [(length)]

where length is an extent for the
number of bits in the string
```

Figure 4-18. BIT Attribute Syntax

```
DECLARE T BIT(6);
/*LENGTH AS DECLARED                    */

DECLARE R BIT;
/*DEFAULT LENGTH OF 1                   */

DCL H BIT(12) VARYING;
/*VARYING WITH MAX LENGTH OF 12         */
```

Figure 4-19. BIT Attribute Examples

The string is assumed to be nonvarying unless VARYING is specified. If the bit string is nonvarying, a constant length is maintained. If the bit string is VARYING, the bit string can have a current length smaller than the specified maximum length of the string. Length of the bit string can be from 0 through 131071 bits.

The specification of length is optional. The default length is 1 bit. The specification of length is an extent. The rules for extents are described later in this section under Extents.

BUILTIN ATTRIBUTE

The BUILTIN attribute specifies that an identifier is a builtin function. Builtin functions are described in section 11, Builtin Functions.

The BUILTIN attribute indicates that an identifier is the name of a builtin function. A builtin function can be declared INTERNAL and defaults to INTERNAL. The BUILTIN attribute conflicts with any attribute other than INTERNAL. The BUILTIN attribute can be declared by keyword. The syntax of BUILTIN is shown in figure 4-20, and examples of the BUILTIN attribute are shown in figure 4-21.

```
BUILTIN
```

Figure 4-20. BUILTIN Attribute Syntax

Explicit DECLARE statement declaration of the BUILTIN attribute is usually not necessary. If the builtin function name is used in a containing block for some other purpose, explicit declaration must be used to reestablish the name as the name of the builtin function. Contextual declaration of the BUILTIN attribute is performed whenever the name of the builtin function is used as a function reference. The BUILTIN attribute also applies to pseudovariables.

```
DECLARE DATE BUILTIN;

STT = TIME();
/*CONTEXTUAL DECLARATION OF TIME        */
```

Figure 4-21. BUILTIN Attribute Examples

CHARACTER ATTRIBUTE

The CHARACTER attribute specifies that a value is a character string value. CHARACTER and BIT are possible string types.

The CHARACTER attribute is a string data type attribute for variables, descriptors, and literal constants. CHARACTER conflicts with BIT and any other data type attributes except VARYING and nonvarying. CHARACTER can be specified for a variable that is a scalar or an array. CHARACTER conflicts with the structure attribute. The CHARACTER attribute is specified by keyword and can be abbreviated. The syntax of CHARACTER is shown in figure 4-22. and examples of the CHARACTER attribute are shown in figure 4-23.

```
{CHARACTER} [(length)]
{CHAR     }

where length is an extent for the
number of characters in the string
```

Figure 4-22. CHARACTER Attribute Syntax

```
DECLARE PLT CHAR(4);
/*LENGTH AS DECLARED                    */

DCL F CHARACTER;
/*DEFAULT LENGTH OF 1                   */

DECLARE GHK CHAR(31) VARYING;
/*VARYING WITH MAX LENGTH OF 31         */
```

Figure 4-23. CHARACTER Attribute Examples

The character string is assumed to be nonvarying unless VARYING is specified. If the character string is nonvarying, a constant length is maintained. If the string is VARYING, the current length can be smaller than the specified maximum length of the string. The length of the character string can be from 0 through 131071 characters.

The specification of length is optional. The default length is 1 character. The specification of length is an extent. Extents are described later in this section under Extents.

CONDITION ATTRIBUTE

The condition attribute is used for a programmer-named condition. Conditions are described in section 10, Conditions.

The condition attribute indicates that an identifier is the name of a programmer-named condition. The condition attribute cannot be declared explicitly for an identifier in a programmer-named condition. The assumed scope of any condition is EXTERNAL. No other attributes apply.

The condition attribute is declared contextually for an identifier referenced after the keyword CONDITION in an ON, SIGNAL, or REVERT statement. Examples of the condition attribute are shown in figure 4-24.

```
ON CONDITION(NOANS) CALL X;
/*CONTEXTUAL DECLARATION OF NOANS      */

SIGNAL CONDITION(NOANS);
/*CONTEXTUAL DECLARATION OF NOANS      */
```

Figure 4-24. Condition Attribute Examples

CONSTANT ATTRIBUTE

The constant attribute applies to identifiers used as named constants and to literal constants. Named constants and literal constants are described in section 3, Data Elements.

The constant attribute indicates that an identifier is the name of a constant. The constant attribute cannot be declared explicitly by keyword. Explicit declarations of constant are possible, and contextual declarations are also possible. The constant attribute applies to entry constants, label constants, format constants, and file constants. The constant attribute also applies to literal constants used in the program. The constant attribute conflicts with the BUILTIN, condition, and variable attributes. Examples of the constant attribute are shown in figure 4-25.

```
TR: PROCEDURE OPTIONS(MAIN);
/*TR IS ENTRY CONSTANT                 */

D: ENTRY(ONE,TWO);
/*D IS ENTRY CONSTANT                  */

DECLARE W EXTERNAL ENTRY;
/*W IS ENTRY CONSTANT                  */

Y: GOTO NEW;
/*Y IS LABEL CONSTANT                  */

F12: FORMAT((4)E(12,3));

DCL DATA FILE;
/*DATA IS FILE CONSTANT                */

AVAL = 12.5;
/*12.5 IS LITERAL CONSTANT             */
```

Figure 4-25. Constant Attribute Examples

The constant and ENTRY attributes are explicitly declared for an entry prefix on a PROCEDURE or ENTRY statement. The constant attribute is assumed for any DECLARE statement declaration if ENTRY is specified and parameter is not present.

The constant and LABEL attributes are explicitly declared for a label prefix on any statement other than a PROCEDURE, ENTRY, or FORMAT statement. The constant attribute is supplied if a DECLARE statement declaration involves the LABEL and dimension attributes and the declared identifier matches a subscripted label prefix in the same block.

The constant and format attributes are explicitly declared for a format prefix on a FORMAT statement.

The constant attribute is assumed for any DECLARE statement declaration if FILE is specified and parameter is not present. The constant and FILE attributes are contextually declared for file references in FILE options, COPY options, and I/O condition names.

CONTROLLED ATTRIBUTE

The CONTROLLED attribute specifies that a variable is a controlled variable. The allocation and freeing of controlled storage is under direct control of the programmer, as described in section 3, Data Elements. Generations for a controlled variable can be allocated by ALLOCATE statements and freed by FREE statements. The generations exist in a pushdown stack, and the generation at the top of the stack is the only one accessible.

The CONTROLLED attribute is a storage type attribute for variables. CONTROLLED conflicts with any other storage type. A controlled variable is assumed to be INTERNAL unless declared as EXTERNAL. CONTROLLED can be specified for a variable of any aggregate type; CONTROLLED cannot be specified for any individual array element or structure member. The CONTROLLED attribute is specified by keyword and can be abbreviated. The syntax of CONTROLLED is shown in figure 4-26, and examples of the CONTROLLED attribute are shown in figure 4-27.

```
{CONTROLLED}
{CTL       }
```

Figure 4-26. CONTROLLED Attribute Syntax

```
DECLARE BETAWT(20,20) CTL;
DCL EIGEN(20) CONTROLLED;
```

Figure 4-27. CONTROLLED Attribute Examples

DECIMAL ATTRIBUTE

The DECIMAL attribute specifies that an arithmetic value has the decimal base. DECIMAL and BINARY are possible arithmetic bases.

The DECIMAL attribute is an arithmetic data type attribute for variables, descriptors, and literal constants. DECIMAL conflicts directly with BINARY and cannot be used as an attribute of any nonarithmetic quantity. DECIMAL is compatible with REAL and with an arithmetic scale FIXED or FLOAT. DECIMAL can be specified for a variable that is a scalar or an array. DECIMAL conflicts with the structure attribute. The DECIMAL attribute can be specified by keyword and can be abbreviated. The syntax of DECIMAL is shown in figure 4-28, and examples of DECIMAL are shown in figure 4-29.

DECIMAL can be explicitly declared in the DECLARE statement and can be supplied by default in some cases. DECIMAL can be supplied by default only when INRULE arithmetic defaults are applied. A precision attribute suffix of the form (p) or (p,q) can follow the DECIMAL attribute in an explicit declaration.

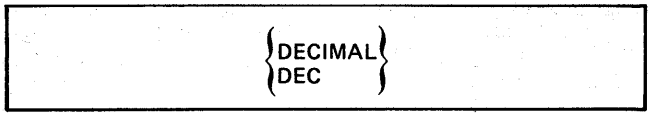


Figure 4-28. DECIMAL Attribute Syntax

```
DECLARE TT FLOAT DECIMAL(10);
DCL R FIXED DEC(12,5);
```

Figure 4-29. DECIMAL Attribute Examples

DEFINED ATTRIBUTE

The DEFINED attribute specifies a variable that is a defined variable. A defined variable has no storage of its own and is used to access storage allocated for another variable that is called the host variable. Either the defined variable or the host variable can access the shared storage.

The DEFINED attribute is a storage type attribute for variables. DEFINED conflicts with any other storage type. DEFINED conflicts with EXTERNAL and defaults to INTERNAL. DEFINED can be specified for a variable with any aggregate type; DEFINED cannot be specified for any individual array element or structure member. The defined variable cannot be a VARYING string and cannot be an aggregate that contains any VARYING strings. DEFINED conflicts with the INITIAL attribute, and INITIAL cannot be used to initialize any member of a defined structure. Storage type for the host variable can be STATIC, AUTOMATIC, CONTROLLED, or parameter. The host variable must be declared in the same block as the defined variable, or in a containing block. The host variable cannot be a VARYING string and cannot be an aggregate that contains any VARYING strings. The DEFINED attribute is specified by keyword and can be abbreviated. The POSITION attribute can accompany DEFINED and can also be abbreviated. The syntax of DEFINED and POSITION is shown in figure 4-30.

Extents in the declaration are evaluated when the block containing the declaration is activated, as described later in this section under Extents. Each time the defined variable is referenced, any subscripts in the host reference are evaluated, the appropriate generation of the host-reference is determined, and any POSITION attribute in the declaration of the defined variable is evaluated.

Three types of defining can be indicated by the DECLARE statement declaration of a defined variable. Each type of defining is described separately.

Simple Defining

The defined variable and the host reference must have the same aggregate type, except that:

- If the defined variable is an array, the array bounds must be such that the defined generation does not extend outside the host generation.
- If the defined variable has the BIT, CHARACTER, or PICTURE attribute, the string length of the defined generation must be less than or equal to the string length of the host generation.
- If the defined variable is a structure, all extents of the members must precisely match the corresponding extents of the host generation. The components (except structures) of the defined variable and the host reference must correspond completely in data type and alignment, except that corresponding area references in OFFSET attributes can be different.

The host reference can contain asterisk subscripts. Array element correspondence between the defined variable and the host reference is established by subscript value, not by indexing relative to lower bounds.

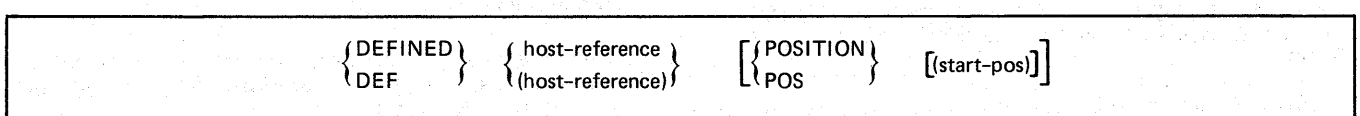
Figure 4-31.1 shows an example of simple defining.

```
DECLARE A(12);
DECLARE B(3:6) DEFINED(A);
.
.
.
/* B IS SIMPLY DEFINED ON A */
.
.
```

Figure 4-31.1. Simple Defining Example

String Overlay Defining and POSITION Attribute

The defined variable and the host reference each represents either a string or an aggregate of strings, and every string in each is the same type, either all bit strings or an arbitrary



where host-reference is a simple, subscripted, structure-qualified, or locator-qualified reference to the host variable

and where start-pos is an unsigned decimal integer

Figure 4-30. DEFINED and POSITION Attribute Syntax

mixture of character strings and pictured variables. No string can be VARYING or ALIGNED. The defined variable and the host reference can have any aggregate type. The total length of each is the sum of the string lengths. The sum of the value of the POSITION attribute and the total length of the defined variable must not be greater than the total length of the host generation, plus one.

The POSITION attribute optionally specifies an unsigned decimal integer greater than zero. The value specifies the starting position of the definition with respect to the first character or bit of the host string. If the POSITION attribute or the start-pos specification is omitted, POSITION (1) is assumed.

Figure 4-31.2 shows an example that uses the POSITION attribute. The character array D overlays the structure C beginning at the second position. Therefore, D(1), D(2), D(3), and D(4) reference CA(2), CA(3), CA(4), and CA(5) respectively.

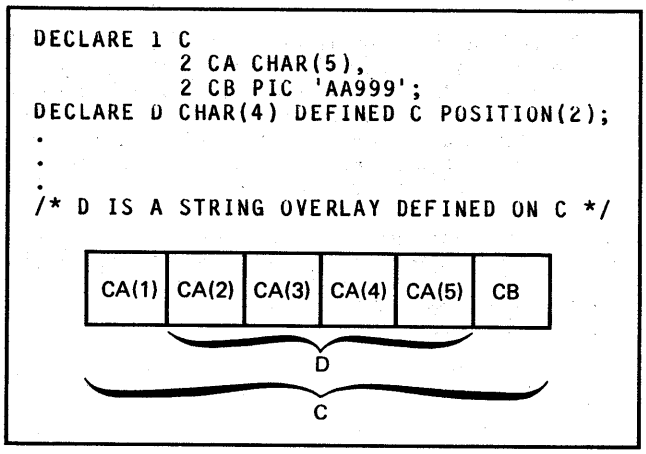


Figure 4-31.2. Defining with POSITION Example

Array Defining with iSUB

Using iSUBs allows the creation of a defined array that consists of designated elements from a host array. This allows an array to be defined as a row, column, diagonal, or other part of the host array.

The defined variable and the host-reference must be arrays with the same aggregate type. Corresponding elements, except structures, must have the same alignment and data type attributes. Both the defined array and host array can be arrays of structures.

The defined variable is defined on selected elements of the host array using iSUBs. The i of iSUB is replaced with a decimal integer corresponding to a subscript of the host array. For example, 1SUB specifies the first subscript of the host array, 2SUB specifies the second subscript, and so forth.

Expressions and constants can also be used as subscripts of the host-reference. References to the defined variable result in references to the host variable. Each use of the defined variable causes evaluation of the host-reference subscripts.

The first evaluated subscript in the reference to the defined variable replaces all occurrences of 1SUB in the subscripted host-reference. The second evaluated subscript replaces all occurrences of 2SUB in the subscripted host-reference, and so forth.

When all of the evaluated subscripts in the reference to the defined variable have replaced all occurrences of iSUB in the host-reference, the subscript expressions in the subscripted host-reference are evaluated. This results in a subscripted reference to an element of the host array. The program must not depend upon the order of evaluation of the subscripts.

A defined variable that is an iSUB-defined array cannot be referenced without subscripts or with asterisk subscripts.

Figure 4-31.3 is an example of how iSUB defining can be used to reference a diagonal of a host array. A reference to B(1) causes both 1SUBs to be replaced with 1, which results in a reference to A(1,1). A reference to B(2) results in a reference to A(2,2), and so forth.

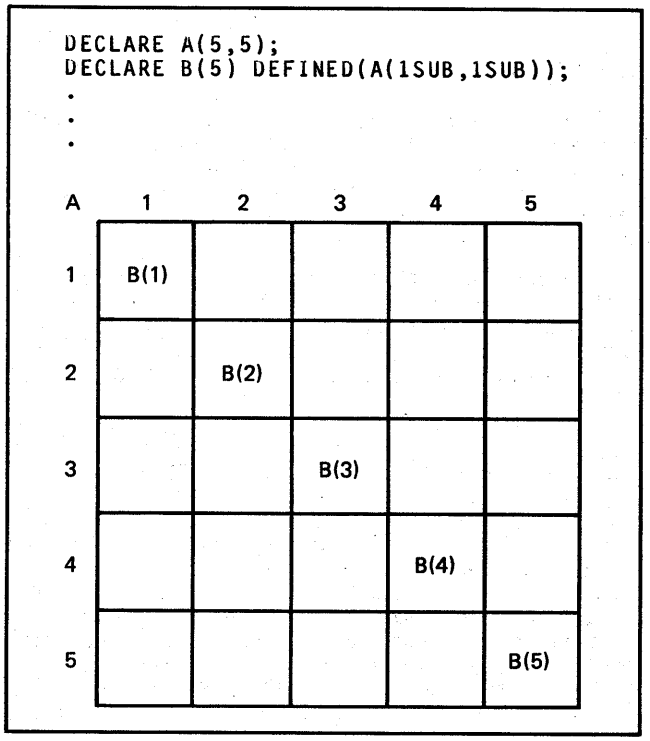


Figure 4-31.3. Diagonal Selection with iSUBs

Figure 4-31.4 shows another example of the use of iSUBs. In this case the iSUBs are part of a subscript expression.

DIMENSION ATTRIBUTE

The dimension attribute is used for an array that contains array elements. Arrays are described under Aggregates in section 3, Data Elements.

Dimension is an aggregation attribute used for variables, named constants, or parameter descriptors. The dimension attribute cannot be used for file constants, file variables, entry constants, entry variables, or format constants. A returns descriptor cannot have the dimension attribute. The dimension attribute cannot be explicitly declared by key-

```
DCL A(3,3);
DCL B(9) DEF A(((1SUB+2)/3),(1SUB-(3*((1SUB-1)/3))));
```

A	1	2	3
1	B(1)	B(2)	B(3)
2	B(4)	B(5)	B(6)
3	B(7)	B(8)	B(9)

Figure 4-31.4. Expressions with iSUBs

word. Dimension is established by appearance of a dimension suffix. The syntax of the dimension suffix is shown in Figure 4-32, and examples of dimension are shown in Figure 4-33.

```
(array-bounds , , , )

where array-bounds is

{ upper-bound                  }
{ lower-bound:upper-bound      }
{ *                            }

where upper-bound is an extent expression

where lower-bound is an extent expression

where * is an asterisk extent indicating that array-bounds
are to be acquired from the argument
```

Figure 4-32. Dimension Attribute Suffix Syntax

```
DECLARE K(30);
/*K HAS ONE DIMENSION 1:30                    */

DCL L(-4:5,J-3);
/* L HAS TWO DIMENSIONS,                      */
/* -4:5 (SPAN IS 10)                          */

/* 1:J-3 (SPAN DEPENDS ON VALUE OF J)         */

DECLARE 1 X(5),
          2 Y(3) CHARACTER(15);
/* Y EFFECTIVELY HAS TWO DIMENSIONS,          */
/* 1:5 INHERITED FROM X                       */
/* 1:3 (SPAN IS 3)                            */

DECLARE E ENTRY((10,10) FIXED);
/*DIMENSIONED PARAMETER DESCRIPTOR            */
```

Figure 4-33. Dimension Attribute Examples

The specification of dimension in a DECLARE statement must follow the name of the identifier and must precede any other attributes declared for the identifier. In a parameter descriptor, dimension must be specified as the first attribute if used.

Each dimension of the array is specified as an array-bounds that is an extent. Each array-bounds indicates the span, that is, the number of elements in each dimension. If only the upper-bound is specified, the upper-bound must be greater than or equal to one. The lower-bound is assumed to be one, and the span is the upper-bound. If the lower-bound and upper-bound are both specified, the upper-bound must be greater than or equal to the lower-bound. The span is calculated as (upper-bound)-(lower-bound)+1. An asterisk extent can be used for a parameter to indicate that the array-bounds are to be acquired from the array-bounds for the corresponding dimension of the argument. Extents are described later in this section under Extents.

The upper-bound cannot exceed 131071 and the lower-bound cannot be less than -131071. The total number of dimensions is restricted to 32, including all bounds specified and all bounds inherited from any containing structure. A member of a structure inherits any dimensions of each containing structure.

DIRECT ATTRIBUTE

The DIRECT attribute specifies that access to records in a file is to be by direct reference to the key of each record. DIRECT is only used for record files.

DIRECT is a file description attribute used for a file constant. DIRECT cannot be specified for a file variable. DIRECT conflicts with the STREAM, PRINT, and

SEQUENTIAL attributes. SEQUENTIAL is an alternative type of access to records of a file. STREAM and PRINT apply to files used for stream I/O rather than record I/O. The DIRECT attribute is specified by keyword. The syntax of DIRECT is shown in figure 4-34, and examples of DIRECT are shown in figure 4-35.

```
DIRECT
```

Figure 4-34. DIRECT Attribute Syntax

```
DCL U FILE DIRECT;
DCL CTFILE INPUT KEYED RECORD DIRECT;
```

Figure 4-35. DIRECT Attribute Examples

Appearance of the DIRECT attribute in a DECLARE statement declaration implies the FILE attribute. The DIRECT attribute also implies the KEYED attribute and the RECORD attribute.

ENTRY ATTRIBUTE

The ENTRY attribute specifies that the identifier represents an entry value. The ENTRY attribute can be associated with a constant, parameter, or parameter descriptor.

The ENTRY attribute is a noncomputational data type attribute used for a variable, named constant, or parameter descriptor. ENTRY conflicts with any other data type except RETURNS. ENTRY and RETURNS are used together for any entry value that represents a function. Entry constants must be scalars. ENTRY variables can only be used as entry parameters. ENTRY conflicts with the structure and member attributes. The ENTRY attribute can be specified by keyword. The syntax of ENTRY is shown in figure 4-36, and examples of the ENTRY attribute are shown in figure 4-37.

```
{ ENTRY                                  }
{ ENTRY ( )                              }
{ ENTRY (parameter-descriptor , , , )    }
```

where each parameter-descriptor specifies any consistent combination of the attributes dimension, structure, member, ALIGNED, UNALIGNED, ENTRY, RETURNS, LABEL, FILE, AREA, POINTER, OFFSET, REAL, BINARY, DECIMAL, FIXED, FLOAT, precision, BIT, CHARACTER, VARYING, and PICTURE.

Figure 4-36. ENTRY Attribute Syntax

Each statement prefix on a PROCEDURE or ENTRY statement is explicitly declared with the constant and ENTRY attributes. In addition, an entry constant declared by use as the entry name of an external procedure receives the EXTERNAL attribute. An entry name of an internal procedure receives the INTERNAL attribute.

```
P:
PROCEDURE OPTIONS(MAIN);
/*P IS EXTERNAL ENTRY CONSTANT          */
   DCL ZC ENTRY(CHAR(5));
   /*DECLARATION FOR EXTERNAL ENTRY     */
   DCL STR CHAR(5);
ZA:
   BEGIN;
      .
      .
      .
      CALL ZB(STR); /*CALL INTERNAL     */
      CALL ZC(STR); /*CALL EXTERNAL     */
   END ZA;
ZB:
   PROCEDURE(HJK);
   /*ZB IS INTERNAL ENTRY CONSTANT      */
      DCL HJK CHARACTER(5);
      .
      .
      .
   END ZB;
END P;

ZC:
PROCEDURE(ASTRING);
/*ZC IS EXTERNAL ENTRY CONSTANT         */
   DCL ASTRING CHAR(5);
   .
   .
   .
END ZC;
```

Figure 4-37. ENTRY Attribute Examples

For procedure calls or function references to another external procedure, declaration of an external entry constant is established by DECLARE statement declaration. When an identifier is declared in the DECLARE statement with the ENTRY attribute, the constant and EXTERNAL attributes are supplied by default. Explicit declaration of entry constants is accomplished by use of the entry prefix for internal or external entry constants. Entry constants naming entry points in other external procedures can only be known by DECLARE statement declaration. Parameter descriptors describe all parameters that the external procedure expects to receive when invoked.

An entry parameter is declared by DECLARE statement in the procedure that receives the entry parameter. Since all parameters in the parameter list of a PROCEDURE or ENTRY statement are explicitly declared with the parameter attribute, the variable attribute is applied by default to a parameter declared as ENTRY.

Parameter Descriptors

Parameter descriptors must be specified for all parameters to be passed to an external procedure. The parameter descriptors are declared in the DECLARE statement declaration of the external entry constant. When no parameters are to be passed to the external procedure, the declaration can be ENTRY or ENTRY( ). A single parameter descriptor can be supplied, or several parameter descriptors can be supplied with commas used to separate the descriptors. A null descriptor is a parameter descriptor with no declared attributes. Commas keep the place of a null descriptor.

Each parameter descriptor can have the same aggregation, alignment, and data type attributes as any variable. A parameter descriptor cannot have the INITIAL attribute. Scope and storage type do not apply to a parameter descriptor. Each declared parameter descriptor is completed with default attributes. The name of the entry constant affects any INRULE arithmetic defaults supplied for each descriptor. A null descriptor always receives arithmetic defaults. Since the entry constant is external, the completed set of attributes for each parameter descriptor must exactly match the attributes of each corresponding parameter in the external procedure to be invoked. Extents that can be used for the parameter descriptors are described later in this section under Extents.

Nested ENTRY Attributes

If a parameter descriptor is declared as type ENTRY, the ENTRY attribute is nested. The nested ENTRY attribute occurs when an entry value is passed to an external procedure. The parameter descriptor declared with type ENTRY can be declared with its own parameter descriptors. Parameter descriptors for the nested ENTRY attribute can be omitted; if specified, they are checked for consistency but not used.

ENVIRONMENT ATTRIBUTE

The ENVIRONMENT attribute is used to specify CYBER Record Manager (CRM) file processing options for a file accessed by the program. The ENVIRONMENT attribute is implementation-defined for CYBER Record Manager.

The ENVIRONMENT attribute is a file description attribute associated with a file constant. The ENVIRONMENT attribute can be used in conjunction with any other file description attributes. ENVIRONMENT cannot be specified for a file variable. The ENVIRONMENT attribute is specified by keyword and can be abbreviated. The syntax of ENVIRONMENT is shown in figure 4-38, and examples of the ENVIRONMENT attribute are shown in figure 4-39.

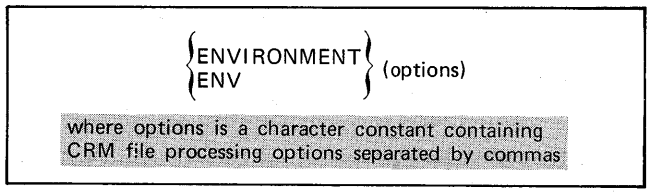


Figure 4-38. ENVIRONMENT Attribute Syntax

```
DCL TIKL FILE INPUT ENV(*FO=SQ,RT=F*);
DECLARE AB DIRECT INPUT ENV(*FO=WA*);
```

Figure 4-39. ENVIRONMENT Attribute Examples

ENVIRONMENT can be established by DECLARE statement declaration. The ENVIRONMENT options are specified as a simple character string constant containing a list of CRM file processing options separated by commas. The processing of ENVIRONMENT is described in section 9, CYBER Record Manager Interface.

EXTERNAL ATTRIBUTE

The EXTERNAL attribute establishes an external scope for the identifier. An external identifier is known in the same scope as an internal identifier and can also be known outside the external procedure. When declarations of the same identifier as EXTERNAL exist in separate internal or external procedures, all declarations of the identifier are effectively merged to reference the same identifier.

The EXTERNAL attribute is a scope attribute for variables, named constants, or programmer-named conditions. EXTERNAL and INTERNAL are conflicting attributes. For variables, EXTERNAL conflicts with any storage type except STATIC and CONTROLLED. EXTERNAL can be specified for a variable with any aggregate type; EXTERNAL cannot be specified for any individual array element or structure member. For named constants, EXTERNAL conflicts with any data type except FILE or ENTRY. EXTERNAL is the only possible scope for condition names. EXTERNAL cannot be used for builtin function names. The EXTERNAL attribute is specified by keyword and can be abbreviated. The syntax of EXTERNAL is shown in figure 4-40, and examples of the EXTERNAL attribute are shown in figure 4-41.

Figure 4-40. EXTERNAL Attribute Syntax

```
X: PROC(A,B);
/*ENTRY CONSTANT X IS EXTERNAL IF      */
/*PROCEDURE X IS EXTERNAL              */

DCL TT ENTRY(BIT(10));
/*DEFAULTS TO EXTERNAL                 */

DECLARE GG(100,20) EXTERNAL;

DCL F23456789 FILE;
/*DEFAULTS TO EXTERNAL                 */
/*EXTERNAL NAME IS F234789             */
```

Figure 4-41. EXTERNAL Attribute Examples

EXTERNAL can be explicitly declared by DECLARE statement for a static or controlled variable, a file constant, or an entry constant. The entry prefix on a PROCEDURE or ENTRY statement for an external procedure is explicitly declared by usage as an external entry constant. Identifiers used in programmer-named conditions are contextually declared as condition and EXTERNAL. The EXTERNAL attribute is supplied by default when FILE or ENTRY is declared, unless the identifier is a parameter. When EXTERNAL is declared in a DECLARE statement for a variable and no storage type is specified, STATIC is supplied.

Declarations of an external identifier are merged to reference the same identifier. Each declaration of an identifier involves the specified attributes and the attributes

supplied by default. The completed attribute sets that result from all declarations of an external identifier must be identical, except that:

- Corresponding members names for an external structure can be different.
- Corresponding area references for an external offset variable can be different.
- Corresponding extents and INITIAL attribute for an external controlled variable can be different.

> NOTE
>
> Correspondence of attributes of external identifiers is not examined. Identical declarations for the external identifier should be used.

The operating system restricts external names and local CRM file names to a maximum of seven alphanumeric characters. The characters $ # @ and _ should not be used in any external identifier. External identifiers in PL/I are modified as necessary for use as names known to the operating system, in the following way:

- If any external name is longer than 7 characters, the name known to the operating system is formed from the first four and the last three characters.

Static external variables can be initialized only in the outermost block of the main external procedure.

FILE ATTRIBUTE

The FILE attribute specifies that an identifier represents a file value. Input/output operations are performed on file constants associated with local files, as described in section 8, Input/Output.

The FILE attribute is a noncomputational data type attribute for variables, named constants, or parameter descriptors. FILE conflicts with any other data type attribute, but a file constant can have file description attributes. Arrays of file constants or file variables are not supported. File variables are supported only as file parameters. File conflicts with the structure attribute. The FILE attribute can be specified by keyword. The syntax of FILE is shown in figure 4-42, and examples of the FILE attribute are shown in figure 4-43.

```
FILE
```

Figure 4-42. FILE Attribute Syntax

```
DCL DATA1 FILE INPUT;

DCL DATA3 PRINT;
/*PRINT IMPLIES FILE FOR DATA3          */

OPEN FILE(LIST) RECORD DIRECT OUTPUT;
/*CONTEXTUAL DECLARATION OF LIST        */

GET COPY(PARTS) LIST(X,Y,Z) SKIP;
/*CONTEXTUAL DECLARATION OF PARTS       */
```

Figure 4-43. FILE Attribute Examples

A file constant, file parameter, or file parameter descriptor can be explicitly declared with the FILE attribute. Appearance of STREAM, RECORD, INPUT, OUTPUT, UPDATE, SEQUENTIAL, DIRECT, KEYED, or PRINT in a DECLARE statement declaration implies the FILE attribute. Contextual declaration of FILE and constant occurs for an identifier used as the file reference in the FILE option in any I/O statement, in an ON, SIGNAL, or REVERT statement for an I/O condition, and in the COPY option of the GET statement. The constant and EXTERNAL attributes are applied by default if parameter is not present. If parameter is present, the variable and INTERNAL attributes are applied by default. File description attributes cannot be associated with a file parameter. Default file description attributes are supplied when the file constant is opened, as described in section 8, Input/Output. If no file description attributes are specified, STREAM and INPUT are supplied by default.

FIXED ATTRIBUTE

The FIXED attribute indicates an arithmetic value with a fixed point scale. The precision of a fixed point value indicates the total number of maintained decimal or binary digits and the number of fractional digits following the decimal or binary point.

The FIXED attribute is an arithmetic data type attribute for variables, parameter descriptors, returns descriptors, and literal constants. The FIXED attribute can be implied by the picture specification for a pictured numeric item. The FIXED attribute conflicts directly with FLOAT and cannot be used with any nonarithmetic data type. The FIXED attribute can be specified for a variable that is a scalar or an array. FIXED conflicts with the structure attribute. The FIXED attribute is specified by keyword. The syntax of the FIXED attribute is shown in figure 4-44, and examples of FIXED are shown in figure 4-45.

```
FIXED
```

Figure 4-44. FIXED Attribute Syntax

```
DECLARE FGH FIXED DECIMAL(8,1);
DCL JVALUE FIXED BINARY(18,5);
```

Figure 4-45. FIXED Attribute Examples

The FIXED attribute can be declared explicitly by DECLARE statement. Precision of the form (p) or (p,q) can follow FIXED. A pictured numeric item can imply FIXED but cannot be declared as FIXED. When a precision of the form (p,q) is declared for an arithmetic quantity, the FIXED attribute is implied. When arithmetic defaults are applied, FIXED is the standard arithmetic default. FIXED can also be supplied in some cases when INRULE arithmetic defaults are applied.

FLOAT ATTRIBUTE

The FLOAT attribute indicates an arithmetic value with a floating point scale. The effective location of the decimal or binary point is dependent on the exponent value. The number of significant decimal or binary digits does not depend on the arithmetic value. The precision of a floating point value is the total number of significant digits.

The FLOAT attribute is an arithmetic data type attribute for variables, parameter descriptors, returns descriptors, and literal constants. The FLOAT attribute can be implied by the picture specification for a pictured numeric item. The FLOAT attribute conflicts with the FIXED attribute and cannot be used with any nonarithmetic data type. The FLOAT attribute can be specified for a scalar or an array. FLOAT conflicts with the structure attribute. The FLOAT attribute is specified by keyword. The syntax of FLOAT is shown in figure 4-46, and examples of FLOAT are shown in figure 4-47.

Figure 4-46. FLOAT Attribute Syntax

```
DECLARE KLM FLOAT DECIMAL(6);
DCL NUMBER BINARY FLOAT(12);
```

Figure 4-47. FLOAT Attribute Examples

The FLOAT attribute can be declared explicitly by DECLARE statement. Precision of the form (p) can follow FLOAT. A pictured numeric item can imply FLOAT but cannot be declared as FLOAT. FLOAT can be supplied by default only when INRULE arithmetic defaults are applied.

FORMAT ATTRIBUTE

The format attribute indicates a format value that can be used in edit-directed stream I/O operations.

The format attribute is a noncomputational data type attribute for named constants. The format attribute conflicts with any other data type. Arrays of format constants are not supported. Format variables are not supported. The format attribute is implied and cannot be specified by keyword. An example of the format attribute is shown in figure 4-48.

```
DECLARE F FILE PRINT;
•
•
•
FMT2: FORMAT(F(8,2), E(12,3,2), A(16));
•
•
•
PUT FILE(F) LINE EDIT(I,J,K) (R(FMT2));
```

Figure 4-48. Format Attribute Example

The format and constant attributes are explicitly declared for each identifier used as a format prefix on a FORMAT statement. The INTERNAL attribute is applied by default to format constants.

INITIAL ATTRIBUTE

The INITIAL attribute is used to initialize a variable. The assignment of an initial value to a scalar or initial values to an array takes place whenever a generation is allocated for the variable.

The INITIAL attribute is the initialization attribute for variables. INITIAL can be specified for a scalar or an array. INITIAL conflicts with the structure attribute. INITIAL cannot be used to initialize any member of an array of structures. Static external variables can be initialized only in the outermost block of the main external procedure. Static label variables cannot be initialized. INITIAL cannot be used to initialize a defined variable or a member of a defined structure. INITIAL cannot be used to initialize a parameter or a member of a structure that is a parameter. The INITIAL attribute is specified by keyword and can be abbreviated. The syntax of INITIAL is shown in figure 4-49, and examples of INITIAL are shown in figure 4-50.

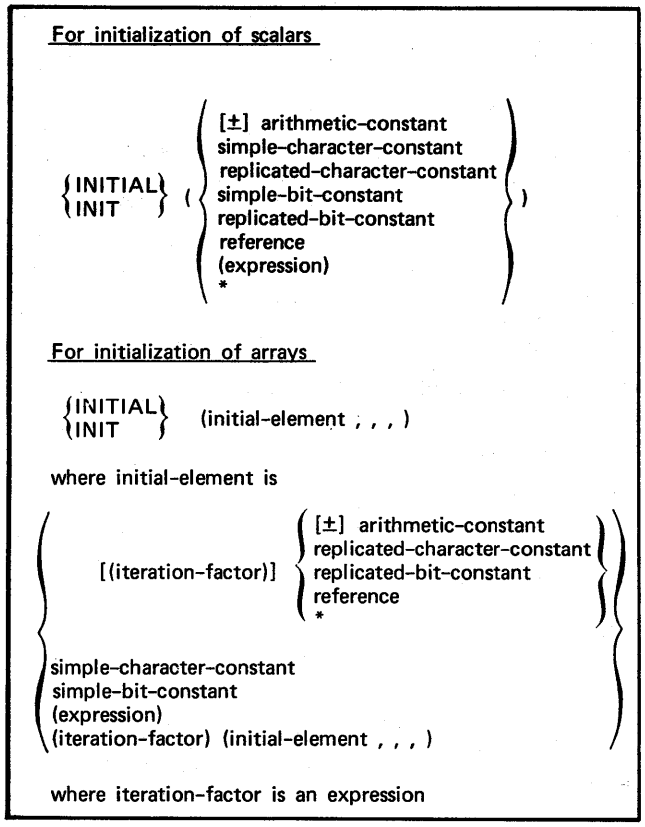


Figure 4-49. INITIAL Attribute Syntax

```
/*INITIALIZATION OF SCALARS                    */
DECLARE T FIXED DEC(8,2) INIT(12.34);
DCL U FLOAT BIN INIT((T+5));
DCL BA BIT(5) INIT(^11011^B);
DCL BB BIT(6) INIT((2)^101^B);
DCL P POINTER INITIAL(NULL());
DECLARE SWITCH INITIAL(0);
DCL AREA4 AREA INIT(AREA5);
DCL (C FIXED, D FLOAT) STATIC INIT(5);

/*INITIALIZATION OF ARRAYS                     */
DCL A(10) INITIAL((5)-3, (5)((-X));
DCL X(10,10) DEC INIT((50)*, (50)1.5);
DCL BC(5) BIT(3) INIT((5)(1)^011^B);
DCL BD(5) BIT(3) INIT((5)(3)^1^B);
DCL B(6) CHAR(5) INIT((6)((^ABC^)));
```

Figure 4-50. INITIAL Attribute Examples

INITIAL is explicitly specified by DECLARE statement, and the initial values to be assigned are supplied. A scalar initial value can be assigned to a scalar variable. Scalar initial values can be assigned to individual elements of an array. References and expressions in the INITIAL attribute are evaluated when allocation occurs for the variable to be initialized.

When a single initial value is assigned, the initial value can be specified as a literal constant, a reference, or an expression enclosed in parentheses. A literal constant can be an arithmetic, character or bit constant. Some prefix operators can be used without the need for enclosing parentheses; all other operators must be used in a parenthesized expression. A reference can be a variable, named constant, or function reference. For initialization of an automatic or controlled variable, a reference must not refer to an earlier generation of the same variable. Multiple initial values cannot be specified for initialization of a scalar.

When INITIAL is used to initialize elements of an array, the specification of initial values includes a number of initial-elements. Any initial-element can be preceded by a parenthesized iteration factor. Every iteration factor must be scalar and computational. The initial elements in a parenthesized list are processed in order from left to right, except as controlled by iteration factors. When an iteration factor is encountered, it is evaluated and converted if necessary to fixed binary integer. A negative iteration factor is assumed to be zero. The following value, asterisk, or entire parenthesized list of initial elements is repeated the specified number of times. For example,

```
( (2) (5,4), *, (3) 6, (7*A) )
```

is processed in the order

```
5 4 5 4 * 6 6 6 7*A
```

For a STATIC variable, iteration factors can only include expressions with operands that are literal constants and operators that are + - * / or ¬. For a STATIC variable, each initial element can only be a reference to a NULL or EMPTY builtin function or an expression with operands that are literal constants and operators that are + - * / ¬or | |.

Note that an iteration factor cannot be specified for a simple character constant or simple bit constant. If a parenthesized factor precedes a simple character or bit constant, the factor is assumed to be a repetition factor for the constant. The simple character or bit constant can be specified with an iteration factor and the repetition factor (1). The simple character or bit constant can also be used as an expression.

The initial values for an array are applied in the order in which the array elements are stored. If the total number of initial values is greater than the number of elements of the array, the excess initial values are ignored. If the total number is fewer, only part of the array is initialized. Assignment of initial values begins with the first element of the array. An asterisk specifies no initialization of the individual element. When asterisks are used, some elements of the array can be uninitialized even though elements that follow in the storage order of the array are initialized.

INPUT ATTRIBUTE

The INPUT attribute indicates that a file is to be used for input. An input file can be used for stream I/O or record I/O operations.

The INPUT attribute is a file description attribute used for a file constant. INPUT cannot be specified for a file variable. INPUT conflicts with OUTPUT, UPDATE, and PRINT. OUTPUT and UPDATE are alternatives for the usage of the file. PRINT applies only to OUTPUT files. The INPUT attribute is specified by keyword. The syntax of INPUT is shown in figure 4-51, and examples of INPUT are shown in figure 4-52.

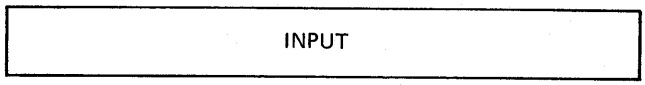


Figure 4-51. INPUT Attribute Syntax

```
DCL RGH FILE INPUT;
DECLARE TEST#FILE RECORD INPUT DIRECT;
```

Figure 4-52. INPUT Attribute Examples

Appearance of the INPUT attribute in a DECLARE statement declaration implies the FILE attribute. The INPUT attribute is the default for a record file or stream file.

INTERNAL ATTRIBUTE

The INTERNAL attribute establishes an internal scope for the identifier. An internal identifier is known in the block that immediately contains the declaration of the identifier. In addition, the identifier is known in all contained blocks except those in which another declaration for the same name is effective.

The INTERNAL attribute is a scope type attribute for variables, named constants, and builtin functions. INTERNAL and EXTERNAL are alternatives for the scope of an identifier. INTERNAL can be specified for a variable with any aggregate type. INTERNAL cannot be specified for any individual array element, but INTERNAL can be specified for a structure member. INTERNAL conflicts with the condition attribute. The INTERNAL attribute can be specified by keyword and can be abbreviated. The syntax of INTERNAL is shown in figure 4-53, and examples of INTERNAL are shown in figure 4-54.

Figure 4-53. INTERNAL Attribute Syntax

```
DECLARE VB INTERNAL AUTOMATIC;

DCL S AUTOMATIC;
/*DEFAULTS TO INTERNAL                    */
```

Figure 4-54. INTERNAL Attribute Examples

INTERNAL can be explicitly declared in the DECLARE statement for a variable of any storage type, a file constant, an array of label constants, or a builtin function name. INTERNAL is the default scope for any identifier except a file constant, an entry constant, or a programmer-named condition.

KEYED ATTRIBUTE

The KEYED attribute indicates that a file is to be accessed by record keys. The KEYED attribute can only be used for record I/O operations.

The KEYED attribute is a file description attribute used for a file constant. KEYED cannot be specified for a file variable. The KEYED attribute conflicts with the STREAM and PRINT attributes that apply to stream I/O operations. The KEYED attribute is specified by keyword. The syntax of KEYED is shown in figure 4-55, and examples of KEYED are shown in figure 4-56.

Figure 4-55. KEYED Attribute Syntax

```
DECLARE BIN FILE UPDATE DIRECT KEYED;
DCL LOOPRUN FILE KEYED;
```

Figure 4-56. KEYED Attribute Examples

Appearance of the KEYED attribute in a DECLARE statement declaration implies the FILE attribute. The KEYED attribute also implies RECORD and can be used with SEQUENTIAL or DIRECT. For a record file, the default is SEQUENTIAL.

LABEL ATTRIBUTE

The LABEL attribute is used for a label value that identifies a statement. A label value also identifies a particular activation of the block containing the statement.

The LABEL attribute is a noncomputational data type attribute for variables and named constants. The LABEL attribute conflicts with any other data type. Label constants can be scalars or arrays. Label constants cannot be external. LABEL can be specified for a variable that is a scalar or an array. LABEL conflicts with the structure attribute. Static label variables cannot be initialized. Automatic, controlled, or based label variables can be initialized. The LABEL attribute can be specified by keyword. The syntax of LABEL is shown in figure 4-57, and examples are shown in figure 4-58.

Figure 4-57. LABEL Attribute Syntax

The LABEL attribute is explicitly declared by usage for all scalar label constants. Each label prefix on a statement other than a PROCEDURE, ENTRY, or FORMAT statement is explicitly declared as a label constant. When defaults are applied, each label constant defaults to INTERNAL.

An array of label constants must be declared by DECLARE statement, and at least one of the elements of the array must be declared explicitly by appearing as a subscripted label prefix. The array of label constants can only be declared by DECLARE statement with the attributes dimension, LABEL, and INTERNAL. The dimension and LABEL attributes are required to establish the array of label constants. The subscripted label prefixes that belong to the array must be immediately contained in the same block as the DECLARE statement declaration of the array. The subscript for each subscripted label prefix must be within the array bounds specified in the declaration.

```
YY:
PROCEDURE;
   DECLARE LABVAR LABEL;
   /*LABVAR IS LABEL VARIABLE          */
   LABVAR = LOOP1;
   •
   •
   •
LOOP1:
/*LOOP1 IS LABEL CONSTANT             */
   BEGIN;
      •
      •
      •
   END;
   •
   •
   •
   GOTO LABVAR;
   •
   •
   •
END YY;
```

Figure 4-58. LABEL Attribute Examples

Label variables must be declared explicitly by DECLARE statement. Label values can be passed as arguments, and the LABEL attribute can be declared for a parameter descriptor. The LABEL attribute cannot be used for a returns descriptor.

OFFSET ATTRIBUTE

The OFFSET attribute indicates an offset value. An offset value is used for references to based variables to identify the displacement of a specific based generation within an area.

The OFFSET attribute is a noncomputational data type attribute for variables, parameters descriptors, or returns descriptors. An offset variable can have any scope or storage type. The OFFSET attribute conflicts with any other data type. OFFSET can be specified for a variable that is a scalar or an array. OFFSET conflicts with the structure attribute. The OFFSET attribute is specified by keyword. The syntax of OFFSET is shown in figure 4-59, and an example of OFFSET is shown in figure 4-60.

The OFFSET attribute must be declared by DECLARE statement. The area-reference is an optionally specified area in which the offset is to be effective. The area-reference is used implicitly for the OFFSET variable in the following circumstances:

- For allocating the based variable, if the offset variable is used in the SET option or assumed SET option of an ALLOCATE statement with no IN option
- For freeing the based variable, if the offset variable is used as the locator or implicit locator in a FREE statement with no IN option

- In all references to based variables if the offset variable is used as the locator or implicit locator.

If an area value is written to a record file and read in again, the offset value remains valid. If the area value is assigned to another area by assignment or by INITIAL attribute, the offset value also remains valid.

```
OFFSET [(area-reference)]
```

Figure 4-59. OFFSET Attribute Syntax

```
DCL BVAR BASED;
DCL AREA4 AREA;
DCL A4OFF OFFSET(AREA4);
/*EXPLICIT DECLARATION OF A4OFF        */
.
.
.
ALLOCATE BVAR SET(A4OFF);
A4OFF->BVAR = TEST_RES;
```

Figure 4-60. OFFSET Attribute Example

OUTPUT ATTRIBUTE

The OUTPUT attribute indicates that a file is to be used for output. An output file can be used for stream I/O or record I/O operations. An output file for stream I/O can have the PRINT attribute.

The OUTPUT attribute is a file description attribute used for a file constant. OUTPUT cannot be specified for a file variable. OUTPUT conflicts with INPUT and UPDATE. INPUT and UPDATE are alternatives for the use of the file. The OUTPUT attribute is specified by keyword. The syntax of OUTPUT is shown in figure 4-61, and examples of OUTPUT are shown in figure 4-62.

```
OUTPUT
```

Figure 4-61. OUTPUT Attribute Syntax

```
DCL RESULT OUTPUT PRINT;
DCL THJ FILE RECORD OUTPUT;
```

Figure 4-62. OUTPUT Attribute Examples

Appearance of the OUTPUT attribute in a DECLARE statement declaration implies the FILE attribute.

PARAMETER ATTRIBUTE

The parameter attribute is used for a parameter variable. Each parameter is associated with an entry point of a procedure. Each variable used as a parameter appears in the parameter list of a PROCEDURE or ENTRY statement. Arguments passed when the procedure is invoked are received as parameters in the invoked procedure.

The parameter attribute is a storage type attribute for variables. The parameter attribute conflicts with any other storage type. The parameter attribute also conflicts with the INITIAL attribute. A parameter can have any data type except format. The parameter attribute cannot be specified by keyword. Examples of the parameter attribute are shown in figure 4-63.

```
/*PARAMETER IN INTERNAL PROCEDURE      */

JKL;
PROCEDURE;
   DCL DD BIT(12);
   .
   .
   .
   CALL YG(DD);
YG;
   PROCEDURE(PRES_JV);
      DCL PRES_JV BIT(12); /*PARAMETER*/
      .
      .
      .
   END YG;
END JKL;

/*-----------------------------------*/
/*PARAMETER IN EXTERNAL PROCEDURE      */

TF;
PROCEDURE OPTIONS(MAIN);
   DCL EX ENTRY(CHAR(20),
                (3,3) DECIMAL FIXED);
   .
   .
   .
   CALL EX(STR,ARR);
END TF;

EX;
PROCEDURE(X,Y);
   DECLARE X CHAR(20);
   DECLARE Y(3,3) DECIMAL FIXED ALIGNED;
   .
   .
   .
END EX;

/*-----------------------------------*/
/*PARAMETER WITH ASTERISK EXTENT       */

H45;
PROCEDURE;
   DCL MAT(100,5) FLOAT DECIMAL;
   DCL GYF(100,2) FLOAT DECIMAL;
   .
   .
   .
   CALL N(MAT);

   .
   .
   .
   CALL N(GYF);
N;
   PROCEDURE(PASS);
      DCL PASS(100,*) FLOAT DECIMAL;
      .
      .
      .
   END N;
END H45;
```

Figure 4-63. Parameter Attribute Examples

Each parameter that appears in the parameter list of a PROCEDURE or ENTRY statement is explicitly declared by usage with the parameter attribute. Within the procedure, each parameter can be further declared by DECLARE statement with additional attributes that define the use of the parameter. When default attributes are applied, parameters are supplied with the variable and INTERNAL attributes. A parameter that appears in a parameter list but is not declared in any other manner receives standard arithmetic defaults. The parameter can instead receive INRULE arithmetic defaults.

PICTURE ATTRIBUTE

The PICTURE attribute is used for a pictured variable that can be a pictured character item or pictured numeric item. Picture-controlled Conversion is described in section 7, Data Manipulation.

The PICTURE attribute is a computational data type attribute. The PICTURE attribute conflicts with any noncomputational data type attribute. Other computational data type attributes can be associated with the pictured item but are not declared for the pictured item. The exception is the REAL attribute. PICTURE can be specified for a variable that is a scalar or an array. PICTURE conflicts with the structure attribute. The PICTURE attribute is specified by keyword. The syntax of the PICTURE attribute is shown in figure 4-64, and examples of PICTURE are shown in figure 4-65.

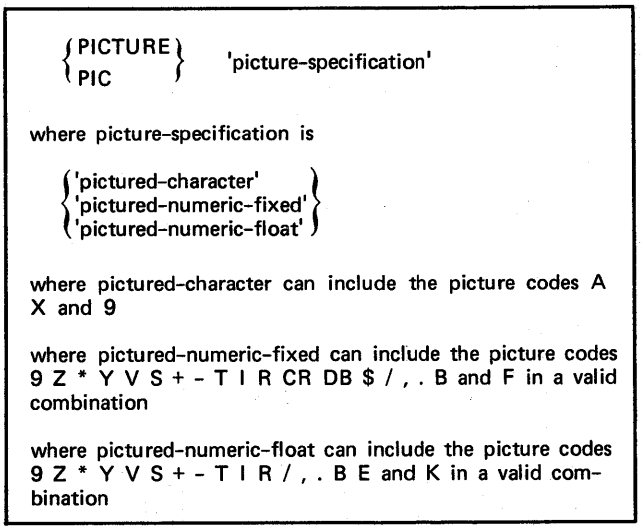


Figure 4-64. PICTURE Attribute Syntax

```
DCL RIGHT PICTURE↑9AXXXXX↑;
DCL RESULT PIC↑$ZZZZV.ZZCR↑;
DCL KPL PIC↑9V.9999ES999↑ REAL;
```

Figure 4-65. PICTURE Attribute Examples

If a PICTURE attribute that specifies pictured-character is declared for a variable or specified for a parameter or returns descriptor, no other data type attribute can be declared. The CHARACTER and nonvarying attributes are implied from the picture. If a PICTURE attribute that specifies pictured-numeric-fixed or pictured-numeric-float is declared for a variable or specified for a parameter or returns descriptor, no data type attribute except REAL can accompany the PICTURE attribute. The picture for a pictured numeric item implies REAL, DECIMAL, FIXED or FLOAT, and the precision. The UNALIGNED attribute is applied by default to a pictured item. A pictured item can be ALIGNED or UNALIGNED.

POINTER ATTRIBUTE

The POINTER attribute indicates a pointer value. A pointer value is a locator value used to specify the location of a generation.

The POINTER attribute is a noncomputational data type attribute for variables, parameter descriptors, and returns descriptors. A pointer variable can have any scope or storage type. The POINTER attribute conflicts with any other data type. POINTER can be specified for a variable that is a scalar or an array. POINTER conflicts with the structure attribute. The POINTER attribute is specified by keyword and can be abbreviated. The syntax of POINTER is shown in figure 4-66, and examples of POINTER are shown in figure 4-67.

Figure 4-66. POINTER Attribute Syntax

```
DCL P POINTER;
/*EXPLICIT DECLARATION OF P              */
DCL BRESULT BASED(P);

DCL DVALUE BASED(Q);
/*CONTEXTUAL DECLARATION OF Q            */
```

Figure 4-67. POINTER Attribute Examples

The POINTER attribute can be declared by DECLARE statement. Contextual declaration of the POINTER and variable attributes occurs for an identifier appearing in the SET option of an ALLOCATE statement, an identifier declared as the implicit locator in the BASED attribute, or an identifier used as the locator in a locator-qualified reference to a based variable. The default attributes INTERNAL, AUTOMATIC, and ALIGNED are applied to a pointer variable. The default attribute ALIGNED is applied to a parameter descriptor or returns descriptor that specifies POINTER.

PRECISION ATTRIBUTE

The precision attribute indicates the computational precision of an arithmetic value. The precision is the number of digits to be maintained for an arithmetic quantity. Precision of a fixed point value indicates the total number of maintained decimal or binary digits and the scale factor of the value. Precision of a floating point value indicates only the total number of significant decimal or binary digits.

The precision attribute is an arithmetic data type attribute used for variables, parameter descriptors, returns descriptors, and literal constants. Precision is implied by the picture specification for pictured numeric items. Precision

is compatible with the arithmetic attributes REAL, DECIMAL or BINARY, and FIXED or FLOAT. Precision cannot be specified by keyword. Precision is established by appearance of a precision suffix. The syntax of the precision suffix is shown in figure 4-68, and examples of precision are shown in figure 4-69.

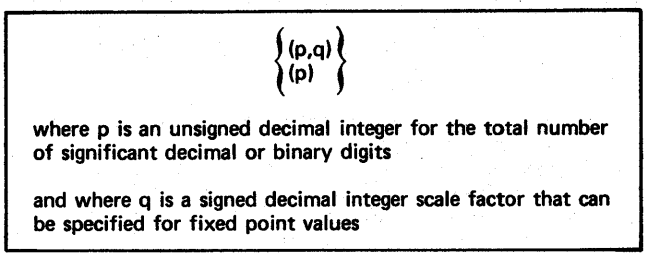


Figure 4-68. Precision Attribute Suffix Syntax

```
DCL TT FIXED DECIMAL(8,5);
DCL YY FLOAT DECIMAL(6);
DCL SDF BIN FIXED(12,3);
DCL KGH REAL(8,6);
```

Figure 4-69. Precision Attribute Examples

Any specification of precision must immediately follow one of the attributes REAL, DECIMAL, BINARY, FIXED, or FLOAT. Precision can be specified for an arithmetic variable that is a scalar or an array. Precision conflicts with the structure attribute. The specification (p) or (p,q) is used for FIXED arithmetic and (p) is used for FLOAT arithmetic. The p specification is an unsigned decimal integer, and the q specification (if used) can be an optionally signed decimal integer. The possible values of p and q are

- For FIXED DECIMAL: 0<p and p<=14, -255<=q and q<=255
- For FLOAT DECIMAL: 0<p and p<=14
- For FIXED BINARY: 0<p and p<=48, -255<=q and q<=255
- For FLOAT BINARY: 0<p and p<=48

When default arithmetic attributes are applied to an arithmetic quantity, the default precisions are:

- For FIXED DECIMAL: (5,0)
- For FLOAT DECIMAL: (14)
- For FIXED BINARY: (15,0)
- For FLOAT BINARY: (48)

PRINT ATTRIBUTE

The PRINT attribute indicates that an output file is intended for printing on a line printer. The PRINT attribute is used only for stream output files.

The PRINT attribute is a file description attribute for a file constant. PRINT cannot be specified for a file variable. PRINT conflicts with any file description attributes except STREAM, OUTPUT, and ENVIRONMENT. The PRINT attribute is specified by keyword. The syntax of PRINT is shown in figure 4-70, and examples of PRINT are shown in figure 4-71.

Figure 4-70. PRINT Attribute Syntax

```
DECLARE SFILE PRINT;
DECLARE GHT FILE STREAM OUTPUT PRINT;
```

Figure 4-71. PRINT Attribute Examples

Appearance of the PRINT attribute in a DECLARE statement declaration implies the FILE attribute. The PRINT attribute also implies the OUTPUT and STREAM attributes.

REAL ATTRIBUTE

The REAL attribute indicates that an arithmetic value has the mode REAL. The mode REAL indicates that the arithmetic value has no imaginary part.

The REAL attribute is an arithmetic data type attribute for variables, parameter descriptors, or returns descriptors. The REAL attribute can be used with other arithmetic data type attributes. REAL can also accompany the PICTURE attribute in the declaration of a pictured numeric item. REAL can be specified for a variable that is a scalar or an array. REAL conflicts with the structure attribute. A precision specification can follow the REAL attribute in the declaration of an arithmetic variable or descriptor. The REAL attribute is specified by keyword. The syntax of REAL is shown in figure 4-72, and examples of REAL are shown in figure 4-73.

Figure 4-72. REAL Attribute Syntax

```
DCL RES REAL FIXED BINARY;
DCL M REAL FLOAT DECIMAL(9);
```

Figure 4-73. REAL Attribute Examples

REAL is compatible with the scale FIXED or FLOAT, the base DECIMAL or BINARY, and a precision. During default processing, any implicitly declared variable and any descriptor with no declared attributes receives the REAL attribute and other arithmetic attributes by default.

RECORD ATTRIBUTE

The RECORD attribute indicates that a file is to be used for transmission of input or output records. RECORD indicates the use of record I/O operations for the file.

The RECORD attribute is a file description attribute used for a file constant. RECORD cannot be used for a file variable. The RECORD attribute conflicts with the STREAM and PRINT attributes that apply to stream I/O operations. The RECORD attribute is specified by keyword. The syntax of RECORD is shown in figure 4-74, and examples of RECORD are shown in figure 4-75.

```
RECORD
```

Figure 4-74. RECORD Attribute Syntax

```
DCL KK FILE RECORD;
DCL OFILE RECORD OUTPUT SEQL KEYED;
```

Figure 4-75. RECORD Attribute Examples

Appearance of the RECORD attribute in a DECLARE statement declaration implies the FILE attribute.

RETURNS ATTRIBUTE

The RETURNS attribute specifies that an entry value represents a function entry point. The RETURNS attribute additionally describes the single value that is returned from the function.

The RETURNS attribute is a noncomputational data type attribute for a variable, named constant, or parameter descriptor. RETURNS conflicts with any data type other than ENTRY. ENTRY and RETURNS are used together for an entry value that represents a function. RETURNS can be associated with an entry constant, an entry variable, or an entry parameter descriptor. The RETURNS attribute is specified by keyword. The syntax of RETURNS is shown in figure 4-76, and examples are shown in figure 4-77.

```
{RETURNS                          }
{RETURNS (returns-descriptor)     }
```

where returns-descriptor specifies any consistent combination of the attributes ALIGNED, UNALIGNED, AREA, POINTER, OFFSET, REAL, BINARY, DECIMAL, FIXED, FLOAT, precision, CHARACTER, BIT, VARYING, and PICTURE

Figure 4-76. RETURNS Attribute Syntax

```
DCL P8 ENTRY(BIT) RETURNS(FLOAT DEC);
/*EXTERNAL ENTRY HAS RETURNS ATTRIBUTE*/

DCL P10 ENTRY(CHAR(6)) RETURNS;
/*ENTRY HAS RETURNS ATTRIBUTE          */
/*STANDARD RETURNS(FIXED BINARY(15,0))*/
/*INRULE   RETURNS(FLOAT DEC(14))     */

PR6;
PROCEDURE RETURNS(CHAR(100));
/*PR6 HAS RETURNS ATTRIBUTE            */
/*SUPPLIED BY STATEMENT OPTION         */
```

Figure 4-77. RETURNS Attribute Examples

The RETURNS attribute can be declared for an entry constant. The entry prefix on a PROCEDURE or ENTRY statement is explicitly declared by usage as an internal or external entry constant. The RETURNS option on the PROCEDURE or ENTRY statement adds the RETURNS attribute to the entry constant. If no attributes are supplied for the returns descriptor, arithmetic defaults are applied. Since the value to be returned is associated with the entry constant, the name of the entry constant affects any INRULE arithmetic defaults applied to the returns descriptor.

The returns descriptor is completed with default attributes. The returns descriptor has no scope or storage type. The returns descriptor cannot be an aggregate and cannot have the data type ENTRY, RETURNS, FILE, format, or LABEL. Extents that can be used in a returns descriptor are described later in this section under Extents. When a value is returned from the function by RETURN statement execution, the value is converted as necessary to match the attributes of the returns descriptor before being returned from the function.

If an external procedure is to be invoked as a function from within a separate external procedure, the RETURNS attribute must be specified in the DECLARE statement declaration of the external entry constant. Rules for the specified returns descriptor are the same as for the RETURNS option used on the PROCEDURE or ENTRY statement. Since attributes of all external identifiers must exactly match, the attributes specified for the returns descriptor (and supplied by default) must exactly match the attributes of the returns descriptor in the external function.

The RETURNS attribute can also be declared for an entry parameter or an entry parameter descriptor. The RETURNS attribute identifies an entry parameter as representing a function entry point.

SEQUENTIAL ATTRIBUTE

The SEQUENTIAL attribute indicates that records in a file can be accessed sequentially. The SEQUENTIAL attribute applies only to record I/O operations, even though stream I/O operations effectively involve sequential access to stream data.

The SEQUENTIAL attribute is a file description attribute used for a file constant. SEQUENTIAL cannot be used for a file variable. SEQUENTIAL conflicts with the STREAM and PRINT attributes that are used for stream I/O rather than record I/O. The SEQUENTIAL attribute is specified by keyword and can be abbreviated. The syntax of SEQUENTIAL is shown in figure 4-78, and examples of SEQUENTIAL are shown in figure 4-79.

```
{SEQUENTIAL}
{SEQL      }
```

Figure 4-78. SEQUENTIAL Attribute Syntax

```
DCL NM SEQUENTIAL OUTPUT KEYED RECORD;
DECLARE ZZ FILE SEQL;
```

Figure 4-79. SEQUENTIAL Attribute Examples

Appearance of the SEQUENTIAL attribute in a DECLARE statement declaration implies the FILE attribute. The records in a SEQUENTIAL file can be created or retrieved in sequential order. The records in a SEQUENTIAL and KEYED file can be accessed sequentially or by record key. The SEQUENTIAL attribute is the default for a record file.

STATIC ATTRIBUTE

The STATIC attribute indicates that a variable has the static storage type, as described in section 3, Data Elements. Static storage is allocated only once, before program execution begins.

The STATIC attribute is a storage type attribute for variables. The STATIC attribute conflicts with any other storage type. STATIC can be used with INTERNAL or EXTERNAL. If a variable is declared with the STATIC attribute, the INTERNAL attribute is applied by default. STATIC can be specified for a variable of any aggregate type; STATIC cannot be specified for any individual array element or structure member. The STATIC attribute can be specified by keyword. The syntax of STATIC is shown in figure 4-80, and examples of STATIC are shown in figure 4-81.

```
STATIC
```

Figure 4-80. STATIC Attribute Syntax

```
DECLARE TMATRIX(40,30) STATIC FLOAT DEC;

DECLARE G CHAR(10) EXTERNAL;
/*DEFAULTS TO STATIC                    */
```

Figure 4-81. STATIC Attribute Examples

If a storage type is not declared and the EXTERNAL attribute is declared for a variable, the STATIC attribute is supplied by default. The STATIC attribute is usually not the default storage type. If scope and storage type are both unspecified, INTERNAL and AUTOMATIC are supplied by default.

STREAM ATTRIBUTE

The STREAM attribute indicates that a file is to be treated as a continuous stream of characters representing input data or output data. The STREAM attribute specifies that stream I/O operations are to be used for the file.

The STREAM attribute is a file description attribute used for a file constant. STREAM cannot be specified for a file variable. The STREAM attribute conflicts with the RECORD, UPDATE, SEQUENTIAL, DIRECT, and KEYED attributes that apply only to record I/O operations. The STREAM attribute is specified by keyword. The syntax of STREAM is shown in figure 4-82, and examples of STREAM are shown in figure 4-83.

```
STREAM
```

Figure 4-82. STREAM Attribute Syntax

Appearance of the STREAM attribute in a DECLARE statement declaration implies the FILE attribute. The STREAM attribute is the default for a file, unless the RECORD attribute is implied.

```
DCL OUT FILE STREAM OUTPUT;
DCL G STREAM;
```

Figure 4-83. STREAM Attribute Examples

STRUCTURE AND MEMBER ATTRIBUTES

The structure attribute is used for identifiers that are structures, and the member attribute is used for identifiers that are members of structures. Structures are described under Aggregates in section 3, Data Elements. The LIKE attribute can be used to expand one structure to the form of another. The REFER option for a based structure is used to preserve the value of an extent expression as the value of one of the members. Each structure is a hierarchical collection of variables. The structure has members. Each member can in turn be a substructure that is both a member of the structure and a structure with members. The members at the bottom level of the hierarchy are variables with data type attributes.

The structure and member attributes are aggregation attributes for variables or parameter descriptors. The structure and member attributes cannot be specified by keyword. The syntax of structures and members is shown in figure 4-84, and examples are shown in figure 4-85.

The structure name is declared as level 1, and members of the structure are declared in outline form with higher level numbers. The list of declared items is processed from left to right, with continual reference to the level number of the previous item. A higher level number indicates that the item is a member contained by the preceding item. The same level number indicates that the item is at the same hierarchical level as the previous item. A lower level number indicates that the item is not contained by the preceding item and that the item is immediately contained by the nearest preceding item with a lower level number. Items that contain one or more members are substructures within the level 1 structure.

The structure variable with level number 1 can be declared with a scope attribute and storage type attribute. Any structure can also be declared with dimension, alignment, and LIKE. Alignment can be declared but is propagated through the structure. After propagation of alignment, alignment is known for all members. A structure cannot be declared with any data type attributes. If structure is specified for a parameter descriptor, scope and storage type cannot be specified. Scope and storage type do not apply to descriptors. The substructures, if any, can be declared only with dimension and alignment.

Each member that is not a structure can be declared with dimension, alignment, and a valid combination of data type attributes. Storage type does not apply to members. Every member has the INTERNAL attribute. INITIAL can be used to assign an initial value to a member that is not a structure. INITIAL cannot be used for the member if any containing structure is dimensioned.

A structure must not occupy more than 120 000 words of memory.

For variables

1 structure-name { [attribute] . . . , { level member-name [attribute] . . . / level member-name LIKE structure-reference [attribute] . . . }, , , / LIKE structure-reference }

where each specified level is an unsigned decimal integer indicating the hierarchical level of the member

For parameter descriptors

1 [attribute] . . . , { level [attribute] . . . } , , ,

where structure and member names are not supplied

Figure 4-84. Structure, Member, and LIKE Attribute Syntax

```
DECLARE 1 A(3) UNALIGNED,
          2 B,
            3 C(5) CHAR(10),
            3 D(10,5) FLOAT BIN,
          2 E,
            3 F FIXED DECIMAL;

/*A IS AN ARRAY OF STRUCTURES          */
/*B IS A SUBSTRUCTURE, MEMBERS C AND D*/
/*E IS A SUBSTRUCTURE, MEMBER F        */
/*                                     */
/*DIMENSIONALITY OF A IS INHERITED     */
/*B IS EFFECTIVELY (3)                 */
/*C IS EFFECTIVELY (3,5)               */
/*D IS EFFECTIVELY (3,10,5)            */
/*E IS EFFECTIVELY (3)                 */
/*F IS EFFECTIVELY (3)                 */
/*                                     */
/*ALIGNMENT PROPAGATES FROM A          */
/*B AND E ARE UNALIGNED                */
/*C, D, AND F ARE UNALIGNED            */

DECLARE 1 M LIKE A;

/*M IS EXPANDED TO THE FORM OF A       */
/*DIMENSIONALITY OF A IS IGNORED       */
/*ALIGNMENT OF A IS IGNORED            */
/*M EXPANDS TO:                        */
/*          2 B                        */
/*            3 C(5) CHAR(10)          */
/*            3 D(10,5) FLOAT BIN      */
/*          2 E                        */
/*            3 F FIXED DECIMAL        */
/*NO DIMENSIONS ARE INHERITED          */
/*C IS UNALIGNED BY DEFAULT            */
/*D AND F ARE ALIGNED BY DEFAULT       */
```

Figure 4-85. Structure, Member, and LIKE Attribute Examples

LIKE Attribute

The LIKE attribute can be used to expand a structure to the form of another structure. The syntax of the LIKE attribute is included in figure 4-84. The LIKE attribute cannot be specified for a parameter descriptor. The structure-reference must be unsubscripted. The referenced structure or substructure cannot be declared with the LIKE attribute. In the structure expansion, any dimension or alignment attributes of the referenced structure are not used. All members of the referenced structure and all declared attributes of the members are copied. Level numbers are modified as necessary to preserve the hierarchical structure. The declaration with the LIKE attribute must not be immediately followed by an item declared with a higher level number.

Since expansion with LIKE is essentially a copy operation, any dimensionality inherited by the referenced structure is ignored. In any given block, the structures referenced in all LIKE attributes are determined before any expansion with LIKE is performed.

Completion of Structure Declarations

Three steps are involved in the completion of structure declarations. The expansion of any LIKE attributes is performed. Level numbers for members are then modified so that the level number of each member is one greater than the level number of the immediately containing structure. The last step in completing structure declarations is the propagation of alignment. During default processing, the declarations of the structure and all members are completed with default attributes.

REFER Option for BASED Structures

The REFER option is used to preserve the value of an extent expression as the value of a member of the structure. The structure must be declared with the BASED attribute. The REFER option is used in an extent in the declaration of a member of the based structure. An example of the REFER option is shown in figure 4-86.

```
DECLARE J AUTOMATIC FIXED DEC INIT(10);
DECLARE 1 GSTRUC BASED,
          2 KEEP FIXED DEC,
          2 VAL CHAR(5*J REFER (KEEP));
/*WHEN GSTRUC IS ALLOCATED, KEEP IS   */
/*SET TO 50. SUBSEQUENT REFERENCE     */
/*TO VAL USES THE VALUE OF KEEP RATHER*/
/*THAN REEVALUATING 5*J               */
```

Figure 4-86. REFER Option Example

The REFER option is specified in an extent for area size, character string length, bit string length, or array bounds of a dimension. Extents are described later in this section under Extents. The extent expression cannot contain

references to any member of the same structure or to the structure itself. The reference in the REFER option must identify a scalar and computational member that is declared earlier in the structure. The extent expression must be scalar and computational.

When the based structure is allocated, all extents are evaluated to determine the size of the based generation. After the generation is allocated, each structure member named in a REFER option is assigned the value of the extent with which it is associated. Therefore, an allocated generation for a based structure has some initial values that define extents for members appearing later in the structure. The REFER option is effective for each allocation of the based structure. When the based structure is referenced or freed, the extent expression is not evaluated. The value of the extent is taken from the member designated by the REFER option.

UPDATE ATTRIBUTE

The UPDATE attribute indicates that a file is to be used either for input or for output. An UPDATE file can only be used for record I/O operations.

The UPDATE attribute is a file description attribute used for a file constant. UPDATE cannot be specified for a file variable. The UPDATE attribute conflicts with INPUT, OUTPUT, STREAM, and PRINT. OUTPUT and INPUT are alternatives for the use of the file. STREAM and PRINT are used for stream I/O rather than record I/O. The UPDATE attribute is specified by keyword. The syntax of UPDATE is shown in figure 4-87, and examples of UPDATE are shown in figure 4-88.

Figure 4-87. UPDATE Attribute Syntax

```
DECLARE YH FILE RECORD SEQL UPDATE;
DECLARE FILE4 UPDATE;
```

Figure 4-88. UPDATE Attribute Examples

Appearance of the UPDATE attribute in a DECLARE statement declaration implies the FILE attribute. The UPDATE attribute also implies the RECORD attribute.

VARIABLE ATTRIBUTE

The variable attribute applies to identifiers used as variables. A variable can have different values during program execution.

The variable attribute indicates that an identifier is the name of a variable. The variable attribute cannot be specified by keyword. Examples of the variable attribute are shown in figure 4-89. The variable attribute can be used with any storage type or scope. Variable conflicts with the constant, BUILTIN, and condition attributes. When defaults are applied to identifiers, most identifiers are supplied with the variable attribute by default. Except if the FILE or ENTRY attribute is present for an identifier that is not a parameter, the variable attribute is applied by default.

```
DECLARE RHVAL FLOAT BINARY;
/*RHVAL IS A VARIABLE                    */
DECLARE D CHAR(5);
/*D IS A VARIABLE                        */
DECLARE AREA2 AREA;
/*AREA2 IS A VARIABLE                    */
DECLARE JN LABEL;
/*JN IS A VARIABLE                       */
```

Figure 4-89. Variable Attribute Examples

VARYING AND NONVARYING ATTRIBUTES

The VARYING attribute and the nonvarying attribute indicate whether the current length of a character string or bit string can ever be shorter than the maximum specified length. A nonvarying string always has the specified length. A VARYING string has a current length and the available maximum length.

The VARYING and nonvarying attributes are string data type attributes for variables, parameter descriptors, and returns descriptors. The VARYING attribute can be specified for a variable that is a scalar or an array. VARYING and nonvarying conflict with the structure attribute. The VARYING attribute is specified by keyword. The syntax of VARYING is shown in figure 4-90. The nonvarying attribute cannot be specified by keyword. Examples of VARYING and nonvarying are shown in figure 4-91.

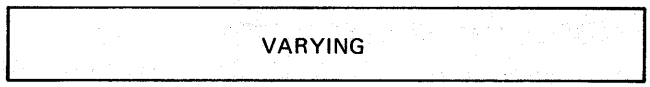


Figure 4-90. VARYING Attribute Syntax

```
DECLARE TSTRING CHAR(65);
/*TSTRING IS NONVARYING BY DEFAULT     */

DCL DSTRING CHAR(105) VARYING;
/*MAXIMUM LENGTH OF DSTRING IS 105     */
/*CURRENT LENGTH OF DSTRING IS NOT SET*/

DCL BITL BIT(100) VARYING INIT('11'B);
/*MAXIMUM LENGTH OF BITL IS 100        */
/*CURRENT LENGTH OF BITL IS 2          */
```

Figure 4-91. VARYING and Nonvarying Attribute Examples

A string variable or descriptor not declared as VARYING is known to be nonvarying. Strings involved in string overlay defining must be nonvarying. All character constants and bit constants are nonvarying. A string declared as VARYING has a string value with a current length established by the most recent assignment.

EXTENTS

An extent is a specification of one of the following values:

- Size in the AREA attribute
- Length in the BIT attribute

- Length in the CHARACTER attribute
- Array bounds for each dimension in the dimension attribute

An extent can be specified in a DECLARE statement, or in a RETURNS option in a PROCEDURE or ENTRY statement. An area size, bit string length, or character string length can be specified as an extent expression or an asterisk extent. The array bounds for a dimension of an array can be specified as an extent expression for the upper bound, a pair of extent expressions separated by a colon for the lower bound and the upper bound, or an asterisk extent.

The appearance of an asterisk as an extent indicates that any area size, bit string length, character string length, or array bounds are acceptable. The actual extent is determined each time a generation is allocated for the variable or associated with the descriptor. For example, a parameter declared as BIT(*) can represent a bit string of a different length each time the procedure is invoked. The length is dependent on the length of the corresponding argument passed to the procedure.

The storage type of the variable, or the type of descriptor, determines when the extent expressions are evaluated, as shown in table 4-1.

The legal usage of extent expressions and asterisk extents depends on the storage type of the variable, or depends on the type of descriptor, as shown in table 4-2. An expression used as an extent must be scalar and computational. Extent expressions are evaluated and converted to fixed binary integer. An evaluated extent can be negative only if the extent expression specifies a lower bound or upper bound of a dimension.

SUMMARY OF ATTRIBUTES

A summary of attributes is provided in table 4-3. The attributes are listed in alphabetic order. A similar table in section 5, Declarations, is intended as a summary of attribute defaults and implications.

TABLE 4-1. EVALUATION OF EXTENTS

Storage type or type of descriptor	Time of evaluation of extent
STATIC	Before program execution begins, when storage is allocated for all static variables.
AUTOMATIC	When storage is allocated for the variable, during activation of the block immediately containing the declaration.
CONTROLLED	When storage is allocated for the variable by execution of an ALLOCATE statement.
BASED	When the variable is allocated, and each time the variable is referenced.
DEFINED	During activation of the block immediately containing the declaration. Note that the defined variable is not associated with a generation at this time.
Parameter	During activation of the block immediately containing the declaration, when the parameter is associated with the generation belonging to an argument passed to the procedure.
Parameter descriptor	During evaluation of the argument list in a procedure reference.
Returns descriptor	During activation of the procedure.

TABLE 4-2. EXTENTS

Storage type or type of descriptor	Legal form of extent expression or asterisk extent	Storage type or type of descriptor	Legal form of extent expression or asterisk extent
STATIC	Optionally signed decimal integer	BASED, for structure member without REFER option	Expression When the based structure is allocated, referenced, or freed, the expression is evaluated and used as the extent. The expression can reference a scalar member of the same based structure. The referenced member must be declared earlier in the structure. If the extent references an earlier member, the declaration cannot be used to allocate the based structure, and the referenced member must contain the correct extent.
AUTOMATIC	Expression The expression can reference other automatic and defined variables declared in the same block. The evaluation of the extent of a variable must not require a reference to the value of the variable itself.		
CONTROLLED	Expression The extent of a controlled variable cannot reference the controlled variable itself.		
BASED	Expression The extent of a based variable cannot reference the based variable itself, except as described for structure member with or without REFER option.	DEFINED	Expression The expression can reference other automatic or defined variables declared in the same block. The evaluation of the extent of a variable must not require a reference to the variable itself.
BASED, for structure member with REFER option	Expression REFER (refer-variable) When the based structure is allocated, the expression is evaluated and used to initialize the refer-variable. When the BASED structure is referenced or freed, the value of the refer-variable is used. The refer-variable must be a scalar member of the same based structure and must be declared earlier in the structure. The user must not assign values to the refer-variable.	Parameter	Asterisk or optionally signed decimal integer
		Parameter descriptor	Asterisk or optionally signed decimal integer
		Returns descriptor	Optionally signed decimal integer Array bounds are not pemitted in a returns descriptor. Asterisk extents are not permitted in a returns descriptor.

TABLE 4-3. SUMMARY OF ATTRIBUTES

Attribute:	Belongs to the following category of attributes:	Can be in the completed attribute set for:	And conflicts with the following attributes:
ALIGNED	Alignment	variable parameter descriptor returns descriptor	UNALIGNED
AREA(size)	Noncomputational data type	variable parameter descriptor returns descriptor	any other data type structure
AUTOMATIC or AUTO	Storage type	variable	any other storage type EXTERNAL member
BASED	Storage type	variable	any other storage type EXTERNAL member

TABLE 4-3. SUMMARY OF ATTRIBUTES (Cont'd)

Attribute:	Belongs to the following category of attributes:	Can be in the completed attribute set for:	And conflicts with the following attributes:
BINARY or BIN	Arithmetic data type	variable parameter descriptor returns descriptor literal constant	any data type other than arithmetic DECIMAL structure
BIT(length)	String data type	variable parameter descriptor returns descriptor literal constant	any data type other than string CHARACTER structure
BUILTIN	None	identifier used as builtin function name	condition constant variable EXTERNAL
CHARACTER(length) or CHAR(length)	String data type	variable parameter descriptor returns descriptor literal constant	any data type other than string BIT structure
condition	None	identifier used in programmer-named condition	BUILTIN constant variable INTERNAL
constant	None	identifier used as named constant literal constant	BUILTIN condition variable
CONTROLLED or CTL	Storage type	variable	any other storage type member
DECIMAL or DEC	Arithmetic data type	variable parameter descriptor returns descriptor literal constant	any data type other than arithmetic BINARY structure
DEFINED or DEF	Storage type	variable	any other storage type EXTERNAL member INITIAL VARYING
dimension	Aggregation	variable label constant parameter descriptor	ENTRY format FILE
DIRECT	File description	file constant	STREAM PRINT SEQUENTIAL

TABLE 4-3. SUMMARY OF ATTRIBUTES (Cont'd)

Attribute:	Belongs to the following category of attributes:	Can be in the completed attribute set for:	And conflicts with the following attributes:
ENTRY	Noncomputational data type	variable named constant parameter descriptor	any data type other than RETURNS dimension structure any storage type other than parameter member
ENVIRONMENT or ENV	File description	file constant	None
EXTERNAL or EXT	Scope	variable named constant identifier used in programmer-named condition	BUILTIN INTERNAL LABEL format member AUTOMATIC BASED DEFINED parameter
FILE	Noncomputational data type	variable named constant parameter descriptor	any other data type any storage type other than parameter dimension structure member
FIXED	Arithmetic data type	variable parameter descriptor returns descriptor literal constant	any data type other than arithmetic FLOAT structure
FLOAT	Arithmetic data type	variable parameter descriptor returns descriptor literal constant	any data type other than arithmetic FIXED precision (p,q) structure
format	Noncomputational data type	named constant	any other data type dimension
INITIAL	Initialization	variable	parameter DEFINED structure ENTRY FILE format LABEL with STATIC

TABLE 4-3. SUMMARY OF ATTRIBUTES (Cont'd)

Attribute:	Belongs to the following category of attributes:	Can be in the completed attribute set for:	And conflicts with the following attributes:
INPUT	File description	file constant	OUTPUT PRINT UPDATE
INTERNAL or INT	Scope	variable named constant builtin function	condition EXTERNAL
KEYED	File description	file constant	STREAM PRINT
LABEL	Noncomputational data type	variable named constant parameter descriptor	any other data type STATIC with INITIAL structure
LIKE	None	None	any data type INITIAL
member	Aggregation	variable parameter descriptor	EXTERNAL any storage type
nonvarying	String data type	variable parameter descriptor returns descriptor literal constant	any data type other than string VARYING structure
OFFSET	Noncomputational data type	variable parameter descriptor returns descriptor	any other data type structure
OUTPUT	File description	file constant	INPUT UPDATE
parameter	Storage type	variable	any other storage type member INITIAL
PICTURE or PIC	Pictured data type	variable parameter descriptor returns descriptor	any other data type structure
POINTER or PTR	Noncomputational data type	variable parameter descriptor returns descriptor	any other data type structure
POSITION(start-pos) or POS(start-pos)	None	variable (for string overlay defining only)	any storage type except DEFINED any data type other than string and pictured EXTERNAL ALIGNED member VARYING INITIAL

TABLE 4-3. SUMMARY OF ATTRIBUTES (Cont'd)

Attribute:	Belongs to the following category of attributes:	Can be in the completed attribute set for:	And conflicts with the following attributes:
precision	Arithmetic data type	variable parameter descriptor returns descriptor literal constant	any data type other than arithmetic structure
PRINT	File description	file constant	INPUT UPDATE RECORD SEQUENTIAL DIRECT KEYED
REAL	Arithmetic data type	variable parameter descriptor returns descriptor literal constant	any data type other than arithmetic structure
RECORD	File description	file constant	STREAM PRINT
RETURNS	Noncomputational data type	variable named constant parameter descriptor	any data type other than ENTRY structure member
SEQUENTIAL or SEQL	File description	file constant	STREAM PRINT DIRECT
STATIC	Storage type	variable	any other storage type member LABEL with INITIAL
STREAM	File description	file constant	RECORD UPDATE SEQUENTIAL DIRECT KEYED
structure	Aggregation	variable parameter descriptor	any data type INITIAL
UNALIGNED	Alignment	variable parameter descriptor returns descriptor	ALIGNED
UPDATE	File description	file constant	INPUT OUTPUT STREAM PRINT

TABLE 4-3. SUMMARY OF ATTRIBUTES (Cont'd)

Attribute:	Belongs to the following category of attributes:	Can be in the completed attribute set for:	And conflicts with the following attributes:
variable	None	identifier used as variable	BUILTIN condition constant
VARYING	String data type	variable parameter descriptor returns descriptor	any data type other than string nonvarying DEFINED structure

Declarations of identifiers used in a PL/I program are accumulated and completed during program compilation. The completed declarations are used during program execution to control the use of variables, named constants, builtin functions, and programmer-named conditions. A specific identifier can be used in different ways in different blocks. Each identifier used in a PL/I program has at least one declaration.

This section describes the ways in which identifiers are declared, either by DECLARE statement or by usage of the name. The operation of explicit, contextual, and implicit declarations is described, as well as the application of default attributes. Declaration of an identifier usually involves declaring attributes for the identifier. Default attributes are added as necessary to complete the description of the declared identifier.

Parameter descriptors and returns descriptors can also contain attributes. Descriptors are completed with default attributes in the same way as declarations are completed.

SCOPE OF DECLARATIONS

The scope of a declaration specifies the parts of the program within which the declared identifier is known. The scope of an identifier is determined by its position within the source program and by the INTERNAL or EXTERNAL attribute.

An identifier with the INTERNAL attribute is known in the block that immediately contains the declaration of the identifier. In addition, the identifier is known in all contained blocks except those in which another declaration for the same name is effective. When another declaration of the same name exists in a contained block, the name is redeclared in that block (and can be INTERNAL or EXTERNAL). Each declaration of an identifier as INTERNAL constitutes a separate declaration for the identifier. If an identifier is declared as INTERNAL and declared again in a contained block, two separate identifiers are declared for use in different blocks.

An identifier with the EXTERNAL attribute is known in contained blocks in the same way as an identifier declared INTERNAL. In addition, an identifier declared EXTERNAL is known outside the external procedure. All declarations of an identifier as EXTERNAL are merged when the program is loaded for execution. Therefore, all declarations of an identifier as EXTERNAL must declare the identifier with the same attributes. For the few minor exceptions to this rule, see the EXTERNAL attribute in section 4, Attributes. If an identifier is declared as EXTERNAL, the same identifier can be declared in a contained block as INTERNAL and with different attributes.

Declaration of an identifier as EXTERNAL sets up communication between external procedures. A name declared as EXTERNAL is recognized as the same in all external procedures that declare the name as EXTERNAL. Effectively, external entry constants, external file constants, and external programmer-named conditions can be known in several external procedures. External variables can be known in other external procedures without being passed as arguments. An example of the use of INTERNAL and EXTERNAL is shown in figure 5-1.

```
PR:
PROCEDURE OPTIONS(MAIN);
   DCL A INTERNAL FIXED BINARY;
   /* DECLARED A IS KNOWN IN PR.          */
   DCL B INTERNAL FIXED BINARY;
   /* DECLARED B IS KNOWN IN PR AND       */
   /* KNOWN IN B1 AND B2.                 */
   DCL C EXTERNAL CHAR(15);
   /* DECLARED C IS KNOWN IN PR AND       */
   /* MERGED WITH DECLARED C IN B2.       */
   .
   .
   .
B1:
   BEGIN;
      DCL A EXTERNAL POINTER;
      /* DECLARED A IS KNOWN IN B1        */
      /* AND KNOWN IN B2.                 */
      DCL C INTERNAL AREA;
      /* DECLARED C IS KNOWN IN B1.       */
      .
      .
      .
B2:
      BEGIN;
         DCL C EXTERNAL CHAR(15);
         /* DECLARED C IS KNOWN IN B2     */
         /* AND MERGED WITH C IN PR.      */
         .
         .
         .
      END B2;
   END B1;
END PR;
```

Figure 5-1. Scope of Declared Identifiers

DECLARATION PROCESSING

The PL/I compiler accumulates declarations during compilation. The order of declaration processing is as follows:

1. Explicit declarations are recognized and established for statement prefixes, parameters in parameter lists, and DECLARE statement declarations.

2. Contextual declarations are established for all identifiers that are used in certain contexts but are not declared explicitly.

3. Implicit declarations are established for all identifiers that are used but are not declared explicitly or contextually.

After all explicit, contextual, and implicit declarations have been established, the compiler completes the declaration for each identifier with default attributes.

If attributes are incorrectly supplied in an explicit DECLARE statement declaration, the compiler produces diagnostic messages during compilation. Default attributes are selectively added to each declaration in such a manner that no conflict results from completion of the declaration.

SYSIN ASSUMPTIONS

Before the processing of declarations begins, the compiler makes the following additions to the source text of the program:

- For each GET statement with no FILE option or STRING option, the compiler supplies a FILE option of the form

  FILE(SYSIN)

SYSPRINT ASSUMPTIONS

Before the processing of declarations begins, the compiler makes the following additions to the source text of the program:

- For each PUT statement with no FILE option or STRING option, the compiler supplies a FILE option of the form

  FILE(SYSPRINT)

- For each COPY option with no file reference, the compiler substitutes a COPY option of the form

  COPY(SYSPRINT)

The compiler also generates an explicit declaration in each external procedure for SYSPRINT with the attributes EXTERNAL FILE STREAM OUTPUT PRINT.

EXPLICIT DECLARATIONS

Explicit declarations are established for entry prefixes, label prefixes, format prefixes, parameters in parameter lists, and DECLARE statement declarations.

For each identifier, only one explicit declaration can be made in any given block. The compiler issues diagnostics for multiple explicit declarations of the same identifier, except in the following cases:

- A member name can be identical to another identifier declared in the same block, but any member name must be unique within the structure that immediately contains the member. The fully structure-qualified member name must be unique within the block.

- An identifier can be explicitly declared by appearance as a parameter in the parameter list of a PROCEDURE or ENTRY statement and can also be explicitly declared in a DECLARE statement. Any given variable can also appear as a parameter at several entry points in the same procedure.

- An array of label constants is explicitly declared in a DECLARE statement and also explicitly declared by the appearance of one or more of the array elements as statement prefixes.

DECLARATION OF STATEMENT PREFIXES

Explicit declaration occurs for identifiers contained in entry prefixes, label prefixes, and format prefixes. The explicit declarations by usage are as follows:

- Each entry prefix on a PROCEDURE or ENTRY statement is explicitly declared with the constant and ENTRY attributes. In addition, a scope attribute is supplied explicitly. If the PROCEDURE or ENTRY statement belongs to an external procedure, the EXTERNAL attribute is supplied and the declaration is not contained in any block. If the procedure is an internal procedure, the INTERNAL attribute is supplied and the declaration is made in the block that immediately contains the procedure. If the PROCEDURE or ENTRY statement has the RETURNS option, the RETURNS attribute and the specified returns descriptor are added to the declaration.

- Each label prefix is explicitly declared with the constant, LABEL, and INTERNAL attributes. Any label prefix attached to a BEGIN statement is declared in the block that immediately contains the BEGIN block. Each label prefix of a statement other than BEGIN is declared in the block in which the statement occurs.

- Each format prefix on a FORMAT statement is explicitly declared with the constant, format, and INTERNAL attributes. Each format prefix is declared in the block in which the FORMAT statement occurs.

Default attributes are added after all explicit, contextual, and implicit declarations have been processed.

DECLARATION OF PARAMETERS

Explicit declaration occurs for identifiers contained in the parameter list of a PROCEDURE or ENTRY statement. The declaration applies the parameter attribute to each identifier in the parameter list. Other attributes for each identifier can be supplied by an explicit DECLARE statement declaration of the identifier in the same block. Default attributes are added after all explicit, contextual, and implicit declarations have been processed.

DECLARE STATEMENT DECLARATIONS

Explicit declarations for identifiers can be made with the DECLARE statement. Each declaration made by DECLARE statement is established in the block that immediately contains the DECLARE statement.

A declaration is established for each identifier in the DECLARE statement. Each declaration has an attribute set consisting of the explicitly declared attributes. Default attributes are added after all explicit, contextual, and implicit declarations have been processed.

NOTE

Duplicate attribute keywords for the same identifier must not be supplied in the DECLARE statement. The compiler diagnoses duplicate attribute keywords as errors.

A DECLARE statement can create declarations for more than one identifier. When a factored declaration is used to apply the same attribute to several identifiers, the attribute is applied to each identifier.

If an identifier is declared with the dimension attribute, a declaration is established for an array of identical data items. The dimensioned identifier can be a structure, in which case the identical elements of the array are structures and the members inherit the dimensions of the structure. If the members of the structure are also dimensioned, the members are established with the declared dimensions and with the inherited dimensions, as described under Aggregates in section 3, Data Elements.

For a declared structure, a declaration is established for the structure and for each member. The completion of structure declarations is performed in the following steps:

1. The expansion of LIKE attributes is performed if the structure is declared with the LIKE attribute, as described under LIKE attribute in section 4, Attributes.
2. The level numbers declared for members are modified as necessary so that the level number of each member is one greater than the level number of the immediately containing structure, as described under Structure and member attributes in section 4.
3. Alignment attributes are propagated through the structure, as described under ALIGNED attribute in section 4.

If an identifier is declared as BUILTIN in a DECLARE statement, the declared identifier must match the name of a supplied builtin function. Explicit declaration of an identifier as BUILTIN is usually not necessary, as described under BUILTIN attribute in section 4.

CONTEXTUAL DECLARATIONS

The appearance of an identifier in a specific context is often sufficient to establish the intended use of the identifier. A contextual declaration is not made if an explicit declaration exists for the identifier in the same block or in a containing block. If no explicit declaration exists, a contextual declaration with certain attributes is established in the following cases:

- AREA and variable attributes are supplied for an identifier referenced as an area in an OFFSET attribute or the IN option of an ALLOCATE or FREE statement.
- BUILTIN attribute is supplied for an identifier that matches the name of a builtin function if the identifier is referenced and is followed by a parenthesized argument list.
- Condition attribute is supplied for an identifier referenced in a programmer-named condition in an ON, SIGNAL, or REVERT statement.
- FILE and constant attributes are supplied for an identifier referenced as a file in a FILE or COPY option of an I/O statement and for an identifier referenced as a file in an I/O condition name in an ON, SIGNAL, or REVERT statement.
- POINTER and variable attributes are supplied for an identifier referenced as a locator in the SET option of an ALLOCATE, READ, or LOCATE statement, for an identifier used as a locator in a locator-qualified reference, and for an identifier referenced as a locator in a BASED attribute.

When a contextual declaration is processed, the declaration is established in the outermost block of the external procedure. The contextual declaration therefore is known in all blocks except any internal blocks in which an explicit declaration of the same identifier is effective. The scope of the identifier can default to INTERNAL or EXTERNAL. Default attributes are added after all explicit, contextual, and implicit declarations are processed.

IMPLICIT DECLARATIONS

An implicit declaration is established for each identifier not declared explicitly or contextually. An implicit declaration is a declaration with no attributes. Each implicit declaration is established in the outermost block of the external procedure. An implicit declaration is not made if an explicit or contextual declaration exists for the identifier in the block or in a containing block. Any implicit declaration operates as if the statement:

```
DECLARE identifier;
```

had been written in the external procedure. Default attributes are added after all explicit, contextual, and implicit declarations are processed. An implicit declaration sets up arithmetic defaults when the default attributes are supplied.

DEFAULT ATTRIBUTES

Default attributes are added as necessary to each explicit, contextual, and implicit declaration. The compiler adds default attributes to complete the attribute set for each declaration.

After defaults have been applied, the attribute set for each identifier is a complete and consistent set of attributes, with one exception. The set of file description attributes for each file constant is completed at open time for the file. The rules for default file description attributes are described in section 8, Input/Output.

The application of default attributes occurs in a specific order. Each step of the sequence can add attributes to the attribute set. In each step, the decision to add default attributes is based on the existing attributes in the attribute set for the identifier. The steps in default attribute processing are as follows:

1. ENTRY: If RETURNS is present and if ENTRY is not present, ENTRY is added. Only the keyword ENTRY is added; parameter descriptors for ENTRY are not created.
2. FILE: If one or more of the file description attributes STREAM, RECORD, INPUT, OUTPUT, UPDATE, SEQUENTIAL, DIRECT, KEYED, or PRINT is present and if FILE is not present, FILE is added. Presence of the ENVIRONMENT file description attribute by itself does not add FILE.
3. Arithmetic: Arithmetic defaults are added for each identifier or descriptor not already known to be nonarithmetic. Arithmetic defaults are not added if the ENTRY, LABEL, FILE, format, AREA, POINTER, OFFSET, BIT, CHARACTER, PICTURE, structure, BUILTIN, condition, or constant attribute is present. Arithmetic defaults are either the standard defaults or the INRULE defaults.

   The standard (ANSI standard) arithmetic defaults are added in the following steps:

   If no base is present, BINARY is added.

   If no scale is present, FIXED is added. Precision of the form (p,q) also establishes the FIXED attribute.

   If no mode is present, REAL is added.

   If INRULE processing is specified with the INRULE parameter of the PLI control statement, then INRULE arithmetic defaults are added in the following steps:

   If the identifier begins with one of the letters I through N, and if the base, scale, and mode are all absent, then FIXED and BINARY are added.

If a precision of the form (p,q) is present, FIXED is added.

If no base is present, DECIMAL is added.

If no scale is present, FLOAT is added.

If no mode is present, REAL is added.

Samples of default arithmetic attributes for ANSI standard and for INRULE are provided in table 5-1.

4. Precision: If the identifier is arithmetic, the default precision is supplied in the following way:

   For FIXED BINARY, if only p is present, q is supplied as 0. If both are unspecified, precision is (15,0).

   For FIXED DECIMAL, if only p is present, q is supplied as 0. If both are unspecified, precision is (5,0).

   For FLOAT BINARY, precision is (48).

   For FLOAT DECIMAL, precision is (14).

5. Length, size, or position: If CHARACTER, BIT, AREA, or POSITION is present without a parenthesized length, size, or position, the default is:

   CHARACTER(1)

   BIT(1)

   AREA(150)

   POSITION(1)

6. Nonvarying: If CHARACTER or BIT is present and VARYING is not present, then nonvarying is added.

7. Variable or constant: If variable is not present, constant is not present, BUILTIN is not present, condition is not present, and if the declared item is not a descriptor, then variable or constant is added according to the following rules:

   Constant is added if ENTRY or FILE is present and parameter is not present.

   Variable is added in all other cases.

8. INTERNAL or EXTERNAL: If INTERNAL is not present, EXTERNAL is not present, and the declared item is not a descriptor, then a scope attribute is added according to the following rules:

   EXTERNAL is added if condition is present.

   EXTERNAL is added if FILE and constant are present.

   EXTERNAL is added if ENTRY and constant are present.

   INTERNAL is added in all other cases.

9. Storage type: If variable is present, member is not present, and no storage type (AUTOMATIC, BASED, CONTROLLED, STATIC, DEFINED or parameter) is present, then AUTOMATIC or STATIC is added according to the following rules:

   If EXTERNAL is present, STATIC is added.

   If INTERNAL is present, AUTOMATIC is added.

10. Alignment: If ALIGNED is not present, UNALIGNED is not present, and the declared item is a variable or a descriptor, an alignment attribute is added according to the following rules:

    If CHARACTER, BIT, or PICTURE is present, UNALIGNED is added.

    ALIGNED is added in all other cases.

SUMMARY OF DEFAULTS

A summary of default attributes is provided in table 5-2. The full list of attributes is included in the summary, even though some attributes are not involved in default processing (for example, dimension and ENVIRONMENT). By convention, attributes that can be recognized by keyword are shown in upper case.

The default file description attributes are covered in this summary. File description defaults are applied when the file constant is opened and removed when the file is closed, as described in section 8, Input/Output.

TABLE 5-1. DEFAULTS FOR PARTIALLY DECLARED ARITHMETIC ITEMS

Arithmetic Attribute Declared†	Standard Arithmetic Default	INRULE Default for I-N	INRULE Default for all except I-N
--	REAL FIXED BINARY (15,0)	REAL FIXED BINARY (15,0)	REAL FLOAT DECIMAL (14)
REAL	REAL FIXED BINARY (15,0)	REAL FLOAT DECIMAL (14)	REAL FLOAT DECIMAL (14)
FIXED	REAL FIXED BINARY (15,0)	REAL FIXED DECIMAL (5,0)	REAL FIXED DECIMAL (5,0)
FLOAT	REAL FLOAT BINARY (48)	REAL FLOAT DECIMAL (14)	REAL FLOAT DECIMAL (14)
BINARY	REAL FIXED BINARY (15,0)	REAL FLOAT BINARY (48)	REAL FLOAT BINARY (48)
DECIMAL	REAL FIXED DECIMAL (5,0)	REAL FLOAT DECIMAL (14)	REAL FLOAT DECIMAL (14)
REAL (p)	REAL FIXED BINARY (p,0)	REAL FLOAT DECIMAL (p)	REAL FLOAT DECIMAL (p)
REAL (p,q)	REAL FIXED BINARY (p,q)	REAL FIXED DECIMAL (p,q)	REAL FIXED DECIMAL (p,q)

† More complete combinations are not shown.

TABLE 5-2. SUMMARY OF DEFAULT ATTRIBUTES FOR DECLARATIONS

An identifier with this attribute:	Will necessarily acquire these attributes:	And will acquire these attributes unless contradicted:
ALIGNED	variable	None
AREA with or without (size)	variable	ALIGNED † size = 150 words †
AUTOMATIC or AUTO	variable INTERNAL	None
BASED	variable INTERNAL	None
BINARY or BIN	variable REAL †	scale FIXED or FLOAT † precision †
BIT with or without (length)	variable	nonvarying † UNALIGNED † length = 1 bit †
BUILTIN	INTERNAL	None
CHARACTER or CHAR with or without (length)	variable	nonvarying † UNALIGNED † length = 1 character †
condition	EXTERNAL	None
constant	None	None
CONTROLLED or CTL	variable	INTERNAL
DECIMAL or DEC	variable REAL †	scale FIXED or FLOAT † precision †
DEFINED or DEF	variable INTERNAL	None
dimension	None	None
DIRECT	FILE constant RECORD (open time) KEYED (open time)	INPUT (open time)
ENTRY	variable (if parameter) constant (if not parameter)	ALIGNED (if parameter) †† EXTERNAL (if constant)
ENVIRONMENT or ENV	None	None
EXTERNAL or EXT	None	STATIC (if variable)
FILE	variable (if parameter) constant (if not parameter)	ALIGNED (if parameter) †† EXTERNAL (if constant) STREAM (open time) INPUT (open time)
FIXED	variable REAL †	base BINARY or DECIMAL † precision †
FLOAT	variable REAL †	base BINARY or DECIMAL † precision †
format	constant INTERNAL	None
INITIAL	None	None
INPUT	FILE constant	STREAM (open time)
INTERNAL or INT	None	AUTOMATIC (if variable)
KEYED	FILE constant RECORD (open time)	SEQUENTIAL (open time) INPUT (open time)

TABLE 5-2. SUMMARY OF DEFAULT ATTRIBUTES FOR DECLARATIONS (Cont'd)

An identifier with this attribute:	Will necessarily acquire these attributes:	And will acquire these attributes unless contradicted:
LABEL	INTERNAL (if constant)	variable ALIGNED (if variable)†† INTERNAL (if variable)
LIKE	None	None
member	None	None
nonvarying	None	None
OFFSET	variable	ALIGNED†
OUTPUT	FILE constant	STREAM (open time)
parameter	variable INTERNAL	None
PICTURE or PIC with 'picture-specification'	variable	INTERNAL AUTOMATIC UNALIGNED†
POINTER or PTR	variable	ALIGNED†
POSITION or POS with or without (start-pos)	None	start-pos = 1
precision	REAL†	(15,0) for FIXED BINARY (48) for FLOAT BINARY (5,0) for FIXED DECIMAL (14) for FLOAT DECIMAL
PRINT	FILE constant STREAM (open time) OUTPUT (open time)	None
REAL	variable	scale FIXED or FLOAT† base BINARY or DECIMAL† precision†
RECORD	FILE constant	SEQUENTIAL (open time) INPUT (open time)
RETURNS	ENTRY††	None
SEQUENTIAL or SEQL	FILE constant RECORD (open time)	INPUT (open time)
STATIC	variable	INTERNAL
STREAM	FILE constant	INPUT (open time)
structure	variable	INTERNAL AUTOMATIC
UNALIGNED	variable	None
UPDATE	FILE constant RECORD (open time)	None
variable	None	INTERNAL AUTOMATIC
VARYING	None	None

† Also effective for any parameter descriptor or returns descriptor.

†† Also effective for any parameter descriptor.

A reference is the appearance of an identifier in any context other than one that represents an explicit declaration of the identifier. This section summarizes the various types of references and describes how they are interpreted during program execution. The descriptions of statements and their components given in section 12 indicate precisely what must be represented by references.

DECLARATION APPLICABLE TO A REFERENCE

The identifier in a reference has an applicable declaration that is either in the same block as the reference or in a containing block. A reference to an identifier that is declared in the same block as the reference is called a local reference. A reference to an identifier that is declared in a containing block is called a nonlocal reference. Examples of local and nonlocal references are shown in figure 6-1.

```
P:
PROC;
   DCL A FIXED DECIMAL;
L:
   A=5;              /*LOCAL REFERENCE*/
B:
   BEGIN;
      A=7;           /*NONLOCAL REFERENCE*/
      .
      .
      .
      GOTO B;        /*NONLOCAL REFERENCE*/
      .
      .
      .
      GOTO L;        /*NONLOCAL REFERENCE*/
   END B;
   .
   .
   .
   GOTO L;           /*LOCAL REFERENCE*/
   .
   .
   .
   GOTO B;           /*LOCAL REFERENCE*/
END P;
```

Figure 6-1. Local/Nonlocal References

When a reference is encountered during program execution, the first block examined for the declaration of the identifier is the block containing the reference. If the declaration is not found in that block, the immediate containing block is examined. Each containing block is examined until the declaration is found. The first declaration found by this process is the applicable declaration. References and applicable declarations are illustrated in figure 6-2.

```
CALC;
PROC;
   DCL X FIXED;
   .
   .
   .
   CALL P(X);
   .
   .
   .
P:
   PROC(A);
      DCL (A,B,P) FIXED;
      .
      .
      .
      B = A + P * X;
   END P;
END CALC;
```

The declaration applicable to both references to X is the one established in block CALC. Entry constant P is declared in block CALC, and the arithmetic variable P is declared in the inner block; consequently, the reference to P in the assignment statement refers to the arithmetic variable and not to the entry name.

Figure 6-2. References and Applicable Declarations

GENERATION OR VALUE ACCESSED BY A REFERENCE

During the execution of a PL/I program, an identifier can be associated with several generations or values at the same time. This can occur under the following circumstances:

- Multiple generations are allocated for a controlled variable.
- Multiple on-unit activations are contained in the dynamic stack. This results in multiple values being associated with some condition builtin functions.
- Multiple activations of the same block are contained in the dynamic stack. This results in multiple generations being associated with each automatic variable, defined variable, and parameter variable declared in the block; and multiple values being associated with each label constant, entry constant, and format constant declared in the block.

When the program evaluates a reference to an identifier associated with multiple generations or values, the system must determine which generation or value to use. The appropriate generation or value is selected as follows:

- A reference to a controlled variable always accesses the top generation in the stack of generations for the variable. This is the most recently allocated generation that has not been freed.

- A reference to a condition builtin function always accesses the most recently established value still available for that builtin function. If the most recent on-unit activation in the dynamic stack does not have a value for the builtin function, a value is obtained from a dynamic predecessor of that on-unit activation.

- A local reference to a variable or named constant accesses the value associated with the current block activation when multiple activations of the same block exist in the dynamic stack; a nonlocal reference accesses the value associated with an activation that is part of the environment of the current block activation.

As described in section 2, Dynamic Program Structure, the environment of an activation of an internal block is a set of block activations; this set consists of precisely one activation of each containing block. The environment is used to evaluate nonlocal references of the following types:

- Automatic variable – The appropriate generation is selected.

- Defined variable – The appropriate generation of the defined variable is selected. The host reference named in the DEFINED attribute is evaluated according to its own storage type, which may or may not involve the environment.

- Parameter variable – The appropriate generation of the parameter is selected. The generation associated with the parameter was established when the procedure was invoked.

- Label constant – The block activation that contains the label value is selected. The referenced block activation is established when a label constant is referenced by a GOTO statement, assigned to a label variable, or passed as an argument; that is, the label value includes both the referenced statement and the selected block activation. Nonlocal reference to a label constant is illustrated in figure 6-3.

```
    CALL P(1);
P:
    PROC(A) RECURSIVE;
       DECLARE A FIXED;
       DECLARE LV STATIC LABEL;
       IF A=5 THEN ON COND(BLK) LV=LC;
       IF A=7 THEN SIGNAL COND(BLK);
       IF A+12 THEN CALL P(A+1);
       GOTO LV;
LC:
       PUT SKIP EDIT('A=',A) (A,F(3));
       STOP;
    END P;
```

The environment of the on-unit is established when the ON statement is executed; therefore, the environment of the on-unit is the activation of P in which A has the value 5. (The dynamic predecessor of the on-unit activation is the activation of P in which A has the value 7.) This example prints A=5; the GOTO causes the termination of seven activations of P.

Figure 6-3. Nonlocal Reference to a Label Constant

- Entry constant – The environment associated with the entry constant is selected. This environment is established when an entry constant is referenced by a CALL statement or function reference, or when it is passed as an argument in a procedure invocation; it is then used by activations resulting from invocations of that entry.

- Format constant – Nonlocal references to format constants are not permitted.

DATA REFERENCE

A data reference is used for one of three purposes. It can be a value reference, a target reference, or a storage control reference. A value reference is written for the purpose of obtaining a value. A target reference is written for the purpose of assigning a value. A storage control reference is written for the purpose of allocating or freeing storage.

A data reference can be written in one of four forms: simple, subscripted, structure-qualified, or locator-qualified. Each form is described in the following paragraphs.

SIMPLE REFERENCE

A simple reference is a reference to a variable or a named constant. Simple reference syntax, which consists only of an identifier, is shown in figure 6-4.

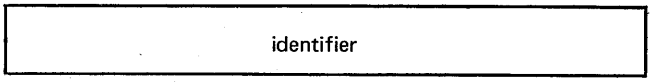


Figure 6-4. Simple Reference Syntax

A simple reference to an array or structure is a reference to the entire array or structure.

SUBSCRIPTED REFERENCE

A subscripted reference is a reference to one or more elements of an array. Subscripted reference syntax is shown in figure 6-5.

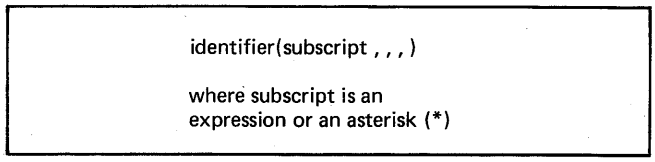


Figure 6-5. Subscripted Reference Syntax

The number of subscripts must equal the number of dimensions declared for the array. The number of dimensions includes inherited dimensions when the array is a member of an array of structures.

A subscript can be either an expression or an asterisk. A subscripted reference with no asterisk subscripts is a scalar reference; it references a single element. A subscripted reference with one or more asterisk subscripts is an array reference; it references a cross section of the array. A cross section is an array of lower dimensionality. For example, it is possible to reference a one- or two-dimensional cross section of a three-dimensional array. A cross section is formed by specifying one or more subscripts uniquely (as expressions) and utilizing all elements that have those subscripts. The dimensionality of a cross section is the number of asterisk subscripts. Other subsets of an array (for example, a diagonal, or a 3x3 piece of a 5x5 array) can be referenced as described under the DEFINED attribute in section 4, Attributes.

An asterisk can be used as a subscript only in the following contexts:

- In an argument of a subroutine or function reference

- In the host reference of a DEFINED attribute (simple defining only)

A subscript that is an expression is evaluated and converted to a fixed binary integer when the reference is encountered during program execution. If the value is less than the lower bound or greater than the upper bound of the dimension, the SUBSCRIPTRANGE condition is raised.

Sample subscripted references are shown in figure 6-6.

Array A is declared as A(2,3,4)

A(1,2,2)	references 1 element
A(1,2,*)	references 4 elements
A(1,*,3)	references 3 elements
A(*,I,I+J)	references 2 elements
A(I-1,*,*)	references 12 elements

Figure 6-6. Sample Subscripted References

STRUCTURE-QUALIFIED REFERENCE

A structure-qualified reference is one form of reference to a member of a structure. Structure-qualified reference syntax is shown in figure 6-7.

{structure-qualifier.}... member-reference

where each structure-qualifier and member-reference is a simple reference or a subscripted reference

Figure 6-7. Structure-Qualified Reference Syntax

A structure-qualified reference with one structure-qualifier identifies a structure and a contained member. For example:

A.B Structure A contains member B.

A structure-qualified reference with more than one structure-qualifier identifies a structure, contained substructures, and the contained member. For example:

A.B.C Structure A contains substructure B, which contains member C.

A partially qualified reference identifies a structure and a member of a contained substructure. For example:

A.C Structure A contains substructure B, which contains member C. This is a partially qualified reference because substructure B is not specified.

A reference to an identifier is said to be fully qualified if it is within the scope of a declaration for which one of the following is true:

- The declared identifier is a structure member, and the reference includes a structure-qualifier for each of the containing structures.
- The declared identifier is not a structure member.

A reference to a structure member need not be fully qualified; some or all of the structure-qualifiers can be omitted and this is termed a partially qualified reference. Partially qualified references can be used whenever no ambiguity results. Fully qualified references eliminate any possibility of ambiguity.

A reference is resolved as follows:

1. If the reference is a fully qualified reference to an identifier, it specifies that identifier.
2. If the preceding case does not apply and if the reference is a partially qualified reference to exactly one identifier, it specifies that identifier.
3. If neither of the preceding cases applies, the reference is illegal or ambiguous and cannot be used.

If the member-reference is an array or inherits dimensions from a containing structure or substructure, the total number of subscripts in the structure-qualified reference must equal the total number of dimensions for the member. The subscripts must appear in the same order as the declared dimensions for the structure and its members, but can appear following any identifier in the reference.

Sample structure-qualified references are shown in figure 6-8.

```
DCL 1 A(5), 2 B(2,2), 3 C;
```

```
A(I).B(2,J).C
A(I,2,J).B.C
A.B.C(I,2,J)
A(I).B(2).C(J)
```

Fully qualified reference to C; references are equivalent.

```
A(4,I,J).C
A(4).C(I,J)
B(4,I,J).C
B(4,I).C(J)
```

Equivalent partially qualified references to C.

```
DCL P(3,3) BIT(2);
DCL 1 A, 2 P, 3 Q FIXED;
DCL 1 B, 2 P, 3 Q POINTER;
```

P has three declarations and Q has two. Any reference to Q or P.Q is ambiguous because it is not sufficiently qualified to identify one structure member uniquely. A reference to P, with or without subscripts, is considered an unambiguous reference to the array, while A.P or B.P would identify a structure member unambiguously. The presence or absence of arguments or subscripts in a reference is not significant for purposes of resolving ambiguity of reference.

Figure 6-8. Sample Structure-Qualified References

LOCATOR-QUALIFIED REFERENCE

A locator-qualified reference is one form of reference to a based variable. Locator-qualified reference syntax is shown in figure 6-9.

locator-reference -> based-reference

where locator-reference is a simple reference, a subscripted reference, a structure-qualified reference, or a locator-qualified reference; and based-reference is a simple reference, a subscripted reference, or a structure-qualified reference.

Note: The term locator qualifier is used to refer to the locator-reference and the symbols ->.

Figure 6-9. Locator-Qualified Reference Syntax

The locator-reference must be a variable or function that has a locator value; the locator value, which can be offset or pointer, addresses a generation of storage. The based-reference must be a based variable or a member of a based structure.

A locator-qualified reference is required only when a locator value cannot be supplied implicitly. A locator value is supplied implicitly when the locator is specified by the BASED attribute of the variable. An explicit locator qualifier overrides any implicitly supplied locator values.

A locator-qualified reference is not permitted for the following:

- A based variable named for allocation in an ALLOCATE or LOCATE statement
- An identifier named in the REFER option of an extent expression

Examples of locator-qualified references are shown in figure 6-10.

```
P->B

P(I,J)->B

P->B(1,X)

A.P->Q->B        Illustrates the use of a locator-
                 qualified reference as a locator
                 qualifier.
```

Figure 6-10. Sample Locator-Qualified References

PROCEDURE REFERENCE

A procedure is invoked at an entry point by execution of a CALL statement or by evaluation of a function reference. When the procedure is invoked, the procedure block is activated and control is transferred to the statement immediately following the PROCEDURE or ENTRY statement. If the procedure activation terminates normally, control returns to the point of invocation.

With the exception of the primary entry point of the main procedure, each entry point can have its own set of parameters. The parameters are denoted by a parameter list included in the PROCEDURE or ENTRY statement. Each parameter is a variable that is declared in the procedure block. External entry points must be declared by DECLARE statement in the calling block; attributes for all external identifiers must be identical in all external procedures that declare the identifiers. Rules relating to procedure invocation are summarized in table 6-1.

TABLE 6-1. PROCEDURE INVOCATION

Method of Invocation	Entry Point Characteristics	Normal Termination
CALL statement	Parameters are optional; RETURNS is not allowed.	RETURN statement without return-value or END statement
Function reference	Parameters are optional; RETURNS is required.	RETURN (return-value)
Operating system control statement	OPTIONS(MAIN) required; parameters and RETURNS are not allowed.	RETURN statement without return-value or END statement.

FUNCTION REFERENCE

A function reference is a reference to an entry point. Function reference syntax is shown in figure 6-11.

simple-reference ([argument,,,])

Figure 6-11. Function Reference Syntax

In a function reference, the simple-reference must identify an item declared with the ENTRY and RETURNS attributes. The number of arguments must equal the number of parameters declared for the invoked entry point. If the entry point has no parameters, the reference must have a null argument list of the form ( ).

Any reference to a function entry name followed by a parenthesized argument list, including a null argument list, is a function reference; that is, a reference to the value returned when the function is invoked. Any reference to a function entry name without a following parenthesized argument list is a reference to an entry value; it does not cause the procedure to be invoked.

Examples of function references are shown in figure 6-12.

```
DCL X AREA;
DCL A ENTRY() RETURNS(AREA);
DCL C ENTRY(FIXED,BIT(2)) RETURNS(FLOAT);
.
.
.
X = A();
/*FUNCTION REFERENCE WITH AREA VALUE*/
Z = C(I+1,↑10↑B);
/*FUNCTION REFERENCE WITH FLOAT VALUE*/
```

Figure 6-12. Sample Function References

PARAMETER AND RETURNS DESCRIPTORS

An entry point denoted by a PROCEDURE or ENTRY statement is declared explicitly by usage as a named constant with the ENTRY attribute, and with the RETURNS

attribute when one is included in the statement. If a parameter list is specified for the entry point, the entry constant declaration is completed with parameter descriptors taken from the declarations of each of the parameter variables.

Parameter descriptors enable the compiler to properly evaluate arguments of function references and CALL statements. A returns descriptor specifies the attributes of the value that is to be returned to the point of invocation.

ARGUMENTS AND PARAMETERS

An argument is an expression that is specified in a function reference or CALL statement. A parameter is a variable that is included in a parameter list of a PROCEDURE or ENTRY statement.

When an entry point is invoked by a function reference or CALL statement that has an argument list, each argument is passed to the corresponding parameter declared for the invoked entry point. The first argument is passed to the first parameter, the second argument to the second parameter, and so forth.

Arguments and parameters must meet the following requirements:

- The number of arguments must equal the number of parameters.
- Aggregate type of an argument must be the same as that of the corresponding parameter.
- Data type of an argument must be the same as or one that can be converted to the data type of the corresponding parameter.
- An aggregate passed as an argument must meet the requirements for argument passing by reference.

The following types of data cannot be passed as arguments:

- Format constants
- Builtin function names (builtin function results can be passed)
- Programmer-named conditions
- A DEFINED variable that is an iSUB-defined array or a cross section of such an array

Argument Passing by Reference and by Value

Each argument is passed either by reference or by value. The differences between the two types are as follows:

Argument passed by reference

> The parameter generation is the generation of the argument variable, and therefore describes the same storage. If a value is assigned to the parameter, the value of the original argument is changed.

Argument passed by value

> A dummy argument is allocated and a generation created for it; the parameter generation is the generation of the dummy argument. If a value is assigned to the parameter, the value of the original argument is not changed.

An argument is passed by reference if it satisfies all of the following rules:

- The argument must be a variable reference. The variable reference can be subscripted, structure qualified, or locator qualified; it cannot be an expression containing operators.
- Data conversion must not be required. Each scalar item of the argument must have the same data type and alignment as the corresponding scalar item of the parameter. The following need not match: scope type, storage type, file description attributes, INITIAL attribute, OFFSET attribute area reference, ENTRY parameter descriptors, returns descriptor, or member names of structures.
- For each extent expression (that is, nonasterisk extent) in the declaration of the parameter, the argument must have a corresponding extent expression. The extent expression for the argument must be a decimal integer (optionally signed if it represents an upper bound or lower bound of an array dimension); must be equal to the extent expression of the parameter; and must not contain a REFER option.
- If the argument is an aggregate, the aggregate argument and parameter must have identical aggregate type, data type, and alignment.
- The argument must not be contained in an iSUB-defined array.

An argument that cannot be passed by reference is passed by value. Enclosing a scalar argument in parentheses forces the argument to be passed by value.

Examples of argument passing by reference and by value are shown in figure 6-13.

Conversion of Arguments

If the data types or alignments of an argument and the corresponding parameter do not match, the argument is converted to the data type and alignment of the parameter. Conversion is performed before the procedure is invoked, and the converted value is passed as a dummy argument. It is an error if the argument cannot be converted. Aggregate arguments that require such conversion cannot be passed.

The conversion rules used for passing dummy arguments are the same as those used for assignment. Conversions are described in section 7, Data Manipulation.

If an argument is a pointer variable and the corresponding parameter is an offset variable, the area of the offset must be known in the calling block and the generation identified must be in that area.

Storage Associated with a Parameter

An argument passed by reference shares its generation of storage with the parameter. The two variables share the generation for the duration of the procedure activation. Either variable can be used to access and assign values to the generation.

An argument passed by value has a dummy argument created by the compiler. The dummy argument has a unique generation of storage allocated for its value or converted value at compilation time. That generation is associated with the parameter for the duration of the procedure

```
DCL(W,X,Y) FIXED BIN;
DCL F ENTRY(FIXED BIN,FIXED BIN) RETURNS(FIXED BIN);
Y=F(X,W);     /*PASS BY REFERENCE*/
Y=F(3,61);    /*PASS BY VALUE*/
Y=F(5,X);     /*FIRST ARGUMENT IS PASSED BY VALUE*/
              /*SECOND ARGUMENT IS PASSED BY REFERENCE*/
```

A literal constant or named constant is passed by value.

```
P: PROC(X);
   DCL LVAR LABEL;              /*LABEL VARIABLE*/
L: RETURN;                      /*LABEL CONSTANT*/
   DCL FL ENTRY(LABEL);
   CALL FL(X);  /*PASS BY REFERENCE*/
   CALL FL(Y);  /*PASS BY VALUE*/
END P;
```

```
DECLARE (X,Y) CHAR(10);
DECLARE (F,G) ENTRY(CHAR(*)) RETURNS(CHAR(10));
Y=F(X);                 /*PASS BY REFERENCE*/
Y=F(G(X));              /*PASS G(X) TO F BY VALUE*/
Y=F(COPY(X,2));         /*PASS COPY(X,2) TO F BY VALUE*/
```

A function reference or a builtin function reference is passed by value: the function or builtin function is invoked, and the value returned by the function or builtin function is passed as a dummy argument.

```
DECLARE (X,Y,Z) FIXED BIN;
DECLARE F       ENTRY(FIXED BIN) RETURNS(FIXED BIN);
Z=F(X);       /*PASS BY REFERENCE*/
Z=F(X+Y);     /*PASS BY VALUE*/
Z=F(+X);      /*PASS BY VALUE*/
Z=F(-X);      /*PASS BY VALUE*/
Z=F((X));     /*PASS BY VALUE*/
```

An expression that is not simply a variable reference is passed by value. Note that a variable reference can contain expressions in subscripts or locator qualifiers; this does not force pass-by-value.

```
DECLARE X(8)   CHAR(10);
DECLARE F      ENTRY(CHAR(10)) RETURNS(CHAR(10));
Z=F(X(3));     /*PASS BY REFERENCE*/
Z=F(X(A-2));   /*PASS BY REFERENCE*/
```

```
DCL B(5)      FIXED BIN BASED;
DCL P         POINTER;
DCL Q         ENTRY(FIXED BIN) RETURNS(POINTER);
DCL F         ENTRY(FIXED BIN) RETURNS(FIXED BIN);
Z=F(P->B(3));         /*PASS BY REFERENCE*/
Z=F(Q(2)->B(A+1));    /*PASS B(A+1) TO F BY REFERENCE*/
```

```
DCL A(2,3,4) FIXED BIN;
DCL D(2,4)   FIXED BIN DEFINED(A(*,1,*));
DCL F        ENTRY((*,*) FIXED BIN);
CALL F(A(*,2,*));  /*PASS BY REFERENCE*/
CALL F(D);         /*PASS BY REFERENCE*/
```

An array or array cross section can be passed by reference if it satisfies the rules for pass-by-reference.

```
DECLARE A(8) CHAR(20);
DECLARE B(8) CHAR(20) VARYING;
DECLARE C    CHAR(20);
DECLARE P ENTRY(CHAR(*));
CALL P(A(1));  /*PASS BY REFERENCE*/
CALL P(B(1));  /*PASS BY VALUE*/
```

If data conversion is required, the argument is passed by value.

```
DCL C CHAR(10);
DCL P ENTRY(CHAR(20));
DCL Q ENTRY(CHAR(*));
CALL P(C);  /*PASS BY VALUE*/
CALL Q(C);  /*PASS BY REFERENCE*/
```

If the parameter has an AREA, BIT, CHARACTER, or dimension attribute, and the corresponding argument has a different extent, the argument is passed by value. (If the parameter has an asterisk extent, the argument can be passed by reference, regardless of its extent, unless pass-by-value is forced for some other reason.)

Figure 6-13. Argument Passing by Reference and by Value

activation; it is freed automatically when control returns to the point of invocation. The original argument is not affected by any changes made to the dummy argument; conversely, the dummy argument is not affected by any changes made to the value of the original argument, including the freeing of the generation of the original argument.

A parameter can never be explicitly allocated. A generation can be associated with a parameter only when arguments are passed to it during procedure invocations. Parameters other than those named for the entry point used for the current invocation have undefined values and should not be accessed during that invocation; they are effectively in an unallocated state.

Extents of a Parameter

The extents of a parameter generation are those of the argument or converted argument. When an asterisk extent is used in a declaration of a parameter, the size, length, or bounds of the parameter are replaced by the corresponding size, length, or bounds of the argument or converted argument.

In the following example, the length of the string represented by P is the length of the argument associated with P each time the procedure is invoked.

```
I: PROC (P);
DCL P CHAR(*); /* PARAMETER */
:
:
:
```

CALLING FORTRAN SUBPROGRAMS

PL/I programs can reference FORTRAN functions and subroutines.

There are two methods of passing parameters between PL/I and FORTRAN routines: using common storage areas, and using argument lists.

Using common storage areas is the only method of calling FORTRAN routines directly from PL/I. This method involves creating a PL/I structure that can be accessed in the FORTRAN routine as a named common block.

The other method of passing parameters involves using an argument list. A PL/I program cannot directly call a FORTRAN subprogram if an argument list is used to pass parameters. Instead, the user must write a COMPASS interface routine to modify the argument list generated by PL/I.

Not all types of PL/I variables can be passed to FORTRAN subprograms. PL/I provides more data types than FORTRAN; therefore, only those data types that are allowed by FORTRAN can be passed as parameters.

Furthermore, arrays are stored differently in PL/I than in FORTRAN, so the programmer must compensate for the difference in the logic of the program.

Common Storage Areas

One method of passing parameters between PL/I and FORTRAN routines is by using common storage areas. This method can be used only when calling a FORTRAN subroutine with the PL/I CALL statement.

The name of the FORTRAN routine must be explicitly declared in the PL/I routine with the ENTRY attribute. References to the FORTRAN subroutine are written in the same manner as references to PL/I procedures.

A common storage area must be created in the PL/I routine and in the FORTRAN subprogram. A common storage area in PL/I is a structure with the STATIC and EXTERNAL attributes. A common storage area in FORTRAN is a named common block.

The name of the PL/I structure must correspond to the name of the FORTRAN common block. The structure can contain scalars and arrays, but not substructures. The scalars and arrays in the structure must correspond to the variables and arrays in the FORTRAN common block.

An example of the use of common storage areas to pass parameters is shown in figure 6-14. The FORTRAN routine is declared with the ENTRY attribute in the PL/I program. The structure MYCMBLK is declared STATIC EXTERNAL, and corresponds to the common block MYCMBLK in the FORTRAN subroutine. The structure members are initialized in the PL/I routine and printed. Then the FORTRAN routine is called to modify the value of the structure members. Upon returning from the FORTRAN subprogram, the new values of the structure members are printed.

Arguments Lists

The other method of passing parameters between PL/I and FORTRAN routines is by using argument lists. This method can be used for calling both FORTRAN functions and subroutines.

This method requires the programmer to supply a COMPASS interface. The name of the interface must be explicitly declared in the PL/I routine with the ENTRY attribute. References to the COMPASS routine are written in the same manner as references to PL/I functions and procedures.

The COMPASS routine must modify the argument list generated by PL/I so that it can be used by the FORTRAN routine. After modifying the list, it must invoke the FORTRAN subprogram.

The argument list is located by accessing register A1, which contains the address of the argument list.

The FORTRAN routine expects the argument list to contain an address for each argument, and expects the list to be terminated by a word of zeros.

However, the argument list generated by PL/I contains the address of a descriptor for each argument, rather than the address of the argument itself. The descriptor word contains the address of the argument in the low order 18 bits. Additional information about the argument might be stored in the rest of the descriptor word. Lists generated by PL/I CALL statements are terminated with a negative zero followed by a positive zero. Lists generated by PL/I function references are terminated with the address of the descriptor of the function result followed by a positive zero.

Figure 6-15 shows the format of the PL/I argument list. Figure 6-16 shows the format of the argument list

```
COMMON:  PROCEDURE OPTIONS(MAIN);

         DCL FORT ENTRY;
         DCL 1 MYCMBLK STATIC EXTERNAL,
               2 X FLOAT DECIMAL,
               2 I FIXED DECIMAL(3,0),
               2 MESS(2) CHAR(10) ALIGNED;

         X = 1.234;
         I = 5;
         MESS(1) = 'HELLO FROM';
         MESS(2) = ' PL/I';

         PUT EDIT (MESS(1),MESS(2),X,I) (SKIP,2(A),F(7,3),F(5));
         CALL FORT;
         PUT EDIT (MESS(1),MESS(2),X,I) (SKIP,2(A),F(7,3),F(5));

END COMMON;


       SUBROUTINE FORT
C
       COMMON /MYCMBLK/ X, I, MESS(2)
C
       X = X * 2.0
       I = I + 25
       MESS(2) = 8H FORTRAN
C
       RETURN
       END


OUTPUT:

HELLO FROM PL/I        1.234    5
HELLO FROM FORTRAN     2.468   30
```

Figure 6-14. Common Storage Area Example

expected by the FORTRAN subprogram. The COMPASS interface routine must transform the argument list from the PL/I format to the FORTRAN format.

The PL/I argument list contains an extra level of indirection compared to FORTRAN. Therefore, the COMPASS interface must replace the descriptor addresses with the addresses of the arguments themselves. The original PL/I argument list can be overwritten with the modified list. The COMPASS routine must preserve the contents of register A0.

An example of a PL/I reference to a FORTRAN subroutine is shown in figure 6-17. The name of the COMPASS interface is declared with the ENTRY attribute. The interface reformats the argument list by reading each descriptor word and storing it back into the list over the original pointer.

For PL/I references to FORTRAN functions, a COMPASS interface is always required; the common storage area method cannot be used. FORTRAN functions return the result in register X6. PL/I functions return the result in an address specified in the argument list. The address of the descriptor for the last argument is followed by the address of the descriptor for the function result. This address takes the place of the negative zero which precedes the terminating positive zero.

An example of a PL/I reference to a FORTRAN function is shown in figure 6-18. The argument list is transformed as before, but the descriptor for the function result is read in addition to those of the other arguments. After the return from the FORTRAN routine, the COMPASS interface reads the descriptor and stores the contents of X6 at the specified address (X6 contains the result returned from FORTRAN).

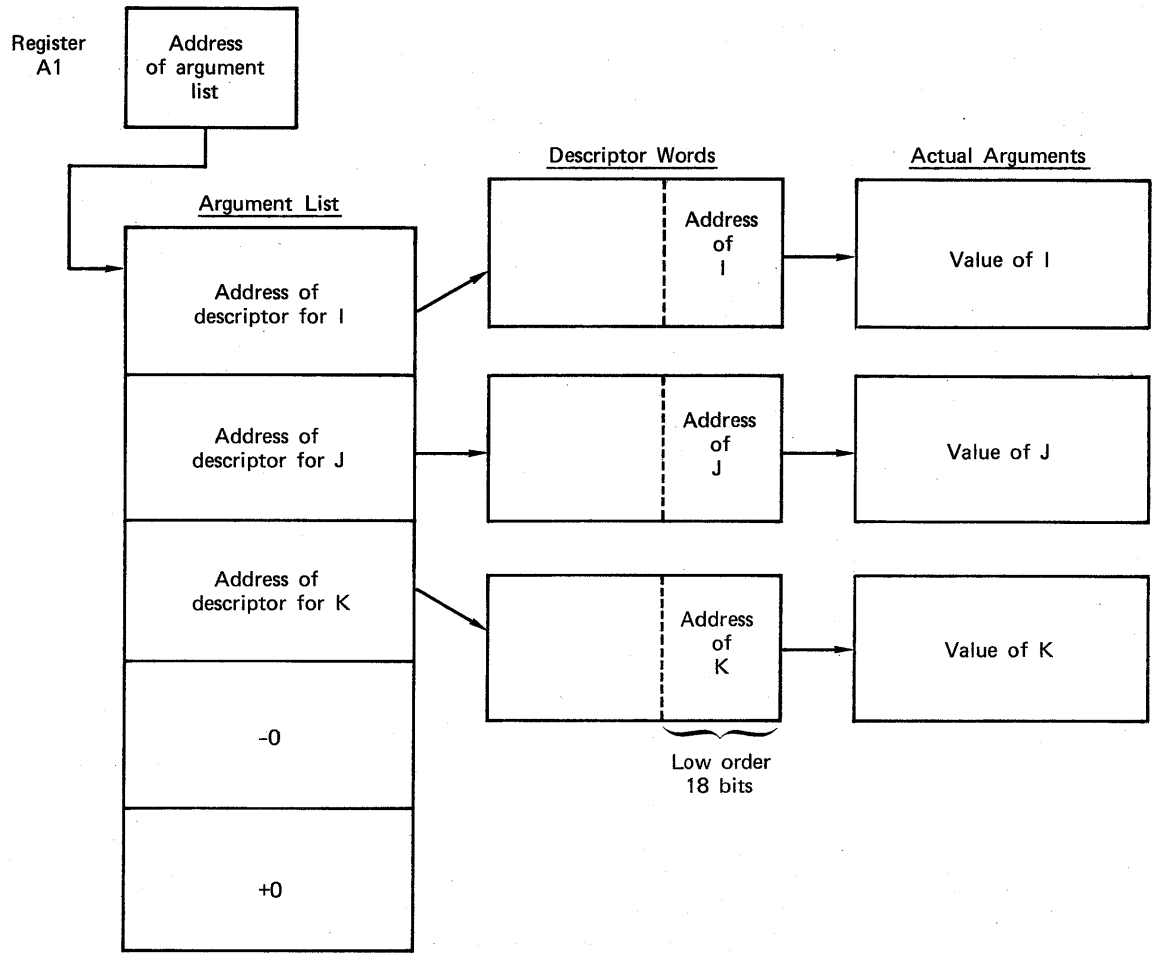
Data Type Restrictions

PL/I provides more data types for variables than FORTRAN. Therefore, not all types of variables can be passed between PL/I and FORTRAN. Only those data types that are common between the two languages can be shared. Table 6-2 shows the PL/I data types that can be passed to FORTRAN subprograms, and gives the corresponding FORTRAN data types.

Other restrictions are:

PL/I strings passed to FORTRAN routines must be declared ALIGNED.

CHARACTER and BIT strings longer than one word must be treated like a 1-dimensional array within the FORTRAN routine. Each group of 10 characters or 60 bits in the PL/I string corresponds to one word of the FORTRAN array. The length of the FORTRAN array must be a multiple of 10 characters or 60 bits.



This shows the PL/I argument list generated by the statement CALL F(I,J,K). For PL/I function references, the -0 in the argument list is replaced with the address of the descriptor word for the function result.

Figure 6-15. Format of PL/I Argument List

TABLE 6-2. DATA TYPE CORRESPONDENCE

PL/I Data Type	FORTRAN Data Type
FLOAT DECIMAL	REAL
FIXED DECIMAL (q=0)	INTEGER
CHAR(10) ALIGNED Nonvarying	Hollerith
FLOAT BINARY	REAL
FIXED BINARY (q=0)	INTEGER
BIT(1) ALIGNED	LOGICAL

PL/I strings of varying length can be passed to FORTRAN routines, but the FORTRAN routine must not alter the current length of the string. Varying length strings cannot be passed through common storage areas. Arrays of varying strings must not be passed to FORTRAN subprograms.

PL/I pictured variables are nonvarying character strings and are treated like FORTRAN character data.

PL/I AREA, POINTER, OFFSET, FILE, ENTRY, or LABEL data and FORTRAN COMPLEX or DOUBLE PRECISION data cannot be shared.

Array Storage Differences

Arrays are stored differently in PL/I than in FORTRAN, so the programmer must compensate for the differences in the logic of the program.

In general, PL/I arrays are stored in row order, whereas FORTRAN arrays are stored in column order. Thus, the programmer should reverse the order of the subscripts in the FORTRAN routine.

For example, the array element referenced as ARRAY(2,3) in the PL/I routine, is referenced as ARRAY(3,2) in the FORTRAN routine.

There are two methods of passing arrays using argument lists: by specifying the unsubscripted array name in the

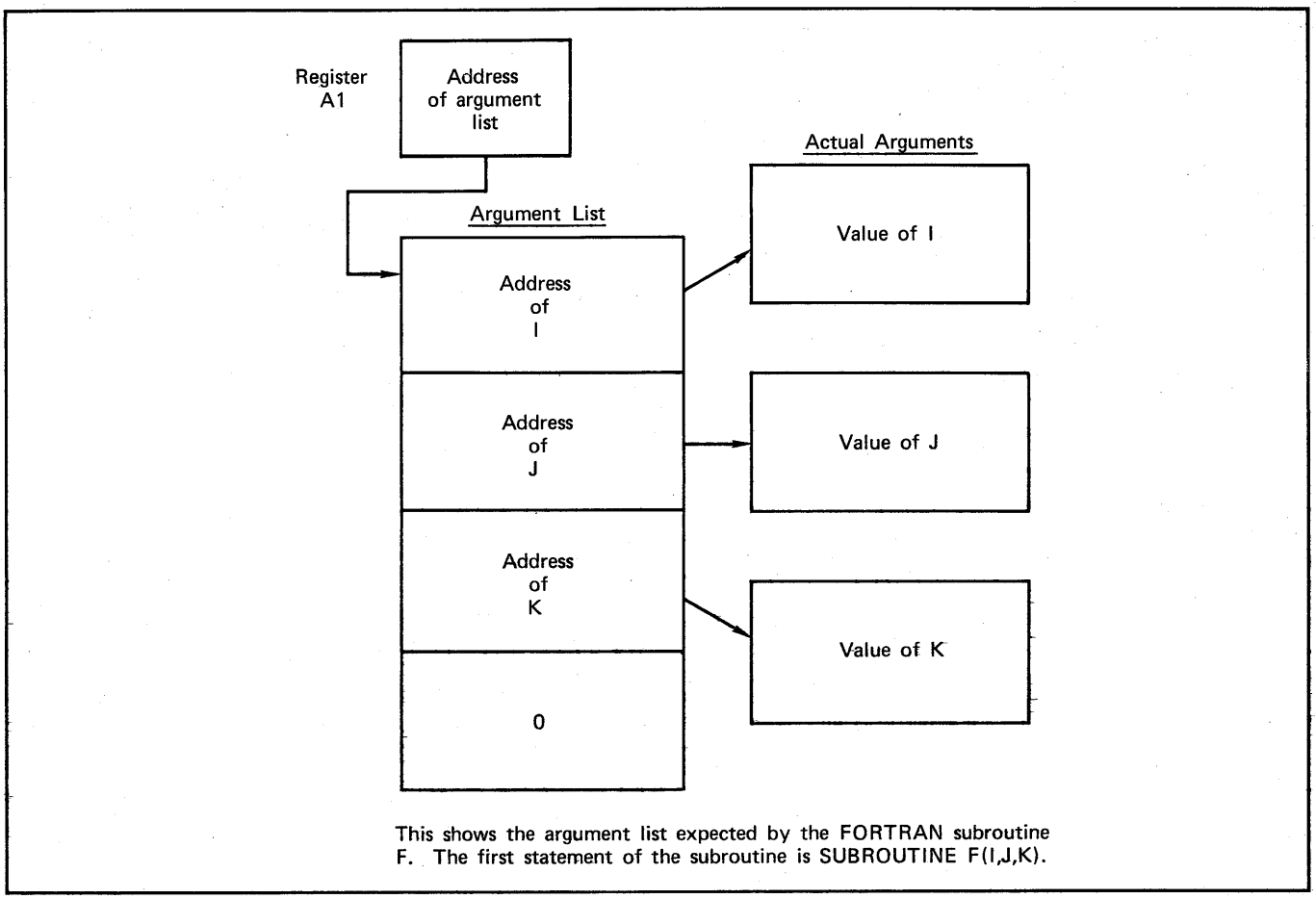


Figure 6-16. Format of FORTRAN Argument List

argument list, and by specifying the subscripted array name.

If the unsubscripted array name is specified, array element zero is associated with the first element of the FORTRAN array. This occurs even if the PL/I array does not have its origin at zero. For example, if a PL/I array is declared without explicitly specifying the array bounds, the first subscript defaults to 1. If this array is passed to FORTRAN, array element zero is associated with the first element of the FORTRAN array, which results in a reference outside of the array bounds. To avoid this problem, the programmer can declare all PL/I arrays to have origins at zero if they are to be passed to FORTRAN in this way.

If a subscripted array name appears in the argument list, the specified element of the PL/I array is associated with the first element of the FORTRAN array; all subsequent elements correspond.

To pass arrays through common storage areas, the programmer must simply reverse the subscripts as explained previously.

Arrays with elements longer than one word should not be passed to FORTRAN subprograms. If this is done, the programmer must develop an algorithm for computing proper subscripts in the FORTRAN routine.

Arrays of variable length strings must not be passed to FORTRAN subprograms.

Additional Considerations

Input and output operations should be restricted to PL/I routines.

PL/I allows recursive procedures, but FORTRAN does not. FORTRAN routines that are called recursively can result in infinite loops.

PL/I snap output normally contains a traceback of all active procedures and on-units; however, FORTRAN and COMPASS routines called from PL/I will not appear in the snap output.

```
ARGLIST:  PROCEDURE OPTIONS(MAIN);
          DCL COMPINT ENTRY (FLOAT DEC, FIXED DEC, CHAR(10) ALIGNED);
          DCL A FLOAT DECIMAL;
          DCL B FIXED DECIMAL;
          DCL C(2) CHAR(10) ALIGNED;

          A = 1.234;
          B = 5;
          C(1) = 'HELLO FROM';
          C(2) = ' PL/I';

          PUT EDIT (C(1),C(2),A,B) (SKIP,2(A),F(7,3),F(3));
          CALL COMPINT ( A, B, C(2) );
          PUT EDIT (C(1),C(2),A,B) (SKIP,2(A),F(7,3),F(3));
END ARGLIST;



      SUBROUTINE FORT ( A, B, C )
C
      INTEGER B,C
C
      A = A * 2
      B = B + 25
      C = 8H FORTRAN
C
      RETURN
      END



          IDENT  COMPINT
          ENTRY  COMPINT
COMPINT   BSS    1
          SB6    0
          SB7    3           SET B7 TO NUMBER OF ARGUMENTS
LOOP      SA2    A1+B6       READ ADDR OF DESCRIPTOR INTO X2
          SA3    X2          READ ADDR OF ARGUMENT INTO X3
          SX6    X3          WRITE ADDR OF ARGUMENT IN ARG LIST
          SA6    A2
          SB6    B6+1
          GT     B7,B6,LOOP LOOP BACK IF MORE ARGUMENTS
          RJ     =XFORT
          EQ     COMPINT
          END


OUTPUT:

 HELLO FROM PL/I       1.234  5
 HELLO FROM FORTRAN    2.468 30
```

Figure 6-17. Argument List Example

```
FORTFUN: PROCEDURE OPTIONS(MAIN);
         DCL AVG ENTRY((*)FLOAT DEC, FIXED DEC) RETURNS(FLOAT DEC);
         DCL ARY1(0:2) FLOAT DEC INIT(12.345,-98.765,246.357);
         DCL ARY2(0:4) FLOAT DEC INIT(1.2,-3.4,5.6,-7.8,9.0);
         DCL ARY3(0:9) FLOAT DEC INIT(.1,.2,.3,.4,.5,.6,.7,.8,.9,.0);

         PUT SKIP EDIT('AVG OF ARY1 IS', AVG(ARY1,3)) (A,F(9,4));
         PUT SKIP EDIT('AVG OF ARY2 IS', AVG(ARY2,5)) (A,F(9,4));
         PUT SKIP EDIT('AVG OF ARY3 IS', AVG(ARY3,10)) (A,F(9,4));

END FORTFUN;
```

```
      FUNCTION AVERAGE (ARRAY,IDIM)
C
      DIMENSION ARRAY(IDIM)
C
      SUM = 0
      DO 100 I = 1,IDIM
         SUM = SUM + ARRAY(I)
100   CONTINUE
      AVERAGE = SUM/IDIM
C
      RETURN
      END
```

```
          IDENT  AVG
          ENTRY  AVG
AVG       BSS    1
          SB6    0
          SB7    3           SET B7 TO NUMBER OF ARGUMENTS + 1
LOOP      SA2    A1+B6       READ ADDR OF DESCRIPTOR INTO X2
          SA3    X2          READ ADDR OF ARGUMENT INTO X3
          SX6    X3          WRITE ADDR OF ARGUMENT IN ARG LIST
          SA6    A2
          SB6    B6+1
          GT     B7,B6,LOOP  LOOP BACK IF MORE ARGUMENTS
          SA6    RESULT      SAVE ADDR FOR RESULT
          RJ     =XAVERAGE
          SA1    RESULT      READ UP ADDR FOR STORING RESULT
          SA6    X1          STORE RESULT IN SPECIFIED ADDR
          EQ     AVG
RESULT    BSS    1
          END
```

OUTPUT:

```
 AVG OF ARY1 IS  53.3123
 AVG OF ARY2 IS   0.9200
 AVG OF ARY3 IS   0.4500
```

Figure 6-18. FORTRAN Function Example

BUILTIN FUNCTION OR PSEUDOVARIABLE REFERENCE

A builtin function reference is a reference to a builtin function. A pseudovariable reference is a reference to a pseudovariable; a pseudovariable is a counterpart of a builtin function and is used as the target of an assignment operation. Builtin function and pseudovariable reference syntax is shown in figure 6-19.

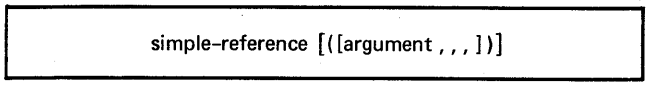


Figure 6-19. Builtin Function and Pseudovariable Reference Syntax

In a builtin function reference, the simple-reference must be the name of one of the system-supplied builtin functions. The number of arguments and the attributes of the arguments must be as described in section 11, Builtin Functions. If the builtin function does not require any arguments, the reference can have a null argument list of the form ( ) or the parentheses can be omitted. Any reference to a builtin function causes the builtin function to be invoked. A builtin function cannot be referenced as an entry value.

In a pseudovariable reference, the simple-reference must be the name of one of the system-supplied pseudovariables. The number of arguments and the attributes of the arguments must be as described in section 11. If the pseudovariable does not require any arguments, the reference can have a null argument list of the form ( ) or the parentheses can be omitted. The name of each pseudovariable is the same as the name of the corresponding builtin function. It is recognized as a pseudovariable reference by the context in which it appears.

This section describes the various types of PL/I expressions that can be written for manipulating data. Other sections of this manual describe where expressions can be used in the language. An expression can be a primitive, prefix, or infix expression. An expression that contains subexpressions is a combination of these simple expressions. Evaluation of an expression proceeds in a specific way, according to the rules given in this section.

Arithmetic, string, and comparison operations are performed as necessary in the evaluation of an expression. The operator in a simple expression establishes whether the operation is to be arithmetic, string, or comparison. Every operator in an expression is either a prefix or infix operator. For each type of operation, this section describes the prefix and infix operators that can be used and the result of the operation. The result of an expression evaluation can be used as an operand in another expression.

PL/I performs conversion of operands as necessary in any arithmetic, string, or comparison operation. Since the operator establishes the type of operation, the operator can force conversion of an operand to a specific computational type, such as arithmetic, bit, or character. In an infix operation, conflict in the data types of the operands can force further conversion, such as from FIXED DECIMAL to FLOAT DECIMAL. In an arithmetic, string, or comparison operation, any necessary conversions occur before the arithmetic, string, or comparison operation is performed.

Assignment involves a source value and a target. The source value is the result of an expression evaluation. Conversion is necessary if the source value has a different data type than the target. If the data type of the source value matches the data type of the target, the source value is assigned to the target without conversion.

Conversion is performed for operations and during assignment. This section briefly describes other situations in which any necessary conversion is performed. The rules for each available conversion from one data type to another are described.

The rules for picture-controlled conversions are described in this section. The picture codes available for pictured character and pictured numeric items are also described.

EXPRESSIONS

Each expression consists of operators and operands arranged in a meaningful way. The operators are categorized as arithmetic, string, or comparison operators and as prefix or infix operators. The complete list of operators is shown in table 7-1. An expression can be a single primitive, prefix, or infix expression. An expression can also be a combination of expressions. An expression used as part of a larger expression is called a subexpression. The syntax of expressions is shown in figure 7-1.

An operand can be a literal constant, a reference, an iSUB reference, or an expression. An expression can be used as an operand in another expression. The result of the final expression evaluation is the result of the complete expression.

TABLE 7-1. LIST OF OPERATORS

Operator	Type	Prefix or Infix	Meaning
+	arithmetic	prefix	positive
-	arithmetic	prefix	negative
+	arithmetic	infix	add
-	arithmetic	infix	subtract
*	arithmetic	infix	multiply
/	arithmetic	infix	divide
**	arithmetic	infix	exponentiate
¬	bit string	prefix	NOT
&	bit string	infix	AND
|	bit string	infix	OR
||	character string or bit string	infix	concatenate
=	comparison	infix	equal
¬=	comparison	infix	not equal
<	comparison	infix	less than
<=	comparison	infix	less than or equal
¬<	comparison	infix	not less than
>	comparison	infix	greater than
>=	comparison	infix	greater than or equal
¬>	comparison	infix	not greater than

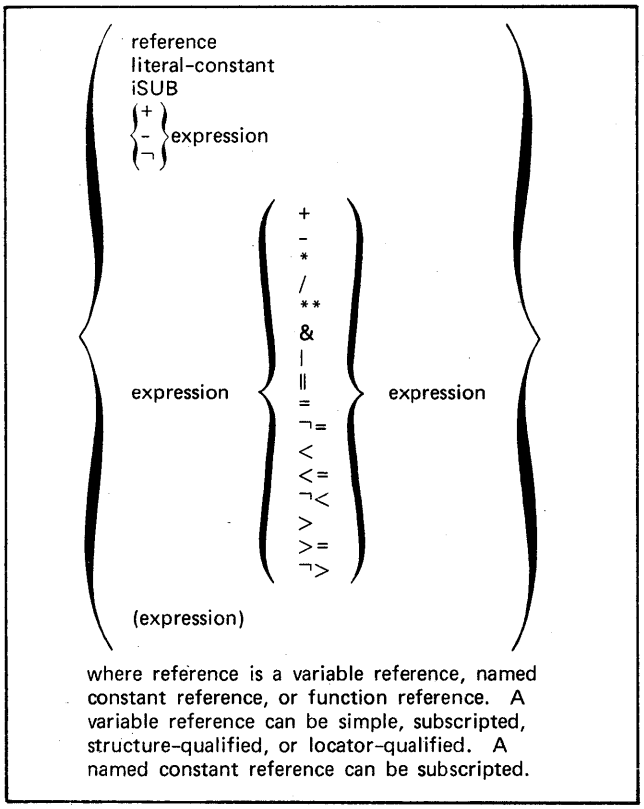
A primitive expression consists of a single operand and no operators. A prefix expression consists of a single operator and a single operand. An infix expression consists of two operands separated by a single operator. Examples of primitive, prefix, and infix expressions are shown in figure 7-2.

The value of an expression is determined each time it is encountered during program execution. An expression yields a result value of a particular data type and aggregate type. The evaluation of each subexpression contained in the full expression produces a result with a particular data type. If the result is used as an operand in another subexpression, the data type and aggregate type are used as the operand type in that subexpression evaluation. The result of the final subexpression evaluation has a result type.

PRIMITIVE EXPRESSIONS

A primitive expression that is a variable reference has the data type and aggregate type of the variable. A primitive expression that is a function reference has the data type and aggregate type of the value returned by the function. If the

primitive expression is used as an operand in a prefix or infix expression, the data type and aggregate type must be acceptable for the prefix or infix operation.



where reference is a variable reference, named constant reference, or function reference. A variable reference can be simple, subscripted, structure-qualified, or locator-qualified. A named constant reference can be subscripted.

Figure 7-1. Expression Syntax

Primitive Expressions

ADDTRY	variable reference
THG(7)	subscripted variable reference
ER.DATA6	structure-qualified variable reference
KPTR->N	locator-qualified variable reference
LABEL7	named constant reference
UTIL(R,T)	function reference
12.6	arithmetic constant
4	integer constant
'JK L'	character constant
'10'B	bit constant
(4)'101'B	replicated bit constant
2SUB	iSUB reference

Prefix Expressions

```
+ADDTRY
-THG(7)
¬'10'B
```

Infix Expressions

```
ER.DATA6 + UTIL(R,T)
ADDTRY ** 4
'10'B | (4)'101'B
KPTR->N ¬< 12.6
-THG(7) = '10'B
3 * 2SUB
```

Figure 7-2. Primitive, Prefix, and Infix Expression Examples

PREFIX EXPRESSIONS

The data type of the expression value is determined by the prefix operator and the data type of the operand. In a prefix expression, the operator requires that the operand be of a certain computational type. The operand is compatible with the operator if the operand has the required computational type or can be converted to that type.

The operand must be scalar but can be an array element or structure member. The operand must represent a single value rather than an aggregate value.

The steps in evaluation are as follows:

1. The operand is evaluated. The value of the primitive expression is used, or an intermediate result from subexpression evaluation is used.
2. The operand is converted to an intermediate value appropriate for the operator, if necessary.
3. The prefix operation produces a result value.

INFIX EXPRESSIONS

The data type of the expression is determined by the infix operator and the data types of the operands. In an infix expression, the operator requires that the operands be of a certain computational type. Each operand is compatible with the operator if the operand has the required computational type or can be converted to that type.

Each operand must be scalar but can be an array element or structure member. Each operand must represent a single value rather than an aggregate value.

The steps in evaluation are as follows:

1. Both of the operands are evaluated. The value of a primitive expression is used, or an intermediate result from subexpression evaluation is used for either operand.
2. Each operand is converted, if necessary, to an intermediate value appropriate for the infix operator. If the two operands have different data types, one or both can require further conversion.
3. The infix operation produces a result value.

ORDER OF EVALUATION

An expression that contains more than one operator is evaluated in an order established by the precedence of operators and by the way the expression is parenthesized. Operator precedence determines the order of evaluation in a static way; operator precedence is not dynamic. If an operand is an expression, the expression is fully evaluated before being used as an operand. Paired parentheses force evaluation of the contained expression before the result is used as an operand of any operator outside the parentheses.

Within the entire expression, and within each pair of parentheses, the order of evaluation depends on the priority of operators. Subexpressions containing operators of the highest priority are evaluated first, then those of the next highest priority, and so forth. If a subexpression or parenthesized subexpression contains operators of equal priority, the order of evaluation is usually left to right but can be right to left. The order of evaluation is shown in table 7-2. Some examples of order of evaluation are shown in figure 7-3.

TABLE 7-2. ORDER OF EVALUATION

Order of Evaluation	Operators	Same Priority Order of Evaluation
First	¬ +(prefix) -(prefix) **	Right to left
Second	* /	Left to right
Third	+(infix) -(infix)	Left to right
Fourth	||	Left to right
Fifth	= ¬= < <= ¬< > >= ¬>	Left to right
Sixth	&	Left to right
Last	|	Left to right

Expression	Order of Evaluation
- A ** B	- (A ** B)
C / - D	C / (- D)
E + F / G	E + (F / G)
H + J | | K	(H + J) | | K
L = M | | N	L = (M | | N)
P > Q & R = S	(P > Q) & (R = S)
S | T & U	S | (T & U)

Figure 7-3. Order of Evaluation Examples

When an operand is evaluated, the order of evaluation for subscripts, locator qualifiers, and function references is not predetermined. The program must not depend on any particular order of evaluation for subscripts, locator qualifiers, and function references in an expression.

For the purpose of optimization, the result from a subexpression evaluation can be reused if the result has already been calculated during the present execution of the statement. The result of a function reference is never reused. The value of a variable reference is not reused if the compiler detects that the generation or the value could change during subexpression evaluation. If any computational conditions are raised during subexpression evaluation, then reuse of subexpression values can change the final result only by reducing the possible number of raised computational conditions.

OPERATIONS

The evaluation of expressions can involve arithmetic, string, or comparison operations. The operator used in an expression establishes what type of operation is to be performed. Depending on the circumstances, an operation can be a prefix or an infix operation.

If operations require conversion of operands, any necessary conversions take place before the operation is performed. Specific conversions are described later in this section under Conversions. The conversions for computational operands are shown in table 7-3. The required intermediate type for each necessary operand conversion in any given operation is described under the particular operation.

ARITHMETIC OPERATIONS

Arithmetic operations can be specified with prefix or infix operators. The prefix operators are + (positive) and - (negative). The infix operators are + (add), - (subtract), * (multiply), / (divide), and ** (exponentiate). Arithmetic operations can involve operands that are arithmetic, character string, bit string, or pictured items. Operands can also be intermediate values that are result values from subexpression evaluations. The result of the operation is always arithmetic with a scale FIXED or FLOAT and a base DECIMAL or BINARY. The result can be used as an operand in a subsequent operation or can be the final result of expression evaluation, depending on the circumstances. Examples of arithmetic operations are shown in figure 7-4.

TABLE 7-3. COMPUTATIONAL CONVERSION FOR OPERATIONS

Operand Type	Intermediate Type		
	Arithmetic	Character	Bit
Arithmetic	Arithmetic to arithmetic conversion	Arithmetic to character conversion	Arithmetic to bit conversion
Pictured Numeric	Interpret for arithmetic value Arithmetic to arithmetic conversion	Use character value Character to character conversion	Interpret for arithmetic value Arithmetic to bit conversion
Character	Character to arithmetic conversion	Character to character conversion	Character to bit conversion
Pictured Character	Use character value Character to arithmetic conversion	Use character value Character to character conversion	Use character value Character to bit conversion
Bit	Bit to arithmetic conversion	Bit to character conversion	Bit to bit conversion

```
DCL FIXDEC FIXED DEC(10,3) INIT(1.10);
DCL FLDEC FLOAT DEC(7) INIT(.21E2);
DCL FIXBIN FIXED BIN(12,4) INIT(10.00B);
DCL FLBIN FLOAT BIN(15) INIT(1.010E2B);
DCL PICNUM PICTURE '+ZZZV.ZZ+' INIT(1.48);
.
.
PUT SKIP LIST(-FIXBIN);
/*RESULT IS -00000010.0000B          */
/*LISTING IS    -2.00                */
.
.
PUT SKIP LIST(FLDEC * 1E1);
/*RESULT IS   3.100000E+001          */
.
.
PUT SKIP LIST(FIXDEC * PICNUM);
/*RESULT IS           2.580          */
.
.
PUT SKIP LIST(FIXDEC * 35);
/*RESULT IS            38.500        */
.
.
PUT SKIP LIST(FLBIN / .1E0B);
/*RESULT IS 1.01000000000000E+003B   */
/*LISTING IS   1.0000E+001           */
.
.
PUT SKIP LIST(FIXDEC ** 3);
/*RESULT IS   1.331000000E+000       */
```

Figure 7-4. Arithmetic Operations Examples

Operand Conversion

Each operand of an arithmetic operator is converted to the appropriate arithmetic type, if necessary, before the operation occurs. The operand type to which the operand value is converted is defined by the following rules.

Arithmetic Operand

An arithmetic operand of a prefix operator requires no conversion. Arithmetic operands of an infix operator can require conversion.

For an infix arithmetic operator, conversion to a common base and common scale is necessary if the operands (after conversion to arithmetic) differ in base or scale. If base differs, the DECIMAL operand is converted to BINARY in an arithmetic to arithmetic conversion. If scale differs, the FIXED operand is converted to FLOAT in an arithmetic to arithmetic conversion. For exponentiation, conversion to a common base and scale is not always necessary.

Character String Operand

A character string operand requires character to arithmetic conversion. The operand is converted to REAL FIXED DECIMAL (14,0). An operand of an infix operator can require further base or scale conversion, as described for arithmetic operands. An operand for a prefix operator requires no further conversion.

Bit String Operand

A bit string operand requires bit to arithmetic conversion. The operand is converted to REAL FIXED BINARY (48,0). An operand of an infix operator can require further base or scale conversion, as described for arithmetic operands. An operand for a prefix operator requires no further conversion.

Pictured Operand

A pictured character item is treated as a character string operand. A pictured numeric item is interpreted for its arithmetic value and treated as a FIXED DECIMAL or FLOAT DECIMAL item, depending on the form of the picture. An operand of an infix operator can require further conversion, as described for arithmetic operands. An operand for a prefix operator requires no further conversion.

Prefix Arithmetic Operation

The result value has the mode, base, scale, and precision of the arithmetic operand (after conversion to arithmetic). The negative (-) operator causes the sign of the operand to be reversed. The positive (+) operator has no effect on the value of the operand. An example of a prefix operation is included in figure 7-4.

Infix Arithmetic Operation

The base and scale of the result value are the common base and scale of the operands. The precision depends on the arithmetic operation performed and on the precision of the operands. The following symbols are used for the description of the operations:

(m,n) or (m)	Actual precision of the result value
N	Maximum number of digits available for the result value (48 for BINARY, 14 for DECIMAL)
(p,q) or (p)	Precision of the first operand
(r,s) or (r)	Precision of the second operand

Examples of infix operations are included in figure 7-4.

Addition or Subtraction

The second operand is added to or subtracted from the first operand.

If both operands are FIXED, the precision of the result is

(m,n) = (MIN(N,MAX(p-q,r-s)+MAX(q,s)+1), MAX(q,s))

The FIXEDOVERFLOW condition can be raised during fixed point addition or subtraction.

If both operands are FLOAT, the precision of the result is

(m) = (MAX(p,r))

The OVERFLOW or UNDERFLOW condition can be raised during a floating point addition or subtraction.

Multiplication

The result value is the product of the two operands.

If both operands are FIXED, the precision of the result is

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

The FIXEDOVERFLOW condition can be raised during fixed point multiplication.

If both operands are FLOAT, the precision of the result is

(m) = (MAX(p,r))

The OVERFLOW or UNDERFLOW condition can be raised during a floating point multiplication.

Division

The first operand is divided by the second operand.

If both operands are FIXED, the precision of the quotient is the maximum available

(m,n) = (N, N-p+q-s)

The FIXEDOVERFLOW or ZERODIVIDE condition can be raised during fixed point division.

If both operands are FLOAT, the precision of the result is

(m) = (MAX(p,r))

The OVERFLOW, UNDERFLOW, or ZERODIVIDE condition can be raised during a floating point division.

Exponentiation

In exponentiation, conversion of the operands to a common base and scale is not always necessary. The following types of exponentiation can apply:

- If the first operand (after conversion to arithmetic) is FIXED BINARY (p,q) and the second operand is a positive integer constant with a value less than or equal to 49/(p+1), then a FIXED to FIXED exponentiation is performed. If the first operand (after conversion to arithmetic) is FIXED DECIMAL (p,q) and the second operand is a positive integer constant with a value less than or equal to 15/(p+1), then a FIXED to FIXED exponentiation is performed. The base, scale and mode of the result are the same as those of the first operand, after conversion. The precision of the result, where y is the value of the second operand, is

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

- The second type of exponentiation can be used if the first type does not apply. If the second operand (after conversion) is FIXED (r,s) with s equal to zero, the first operand is converted, as necessary, to FLOAT and a FLOAT to FIXED exponentiation is performed. The result is FLOAT with the base and mode of the first operand, after conversion. The precision of the result is

  (m) = (p)

- If the first two types of exponentiation do not apply, both operands are converted (as necessary) to FLOAT and a FLOAT to FLOAT exponentiation is performed. The result is FLOAT with the base and mode of the first operand, after conversion. The precision of the result is

  (m) = (MAX(p,r))

In some special cases of exponentiation, the result value can be determined without executing the operation. The result value is determined in the following way, where x represents the value of the first operand, and y represents the value of the second operand:

- For x=0 and y>0, the result is 0
- For x¬=0 and y=0, the result is 1
- For x=0 and y<=0, the ERROR condition is raised
- For x<0 and y not FIXED (r,s), with s=0, the ERROR condition is raised

STRING OPERATIONS

String operations can be specified with prefix or infix operators. The prefix operator is ¬ (NOT). The infix operators are & (AND), | (OR), and || (concatenate). The operators ¬ & and | are for bit strings. The operator || is for character strings or bit strings. For string operations, the operands can be character string, bit string, arithmetic, or pictured items. Operands can also be intermediate values that are result values from subexpression evaluations. The result of the operation is a bit string or character string with a specific length. The result can be used as an operand in a subsequent operation or can be the final result of expression evaluation, depending on the circumstances. Examples of string operations are shown in figure 7-5.

```
DCL CHAR CHARACTER(4) INIT('K');
DCL CHARVAR CHAR(6) VARYING INIT('LMN');
DCL PICSTR PICTURE 'AA99' INIT('AB33');
DCL BIT BIT(5) INIT('11011'B);
DCL BITVAR BIT(8) VARYING INIT('1000'B);
.
.
/*NOT OPERATION                          */
PUT SKIP LIST(¬BIT);
/*RESULT IS  '00100'B                    */
.
.
/*AND OPERATION                          */
PUT SKIP LIST(BIT & BITVAR);
/*RESULT IS  '10000'B                    */
.
.
/*OR OPERATION                           */
PUT SKIP LIST(BITVAR | BIT);
/*RESULT IS  '11011'B                    */
.
.
/*CONCATENATE OPERATION                  */
PUT SKIP LIST(CHAR || CHARVAR);
/*RESULT IS  'K   LMN'                   */
.
.
PUT SKIP LIST(CHARVAR || PICSTR);
/*RESULT IS  'LMNAB33'                   */
```

Figure 7-5. String Operations Examples

Operand Conversion

For bit string operations, each operand is converted, as necessary, to a bit string. For the operator || (concatenate), both operands are converted, as necessary, to character strings unless both are bit strings.

Character String Operand

A character string operand in a bit string operation requires character to bit conversion. The operand is converted to a bit string with the current length of the character string. A character string operand in a concatenate operation requires no conversion.

Bit String Operand

A bit string operand in a prefix or infix bit string operation requires no conversion. Unless both operands are bit strings, a bit string in a concatenate operation requires bit to character conversion. The operand is converted to a character string with the current length of the bit string.

Pictured Operand

A pictured character or pictured numeric operand in a bit string operation requires character to bit conversion on the character value of the pictured item. A pictured character or pictured numeric operand in a concatenate operation is used for its character value.

Arithmetic Operand

An arithmetic operand in a bit string operation requires arithmetic to bit conversion. An arithmetic operand in a concatenate operation requires arithmetic to character conversion. The length of the bit string or character string depends on the precision of the arithmetic operand.

Prefix Bit String Operation

The result value of the prefix operator ¬ (NOT) is the complement of the bit string operand (after conversion to bit). At each bit position, the corresponding bit of the result is set to '1'B if the operand has '0'B, or set to '0'B if the operand has '1'B. The length of the result is the length of the operand. An example of a prefix operation is included in figure 7-5.

Infix Bit String Operation

The result value of the bit string operator & (AND) or | (OR) is a bit string with a length that is the greater of the current lengths of the two operands. Before the operation occurs, the shorter operand is padded on the right with zero bits to the length of the longer one. If either operand has the VARYING attribute, the infix expression can yield a string of a different length each time it is evaluated. Examples of infix bit string operations are included in figure 7-5.

Logical AND

For the infix & operator, a logical AND operation is performed on the strings, bit by bit. At any bit position, the corresponding bit of the result is set to '1'B if both operands have a '1'B. The corresponding bit of the result is set to '0'B if either operand has '0'B or both operands have '0'B.

Logical OR

For the infix | operator, a logical OR operation is performed on the strings, bit by bit. At any bit position, the corresponding bit of the result is set to '1'B if either operand has '1'B or both operands have '1'B. The corresponding bit of the result is set to '0'B if both operands have '0'B.

Concatenate Operation

The result value of the infix operator || (concatenate) is a character string if the two operands are character strings. The result value is a bit string if the two operands are bit strings. The resulting length is the sum of the lengths of the two character string or bit string operands (after conversion). The second string is concatenated with the first so that its first character (or bit) immediately follows the last character (or bit) of the first string. If one of the operands has a VARYING attribute, the length of the result can be different each time the operation is performed. Examples of the concatenate operation are included in figure 7-5.

COMPARISON OPERATIONS

Comparison operations are all infix operations. The comparison operators are = (equal), ¬ = (not equal), < (less than), ¬< (not less than), < = (less than or equal), > (greater than), ¬> (not greater than), and > = (greater than or equal). Examples of comparison operations are shown in figure 7-6.

```
DCL FIXDEC FIXED DEC(10,3) INIT(1.10);
DCL FLBIN FLOAT BIN(15) INIT(1.010E2B);
DCL CHAR CHARACTER(4) INIT('K');
DCL CHARVAR CHAR(6) VARYING INIT('LMN');
.
.
PUT SKIP LIST(FIXDEC > 1);
/*RESULT IS  '1'B                          */
.
.
PUT SKIP LIST(FLBIN >= 111E0B);
/*RESULT IS  '0'B                          */
.
.
PUT SKIP LIST(CHAR > CHARVAR);
/*RESULT IS  '0'B                          */
```

Figure 7-6. Comparison Operations Examples

Operand Conversion

No conversion occurs if both operands have the same data type. Any necessary conversions are done to achieve a common data type.

Arithmetic values can be compared. If at least one operand is arithmetic or pictured numeric, the common type is arithmetic. One or both operands are converted to arithmetic and additionally converted to a common base and scale in the same way as for infix arithmetic operations. A pictured numeric item is interpreted for its arithmetic value.

String values can be compared. If one operand is a character string and the other is a bit string, the common type is character. The bit string operand therefore requires a bit to character conversion. A pictured character item is used for its character value.

Locator values can be compared. If one operand is a pointer value and the other an offset value, pointer is the common type. The offset value is therefore converted to pointer in an offset to pointer conversion.

Area values cannot be compared. Label, format, entry, and file values cannot be compared.

Infix Comparison Operation

The result value of a comparison operation is expressed as the truth of the comparison. If the comparison is true, the result value is '1'B. If the comparison is false, the result value is '0'B.

Arithmetic comparison is an algebraic comparison of arithmetic values.

NOTE

Every floating point value is an approximation. Floating point values are not rounded or truncated to the precision (p) declared for a target variable. The program should not compare two FLOAT values for precise equality.

Character string comparisons are performed character-by-character from left to right. Comparison stops when the first nonmatching pair is encountered. If the strings are not the same length before the comparison begins, the shorter string is padded on the right with blanks so that it is the same length as the other. For all comparisons involving greater-than or less-than relations, the collating sequence is used. In the collating sequence, a blank is less than any nonblank character, special characters are less than alphabetic characters, and alphabetic characters are less than numeric characters. ('A' is less than 'B'..., and '1' is less than '2'...) Two null strings are equal. The collating sequence appears in appendix A.

Bit string comparison involves a bit-by-bit comparison of the two strings from left to right. If one string is shorter than the other, the shorter string is padded on the right with zero bits to the length of the longer string before comparison begins. For all greater-than and less-than tests, a '1'B is greater than a '0'B. Two null strings are equal.

Pointer values are equal if they are both null, or if both identify the same generation. Pointer values can only be compared with the = and ¬= operators. Offset values are equal if they are both null or they both represent the same displacement from the beginning of an area. The attributes of the BASED generations represented by the offset values are not compared. Offset values are also equal if they represent the same displacement from the beginning of different areas. Offset values can only be compared with the = and ¬= operators.

ASSIGNMENT

The assignment operation assigns a source value to a target. Any previous value of the target is replaced with the source value. The source and the target value must be scalar. The target value can be a scalar pseudovariable. The source or the target can be an array element or structure member but cannot be an array or structure. A subscripted source or target reference cannot have any asterisk subscripts.

Assignment occurs in the following circumstances:

- Assignment statement execution
- GET statement execution with LIST or EDIT option
- PUT statement execution with STRING option
- Assignment of a locator value to a locator variable during ALLOCATE, READ, or LOCATE statement execution
- Assignment of a value to a KEYTO option variable during READ statement execution
- Assignment of a value to the target-variable in a REFER option during allocation of a BASED structure
- Assignment of initial values according to the INITIAL attribute during allocation of a STATIC, AUTOMATIC, BASED, or CONTROLLED variable
- Assignment of a value to the index during indexed DO execution or during execution of an embedded-do in a GET or PUT statement

If the source value has a different data type than the target, conversion of the source value is necessary. No conversion is necessary if the data type of the source value matches the data type of the target. The source value is assigned to the target without conversion. If conversion is necessary, the source value is converted to an intermediate value that is assigned to the target. If conversion is necessary, the data types must be compatible. Source type and target type are compatible if both are in one of the following categories:

- Computational (arithmetic, character, bit, or picture)
- Locator (pointer or offset)
- Label
- Area

COMPUTATIONAL ASSIGNMENT

A computational source value must be assigned only to a computational target. The source value and the target value can be arithmetic, character string, bit string, pictured string, or pictured numeric. Any computational value can be converted to any other type of computational value, as shown in table 7-4. Computational conversion produces an intermediate value with computational data type attributes identical to the data type attributes of the target value. For assignment of an arithmetic source value to a target, conversion is required even if mode, base, and scale match but precision does not match. For assignment of a character or bit source value to a target with the same type, conversion is required simply to adjust the length. For conversion from a pictured source value, the pictured source value is a character value or is interpreted for the arithmetic value, depending on the pictured item and the circumstances. For assignment to a pictured target value, an operation to validate through the picture or edit through the picture is necessary. Picture-controlled conversions are described later in this section.

LOCATOR ASSIGNMENT

A locator source value must be assigned only to a locator target value. A pointer source value is directly assigned to a pointer target value. An offset source value is directly assigned to an offset target variable, without regard for any area identified in the declaration of the offset target variable. The offset source value represents a displacement from the beginning of an area. Conversion is possible from pointer to offset or offset to pointer.

LABEL ASSIGNMENT

A label source value must be assigned only to a label target value. A label source value is assigned directly to a label target value. The label source value represents an address.

TABLE 7-4. COMPUTATIONAL CONVERSION FOR ASSIGNMENT

Source Type	Target Type				
	Arithmetic	Pictured Numeric	Character	Pictured Character	Bit
Arithmetic	Arithmetic to arithmetic conversion† Assign	Arithmetic to arithmetic conversion† Edit through picture and assign	Arithmetic to character conversion Assign	Arithmetic to character conversion Validate through picture and assign	Arithmetic to bit conversion Assign
Pictured Numeric	Interpret for arithmetic value Arithmetic to arithmetic conversion† Assign	Interpret for arithmetic value Arithmetic to arithmetic conversion† Edit through picture and assign	Use character value Character to character conversion† Assign	Use character value Character to character conversion† Validate through picture and assign	Interpret for arithmetic value Arithmetic to bit conversion Assign
Character	Character to arithmetic conversion Assign	Character to arithmetic conversion Edit through picture and assign	Character to character conversion† Assign	Character to character conversion† Validate through picture and assign	Character to bit conversion Assign
Pictured Character	Use character value Character to arithmetic conversion Assign	Use character value Character to arithmetic conversion Edit through picture and assign	Use character value Character to character conversion† Assign	Use character value Character to character conversion† Validate through picture and assign	Use character value Character to bit conversion Assign
Bit	Bit to arithmetic conversion Assign	Bit to arithmetic conversion Edit through picture and assign	Bit to character conversion Assign	Bit to character conversion Validate through picture and assign	Bit to bit conversion† Assign

†Not necessary for an arithmetic value if mode, base, scale, and precision are already as required. Not necessary for a string value if the length is already as required.

AREA ASSIGNMENT

An area source value must be assigned only to an area target value. The area target variable must be of sufficient size to contain all based generations presently allocated in the source area. After assignment, all offset variables valid for the source area are valid for the target area. If the target area contains any allocated generations, assignment to the target area destroys all previously allocated generations and invalidates all locator values that previously addressed the allocated generations. The area value of the source area is dependent on the order of allocation and freeing of generations in the source area. If the target area cannot contain all allocated generations in the source area at the same offset positions, the AREA condition is raised.

CONVERSIONS

Conversion of a value is necessary when the data type attributes are not appropriate for the context in which the value is used. Conversion occurs automatically in a number of circumstances. In conversion, a source value is converted to an intermediate value.

During expression evaluation, any operand can be converted as necessary to a computational type (arithmetic, bit string, or character string) appropriate for the operator in a prefix or infix expression. If the operands of an infix arithmetic operator are not the same data type, one operand is converted or both operands are converted to a common arithmetic data type. Conversion of operands to a common

base and scale does not apply to the exponentiation operation. Operands in an infix comparison operation are converted as necessary to a common data type. Individual cases of required conversion are described earlier in this section for specific arithmetic, string, or comparison operations.

For assignment, if the source value does not have the same data type as the target, the source value is converted to the data type of the target before assignment. Assignment is described earlier in this section. Assignment can involve conversion of the source value from one data type to another.

When an argument is passed to a parameter during CALL statement execution or evaluation of a function reference (including a builtin function reference), conversion can be required for the argument.

When a RETURN statement is executed, conversion is required if the value of the expression specified by the statement does not have the data type specified by the RETURNS attribute associated with the entry name used to invoke the procedure.

Conversion to integer is performed as necessary in evaluation of an expression used in a number of program components. The intermediate value has the attributes REAL FIXED BINARY (17,0). The conversion used depends on the result type of the evaluated expression. Conversion to integer is done as necessary for the following program components:

- LINESIZE, PAGESIZE, SKIP, and LINE options of I/O statements for stream files
- Subscripts in a subscripted reference
- Extents that specify area sizes, string lengths, and array bounds
- IGNORE option of a READ statement
- Any POSITION attribute declared for a DEFINED variable
- Certain argument expressions in a SUBSTR builtin function or pseudovariable reference
- Iteration factors used in INITIAL attributes and in format specifications

Conversion to character string is performed as necessary in evaluation of an expression in a number of program components. The intermediate value has the attributes CHARACTER and nonvarying. The exact conversion procedure used depends on the result type of the expression. Conversion to character string is done as necessary for the following program components:

- The STRING option of a GET statement. The length of the intermediate string is determined by the value of the input-source expression.
- Values transmitted to the output stream during PUT statement execution. Length is appropriate for the source value (LIST option) or specified with a format item (EDIT option).
- The KEY option of a READ, REWRITE, or DELETE statement, or the KEYFROM option of a WRITE or LOCATE statement. The length of the target string is determined by the length of the key for the file.

Conversion to bit string is performed as necessary in evaluation of an expression in a number of program components. The intermediate value has the attributes BIT and nonvarying. The length of the intermediate string is determined by the expression value. The exact conversion procedure used depends on the data type of the expression. Conversion to bit string is done as necessary for the following program components:

- Any WHILE option in a DO statement or in an embedded-do in a GET or PUT statement
- The expression in an IF statement

The conversion of any computational value to another computational data type is possible. The computational conversions are the following:

- Arithmetic, character, or bit to arithmetic
- Arithmetic, character, or bit to character
- Arithmetic, character, or bit to bit
- Validate through picture, edit through picture, and interpret for arithmetic value as described later in this section for pictured items

The locator conversions are the following:

- Pointer to offset
- Offset to pointer

If conversion is required and the required conversion is not defined (not legal), the program is in error and a fatal compile-time diagnostic message is issued. The description of each conversion describes the conditions that can be raised during the conversion.

ARITHMETIC TO ARITHMETIC CONVERSION

The mode, base, scale, and precision of the source arithmetic value are known. For conversion during assignment, the required base, scale, and precision of the intermediate value are known. For operand conversion, the required base and scale are known. For operand conversion, the precision is calculated as shown in table 7-5. Operand conversion does not require any FLOAT to FIXED conversion.

The intermediate value resulting from the conversion has the mode REAL and the required base and scale. The intermediate has the required precision or the calculated precision.

The SIZE condition is raised if the target is FIXED with a precision insufficient to represent the integral value of the source. The OVERFLOW or UNDERFLOW condition can be raised in a FLOAT DECIMAL to FLOAT BINARY conversion or FIXED DECIMAL to FLOAT BINARY conversion.

NOTE

A FIXED DECIMAL variable or literal constant with noninteger value and small positive value of precision q should be avoided in an operation that causes conversion to FIXED BINARY. For example, the expression (.1 + 0B) has the result value 00.0001B (0.0625) because of the precision rules. The expression (.1000 + 0B) has the result value 00.0001100110101B (approximately 0.0999435).

Examples of arithmetic to arithmetic conversion are shown in figure 7-7.

TABLE 7-5. PRECISION IN ARITHMETIC TO ARITHMETIC CONVERSION

Source Type	Required Intermediate Type	Resulting Precision of Intermediate
FIXED BINARY (p,q)	FIXED BINARY	(p, q)
FIXED DECIMAL (p,q)	FIXED BINARY	(MIN(CEIL(p*3.32)+1,48), CEIL(q*3.32))
FIXED BINARY (p,q)	FIXED DECIMAL	(MIN(CEIL(p/3.32)+1,14), CEIL(q/3.32))
FIXED DECIMAL (p,q)	FIXED DECIMAL	(p, q)
FIXED BINARY (p,q)	FLOAT BINARY	(MIN(p,48))
FIXED DECIMAL (p,q)	FLOAT BINARY	(MIN(CEIL(p*3.32),48))
FLOAT BINARY (p)	FLOAT BINARY	(p)
FLOAT DECIMAL (p)	FLOAT BINARY	(MIN(CEIL(p*3.32),48))
FIXED BINARY (p,q)	FLOAT DECIMAL	(MIN(CEIL(p/3.32),14))
FIXED DECIMAL (p,q)	FLOAT DECIMAL	(MIN(p,14))
FLOAT BINARY (p)	FLOAT DECIMAL	(MIN(CEIL(p/3.32),14))
FLOAT DECIMAL (p)	FLOAT DECIMAL	(p)

```
DCL FIXDEC FIXED DECIMAL;
DCL FLDEC FLOAT DECIMAL;
DCL FIXBIN FIXED BINARY;
DCL FLBIN FLOAT BINARY;
•
•
FIXDEC = 12.0E1;
PUT SKIP LIST(FIXDEC);
/*RESULT IS        120                    */
•
•
FLDEC = 5;
PUT SKIP LIST(FLDEC);
/*RESULT IS    5.0000000000000E+000       */
•
•
FLBIN = 65.0E-1;
PUT SKIP LIST(FLBIN);
/*RESULT IS 1.000001000E+006B             */
/*LISTING IS    6.5000000000000E+000      */
•
•
FIXBIN = 25;
PUT SKIP LIST(FIXBIN);
/*RESULT IS 00011001B                     */
/*LISTING IS         25                   */
```

Figure 7-7. Arithmetic to Arithmetic Conversion Examples

CHARACTER TO ARITHMETIC CONVERSION

The length of the source character string is known. The source string is expected to contain an arithmetic constant, optionally preceded by a sign. The constant can be preceded and followed by any number of blanks. If the source string consists entirely of blanks, or if the source string is null, the value is zero. The intermediate value resulting from the conversion has the mode REAL. An arithmetic to arithmetic conversion is used to convert the arithmetic constant to the required base, scale, and precision. The required precision is not known if the conversion was caused by an operator in an expression. In that case, the precision for the conversion becomes (14) or (14,0) for DECIMAL, (48) or (48,0) for BINARY.

The CONVERSION condition is raised if the source string does not contain a valid arithmetic constant or all blanks.

If conversion from character string to arithmetic involves an arithmetic precision with q=0, any fractional digits are lost.

Examples of character to arithmetic conversion are shown in figure 7-8.

```
DCL FIXDEC FIXED DECIMAL;
DCL FIXBIN FIXED BINARY;
•
•
FIXDEC = ↑125.3↑;
PUT SKIP LIST(FIXDEC);
/*RESULT IS        125                    */
•
•
FIXBIN = ↑↑101↑;
PUT SKIP LIST(FIXBIN);
/*RESULT IS 000000001100101B              */
/*LISTING IS        101                   */
```

Figure 7-8. Character to Arithmetic Conversion Examples

BIT TO ARITHMETIC CONVERSION

The length of the source bit string is known. The bit string is considered to be a positive binary integer with a length of 48 bits. If the string length is greater than 48, only the rightmost 48 bits are used. If the source string is null, the value is zero. The intermediate value resulting from the conversion has the mode REAL. An arithmetic to arithmetic conversion is used to convert the FIXED BINARY integer to the required base, scale, and precision. The required precision is not known if the conversion was caused by an operator in an expression. In that case, the precision becomes (14) or (14,0) for DECIMAL, (48) or (48,0) for BINARY.

Examples of bit to arithmetic conversion are shown in figure 7-9.

```
DCL FIXDEC FIXED DECIMAL;
DCL FLBIN FLOAT BINARY;
.
.
FIXDEC = '1'B;
PUT SKIP LIST(FIXDEC);
/*RESULT IS          1                    */
.
.
FLBIN = '1101'B;
PUT SKIP LIST(FLBIN);
/*RESULT IS 1.101000...  000E+003B      */
/*LISTING IS   1.3000000000000E+001     */
```

Figure 7-9. Bit to Arithmetic Conversion Examples

ARITHMETIC TO CHARACTER CONVERSION

The mode, base, scale, and precision of the arithmetic source value are known. An arithmetic to arithmetic conversion is used, if necessary, to convert the source value to a DECIMAL value. The DECIMAL value is then converted to a character string. Conversion to character string is described in terms of picture-controlled conversion. Conversion is performed in one of the following ways to generate the result string, S:

- Value FIXED DECIMAL with q=0: The length of S is p+3. The string value is derived as if under the control of the picture

  'BB(p)-9'

- Value FIXED DECIMAL with p>=q and q>0: The length of S is p+3. The string value is derived as if under the control of the picture

  'B(i)-9V.(q)9' where repetition factor i=p-q and q controls the number of digits in the fractional part

- Value FIXED DECIMAL with p=q and q>0: The length of S is p+4. The string value is derived as if under control of the picture

  '-9V.(q)9'

- Value FIXED DECIMAL with p<q or with q<0: The length of S is p+3+k, where k is the number of digits needed to represent the value of q. A partial string value is derived as if under the control of the picture

  '(p)-9' where the value is multiplied by (10**q) before being edited through the picture

  The partial string is completed with the addition of an F followed by a signed decimal integer whose value is -q.

- Value FLOAT DECIMAL: The length of S is p+7. The string value is derived as if under the control of the picture

  '-9V.(i)9ES999' where repetition factor i=p-1

If the required length is known, the length of string S is adjusted to the target length. The string S is padded on the right with blanks or truncated on the right to form the intermediate value.

Examples of arithmetic to character conversion are shown in figure 7-10.

```
DCL CHAR CHARACTER(8) VARYING;
.
.
CHAR = -10;
PUT SKIP LIST(CHAR);
/*RESULT IS '  -10'                    */
/*LISTING IS    -10                    */
.
.
CHAR = 11.1B;
PUT SKIP LIST(CHAR);
/*RESULT IS '  3.5'                    */
/*LISTING IS    3.5                    */
```

Figure 7-10. Arithmetic to Character Conversion Examples

CHARACTER TO CHARACTER CONVERSION

The length of the source character string and the length of the target character string are known. The source string is adjusted to the length of the target string. The source string is padded on the right with blanks or truncated on the right to form the intermediate value.

Examples of character to character conversion are shown in figure 7-11.

```
DCL CHAR CHARACTER(8);
.
.
CHAR = 'G';
PUT SKIP LIST(CHAR);
/*RESULT IS 'G       '                 */
/*LISTING IS  G                        */
.
.
CHAR = 'A12B12C12D12';
PUT SKIP LIST(CHAR);
/*RESULT IS 'A12B12C1'                 */
/*LISTING IS  A12B12C1                 */
```

Figure 7-11. Character to Character Conversion Examples

BIT TO CHARACTER CONVERSION

The length of the source bit string is known. The source string is converted to a character string by changing each '0'B bit of the source to a '0' character and each '1'B bit to a '1' character. If the required length is not known, the changed string is the intermediate value.

If the required length is known, the length of the string is adjusted to the target length. The string is extended on the right with blanks or truncated on the right to form the intermediate value.

An example of bit to character conversion is shown in figure 7-12.

```
DCL CHAR CHARACTER(8);
•
•
CHAR = ↑1101↑B;
PUT SKIP LIST(CHAR);
/*RESULT IS ↑1101    ↑                    */
/*LISTING IS  1101                        */
```

Figure 7-12. Bit to Character Conversion Example

ARITHMETIC TO BIT CONVERSION

The mode, base, scale, and precision of the source arithmetic value are known. The absolute value of the source is taken. An arithmetic to arithmetic conversion is used, if necessary, to convert the source value to a FIXED BINARY value with the precision shown in table 7-6. The FIXED BINARY integer is interpreted as a bit string with a length that is the precision p. If the required length is not known, the intermediate value is the bit string.

TABLE 7-6. PRECISION IN ARITHMETIC TO BIT CONVERSION

Source Type	Required Precision
FIXED BIN (p,q)	(MIN(48,MAX(p-q,0)), 0)
FIXED DEC (p,q)	(MIN(14,MAX(CEIL((p-q)*3.32),0)), 0)
FLOAT BIN (p)	(MIN(48,p), 0)
FLOAT DEC (p)	(MIN(14,CEIL(p*3.32)), 0)

If the required length is known, the length of the bit string is adjusted to the target length. The bit string is padded on the right with '0'B bits or truncated on the right to form the intermediate value.

Examples of arithmetic to bit conversion are shown in figure 7-13.

```
DCL BIT BIT(10);
•
•
BIT = 1010B;
PUT SKIP LIST(BIT);
/*RESULT IS ↑1010000000↑B                 */
•
•
BIT = 33;
PUT SKIP LIST(BIT);
/*RESULT IS ↑0100001000↑B                 */
```

Figure 7-13. Arithmetic to Bit Conversion Examples

CHARACTER TO BIT CONVERSION

The length of the source character string is known. The source string must consist of '0' and '1' characters only. The source string is converted to a bit string by changing each '0' character to a '0'B bit and each '1' character to a '1'B bit. If the required length is not known, the changed string is the intermediate value.

If the required length is known, the length of the bit string is adjusted to the target length. The string is padded on the right with '0'B bits or truncated on the right to form the intermediate value.

The CONVERSION condition is raised if the source string contains any characters other than '0' or '1' characters.

An example of character to bit conversion is shown in figure 7-14.

```
DCL BIT BIT(10);
•
•
BIT = ↑01001↑;
PUT SKIP LIST(BIT);
/*RESULT IS ↑0100100000↑B                 */
```

Figure 7-14. Character to Bit Conversion Example

BIT TO BIT CONVERSION

The length of the source bit string and the length of the target bit string are known. The source string is adjusted to the length of the target string. The source string is padded on the right with '0'B bits or truncated on the right to form the intermediate value.

Examples of bit to bit conversion are shown in figure 7-15.

```
DCL BIT BIT(10);
•
•
BIT = ↑110110↑B;
PUT SKIP LIST(BIT);
/*RESULT IS ↑1101100000↑B                 */
•
•
BIT = ↑1111110110↑B;
PUT SKIP LIST(BIT);
/*RESULT IS ↑1111110110↑B                 */
```

Figure 7-15. Bit to Bit Conversion Examples

POINTER TO OFFSET CONVERSION

A pointer value can be converted to an offset value that identifies the same BASED generation stored within the same area. The program is in error unless the offset was declared with the area reference, or the conversion occurs during evaluation of the OFFSET builtin function where the area reference is supplied as the second argument.

OFFSET TO POINTER CONVERSION

An offset value that identifies a BASED generation allocated within an area can be converted to a pointer value that identifies the same generation. The program is in error unless the offset was declared with the area reference, or the conversion occurs during evaluation of the POINTER builtin function where the area is identified by the second argument.

PICTURE-CONTROLLED CONVERSIONS

Picture-controlled conversions apply to pictured variables and to input/output operations that involve picture specifications. Pictured variables are declared with the PICTURE attribute, as described in section 4, Attributes. Stream I/O operations can be performed with the P format item, as described in section 8, Input/Output. Specification of the PICTURE attribute or the P format item involves the use of picture codes. Syntax of the picture specification is shown in figure 7-16. The following discussion provides the specific rules for using the picture codes.

A picture can be specified for a pictured character, pictured numeric fixed point, or pictured numeric floating point item. Each pictured character item has a character value. Each pictured numeric item has a character value and also a decimal arithmetic value that can be fixed point or floating point. Any pictured variable is stored as a character value. Maximum length of the character value is 1000 characters. If the arithmetic value of any pictured numeric variable is required, the arithmetic value is interpreted from the present character value.

Picture codes can be specified with repetition factors. Each repetition factor is an unsigned decimal integer enclosed in parentheses. The repetition count applies to the picture code immediately to the right of the repetition factor. If the repetition factor is zero, the picture code is ignored rather than repeated. For instance, '(5)Z99' is equivalent to 'ZZZZZ99' and '(0)Z99' is equivalent to '99'.

PICTURED CHARACTER

A pictured character item has a character value that can contain characters, digits, or blanks. The picture codes for a pictured character item specify the type of character that can appear in each position. The available codes are shown in table 7-7.

TABLE 7-7. CODES FOR PICTURED CHARACTER

Category	Code	Use
Character codes	9	Digit or blank
	A	Letter A-Z or blank
	X	Any character

Validate through Picture

A character value that is assigned to a pictured character variable or written out through a P format item is validated through the picture. The character value must match the picture. If an indicated character does not appear, the CONVERSION condition is raised.

Character Codes

The A code indicates any letter (A through Z) or blank. The X code indicates any character. The 9 code indicates any digit or blank. The picture specification for each pictured character item must contain at least one A or X code. The total number of A, X, and 9 codes establishes the number of characters in the pictured character item.

PICTURED NUMERIC FIXED POINT

A pictured numeric fixed point item has a character value representing a FIXED DECIMAL value on which editing has been performed. The character value contains the decimal digits and all edited characters including insertion characters. The character value does not include the assumed decimal point but can include inserted decimal points.

Assignment of an arithmetic value to a pictured numeric fixed point variable can raise the SIZE condition. SIZE is raised if editing the value through the picture specification causes loss of significant digits on the left. SIZE is also raised if the arithmetic value is negative but the picture does not include a sign indication.

The picture codes used for pictured numeric fixed point items can be grouped by function. The available picture codes are shown in table 7-8. A pictured numeric fixed point item consists of one field. If the assumed decimal point (picture code V) is used, the assumed decimal point divides the picture into an integer subfield and a fractional subfield.

TABLE 7-8. CODES FOR PICTURED NUMERIC FIXED POINT

Category	Code	Use
Digit code	9	Digit
	Z	Leading zero suppression digit
	*	Leading zero replacement digit
	Y	Zero suppression digit
Decimal point code	V	Assumed decimal point
Sign position codes	S	Sign + or - (can drift)
	+	Sign if + (can drift)
	-	Sign if - (can drift)
Signed digit codes	T	Overpunch digit with + or -
	I	Overpunch digit if +
	R	Overpunch digit if -
Sign suffix codes	CR	Credit indicator
	DB	Debit indicator
Currency code	$	Dollar sign (can drift)
Insertion codes	/	Inserted slash
	,	Inserted comma
	.	Inserted period
	B	Inserted blank
Scaling factor code	F	Scaling factor for arithmetic value

The picture specification for a fixed point value must contain at least one code that represents a digit (9 Z * Y T I or R) in the field. The maximum number of digits that can be represented in the entire field is 14. Only one assumed decimal point code (V) can be specified. In the entire field, the sign can be represented only once with a sign position code (S + or -), a signed digit code (T I or R), or

{pictured-character / pictured-numeric-fixed / pictured-numeric-float}

where pictured-character must include at least one A or X code and is

{9 / A / X} ...

where pictured-numeric-fixed must include at least one of the codes 9 Z * Y T I or R, must include no more than one sign code S + - T I R CR or DB, can include insertion codes / , . or B at any position, and is

{nondrifting-field / drifting-field} [F([+/-] integer)]

where nondrifting-field is

{ [$] • [S/+/-] • digits
  [$] • digits {CR/DB} }

where digits is

{ [Z ... / * ...] [digit-pos] ... [V] [digit-pos] ...
  [Z ... / * ...] V Z ... / V * ... }

where digit-pos is

{9 / Y / T / I / R}

where drifting-field is

{ [$] • drifting-sign
  [S/+/-] • { $$ ... [digit-pos] ... [V] [digit-pos] ... / $ [$ ...] V $ ... }
  { $$ ... [9/Y] ... [V] [9/Y] ... / $ [$ ...] V $ ... } {CR/DB} }

where drifting-sign is

{SS ... / ++ ... / - - ...} [9/Y] ... [V] [9/Y] ...

where pictured-numeric-float is

mantissa {E/K} exponent

where mantissa must include at least one of the codes 9 Z * Y T I or R, must include no more than one sign code S + - T I or R, can include insertion codes / , . or B at any position, and is

{ [S/+/-] digits
  drifting-sign }

and where exponent must include at least one of the codes 9 Z * Y T I or R, must include no more than one sign code S + - T I or R, can include insertion codes / , . or B at any position, and is

[S/+/-] { digit-pos ...
  {Z ... / * ...} [digit-pos] ... }

Figure 7-16. Picture Specification Syntax

a sign suffix code (CR or DB). A sign position code can be used in a drifting sequence. In the field, a currency code ($) can be used only once and can be used in a drifting sequence. The insertion codes (/ , . or B) cause insertion of a slash, comma, period, or blank anywhere in the field. Any combination and number of insertion characters can be used. The scaling factor code (F) can be used to specify that the arithmetic value is to be scaled. Each picture code for numeric fixed point items is described by category.

Edit through Picture

An arithmetic value that is assigned to a pictured numeric variable or written out through a P format item is edited through the picture specification. The arithmetic value is used to create a character value that can include digits, blanks, a sign indication, a dollar sign, and inserted characters. The created character value depends on the picture codes specified. The picture codes indicate a character value created from a FIXED DECIMAL arithmetic value.

Interpret for Arithmetic Value

A pictured numeric is stored as a character value and can be interpreted for the arithmetic value when the arithmetic value is required for the purpose of operations or assignment. The character value is interpreted for the arithmetic value when an input value is read in through a P format item. The arithmetic value is FIXED DECIMAL with a precision indicated in the picture specification.

Digit Codes

The digit codes specify character positions in which a decimal digit can appear. The digit codes are the following:

- 9 indicates a digit position to hold a digit. The position can hold a leading zero as necessary.
- Z indicates leading zero suppression. When the position would hold a leading zero for the number, the zero is replaced with a blank. When the position would hold a significant digit, the Z code acts as a 9 code.
- * indicates leading zero replacement. When the position would hold a leading zero for the number, the zero is replaced with an asterisk. When the position would hold a significant digit, the * code acts as a 9 code.
- Y indicates zero suppression. When any position of a number would hold a zero digit, the zero is replaced with a blank. Effectively, the Y code unconditionally replaces a zero with a blank. When the position would hold any nonzero digit, the Y code acts as a 9 code.

The 9 code is used for each position that must hold a digit. Leading zeros in the integer subfield and trailing zeros in the fractional subfield are represented as zeros.

The Z code represents a digit position. The Z code suppresses any leading zero and replaces it with a blank. The Z code cannot be used in the same field as the * code. The Z code cannot be used in the same field as a drifting sequence of S + - or $ codes. The Z code also cannot be used to the right of any 9 Y T I or R code. Usually, a number of Z codes precede a number of 9 codes in the integer subfield. If a Z code appears in any position in the fractional subfield, all digit positions in the integer and fractional subfields must contain the Z code. When all integer and fractional digit positions contain the Z code, suppression of leading zeros in the fractional subfield is not performed unless all positions in the number are zero digits. If all positions are zero digits, the character value of the entire picture is a string of blanks.

The * code represents a digit position. The * code replaces any leading zero with an asterisk. The * code cannot be used in the same field as the Z code. The * code cannot be used in the same field as a drifting sequence of S + - or $ codes. The * code also cannot be used anywhere to the right of the code 9 Y T I or R. Usually, a number of * codes precede a number of 9 codes in the field. If a * code appears in any position in a fractional subfield, all digit positions in the integer and fractional subfields must contain the * code. When all integer and fractional digit positions contain the * code, replacement of leading zeros in the fractional subfield is not performed unless all positions in the number are zero digits. If all positions are zero digits, the character value of the entire picture is a string of asterisks.

The Y code represents a digit position but suppresses any zero by replacing it with a blank. The Y code can appear in any position where unconditional replacement of a zero is required.

Examples of digit codes are shown in figure 7-17.

Arithmetic Value	Picture Specification	Character Value
120	999	120
2	999V	002
1.34	999V	001
5	ZZ9	ΔΔ5
19.65	ZZ9V	Δ19
1256	ZZZZ	1256
0	(4)Z	ΔΔΔΔ
6	***9	***6
2.5	**V**	*250
570	YYYY	Δ57Δ
45.0578	YYYVY	Δ45Δ
10005	YYYYYY	Δ1ΔΔΔ5

Figure 7-17. Digit Codes and Decimal Point Code Examples

Decimal Point Code

The decimal point code is the following:

- V specifies the assumed location of the decimal point in the arithmetic value. The V code separates the integer and fractional subfields. The V code does not specify insertion of an actual decimal point character.

The V code cannot appear more than once in the field. If no V code appears, a V code is assumed at the right end of the field. The position of the V code establishes digit codes to the left as being in the integer subfield and digit codes to the right as being in the fractional subfield. Examples of the V code are included in figure 7-17.

Sign Position Codes

The sign position codes represent a sign position in the character value of the picture. The sign position codes are the following:

- S indicates a sign represented as + (value>=0) or - (value<0). A sequence of S codes represents a drifting + or -.
- + indicates a sign represented as + (value>=0) or blank. A sequence of + codes represents a drifting +.
- - indicates a sign represented as - (value<0) or blank. A sequence of - codes represents a drifting -.

Each picture can have only one code for the sign. The sign code can be a sign position code (S + or -) or one of the other sign codes (T I R CR or DB). A sign position code must appear to the left or right of all digit codes in the field. Examples of sign position codes are shown in figure 7-18.

Arithmetic Value	Picture Specification	Character Value
1367	S99999	+01367
5.0	999S	005+
-12.0	+9999	Δ0012
-3	9999-	0003-
553	SSSSS99	ΔΔΔ+553
-152	——9	ΔΔ-152
135	T99	A35
135	9I9	1C5
135	99R	135
-135	99R	13N
1250	(6)ZCR	ΔΔ1250ΔΔ
-8841	ZZZZZZCR	ΔΔ8841CR
145	ZZZZZDB	ΔΔ145ΔΔ

Figure 7-18. Sign Codes Examples

The sign position code can be static or drifting. Static use reserves one position in the character value for the sign associated with the arithmetic value of the pictured numeric item. Drifting use involves a sequence of occurrences of the same sign code. The S + - or $ code can be used as a drifting sequence, but only one drifting sequence can appear in the field. If a drifting $ is used, no drifting sign can be used in the same field.

A drifting sign cannot appear in the same field with any Z code or * code. The drifting sequence must appear to the left of all digit codes (9 and Y) in the field. The sign position moves to the right through leading zeros. Except for the first position, all positions in the drifting sequence can contain a significant digit. The sign occupies the position immediately to the left of the first significant digit. All positions to the left of the sign in the drifting sequence are supplied as blanks. If the drifting sequence occupies all digit positions and the value is zero, the entire field becomes a sequence of blanks.

The drifting sequence can contain the V code and can contain insertion codes (/ , . or B). If the drifting sequence contains the V code, all digit positions in the fractional subfield must be included in the drifting sequence. The V code terminates movement of the sign to the right, except when the value of the picture is zero, in which case the drifting sequence is represented as all blanks.

Insertion codes within the drifting sequence (or following the drifting sequence) are considered part of the sequence and can be altered as appropriate. If an insertion code appears more than one position to the left of the first significant digit, the insertion code becomes a blank. If the insertion code is immediately to the left of the first significant digit, the insertion code is replaced with the drifting symbol. If the insertion code appears to the right of the first significant digit, the insertion code operates normally. Refer to the description of insertion codes later in this section.

If a fractional value between -1.0 and 0.0 is assigned to a picture and the fractional digits are truncated, the sign remains negative because evaluation of the sign occurs before any necessary truncation.

Signed Digit Codes

The signed digit codes specify that the sign of the arithmetic value and the value of a digit are to be represented as a single character. Combination of a sign and a digit yields an overpunch digit that is represented as a character. The signed digit codes are the following:

- T indicates that a 12-punch (value>=0) or 11-punch (value<0) is overpunched in the digit position.
- I indicates that a 12-punch is overpunched in the digit position (value >=0).
- R indicates that an 11-punch is overpunched in the digit position (value <0).

Each picture can have only one code or drifting code for the sign. The T I or R code can be used, or one of the other sign codes (S + - CR or DB). When a signed digit code is used, the value of the digit and the sign are contained in any chosen digit position. Examples of signed digit codes are included in figure 7-18.

The character represented by the combination of a digit and 12-punch or 11-punch is shown in table 7-9.

TABLE 7-9. SIGNED DIGIT CODE REPRESENTATIONS

Digit	Digit with 12-punch	Digit with 11-punch
0	<	!
1	A	J
2	B	K
3	C	L
4	D	M
5	E	N
6	F	O
7	G	P
8	H	Q
9	I	R

Sign Suffix Codes

The sign suffix codes for credit and debit are intended for use in financial reports. The sign suffix codes are the following:

CR indicates that the last two characters of the field are 'CR' (value<0) or two blanks (value>=0).

DB indicates that the last two characters of the field are 'DB' (value<0) or two blanks (value>=0).

The credit or debit indication always appears at the end of the character value; the CR or DB code must be to the right of all digit positions in the field. Each field can have only one indication of sign. The CR or DB code can be used, or one of the other sign codes (S + - T I or R). Examples of sign suffix codes are included in figure 7-18.

Currency Code

The currency code is the following:

$ supplies a dollar sign in the character value of the picture. A sequence of $ codes represents a drifting $.

The currency code $ can be used in a static or drifting manner. Static use involves one occurrence of the $ code to the left or to the right of all digit positions in the field. Drifting use of the $ code involves a sequence of $ codes. Examples of the currency code are shown in figure 7-19.

Arithmetic Value	Picture Specification	Character Value
51	$ZZZZ	$ΔΔ51
23	ZZZZZ$	ΔΔΔ23$
663	(6)$9	ΔΔΔ$663

Figure 7-19. Currency Code Examples

A drifting sequence of $ codes must appear to the left of all digit codes. Only one drifting sequence can be used in a field; the drifting dollar sign cannot be used if a drifting sign (S + or -) is used. The drifting dollar sign also cannot appear in any field containing a Z code or * code. The dollar sign moves to the right through leading zeros. Except for the first position, all positions in the drifting sequence can contain a significant digit. The dollar sign occupies the position to the left of the first significant digit. All positions to the left of the dollar sign in the drifting sequence are supplied as blanks. If the drifting sequence occupies all digit positions and the value is zero, the entire field becomes a sequence of blanks.

The drifting sequence can contain the V code and can contain insertion codes (/ , . or B). The discussion of the V code and insertion codes used in a drifting sequence appears under Sign Position Codes.

Insertion Codes

The insertion codes cause a character to be inserted into the character value of the pictured numeric item. The insertion codes are the following:

/ indicates insertion of a slash '/' character.

, indicates insertion of a comma ',' character.

. indicates insertion of a period '.' character.

B indicates insertion of a blank ' ' character.

Insertion codes do not indicate digit positions. The . code does not specify the position of the assumed decimal point. It can be used with the V picture code to cause the assumed decimal point to be represented as a character. An insertion code which is immediately preceded by a Z picture code is represented as a blank when the Z code causes a leading zero to be suppressed. An insertion code which is immediately preceded by a * picture code is represented as * when the * picture code causes a leading zero to be replaced. Examples of insertion codes are shown in figure 7-20.

Arithmetic Value	Picture Specification	Character Value
122577	Y9/Y9/99	12/25/77
12.578	999V.99	012.57
15.36	ZZZV,ZZ	Δ15,36
12567.03	ZZZ,ZZZV.99	Δ12,567.03
6475.77	ZZZ.ZZZV,99	ΔΔ6.475,77
54.7	SBB99V.999	+ΔΔ54.700
6	Z,ZZZV.ZZBDB	ΔΔΔΔ6.00ΔΔΔ
0.25	ZZZV.99	ΔΔΔ.25
0.25	ZZZ.V99	ΔΔΔΔ25
00345.67	ZZ,ZZ9V.99	ΔΔΔ345.67
00345.67	**,**9V.99	***345.67

Figure 7-20. Insertion Codes Examples

The / code inserts a slash in the character value of the picture.

The , code inserts a comma in the character value of the picture.

The . code inserts a period in the character value of the picture. The . code must not be confused with the V code that specifies the assumed decimal point. The V code can only appear once, while multiple . insertion codes can be used. Usually, the . code is used immediately after the V code, since the assumed decimal point is not represented in the character value of the picture.

The B code inserts a blank in the character value of the picture.

Scaling Factor Code

The scaling factor code is the following:

F(factor) indicates that the decimal digits in the character value must represent a scaled arithmetic value. The factor is an optionally signed decimal integer.

The F code must appear at the right end of the field. The scaling factor indicates that the arithmetic value is divided by ten to the power of the scaling factor, that is, (value/(10**factor)), before being used to create the character value. If the arithmetic value is subsequently required, interpretation of the character value for the arithmetic value involves multiplying the decimal value in the character string by ten to the power of the scaling factor, that is, (value*(10**factor)). The scaling factor must be in the range -255 to 255. Examples of the scaling factor code are shown in figure 7-21.

Arithmetic Value	Picture Specification	Character Value
1800	9999F(2)	0018
1.35	999999F(-3)	001350

Figure 7-21. Scaling Factor Code Examples

PICTURED NUMERIC FLOATING POINT

A pictured numeric floating point item has a character value representing a FLOAT DECIMAL value on which editing has been performed. The character value contains the decimal digits and all edited characters including insertion characters. The character value does not include the assumed decimal point in the mantissa but can include inserted decimal points.

The picture specification for a pictured numeric floating point item consists of two fields. The picture field for the mantissa is similar to the field used for a fixed point item and can contain most of the picture codes available for a numeric fixed point item. The picture field for the exponent can contain a more restricted set of picture codes. The two fields are included in the same picture specification; the mantissa field is followed by the E or K exponent code that precedes the exponent field. The available picture codes are shown in table 7-10.

Edit through Picture

An arithmetic value that is assigned to a pictured numeric variable or written out through a P format item is edited through the picture specification. The arithmetic value is used to create a character value that can include digits, blanks, a sign indication for the mantissa, and inserted characters, as well as an exponent indication and a sign indication for the exponent. The created character value depends on the picture codes specified. The picture codes indicate a character value created from a FLOAT DECIMAL arithmetic value.

Interpret for Arithmetic Value

A pictured numeric item is stored as a character value and can be interpreted for the arithmetic value when the arithmetic value is required for the purpose of operations or assignment. The character value is interpreted for the arithmetic value when an input value is read in through a P format item. The arithmetic value is FLOAT DECIMAL with a precision indicated in the picture specification.

Codes for the Mantissa

For the mantissa, digit codes (9 Z * or Y) can be specified in the same way as for a pictured fixed point numeric item.

TABLE 7-10. CODES FOR PICTURED NUMERIC FLOATING POINT

Category	Code	Use
Mantissa		
Digit codes	9	Digit
	Z	Leading zero suppression digit
	*	Leading zero replacement digit
	Y	Zero suppression digit
Decimal point code	V	Assumed decimal point
Sign position codes	S	Sign + or - (can drift)
	+	Sign if + (can drift)
	-	Sign if - (can drift)
Signed digit codes	T	Overpunch digit with + or -
	I	Overpunch digit if +
	R	Overpunch digit if -
Insertion codes	/	Inserted slash
	,	Inserted comma
	.	Inserted period
	B	Inserted blank
Exponent		
Exponent codes	E	Start exponent field (supply E)
	K	Start exponent field (suppress E)
Digit codes	9	Digit
	Z	Leading zero suppression digit
	*	Leading zero replacement
	Y	Zero suppression digit
Sign position codes	S	Sign + or -
	+	Sign if +
	-	Sign if -
Signed digit codes	T	Overpunch digit with + or -
	I	Overpunch digit if +
	R	Overpunch digit if -
Insertion codes	/	Inserted slash
	,	Inserted comma
	.	Inserted period
	B	Inserted blank

The assumed decimal point can be indicated with the V code. The sign can be represented with a sign position code (S + or -) or a signed digit code (T I or R). A drifting sequence of the sign code S + or - can be used. Insertion codes (/ . , B) can be used. The codes $ CR DB and F cannot be used and are used only for pictured numeric fixed point items.

The value of the mantissa is adjusted so that the first significant digit appears in the first available digit position, even if leading zero suppression or leading zero replacement is specified with Z codes or * codes.

Codes for the Exponent

The exponent code separates the mantissa field and the exponent field. The exponent codes are the following:

- E starts the exponent field and inserts the exponent indicator E.
- K starts the exponent field but suppresses the exponent indicator E.

Either the E code or K code must be used in any floating point picture specification. The exponent field follows the E code or K code. The digit codes (9 Z * or Y) can be used. A sign can be represented with a sign code (S + or -) or a signed digit code (T I or R). The sign code S + or - cannot be used in a drifting sequence. Insertion codes (/ , . or B) can be included. Examples of pictured numeric floating point items are shown in figure 7-22.

Arithmetic Value	Picture Specification	Character Value
1.1E1	99999ES99	11000E-03
3.456E3	ZZZZV.9ES99	3456.0E+00
3.4E-7	9V.99BBES99	3.40ΔΔE-07
7.7E-1	V.(4)9K-(3)9	.7700Δ000
.352E-12	V.9999KB-99	.3520Δ-12

Figure 7-22. Pictured Numeric Floating Point Examples

This section describes the available PL/I data transmission operations. Input/output operations are either stream I/O operations or record I/O operations.

Stream I/O operations involve the input or output of streams of legible characters. Data items transmitted with stream I/O operations can be converted from one computational type to another, and can be formatted under program control. The two methods of stream transmission are list-directed I/O and edit-directed I/O. The stream I/O statements are

GET	Stream input
PUT	Stream output

Record I/O operations involve the input or output of complete records transmitted in internal form. No computational conversions occur, and no formatting of data items is possible. The record I/O statements are

READ	Record input
WRITE	Record output
REWRITE	Record replacement
DELETE	Record deletion
LOCATE	Record output

Stream I/O operations and record I/O operations (except for stream I/O with the STRING option) are performed on files that are opened and closed during program execution. The I/O statements that can be used to open and close files are

OPEN	Open file
CLOSE	Close file

Various conditions can be raised during input/output operations. The possible I/O conditions are UNDEFINEDFILE, ENDFILE, ENDPAGE, KEY, RECORD, and TRANSMIT.

FILES

The file constants in a program have no intrinsic connection to CYBER Record Manager (CRM) local files. Association of a particular file constant with a CRM file is made when the file constant is opened during program execution. A file is defined as a file constant that is open, together with the completed file description attributes and the associated CRM file.

FILE OPENING

File opening is the process of establishing all necessary information so that the file can be used. If a file constant is not open, no association with a CRM file exists. When the file constant is opened, an association exists. When the file is closed, the established association is discarded. Subsequent reopening of the file constant can establish association with a different CRM file. Attributes specified in the declaration of a file constant cannot be altered during program execution. When a file constant is opened, a complete set of file description attributes is established for the file, the file constant is associated with a CRM file (the file title is established), and all necessary CRM file processing options are established from ENVIRONMENT information and defaults.

A file can be opened explicitly by an OPEN statement, or can be opened implicitly by execution of a stream or record I/O statement. An attempt to open a file constant is ignored if the file constant is already open.

FILE DESCRIPTION

The available file description attributes are STREAM, RECORD, INPUT, OUTPUT, UPDATE, PRINT, SEQUENTIAL, DIRECT, KEYED, and ENVIRONMENT. The ENVIRONMENT attribute specifies CRM file processing options. A file constant can be declared with file description attributes in the DECLARE statement.

If a file is opened explicitly by an OPEN statement, options in the OPEN statement can be used to supply file description attributes. The attributes specified in the OPEN statement are combined with the attributes specified in the declaration of the file constant.

If a file is opened implicitly by execution of a stream or record I/O statement, the statement implies file description attributes that are combined with the attributes specified in the declaration of the file constant. The attributes implied by the statement that causes implicit opening of the file are shown in table 8-1.

TABLE 8-1. FILE DESCRIPTION FROM IMPLICIT OPENING

Input/Output Statement Causing Implicit Opening	Implied File Description Attributes
GET	STREAM INPUT
PUT	STREAM OUTPUT
READ	RECORD (if the declaration does not specify UPDATE, INPUT is also implied)
WRITE	RECORD (if the declaration does not specify UPDATE, OUTPUT is also implied)
REWRITE	RECORD UPDATE
DELETE	RECORD UPDATE
LOCATE	RECORD OUTPUT

Certain file description attributes imply other file description attributes. PRINT implies STREAM and OUTPUT. UPDATE, SEQUENTIAL, DIRECT, or KEYED implies RECORD. DIRECT implies KEYED.

In some cases, a default file description attribute can be supplied. If STREAM or RECORD is not indicated, STREAM is supplied. If INPUT, OUTPUT, or UPDATE is not indicated, INPUT is supplied. If SEQUENTIAL or DIRECT is not indicated for a record file, SEQUENTIAL is supplied. If a file constant named SYSPRINT has the attributes STREAM and OUTPUT, PRINT is supplied.

The completed set of file description attributes must be a consistent set. If conflicting attributes exist in the set, the UNDEFINEDFILE condition is raised. In the completed set, STREAM and RECORD are alternative types of file operations. INPUT, OUTPUT, and UPDATE are alternative types of usage for the file. INPUT and OUTPUT can apply to a stream file or a record file. UPDATE can apply only to a record file. PRINT can apply only to a stream output file. For a record file, SEQUENTIAL and DIRECT are alternative types of access to records. KEYED can apply to a sequential record file or a direct record file.

The completed file description attributes are shown in table 8-2 for all consistent combinations of attributes, including declared attributes, and additional attributes specified in the OPEN statement or implied by the I/O statement that implicitly opens the file. The ENVIRONMENT attribute can apply to any consistent combination shown in the table. Section 9, CYBER Record Manager Interface, describes the available CRM file processing options for each attribute combination.

FILE TITLE AND LOCAL FILE NAME

When a file constant is opened, a file title is established. The file title becomes the local CRM file name unless the title is too long to be a legal CRM file name.

The local CRM file name must consist only of letters and digits, must begin with a letter, and must be 1 to 7 characters long. If the file title is longer than 7 characters, the local file name is constructed from the first four and last three characters of the file title. For example, the file title LONGTITLE would specify the local CRM file LONGTLE.

The file title can be specified in one of three ways. If the file is opened explicitly with an OPEN statement, the TITLE option can be used to specify the file title. If the file is opened with an OPEN statement and the TITLE option is not used, the file constant specifies the file title. If the file is opened implicitly by execution of an I/O statement, the name of the referenced file constant specifies the file title.

Treatment of the file title is identical to the treatment of external names in general. The file title must not duplicate the local file name of an open file. Note that the file titles TEST1INPUT and TEST2INPUT would cause an invalid duplication of the local file name TESTPUT. A local file name can duplicate the name of an external variable, named constant, or condition without conflict.

NOTE

Two special cases exist for the file title. If a file constant named SYSPRINT with the attributes STREAM OUTPUT PRINT is opened without an explicit TITLE option, the file title OUTPUT is used. If a file constant named SYSIN with the attributes STREAM INPUT is opened without an explicit TITLE option, the file title INPUT is used.

STREAM I/O

In stream I/O operations, the data in the file is considered as a continuous stream of characters organized into lines. A file with the PRINT attribute can also be organized into pages. Stream I/O statements specify the input/output of data items and direct any editing or formatting of the data. Stream I/O operations are different from record I/O operations described later in this section.

STREAM FILE STATUS

Each file constant that can be used for a stream file carries the following information as the result of program compilation:

- Declared file description attributes, if any.
- Declared ENVIRONMENT, if any.

When the file constant is opened either explicitly by OPEN statement or implicitly by execution of a stream I/O statement, the following information is added:

- The file title, either supplied in the TITLE option of the OPEN statement or created from the file constant.
- Additional attributes specified on the OPEN statement or implied by the statement that implicitly opens the file constant.
- Any additional attributes required to complete the set of file description attributes for the file.
- Creation of a file information table (FIT) used by CRM. The ENVIRONMENT attribute specifies file processing options that are used for option fields in the FIT (see section 9).
- The line size for the file is established. If the OPEN statement specifies the LINESIZE option, that line size is used. If LINESIZE is not specified, the following line size is used:

80	STREAM INPUT
80	STREAM OUTPUT
136	STREAM OUTPUT PRINT

  Line size is the maximum number of characters per line of user data. For a PRINT file, the line size is the maximum number of printable characters in each line. The line size does not count any carriage control information for the line. A line size specified with the ENVIRONMENT attribute overrides the LINESIZE option of the OPEN statement or the default line size shown.

- The current column position is established. Column position is set to 1 when the file constant is opened and reset to 1 for each new line. The current column position is the number of characters already transmitted in the line, plus one.
- For a PRINT file only, the page size is established. If the OPEN statement specifies the PAGESIZE option, that page size is used. If page size is not specified, the default page size is 60 lines. The page size establishes the maximum number of lines that can be written on a page before the ENDPAGE condition is raised. The ENDPAGE condition is raised when the current line number is incremented and the increment causes the current line number to exceed the maximum page size.

TABLE 8-2. COMPLETED FILE DESCRIPTION

Combinations of Declared Attributes and Attributes Specified or Implied in File Opening				Resulting Completed File Description
STREAM	INPUT			STREAM INPUT
STREAM	INPUT			
(no attributes supplied)				
STREAM	OUTPUT			STREAM OUTPUT
	OUTPUT			
STREAM	OUTPUT		PRINT	STREAM OUTPUT PRINT
STREAM			PRINT	
	OUTPUT		PRINT	
			PRINT	
RECORD	INPUT	SEQUENTIAL		RECORD INPUT SEQUENTIAL
RECORD	INPUT			
RECORD		SEQUENTIAL		
RECORD				
	INPUT	SEQUENTIAL		
		SEQUENTIAL		
RECORD	OUTPUT	SEQUENTIAL		RECORD OUTPUT SEQUENTIAL
RECORD	OUTPUT			
	OUTPUT	SEQUENTIAL		
RECORD	UPDATE	SEQUENTIAL		RECORD UPDATE SEQUENTIAL
RECORD	UPDATE			
	UPDATE	SEQUENTIAL		
	UPDATE			
RECORD	INPUT	SEQUENTIAL	KEYED	RECORD INPUT SEQUENTIAL KEYED
RECORD	INPUT		KEYED	
RECORD		SEQUENTIAL	KEYED	
RECORD			KEYED	
	INPUT	SEQUENTIAL	KEYED	
	INPUT		KEYED	
		SEQUENTIAL	KEYED	
			KEYED	
RECORD	OUTPUT	SEQUENTIAL	KEYED	RECORD OUTPUT SEQUENTIAL KEYED
RECORD	OUTPUT		KEYED	
	OUTPUT	SEQUENTIAL	KEYED	
	OUTPUT		KEYED	
RECORD	UPDATE	SEQUENTIAL	KEYED	RECORD UPDATE SEQUENTIAL KEYED
RECORD	UPDATE		KEYED	
	UPDATE	SEQUENTIAL	KEYED	
	UPDATE		KEYED	
RECORD	INPUT	DIRECT	KEYED	RECORD INPUT DIRECT KEYED
RECORD	INPUT	DIRECT		
RECORD		DIRECT	KEYED	
RECORD		DIRECT		
	INPUT	DIRECT	KEYED	
	INPUT	DIRECT		
		DIRECT	KEYED	
		DIRECT		
RECORD	OUTPUT	DIRECT	KEYED	RECORD OUTPUT DIRECT KEYED
RECORD	OUTPUT	DIRECT		
	OUTPUT	DIRECT	KEYED	
	OUTPUT	DIRECT		
RECORD	UPDATE	DIRECT	KEYED	RECORD UPDATE DIRECT KEYED
RECORD	UPDATE	DIRECT		
	UPDATE	DIRECT	KEYED	
	UPDATE	DIRECT		

- For a PRINT file only, the current line number is established. The current line number is set to 1 when the file constant is opened. The current line number is the number of complete lines written since the beginning of the page, plus one.
- For a PRINT file only, the current page number is established. The current page number is set to 1 when the file constant is opened. The current page number is incremented whenever a new page is established and can be reset with the PAGENO pseudovariable.

The following I/O conditions can be raised during stream I/O operations using GET and PUT statements with the FILE option:

UNDEFINEDFILE	Failure to open a file correctly.
ENDFILE	End of input file exceeded.
ENDPAGE	End of page exceeded on a print file.
TRANSMIT	Data transmission error.

In addition, the SIZE or CONVERSION condition can be raised during list-directed or edit-directed operations involving conversion of data.

STREAM I/O STATEMENTS

The GET and PUT statements are used for stream I/O. The FILE and STRING options are mutually exclusive. The GET statement syntax and the PUT statement syntax are shown in section 12, Statements. The FILE option is used for data transmission to and from stream files on external input/output devices. The STRING option is used for data transmission to and from character strings in internal storage. If both the FILE option and the STRING option are omitted from a GET or PUT statement, a FILE option is assumed.

GET and PUT statements with the FILE option are used for stream files. The available GET and PUT statements for each possible set of file description attributes are shown in table 8-3.

GET and PUT statements with the STRING option are used for character string transmission. The available GET and PUT statements are shown in table 8-4.

The FILE option specifies the stream file to be used for the input/output operation. If both the FILE option and the STRING option are omitted from a GET or PUT statement, a FILE option of the form FILE(SYSIN) is supplied for GET and a FILE option of the form FILE(SYSPRINT) is supplied for PUT.

The STRING option specifies the character string value from which input is taken, or the character variable or pseudovariable to which output is assigned. The STRING option is used only for internal manipulation of string values, and no input/output is performed on any external I/O device.

The COPY option specifies that all input data is to be copied to an output file. The COPY option can be used for GET FILE or GET STRING operations. Effectively, the COPY option provides a record of all input data used by the statement. If a COPY option appears with no file-reference, COPY(SYSPRINT) is assumed. The COPY option uses all input data accessed by the statement and writes a copy to the output file. The input stream is copied intact, with no editing of the input data and no positioning of the output file. Data on the COPY file is separated into lines in accordance with the line size of the COPY file. The COPY file must be a STREAM OUTPUT or STREAM OUTPUT PRINT file. If the COPY file is a print file, the ENDPAGE condition can be raised.

The SKIP option is used only for GET FILE or PUT FILE operations to specify positioning of the input file or output file. For a GET FILE operation, the SKIP option indicates that the input file is to be positioned forward the specified

TABLE 8-3. STREAM I/O STATEMENTS FOR STREAM FILES

Stream File Attributes	Available I/O Statements
STREAM INPUT	GET [FILE-option†] [LIST-option / EDIT-option] [SKIP-option] [COPY-option]
STREAM OUTPUT	PUT [FILE-option††] [LIST-option / EDIT-option] [SKIP-option]
STREAM OUTPUT PRINT	PUT [FILE-option††] [LIST-option / EDIT-option] [SKIP-option / PAGE-option / LINE-option / PAGE-option LINE-option]

†if omitted, FILE(SYSIN) is assumed.
††if omitted, FILE(SYSPRINT) is assumed.

TABLE 8-4. STREAM I/O STATEMENTS FOR STRING OPERATIONS

String Operation	Available I/O Statements
String Input	GET STRING-option {LIST-option / EDIT-option} [COPY-option]
String Output	PUT STRING-option {LIST-option / EDIT-option}

number of lines. The ENDFILE condition can be raised. For a PUT FILE operation, the SKIP option indicates that the current line is ended and the output file is positioned forward the specified number of lines. If the output file is a print file, the ENDPAGE condition can be raised.

The PAGE option is used only for PUT FILE operations to establish a new page on the output print file. The current page number is incremented by 1, and the next available print position becomes column position 1 in line number 1 on the new page.

The LINE option is used only for PUT FILE operations to establish a new line on the output print file. The LINE option specifies the required line position on the page and is evaluated with respect to the current line position. If the new line number is greater than the current line number but less than the page size, lines on the page are skipped until the required position is reached. If the new line number is less than the current line number, or if the new line number is greater than the established page size, the remaining lines on the page are skipped and the ENDPAGE condition is raised. If the new line number is the same as the current line number, the required line position is already the next available line position and the LINE option has no effect.

The LIST option indicates a list-directed I/O operation. List-directed I/O involves transmission of data in accordance with the data type of each target or source value.

The EDIT option indicates an edit-directed I/O operation. Edit-directed I/O involves transmission of data in accordance with the format specifications supplied in the program.

Stream I/O operations without the LIST option or the EDIT option are file positioning operations. If the LIST option or EDIT option is used, list-directed I/O or edit-directed I/O operations are indicated.

LIST-DIRECTED I/O

In list-directed I/O, each data item in the stream is interpreted or created according to the data type of the target or the source value, rather than according to a format specification supplied in the program. Each list-directed I/O statement can transmit one or more data items.

List-Directed Input

The LIST option specifies one or more targets to which data values are assigned by GET statement execution. Each target can be a computational variable or a pseudovariable. Each target must be scalar.

When a GET statement is executed, the specified list of input-targets is processed in order. The processing order is left to right, except as controlled by embedded-do specifications. The input-targets at the same parenthesis level as the embedded-do are processed left to right once for each iteration of the embedded-do. For example, the specification

```
(A(I), B(I) DO I = 1 TO 3)
```

is processed in the order

```
A(1) B(1) A(2) B(2) A(3) B(3)
```

The statement terminates normally when the input-target list is exhausted.

The input stream is treated as a sequence of literal constants separated by blanks or commas. For each input-target, the next constant in the input stream is obtained, interpreted, and assigned. The following forms are recognized:

- Arithmetic constant, as described in section 3, Data Elements. The arithmetic constant can be preceded by a sign + or -. Blanks are not permitted between the sign and the constant.

- Simple character constant, as described in section 3. The character constant can include blanks or commas as part of the string and can include the apostrophe character (represented by two consecutive apostrophes).

- Character constant not enclosed in apostrophes. The character constant is treated as an evaluated character constant in the program. The constant cannot include any blanks or commas. The constant can include the apostrophe character (represented by a single apostrophe character). To avoid confusion with the standard character constant form, this form of character constant must not begin with an apostrophe.

- Simple bit constant, as described in section 3. In addition, the bit constant can include blanks or commas between the apostrophe that begins the constant and the sequence of specified bits, or between the sequence of specified bits and the second apostrophe. Blanks and commas are not allowed in the sequence of specified bit values. No blank is allowed between the second apostrophe and the radix factor B.

- Character constant with bad parse. The character constant is a standard character constant enclosed in apostrophes, but additional characters follow the apostrophe that would normally end the character constant. Until a blank or comma is encountered in the input stream, any additional characters are treated as a continuation of the constant.

The terminator used to separate source items in the input stream is the comma or the blank. The following can represent a single terminator separating source items:

- A single blank.

- A number of consecutive blanks.

- A single comma, preceded by zero or more blanks and followed by zero or more blanks.

For each target item, the input stream is advanced by at least one character from its current position. Line boundaries in the input stream are ignored.

An input field is considered to be null if two commas are separated only by zero or more blanks. If the first nonblank character in the input stream is a comma, the first input field is considered null. If the input field is null, no value is assigned to the input target item and execution continues with evaluation of the next target item.

If the input field is not a valid constant, CONVERSION is raised. The new value of the ONSOURCE builtin function is the input field that raised CONVERSION. If an end of file is encountered before an input field is found, the ENDFILE condition is raised. If the end of the STRING option character string is reached before an input field is obtained, the ERROR condition is raised.

Any necessary computational conversion can be performed to convert the value contained in each input field to another computational type suitable for assignment to the target item. The computational conversions are described in section 7, Data Manipulation.

List-Directed Output

The LIST option specifies the source values to be transmitted to the output stream by PUT statement execution. Each source value can be a computational expression. Each source value must be scalar.

When a PUT statement is executed, the specified list of source values is processed in order. The processing order is left to right, except as controlled by embedded-do specifications, which are processed in the same way as for list-directed input. The statement terminates normally when the list of source values is exhausted.

Each source value is converted, as necessary, to a character string for transmission to the output stream. Each source value appears in the output stream exactly as converted, except in the following cases:

- If the source value is a bit string, the converted string is modified to have the form of a bit constant. Enclosing apostrophes are supplied, and the character B is appended.
- If the source value is a character string or pictured character item, and if the output stream is not being transmitted to a PRINT file, then the value is modified to have the form of a character constant. Enclosing apostrophes are supplied, and each contained apostrophe is replaced by a pair of consecutive apostrophes. For a PRINT file, the converted string is not modified.

The length of the output field is the length of the converted character string and any added characters, including added apostrophes (if any) and an added radix factor B (if any). For a PUT FILE operation, if there are not enough positions remaining in the current line for the output field, a SKIP(1) operation is performed to skip to position 1 of the next line. For a PUT FILE operation, if there are not enough positions in an entire line for the output field, the output field is split as necessary and transmitted to the current and subsequent lines. If there are not enough positions remaining in a STRING option character string for the output field, the ERROR condition is raised.

Each source value transmitted to the output stream is automatically positioned according to system-defined tabs.

EDIT-DIRECTED I/O

For edit-directed transmission, a format is specified for each data item in the data stream. A GET statement can specify input of one or more data items. For each target item, a format item in a format list specifies the characters that are to be obtained from the input stream and the interpretation of the characters. A PUT statement can specify output of one or more data items. For each source item, a format item specifies the form of the transmitted data in the output stream.

In edit-directed operations, the data stream is not considered to have separators between data items for input, and no separators are written during output. The program has complete format control over access to and interpretation of characters in the input stream, and over construction of characters in the output stream.

Edit-Directed Input

The EDIT option specifies one or more targets to which data values are assigned by GET statement execution. Each target can be a computational variable or pseudovariable. Each target must be scalar. A sequence of format items is specified as part of the EDIT option. The specified format items control the access to and interpretation of characters in the input stream.

When a GET statement is executed, each parenthesized list of input targets is paired with the following parenthesized list of format items. The paired groupings are processed in order from left to right. For each paired grouping, the specified list of input targets is processed in order from left to right, except as controlled by embedded-do specifications. The input targets at the same parenthesis level as the embedded-do are processed left to right once for each iteration of the embedded-do. For example, the specification

```
(C(4,J), D(J) DO J = 1 TO 2)
```

is processed in the order

```
C(4,1) D(1) C(4,2) D(2)
```

Processing of each paired grouping terminates normally when the list of input targets is exhausted. The statement terminates normally when the list of paired groupings is exhausted.

Edit-Directed Output

The EDIT option specifies the source values to be transmitted to the output stream by PUT statement execution. Each source value can be a computational expression. Each source value must be scalar. The EDIT option also specifies a sequence of format items that control the construction of characters in the output stream.

When a PUT statement is executed, each parenthesized list of source values is paired with the following parenthesized list of format items. The paired groupings are processed in order from left to right. For each paired grouping, the specified list of source values is processed in order from left to right, except as controlled by embedded-do specifications, which are processed in the same way as for edit-directed input. The processing of each paired grouping terminates normally when the list of source values is exhausted. The statement terminates normally when the list of paired groupings is exhausted.

FORMAT PROCESSING FOR EDIT-DIRECTED I/O

The list of input targets or output source values drives processing of the list of format items. Spacing or positioning is done in the process of a search for the next data transmission format item.

The format items in a parenthesized format specification are processed in order from left to right, except as controlled by iteration factors. When an iteration factor is encountered, it is evaluated and converted to a fixed binary integer that must not be negative. Then the following format item, or entire parenthesized list of format items, is repeated the specified number of times. For example, the specification

```
(2(A(10), F(5)))
```

is processed in the order

```
A(10) F(5) A(10) F(5)
```

When a remote format item is encountered, left-to-right processing of the format specification in which the remote format appears is interrupted. Format processing goes to the first item of the remote list. When the last format item of a remote list has been used and target or source items remain to be transmitted, format processing returns to the format specification that contains the remote format reference. Evaluation continues with the next format item following the used remote format item.

If the last format item has been used and target or source items remain to be transmitted, format evaluation returns to the beginning of the format specification.

The data transmission format items that control the transmission of values are:

F	Fixed point decimal arithmetic
E	Floating point decimal arithmetic
A	Character string
B	Bit string
P	Pictured

The control format items that control the spacing and positioning of the input or output data in the stream are:

X	Ignore a number of characters or create a number of blanks
COLUMN	Move to a new column position
SKIP	Skip a number of lines
LINE	Move to a new line on the page (PRINT only)
PAGE	Move to the next page (PRINT only)

The remote format item that controls the use of the format specifications contained in FORMAT statements is:

R	Use remote format specification in FORMAT statement

The format items for data transmission, spacing and positioning, and use of remote formats are described in alphabetic order.

A Format Item

The A format item is used for stream transmission of a character string value. The syntax is shown in figure 8-1.

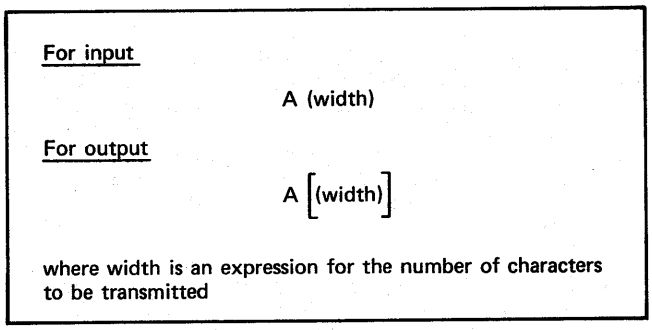


Figure 8-1. A Format Item Syntax

The width specification is evaluated and converted, if necessary, to fixed binary integer.

Input

Width must be specified and must not be negative. The specified number of characters is obtained from the input stream. If width is zero, the source value is a null string. The source string is assigned to the target, and any necessary conversion is done during assignment.

Output

The source value is converted as necessary to character string. If width is not specified, all characters in the string are transmitted to the output stream. If width is specified, the string is padded on the right with blanks or truncated on the right. The string is then transmitted to the output stream. Examples are shown in figure 8-2.

Output Value	Format	Output Stream
'ABCDE'	A(5)	ABCDE
'ABC'	A(7)	ABC△△△△
'ABCDEFGH'	A(6)	ABCDEF

Figure 8-2. A Format Item Output Examples

B Format Item

The B format item is used for stream transmission of a bit string value. The syntax is shown in figure 8-3.

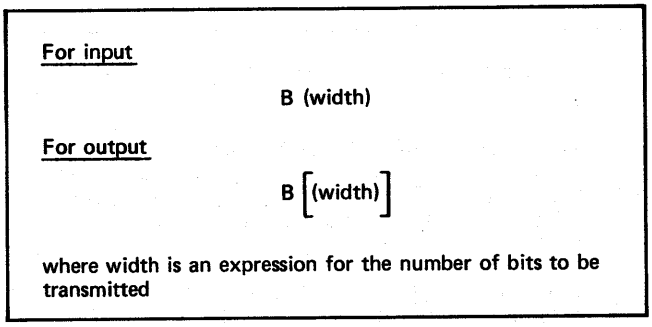


Figure 8-3. B Format Item Syntax

The width specification is evaluated and converted, if necessary, to fixed binary integer.

Input

Width must be specified and must not be negative. The specified number of characters is obtained from the input stream. The stream is expected to contain a representation of a bit string as zero or more consecutive '0' and '1' characters. The bit string can be preceded and followed by blanks. The CONVERSION condition is raised if the input field contains any character that is not '0', '1', or blank. If width is zero or if the string is all blanks, a null character string is the source. The source value is then converted to a bit string. The bit string value is assigned to the target item, and any necessary conversion is done during assignment.

Output

The source value is converted as necessary to bit string. The bit string is then converted to a character string. If width is not specified, all characters in the string are transmitted to the output stream. If width is specified, the source value is padded on the right with blanks or truncated on the right. The string is then transmitted to the output stream. Examples are shown in figure 8-4.

Output Value	Format	Output Stream
'1101111'B	B(7)	1101111
'11'B	B(5)	11ΔΔΔ
'111101010101'B	B(4)	1111

Figure 8-4. B Format Item Output Examples

COLUMN Format Item

The COLUMN format item is used to reposition the data stream at a specific column position in the line. COLUMN is used for FILE option processing and cannot be used for STRING option processing. The syntax is shown in figure 8-5.

where new-column is an expression for the required column position in the line

Figure 8-5. COLUMN Format Item Syntax

The new-column specification is evaluated and converted, if necessary, to fixed binary integer. If new-column is omitted, COLUMN(1) is assumed. If new-column specifies a new column position beyond the line size for the stream file, COLUMN(1) is also assumed.

Action of the COLUMN format item is effective before the next data transmission through an A, B, E, F, or P format item. If no data transmission follows, the COLUMN format item is ignored.

Input

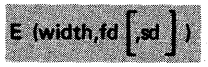
No action is taken if new-column is the next position. If new-column is greater than the next position, characters are ignored in the stream until the new-column position is reached. If new-column is less than the next position, remaining characters in the current line are ignored. Characters are also ignored in the following line until the required new-column position is reached.

Output

No action is taken if the new-column position matches the next available column position. If new-column is greater than the next available position, blanks are transmitted to the output stream until the new-column position is reached. If new-column is less than the next available position, blanks are transmitted for the remainder of the line. Blanks are then transmitted for the following line until the required column position is reached.

E Format Item

The E format item is used for stream transmission of a floating point decimal arithmetic value. The syntax is shown in figure 8-6.



where width is an expression for the number of characters in the field

where fd is an expression for the number of fractional digits in the mantissa

where sd is an expression for the total number of significant digits in the mantissa

Figure 8-6. E Format Item Syntax

The width, fd, and sd specifications are evaluated and converted, if necessary, to fixed binary integers. Width must be specified and must not be negative.

Input

A field of the specified width is obtained from the input stream. The input field is expected to contain a representation of a floating point decimal constant. The representation of the floating point constant can be preceded or followed by one or more blanks, but the constant cannot contain any blanks. If width is zero or the input field consists only of blanks, the source value is zero. The possible valid representations are:

- An optionally signed fixed point decimal constant. The exponent E+0 is assumed. For example, 13.67 is assumed to represent 13.67E+0.
- An optionally signed fixed point decimal constant immediately followed by a sign and a decimal integer. The exponent indication E is assumed. For example, 23+4 is assumed to represent 23E+4.
- An optionally signed floating point decimal constant.

The fd specification for the number of fractional digits is required. The fd specification is used if the mantissa of the constant contains no decimal point. If the constant contains a decimal point, fd does not change the value of the constant. If fd is used, the assumed decimal point is placed so that fd significant digits are to the right of the decimal point. The sd specification can be present but is ignored for input values.

The value is then assigned to the target. The precision of the value is (p), where p is the number of significant digits in the value. Any necessary conversion is done during assignment. Examples are shown in figure 8-7.

Input Stream	Format	Input Value
ΔΔ20	E(4,0)	20E+0
200ΔΔΔ	E(6,2)	2.00E+0
Δ2.3E5Δ	E(7,2)	2.3E+5
-2+3ΔΔΔ	E(7,0)	-2E+3

Figure 8-7. E Format Item Input Examples

Output

The source value is converted as necessary to a decimal floating-point value. The fd specification for the number of fractional digits is required. The sd specification is optional. If sd is not specified, sd is assumed to be fd+1. If sd is specified, sd must be greater than fd. The value of fd, and the value of sd, if specified, must not exceed 14.

The width should be at least 6 greater than the total number of significant digits. Positions must be reserved for the sign of the mantissa, an E, and a signed three-digit exponent. If the mantissa is to contain a decimal point, width should be at least 7 greater than sd, and if all significant digits are to be fractional digits, width should be at least 8 greater than sd. If the specified width is not sufficient to represent the value without loss of sign or high-order significant digits, the SIZE condition is raised.

The value is transmitted to the output stream in one of the following ways:

- E format item with fd=0: all significant digits are to be in the integer part of the mantissa. The output field is created as if under control of the picture

  '(i)B-(j)9VES999' where repetition factor i=width-sd-6 and j=sd

- E format item with sd>fd and fd>=0; significant digits are to be in the integer and the fractional part of the mantissa. The output field is created as if under control of the picture

  '(i)B-(j)9V.(k)9ES999' where repetition factor i=width-sd-7 and j=sd-fd and k=fd

- E format item with sd=fd and sd>0: all significant digits are to be in the fractional part of the mantissa. The output field is created as if under control of the picture

  '(i)B-9V.(j)9ES999' where repetition factor i=width-sd-8 and j=sd

Examples are shown in figure 8-8.

Output Value	Format	Output Stream
20	E(10,3)	2.000E+001
20	E(10,0)	ΔΔΔΔ2E+001
2.3	E(10,3)	2.300E+000
.023	E(10,1,4)	230.0E-004

Figure 8-8. E Format Item Output Examples

F Format Item

The F format item is used for stream transmission of a decimal fixed point arithmetic value. The syntax is shown in figure 8-9.

F (width [,fd [,scale]])

where width is an expression for the number of characters in the field

where fd is an expression for the number of fractional digits

where scale is an expression for the scale factor to be used

Figure 8-9. F Format Item Syntax

The width, fd, and scale specifications are evaluated and converted, if necessary, to fixed binary integers. Width must be specified and must not be negative.

Input

A field of the specified width is obtained from the input stream. The input field is expected to contain a representation of a fixed-point decimal constant. The constant can be immediately preceded by a sign. The constant can be preceded or followed by blanks, but the constant cannot contain any blanks. If the width is zero or the input field consists entirely of blanks, the source value is zero.

The fd specification for the number of fractional digits is optional. If the constant contains a decimal point, the fd specification is ignored. If fd is specified, the assumed decimal point is initially placed so that fd significant digits are to the right of the decimal point. If fd is not specified, fd is assumed to be zero. The scale specification can further alter the position of the decimal point.

If scale is not specified, scale is assumed to be zero. The scale specification alters the source value by changing the location of the decimal point. A positive scale moves the decimal point to the right, and a negative scale moves the decimal point to the left. Effectively, the source value is multiplied by ten raised to the power of the scale.

The value is then assigned to the target. The precision of the value is (p,q), where p is the number of significant digits in the value and q is fd-scale. Any necessary conversion is done during assignment. Examples are shown in figure 8-10.

Input Stream	Format	Input Value
Δ200	F(4)	200
Δ200	F(4,2)	2.00
Δ3.2	F(4)	3.2
200Δ	F(4,3,2)	20.0

Figure 8-10. F Format Item Input Examples

Output

The source value is converted as necessary to a decimal fixed point value. The fd specification for the number of fractional digits is optional. If fd is not specified, fd is assumed to be zero. The scale is optional and is used to scale the output value. If scale is not specified, the scale is assumed to be zero.

The width should be at least 1 greater than the total number of significant digits, to reserve a position for the sign. If the output value is to contain fractional digits, width should be at least 2 greater than the number of significant digits, to reserve an additional position for the decimal point. If the specified width is not sufficient to represent the value without loss of sign or high-order significant digits, the SIZE condition is raised.

The value is multiplied by ten raised to the power of the scale. The number of fractional digits specified by fd is maintained, but fractional digits can be lost when the scale is applied. The value is then transmitted to the output stream in one of the following ways:

- F format item with fd=0: all significant digits are to be in the integer part of the value. The output field is created as if under control of the picture

  '(i)-9' where repetition factor i=width-1

- F format item with fd>0: significant digits are to be in the integer and fractional parts of the value. The output field is created as if under control of the picture

  '(i)-9V.(j)9' where repetition factor i=width-fd-1 and j=fd

Examples are shown in figure 8-11.

Output Value	Format	Output Stream
3.2	F(4)	ΔΔΔ3
3.2	F(4,2)	3.20
0	F(4,1)	Δ0.0
-3.5	F(4,0,2)	-350

Figure 8-11. F Format Item Output Examples

LINE Format Item

The LINE format item is used to position the output stream to the specified line with respect to the top of the page. LINE is used for FILE option processing and cannot be used for STRING option processing. LINE is used only for a PRINT file. The syntax is shown in figure 8-12.

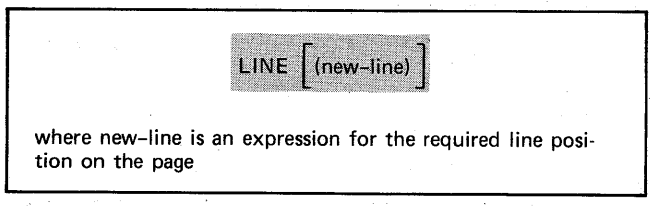


Figure 8-12. LINE Format Item Syntax

The new-line specification is evaluated and converted, if necessary, to fixed binary integer. If the new-line specification is omitted, new-line is assumed to be 1. If new-line is greater than the current line number, lines on the page are skipped until the specified line number is reached. If new-line causes the current line number to exceed page size, the ENDPAGE condition is raised. If new-line is less than the current line number, or new-line is the same as the current line number but characters have already been transmitted to the current line, the ENDPAGE condition is raised. If new-line specifies the current line and the next available position is the first character of the line, the LINE format item has no effect.

Action of the LINE format item is effective before the next data transmission through an A, B, E, F, or P format item. If no data transmission follows, the LINE format item is ignored.

P Format Item

The P format item is used for stream transmission of a pictured value. The pictured value transmitted is dependent on the form of the picture. The syntax is shown in figure 8-13.

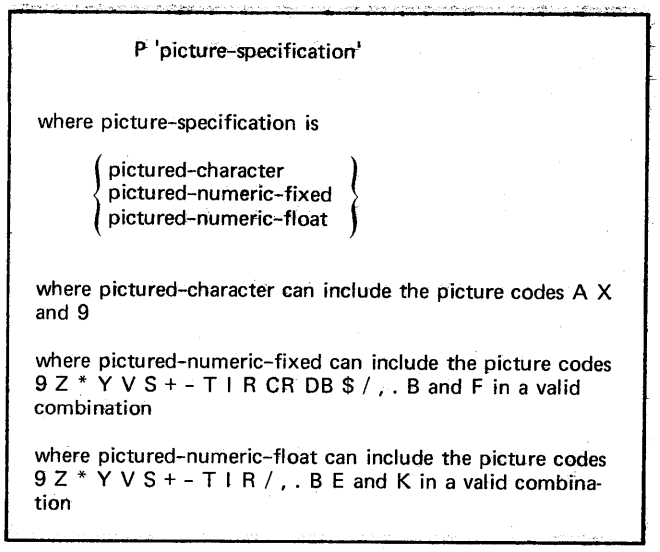


Figure 8-13. P Format Item Syntax

The picture-specification is a character string containing a sequence of picture codes. The length of the picture specification cannot exceed 1000 characters. Picture-controlled conversion is described in section 7, Data Manipulation.

Input

The width of the input field is taken from the picture. For pictured character, the width is the number of picture codes. For pictured numeric fixed, the width is the number of picture codes other than V and F. For pictured numeric float, the width is the number of picture codes other than V and K. The input field is treated as the character value of a pictured item; if the input field does not conform to the specified picture, the CONVERSION condition is raised. The character value or arithmetic value interpreted from the character value is assigned to the target, and any necessary conversion is done during assignment.

Output

The source value is treated as if for assignment through the picture-specification. The width of the output field is determined by the picture. For pictured character, the source value is converted to character, if necessary, and then validated through the picture. For pictured numeric, the source value is converted, if necessary, to FIXED DECIMAL or FLOAT DECIMAL and then edited through the picture. Examples are shown in figure 8-14.

Output Value	Format	Output Stream
'123 ABC'	P'999AAAA'	123ΔABC
12.25	P'S(5)ZV.99'	+ΔΔΔ12.25
3E+7	P'999ES99'	300E+05

Figure 8-14. P Format Item Output Examples

PAGE Format Item

The PAGE format item is used to select the next page of the output stream. PAGE is used for FILE option processing and cannot be used for STRING option processing. PAGE is used only for a PRINT file. The syntax is shown in figure 8-15.

```
PAGE
```

Figure 8-15. PAGE Format Item Syntax

PAGE causes the current page number to be incremented by 1. The first character of the first line of the next page is the next available character position.

Action of the PAGE format item is effective before the next data transmission through an A, B, E, F, or P format item. If no data transmission follows, the PAGE format item is ignored.

R Format Item

The R format item causes use of a remote format specification in a FORMAT statement. The referenced FORMAT statement must be immediately contained in the same block as the GET EDIT or PUT EDIT statement that references the remote format. The syntax is shown in figure 8-16.

```
R(remote-format)

where remote-format specifies a format constant
```

Figure 8-16. R Format Item Syntax

All format items in the format specification of the FORMAT statement identified by remote-format are included in place of the R format item. The remote-format can in turn contain the R format item. Each R format item must not reference a remote-format already referenced for the same GET EDIT or PUT EDIT operation; recursion is not permitted. An example is shown in figure 8-17.

```
FMT1:
   FORMAT(A(5), R(FMT2));
FMT2:
   FORMAT(SKIP, A(15), F(10,3));
   .
   .
   .
   PUT EDIT(CN, CLABEL, CVALUE, CFLAG)
         (R(FMT1), B);

   /* FORMAT ITEMS USED ARE:          */
   /* A(5), SKIP, A(15), F(10,3), B   */
```

Figure 8-17. R Format Item Example

SKIP Format Item

The SKIP format item is used to skip a specified number of lines in the input stream or to advance a specified number of lines in the output stream. SKIP is used for FILE option processing and cannot be used for STRING option processing. The syntax is shown in figure 8-18.

```
SKIP [(line-count)]

where line-count is an expression for the number of lines
to be skipped
```

Figure 8-18. SKIP Format Item Syntax

The line-count specification is evaluated and converted, if necessary, to fixed binary integer. The line-count specification is optional and is assumed to be 1 if omitted.

Action of the SKIP format item is effective before the next data transmission through an A, B, E, F, or P format item. If no data transmission follows, the SKIP format item is ignored.

Input

The input stream is advanced to the beginning of the indicated line. The specified line-count must be greater than zero. SKIP(1) indicates that the input stream is advanced from the current position to the first character position in the next line. If line-count specifies a value greater than 1, additional lines are skipped, with (line-count)-1 lines ignored. If line-count exceeds the number of lines available in the input stream before end-of-file, the ENDFILE condition is raised.

Output

The output stream is advanced the specified number of lines. The line-count specification must not be negative. For a PRINT file only, line-count can be zero to indicate that the current character position is to be reset to 1. The current line number is not changed, and an overprint situation exists. If line-count is 1, the current output line is terminated and

the output stream is positioned at the first character position of the next line. If line-count specifies a value greater than 1, additional lines are skipped, with (line-count)-1 empty lines written. If execution of the SKIP option causes the page size to be exceeded, the ENDPAGE condition is raised. Examples are shown in figure 8-19.

```
PUT EDIT('FIRST LINE',
         'SECOND',
         'LINE',
         'FIFTH LINE')
        (SKIP, A,
         SKIP, A,
         SKIP(0), X(7), A,
         SKIP(3), A);

/*OUTPUT LINES ------------------------
FIRST LINE
SECOND LINE


FIFTH LINE
-----------------------------------*/
```

Figure 8-19. SKIP Format Item Output Example

X Format Item

The X format item is used to ignore a number of characters in the input stream or create a number of blank characters in the output stream. The syntax is shown in figure 8-20.

where space-count is an expression for the number of characters to be ignored or blanks to be created

Figure 8-20. X Format Item Syntax

The space-count specification is evaluated and converted, if necessary, to fixed binary integer. If space-count is not specified, space-count is assumed to be 1. For input, the specified number of characters are ignored. For output, the specified number of blank characters are created in the output stream.

RECORD I/O

In the record I/O operations, the file is considered as a collection of discrete records. Record I/O operations direct the input, output, or replacement of complete records. No conversions are performed during data transmission, and no formatting of the records occurs. Record I/O operations are different from stream I/O operations described earlier in this section.

Records in a file can be accessed sequentially or by record key. Records in a SEQUENTIAL file can be accessed in the order in which the records exist in the file. Records in a SEQUENTIAL KEYED file can be accessed either sequentially or by record key. Records in a DIRECT KEYED file can be accessed only by record key. Each record key is a unique name associated with the complete record.

RECORD FILE STATUS

Each file constant that can be used for a record file carries the following information as the result of program compilation:

- Declared file description attributes, if any.
- Declared ENVIRONMENT, if any.

When the file constant is opened either explicitly by OPEN statement or implicitly by execution of a record I/O statement, the following information is added:

- The file title, either supplied in the TITLE option of the OPEN statement or created from the file constant.
- Additional attributes specified on the OPEN statement or implied by the statement that implicitly opens the file constant.
- Any additional attributes required to complete the set of file description attributes for the file.
- Creation of a file information table (FIT) used by CRM. The ENVIRONMENT attribute specifies file processing options that are used for option fields in the FIT (see section 9).
- For SEQUENTIAL and SEQUENTIAL KEYED files, the current record designator is established.

The following I/O conditions can be raised during record I/O operations:

UNDEFINEDFILE	Failure to open a file correctly.
ENDFILE	End of input file exceeded on a sequential read of a SEQUENTIAL or SEQUENTIAL KEYED record file.
KEY	Invalid record key used for a DIRECT KEYED or SEQUENTIAL KEYED record file.
RECORD	Inconsistency between the size of the allocation unit (variable or allocated buffer) and the record length (MRL or FL) specified with ENVIRONMENT.
TRANSMIT	Data transmission error.

In addition, the CONVERSION condition can be raised during record I/O operations. For example, CONVERSION can be raised during conversion to integer for a key used for a word addressable (WA) file.

RECORD I/O STATEMENTS

The statements used for record I/O are the READ, WRITE, REWRITE, DELETE, and LOCATE statements. Each record I/O statement is described in detail in section 12, Statements. The record I/O statements have a number of statement options that are used or omitted, depending on the circumstances.

The READ statement can be used with an INPUT file or UPDATE file. The WRITE statement can be used with an OUTPUT file or UPDATE file. The REWRITE statement can be used with an UPDATE file. The DELETE statement can be used with an UPDATE KEYED file. The LOCATE statement can be used for an OUTPUT file. The available record I/O statements for each possible set of file description attributes is shown in table 8-5.

TABLE 8-5. RECORD I/O STATEMENTS FOR RECORD FILES

Record File Attributes	Available I/O Statements
RECORD INPUT SEQUENTIAL	READ FILE-option {INTO-option / SET-option / IGNORE-option}
RECORD OUTPUT SEQUENTIAL	WRITE FILE-option FROM-option LOCATE based-variable FILE-option [SET-option]
RECORD UPDATE SEQUENTIAL	READ FILE-option {INTO-option / SET-option / IGNORE-option} REWRITE FILE-option [FROM-option]
RECORD INPUT SEQUENTIAL KEYED	READ FILE-option {INTO-option / SET-option / IGNORE-option} [KEY-option / KEYTO-option]
RECORD OUTPUT SEQUENTIAL KEYED	WRITE FILE-option FROM-option KEYFROM-option LOCATE based-variable FILE-option KEYFROM-option [SET-option]
RECORD UPDATE SEQUENTIAL KEYED	READ FILE-option {INTO-option / SET-option / IGNORE-option} [KEY-option / KEYTO-option] REWRITE FILE-option FROM-option KEY-option DELETE FILE-option KEY-option
RECORD INPUT DIRECT KEYED	READ FILE-option KEY-option {INTO-option / SET-option}
RECORD OUTPUT DIRECT KEYED	WRITE FILE-option FROM-option KEYFROM-option LOCATE based-variable FILE-option KEYFROM-option [SET-option]
RECORD UPDATE DIRECT KEYED	READ FILE-option KEY-option {INTO-option / SET-option} REWRITE FILE-option FROM-option KEY-option DELETE FILE-option KEY-option WRITE FILE-option FROM-option KEYFROM-option

The FILE option for all record I/O statements identifies the file on which record I/O is to be performed. Transmission occurs to or from the file.

The INTO option for READ statements specifies the variable to receive the input record.

The SET option for READ and LOCATE statements specifies a pointer variable. The pointer variable is set to address an allocated buffer used for storage of the record.

The IGNORE option for READ statements specifies the number of records to be skipped during input. The IGNORE option is used in reading records by sequence rather than by key.

The FROM option for WRITE and REWRITE statements specifies the variable whose value is to be written as an output record.

The KEYTO option for READ statements specifies the variable to receive the value of the key when a keyed record is read sequentially.

The KEY option for READ, REWRITE, and DELETE statements specifies the value of the key for an operation to read, replace, or delete a record by key.

The KEYFROM option for WRITE and LOCATE statements specifies the value of the key when a new keyed record is written to a file.

An example of statements that access a SEQUENTIAL file and a DIRECT KEYED file is shown in figure 8-21.

```
.
.
.
DECLARE IN1 FILE RECORD INPUT;
DECLARE ISFILE FILE RECORD OUTPUT;
DECLARE 1 CARDIMAGE,
          2 ANIMAL CHAR(15),
          2 AGE CHAR(5),
          2 COMMENTS CHAR(60);
.
.
.
OPEN FILE(IN1) ENV(↑RT=W,MRL=80,KL=15↑);
OPEN FILE(ISFILE) DIRECT KEYED
               ENV(↑RT=W,MRL=80,KL=15↑);
.
.
.
READ FILE(IN1) INTO(CARDIMAGE);
WRITE FILE(ISFILE) FROM(CARDIMAGE)
                   KEYFROM(ANIMAL);
.
.
.

/*RECORDS FROM FILE IN1 ---------------
BABOON          12    A TYPE OF MONKEY
COW             4     NOT A MONKEY
AARDVARK        92    THIS IS A CRAZY ANIMAL
DOG             10    ENEMY OF CATS
---------------------------------------*/
```

Figure 8-21. RECORD File Example

ALLOCATION UNIT SIZE

All record I/O operations involve the input or output of complete records. For an input operation, the length of the input record must match the size of the allocation unit for the variable referenced in the INTO option. For an output operation, the size of the allocation unit for the variable referenced in the FROM option must match the record length established for records in the output file.

The record length for records in a file can be specified with the ENVIRONMENT attribute for the file constant or with the ENVIRONMENT option in the OPEN statement. A FILE control statement (in the control statements for the job) can also specify the record length. The processing of ENVIRONMENT options is described in section 9, CRM Interface.

Record I/O can only transmit records which represent complete allocation units. The size of an allocation unit is always an integral number of 60-bit words. Each 60-bit word can be divided into 10 characters or bytes containing 6 bits each.

The allocation unit size for a scalar variable depends upon the data type of the variable, and is calculated as follows:

- Arithmetic: 1 word
- CHARACTER nonvarying: CEIL(length/10) words
- CHARACTER VARYING: CEIL(length/10) + 1 words, where length is the maximum length
- BIT nonvarying: CEIL(length/60) words
- BIT VARYING: CEIL(length/60) + 1 words, where length is the maximum length
- Pictured: CEIL(length/10) words, where length is the length of the character value
- AREA: (area-size) words
- All noncomputational types except area: 1 word

The allocation unit size for an array variable is calculated as follows:

- ALIGNED array containing elements of any type: (words-per-element) * (number-of-elements) words
- UNALIGNED array containing CHARACTER nonvarying elements: CEIL((length * (number-of-elements))/60) words
- UNALIGNED array containing BIT nonvarying elements: CEIL((length * (number-of-elements))/60) words
- UNALIGNED array containing pictured elements: CEIL((length * (number-of-elements))/10) words, where length is the length of the character value
- UNALIGNED array containing elements of any type except CHARACTER nonvarying, BIT nonvarying, or pictured: (words-per-element) * (number-of-elements) words

The allocation unit size for a structure that does not contain another structure is calculated as follows:

- Structure in which every member is CHARACTER nonvarying UNALIGNED: CEIL((total-length/10) words, where total-length is the sum of all scalar members including array elements.
- Structure in which every member is BIT nonvarying UNALIGNED: CEIL((total-length)/60 words, where total-length is the sum of the lengths of all scalar members including array elements.
- All other structures that do not contain other structures: (total-size) words, where total-size is the sum of the lengths of all immediately contained members of the structure.

The allocation unit size for a structure variable that contains one or more other structures is calculated as follows:

- All structures that contain other structures: (total-size) words, where total-size is the sum of the lengths of all immediately contained members of the structure.

All input/output operations performed on stream files or record files are done by CYBER Record Manager (CRM). This section describes the connection between PL/I information about a file and the information specified about the local CRM file used for input/output operations. Input/output operations are described in section 8 Input/Output.

The environment of a file consists of information about CRM file processing options. The environment options can be specified with the ENVIRONMENT attribute in the declaration of the file constant, with the ENVIRONMENT option in an OPEN or CLOSE statement, or with the FILE control statement (in the control statements for the job). The file environment information can be specified or omitted. If omitted, environment options are established by default.

The environment options specified for a file must be appropriate for the local CRM file. The file description attribute set for the file determines the compatibility of the environment options. The description of environment compatibility discusses the available environment options.

ENVIRONMENT PROCESSING AND DEFAULTS

When a file constant is opened, the file environment information is established. The environment information is established with default values which can then be overridden. The processing of environment information is therefore different from the completion of a consistent set of file description attributes for the file.

The file environment information is placed in the file information table (FIT) used by CRM. When a file constant is opened, file processing options are established in the newly-created FIT for the file. Default file processing options are supplied, and the specified options can override certain default options or set additional options. The supplied default options are:

- The local file name (LFN) option is supplied. The file title is used as the local file name, as described in section 8.
- The file organization (FO) option is supplied as shown in table 9-1. FO is set to SQ (sequential organization) for all stream files and for all record files that are not KEYED. FO is set to indexed sequential organization (IS) for all record files that are KEYED.
- The record type (RT) option is supplied. RT is set to Z for all stream files, and W for all record files except INPUT and OUTPUT. The record types for INPUT and OUTPUT are always set to Z. See table 9-1.
- The fixed length or maximum record length (FL/MRL) option can be supplied. For a stream file, FL is supplied as shown in table 9-1. For a record file, no FL/MRL value is supplied by default. For a record file, ENVIRONMENT (or the FILE control statement) can be used to set the FL/MRL option.

TABLE 9-1. DEFAULT FILE ENVIRONMENT OPTIONS

File Attributes	Default Environment Options		
	FO	RT	FL/MRL
STREAM INPUT	SQ	Z	80
STREAM OUTPUT	SQ	Z	80
STREAM OUTPUT PRINT	SQ	Z	137
RECORD INPUT SEQL	SQ	W	
RECORD OUTPUT SEQL	SQ	W	
RECORD UPDATE SEQL	SQ	W	
RECORD INPUT SEQL KEYED	IS	W	
RECORD OUTPUT SEQL KEYED	IS	W	
RECORD UPDATE SEQL KEYED	IS	W	
RECORD INPUT DIRECT KEYED	IS	W	
RECORD OUTPUT DIRECT KEYED	IS	W	
RECORD UPDATE DIRECT KEYED	IS	W	

- The key length (KL) option is supplied for a KEYED file. The specification KL=10 is assumed.
- The open flag (OF) option is supplied. For any INPUT file (except file SYSIN) and for any UPDATE file, OF is set to R (rewind). For all other files, OF is set to N (no rewind).
- The block type (BT) option is set to C for all stream files and record files. The BT option must not be overridden for stream files.
- The processing direction (PD) option is supplied. PD is set to INPUT for all INPUT files, OUTPUT for all OUTPUT files, and I-O for all UPDATE files. The PD option must not be overridden.
- The old/new flag (ON) option is supplied. ON is set to NEW or OLD as appropriate. The ON option must not be overriden.
- The ORG option is supplied as NEW for all files. The ORG option must not be overridden.
- The error file control (EFC) option is supplied. EFC is set to 3, which enables error messages and notes to be listed on the error file.
- The dayfile control (DFC) option is supplied. DFC is set to 1, which enables error messages to be listed on the dayfile.

- The error option (EO) is supplied. EO is set to AD which causes parity errors to be disregarded.

After default environment options have been established, the following steps are performed to complete the opening of the file:

1. If the OPEN statement for a stream output file includes the LINESIZE option, the specified line size is used to set FL/MRL. If the file is a print file, FL/MRL is set to a value one greter than the line size. For example, LINESIZE(136) sets line size to 136 and sets FL/MRL to 137. The FL/MRL option includes one character for the carriage control information.

2. If the DECLARE statement for the file constant specifies the ENVIRONMENT attribute, the specified options are effective. The ENVIRONMENT attribute specifies a simple character constant containing a list of CRM options separated by commas. For any option that requires a supplied numeric value, a decimal integer must be used. The CRM options that can be specified include:

FL	File organization
RT	Record type
FL/MRL	Fixed length/maximum record length
KL	Key length (KEYED record file only)
OF	Open flag

3. If the OPEN statement for the file specifies the ENVIRONMENT option, the specified options are effective. The ENVIRONMENT option specifies and expression that is evaluated to yield a character value containing a list of CRM options separated by commas. The CRM options that can be specified are the same as shown for step 2.

4. If a FILE control statement for the file appears in the control statements for the job, the specified options are effective. The FILE control statement specifies the local file name and a sequence of specifications separated by commas.

   The format of the FILE control statement is:

   FILE(lfn,option=value,...)

   The CRM options that can be specified for a file used by the PL/I program are the same as shown in step 2.

5. CYBER Record Manager can further supply values for FIT fields. See the CYBER Record Manager reference manuals.

When the file is closed, the following steps are taken:

1. The close flag (CF) option is set to DET. This detaches the file without rewinding it, releases the buffer space and removes the file name from the active file list.

2. If the DECLARE statement for the file constant specifies the ENVIRONMENT attribute, the specified options are effective. The ENVIRONMENT attribute specifies a simple character constant containing a list of CRM options separated by commas. For any option that requires a numeric value, a decimal integer must be used. The CRM option that can be used is:

   CF Close flag

3. If the CLOSE statement for the file specifies the ENVIRONMENT option, the specified options are effective. The ENVIRONMENT option specifies an expression that is evaluated to yield a character value containing a list of CRM options separated by commas. The CRM options that can be specified are the same as those shown for step 2.

4. If a FILE control statement for the file appears in the control statements for the job, the specified options are effective. The FILE control statement specifies the local file name and a sequence of specifications separated by commas. The format of the FILE control statement is:

   FILE(lfn,option mnemonic=value,...)

   The CRM options that can be specified for a file are the same as those shown in step 2.

5. CYBER Record Manager can further supply values for FIT fields. See the CYBER Record Manager reference manuals.

ENVIRONMENT COMPATIBILITY

When a PL/I file is opened for input/output processing and the environment information has been used to establish CRM options in the file information table (FIT), the CRM options are checked. If any incompatiblity exists between the PL/I file and the established CRM options, the UNDEFINEDFILE condition is raised and the file is not opened.

FILE ORGANIZATION

The FO option for file organization can indicate a sequential CRM file (SQ), a word addressable CRM file (WA), or an indexed sequential CRM file (IS). The possible values for the FO option are shown in table 9-2.

For FO-SQ, the following rules apply:

- A REWRITE statement can only be used for RT=F or RT=W. In either case, the rewritten record must have the same length as the one it replaces.

- The DELETE statement can only be used for RT=F or RT=W. In either case, the rewritten record must have the same length as the one it replaces.

For FO=WA, the following rules apply:

- All records transmitted must have the exact length specified by the FL or MRL option, even if RT=W.

- The file is always rewound before opening, and the OF option is ignored. For example, the following sequence of operations would cause data to be lost:

  OPEN the file with SEQUENTIAL

  WRITE to the file

  CLOSE THE file

TABLE 9-2. FO AND RT OPTIONS FOR FILES

File Attributes	CRM File Organization and Record Type
STREAM INPUT STREAM OUTPUT STREAM OUTPUT PRINT	FO=SQ and RT=F, Z, W, or S
RECORD INPUT SEQUENTIAL RECORD OUTPUT SEQUENTIAL RECORD UPDATE SEQUENTIAL	FO=SQ and RT=F, Z, W, or S FO=WA and RT=F or W
RECORD INPUT SEQUENTIAL KEYED RECORD OUTPUT SEQUENTIAL KEYED RECORD UPDATE SEQUENTIAL KEYED RECORD INPUT DIRECT KEYED RECORD OUTPUT DIRECT KEYED RECORD UPDATE DIRECT KEYED	FO=IS and RT=F, Z, W, or S FO=WA and RT=F or W

OPEN the file with SEQUENTIAL

WRITE to the file

- The DELETE statement cannot be used.
- Overwrite of an existing record cannot be detected. Extension of the file (writing beyond the last existing key) cannot be detected.
- In a READ operation, the program must not attempt to read a nonexistent record or read beyond the last existing record of a file.
- In a REWRITE operation, the program must not attempt to rewrite a nonexistent record or rewrite beyond the last existing record.
- A REWRITE statement must write a record of the same length as the one it replaces.

For FO=IS, the following rules apply:

- All record I/O statements are available.
- A REWRITE statement must write a record of the same length as the one replaced.

RECORD TYPE AND LENGTH

The RT option for record type can indicate fixed records (F), zero byte records (Z), control word records (W), or system records (S). The possible values for the RT option are determined by the FO option and are shown in table 9-2. The fixed length option (FL) is used to specify the record length for RT=F or RT=Z files. The maximum record length option (MRL) is used to set the maximum record length for RT=W or RT=S files.

For RT=F, all records transmitted must have the exact length specified by the FL option. Since no FL/MRL option is supplied by default for a record file, FL must be specified for a record file in the ENVIRONMENT or with the FILE control statement.

For RT=Z on stream input files, each line from the CRM file is padded on the right with blanks to the length specified by the FL option. For RT=Z on stream output files, all blanks at the end of each line will be lost in transmission to the CRM file. No transmitted line can be longer than the length specified by the FL option. No occurrence of two consecutive ':' characters can occur in the data if a 64-character set is in use.

For RT=Z on record files, a record cannot be transmitted if the internal form contains twelve consecutive zero bits at the right end of any word and a 64-character set is in use.

For RT=W, all records transmitted must have a length less than or equal to the length specified by the MRL option.

For RT=S, all recrords transmitted must have a length less than or equal to the length specified by the MRL option.

RECORD KEYS

Record keys are used for a KEYED record file. The file organization for a KEYED record file can be either FO=IS or FO=WA. The program specifies key values for the I/O statements used to read or write the file. The KEY, KEYFROM, and KEYTO options of the record I/O statements specify handling of record keys.

For FO=IS files, all keys are character strings with the attributes CHARACTER nonvarying and a length greater than 0. The KL option for key length specifies the length of all keys for the file. The KL option for an OUTPUT or UPDATE file must agree with the length of the keys in the existing file; no length checking is performed. The length of the string specified by a KEY, KEYFROM, or KEYTO option must agree with the KL option for the file.

For FO=WA files, all keys are character strings containing values that are representations of nonnegative integers. The KL option is not used. All keys used in the program must be record numbers and not word addresses. The first record is addressed with a key of '0', the second record with a key of '1', and so forth. Since all records transmitted are required to have the exact length specified by the FL/MRL option, the word address of the record is calculated at run time for each access to a record. In the evaluation of any expression that specifies a key value, conversion to integer is performed. For example, '3.42' becomes 3.

A condition is an unusual situation that can occur during execution and result in a program interrupt. The situation could be related to computational error, such as dividing by zero; special status recognition, such as sensing an end-of-file; or I/O error, such as using an invalid record key. When the situation occurs, a condition is said to be raised.

PL/I defines a number of standard conditions and the circumstances under which they are raised. Each is identified by a keyword condition name. Standard conditions are raised automatically by the system. Any standard condition can be simulated by the SIGNAL statement for testing of error recovery code.

Conditions can also be defined by the programmer. Each is identified by a programmer-assigned name in conjunction with keyword CONDITION. Programmer-named conditions are raised only by SIGNAL statement execution.

When a condition is raised or simulated by the execution of a statement, the action taken depends on whether the condition is enabled or disabled for that statement. Computational conditions can be enabled or disabled through the use of condition prefixes. All other conditions are always enabled. The raising of an enabled condition causes an interrupt and activates the current established on-unit for that condition. The on-unit specifies the action that is to be taken when the condition is raised. The raising of a disabled condition is an error; the subsequent actions are described for individual conditions later in this section. The simulation of a disabled condition by SIGNAL statement is not an error; it has no effect.

Enabling a condition ensures that errors will be detected. Disabling a condition improves the execution speed of a checked-out program, but errors that occur might not be detected and can produce unpredictable results.

There are two types of on-units: programmed on-units and system on-units. A programmed on-unit is code that specifies action to be taken when the condition is raised. A system on-unit provides standard system action when the condition is raised. Every condition has exactly one current established on-unit, which can be either a programmed on-unit or the system on-unit. The system on-unit is automatically in effect for each condition that has no programmed on-unit.

CONDITION CLASSIFICATION

Conditions can be grouped into six general categories. The categories and their respective conditions are listed in table 10-1.

CONDITION PREFIXES

Condition prefixes are used to enable or disable computational conditions. As described in section 12, Statements, a condition prefix precedes the body of a statement and consists of one or more enabled condition names or disabled condition names enclosed in parentheses. Condition prefixes can appear on any statement except DECLARE, END, and ENTRY. Condition prefix syntax is

$$\left(\left\{\begin{array}{l}\text{enabled-condition-name}\\ \text{disabled-condition-name}\end{array}\right\},\ ,\ ,\right):$$

An enabled condition name can be any of the names listed in the computational category of table 10-1. A disabled condition name is formed by preceding the condition name with the letters NO. Disabled condition names are

NOCONVERSION or NOCONV
NOFIXEDOVERFLOW or NOFOFL
NOOVERFLOW or NOOFL
NOSIZE
NOSTRINGRANGE or NOSTRG
NOSUBSCRIPTRANGE or NOSUBRG
NOUNDERFLOW or NOUFL
NOZERODIVIDE or NOZDIV

A condition is explicitly enabled or disabled only within the scope of a condition prefix that names the condition. Outside that scope or in the absence of any condition prefix naming that condition, the condition assumes a default state. All conditions are enabled by default except SIZE, STRINGRANGE, and SUBSCRIPTRANGE; these three conditions are disabled by default.

A condition prefix on any statement other than BEGIN or PROCEDURE applies only to that statement. It does not apply to blocks activated during the execution of the statement. Specific rules are noted as follows:

BEGIN statement

The condition prefix applies to all statements and blocks contained in the begin block. It does not apply to contained statements or blocks that have condition prefixes identifying the same condition.

CALL statement

The condition prefix applies to expressions or arguments in the CALL statement. It does not apply to the invoked procedure.

DO statement

The condition prefix applies to expressions in the DO statement. It does not apply to statements or blocks contained in the do group.

FORMAT statement

The condition prefix applies to expressions in the FORMAT statement. It does not apply to the referencing GET or PUT statement.

GET statement

The condition prefix applies to expressions in the GET statement. It does not apply to a referenced FORMAT statement.

IF statement

The condition prefix applies to evaluation of the IF statement expression. It does not apply to any statements in the executable units following the keywords THEN and ELSE.

ON statement

The condition prefix applies to execution of the ON statement. It does not apply to the on-unit. A condition prefix on the first or only statement of the on-unit applies to the on-unit.

TABLE 10-1. CLASSIFICATION OF CONDITIONS

Condition Type	Condition Name	Default State	Can it be Disabled?
Computational	CONVERSION or CONV	Enabled	Yes
	FIXEDOVERFLOW or FOFL	Enabled	Yes
	OVERFLOW or OFL	Enabled	Yes
	SIZE	Disabled	Yes
	STRINGRANGE or STRG	Disabled	Yes
	SUBSCRIPTRANGE or SUBRG	Disabled	Yes
	UNDERFLOW or UFL	Enabled	Yes
	ZERODIVIDE or ZDIV	Enabled	Yes
I/O	ENDFILE(file-reference)	Enabled	No
	ENDPAGE(file-reference)	Enabled	No
	KEY(file-reference)	Enabled	No
	RECORD(file-reference)	Enabled	No
	TRANSMIT(file-reference)	Enabled	No
	UNDEFINEDFILE(file-reference) or UNDF(file-reference)	Enabled	No
Storage Control	AREA	Enabled	No
	STORAGE	Enabled	No
Error	ERROR	Enabled	No
Finish	FINISH	Enabled	No
Programmer-named	CONDITION(identifier) or COND(identifier)	Enabled	No

PROCEDURE statement

The condition prefix applies to all statements and blocks contained in the procedure block. It does not apply to contained statements or blocks that have condition prefixes identifying the same condition.

PUT statement

The condition prefix applies to expressions in the PUT statement. It does not apply to a referenced FORMAT statement.

ON-UNITS

Programmed and system on-units are specified with the ON statement. ON statement syntax is

[prefix]. . . ON condition [SNAP] { on-unit / SYSTEM; }

Refer to section 12 for detailed ON statement syntax.

A programmed on-unit is a begin block or a single statement. For example:

```
ON ENDFILE(FILEA) GOTO RECOVER;

        or

ON ENDFILE(FILEA)  BEGIN;
                     .
                     .
                     .
                   END;
```

A system on-unit specifies standard system action for the condition. The SYSTEM option in the ON statement indicates that the system on-unit should be activated when the condition is raised. For example:

```
ON ENDFILE(FILEA) SYSTEM;
```

The SNAP option is described later in this section.

Execution of the ON statement establishes the programmed or system on-unit for the specified condition. Establishing an on-unit does not cause it to be executed; it merely makes the on-unit available for execution when the condition is

subsequently raised. An established on-unit is always associated with the block activation in which the ON statement was executed. The on-unit is said to be established in that block activation.

Note that the ON statement is not a declarative statement. It is executed only when it is encountered in the normal flow of control.

CURRENT ESTABLISHED ON-UNIT

The current established on-unit for a condition is the on-unit that will be activated if the condition is raised. The current established on-unit can be associated with any block activation; it need not be established in the current block activation.

Several established on-units can exist for a condition; one can be associated with each block activation in the dynamic block activation stack. There is always exactly one current established on-unit for each condition, and it is found as follows:

- If no established on-unit exists for the condition in any block activation, the current established on-unit is the system on-unit for the condition. The SNAP option is assumed to be present for the ERROR condition only.
- If exactly one established on-unit exists for the condition, it is the current established on-unit.
- If more than one block activation has an established on-unit for the condition, the current established on-unit is the most recently established on-unit; that is, the one in the most recent block activation.

Figure 10-1 illustrates these three cases.

ESTABLISHING AND REMOVING ON-UNITS

Each block activation in the dynamic block activation stack can have associated with it one established on-unit for each condition. Note that ENDFILE(F) and ENDFILE(G) are two different conditions if F and G are file constants. CONDITION(RED) and CONDITION(BLUE) are also different conditions.

When a block is activated, the new block activation has no established on-units. When the block activation is terminated, all on-units established in it are discarded. While the block is active, on-units can be established and removed as follows:

- Execution of an ON statement for a condition establishes the specified programmed or system on-unit in the current block activation. If an on-unit for that condition is already established in the current block activation, the old on-unit is discarded and replaced by the new one.
- Execution of a REVERT statement for a condition removes the on-unit established for that condition in the current block activation. If there is no on-unit for the condition established in the current block activation, the REVERT statement has no effect.
- Execution of an ON statement or REVERT statement has no effect on on-units established in dynamic predecessors of the current block activation.

```
P: PROC OPTIONS(MAIN);
   CALL Q;

   Q: PROC;
      READ FILE(F) SET(POINT);
      /*SYSTEM ON-UNIT*/
      END Q;
   END P;


P: PROC OPTIONS(MAIN);
   DECLARE FIRST ENTRY;
   ON ENDFILE(F) CALL FIRST;
   CALL Q;

   Q: PROC;
      READ FILE(F) SET(POINT);
      /*CALL FIRST*/
      END Q;
   END P;


P: PROC OPTIONS(MAIN);
   DECLARE(FIRST,SECOND) ENTRY;
   ON ENDFILE(F) CALL FIRST;
   CALL Q;

   Q: PROC;
      ON ENDFILE(F) CALL SECOND;
      READ FILE(F) SET(POINT);
      /*CALL SECOND*/
      END Q;
   END P;


P: PROC OPTIONS(MAIN);
   DECLARE (Q,FIRST) ENTRY;
   ON ENDFILE(F) CALL FIRST;
   CALL Q;
   END P;

Q: PROC;
   READ FILE(F) SET(POINT);
   /*CALL FIRST*/
   END Q;
```

Comments indicate the action that will be taken if ENDFILE(F) is raised during execution of the READ statement.

The last example illustrates that the current established on-unit need not be contained in the same external procedure as the statement that causes it to be activated.

Figure 10-1. The Current Established On-Unit

Figure 10-2 illustrates the use of the ON statement to establish on-units and the use of the REVERT statement to remove established on-units.

NONLOCAL REFERENCES IN AN ON-UNIT

Any reference to an identifier that is not explicitly declared inside the on-unit itself is a nonlocal reference. Any reference in a single statement on-unit is a nonlocal reference.

```
P: PROC OPTIONS(MAIN);
   DECLARE (FIRST,SECOND,THIRD) ENTRY;
   READ FILE(F) SET(POINT);
   /*SYSTEM ON-UNIT*/
   ON ENDFILE(F) CALL FIRST;
   READ FILE(F) SET(POINT);
   /*CALL FIRST*/
   ON ENDFILE(F) CALL SECOND;
   CALL Q;

   Q: PROC;
      READ FILE(F) SET(POINT);
      /*CALL SECOND*/
      ON ENDFILE(F) CALL THIRD;
      READ FILE(F) SET(POINT);
      /*CALL THIRD*/
      REVERT ENDFILE(F);
      READ FILE(F) SET(POINT);
      /*CALL SECOND*/
      REVERT ENDFILE(F);
      READ FILE(F) SET(POINT);
      /*CALL SECOND*/
      ON ENDFILE(F) CALL THIRD;
      READ FILE(F) SET(POINT);
      /*CALL THIRD*/
      END Q;

   READ FILE(F) SET(POINT);
   /*CALL SECOND*/
   REVERT ENDFILE(F);
   READ FILE(F) SET(POINT);
   /*SYSTEM ON-UNIT*/
   END P;
```

Each comment indicates the action that will be taken if ENDFILE(F) is raised during execution of the preceding READ statement.

Figure 10-2. Establishing and Removing On-Units

Execution of a nonlocal reference sometimes requires the use of the environment of the current block activation as described in section 2, Dynamic Program Structure. The immediate environment of an on-unit activation is the block activation in which the on-unit was established.

I/O CONDITION NAMES

The name of each I/O condition contains a file reference. If the file reference is a file variable, the current file constant value of the file variable is used when the ON, REVERT, or SIGNAL statement is executed. Similarly, when an I/O condition is raised during execution of a statement that references a file variable, the current file constant value of the file variable is used in finding the current established on-unit. The use of I/O conditions is illustrated in figure 10-3.

If A and B are file constants, ENDFILE(A) and ENDFILE(B) are two different conditions. An on-unit for ENDFILE(A) and an on-unit for ENDFILE(B) can be established in the same block activation. When ENDFILE(A) is raised, the current established on-unit for ENDFILE(A) is activated; established on-units for ENDFILE(B) are irrelevant.

ON-UNIT TERMINATION

Termination of an on-unit is classified as normal or abnormal. Normal termination returns control to the block activation in which the condition was raised. The point to which control is returned depends upon the particular condition and the manner in which it was raised. Abnormal termination transfers control to some specified point in the program by a nonlocal GOTO. Any GOTO statement that transfers control to a statement outside the on-unit is a nonlocal GOTO. This is true regardless of whether the on-unit is written as a single statement or as a begin block. Normal and abnormal termination is discussed under Block Termination in section 2.

```
P: PROC OPTIONS(MAIN);
   DECLARE (F,G) FILE;
   ON ENDFILE(F) GOTO CLEANUP;
   ON ENDFILE(G) RETURN;
   CALL Q(F);
   CALL Q(G);


   Q: PROC(FPARM);
      DECLARE FPARM FILE;
      ON ENDFILE(FPARM) CLOSE FILE(FPARM);
      READ FILE(F) SET(POINT);
      .
      .
      .
      READ FILE(FPARM) SET(POINT);
      END Q;

   CLEANUP: ...
   END P;
```

During the first execution of procedure Q, an end-of-file encountered by either READ statement will cause the execution of the CLOSE statement, and will close file F.

During the second execution of procedure Q, an end-of-file encountered by the first READ statement will cause the GOTO statement to be executed, terminating the execution of both the on-unit and procedure Q. An end-of-file encountered by the second READ statement will cause the CLOSE statement to be executed, and will close file G.

Figure 10-3. Referencing I/O Conditions

Examples of normal and abnormal on-unit terminations are shown in figure 10-4.

```
ON UFL PUT . . . ;                  Normal Termination

ON CONV BEGIN;
        PUT . . . ;
        END;                        Normal Termination

ON UFL GO TO X;                     Abnormal Termination

ON CONV BEGIN;
        GO TO X;                    Abnormal Termination
        END;
```

Figure 10-4. On-Unit Termination

RAISING A CONDITION

When an enabled condition is raised, the current established on-unit for that condition is activated. It is an error to raise a disabled condition; raising a disabled condition can produce mode errors, program looping, or incorrect results.

Conditions that are not classified as computational cannot be disabled. When such conditions are raised, the associated system on-unit is always executed in the absence of a programmed on-unit. System action can be bypassed, however, by specifying a null on-unit. For example:

```
ON ENDPAGE(SYSPRINT);
```

This statement allows printing to be continued after the maximum number of lines are printed on a page.

CONDITION BUILTIN FUNCTIONS

Condition builtin functions provide information concerning interrupts. They can be referenced in an on-unit or a dynamic successor of an on-unit to determine the location and cause of the interrupt. The condition builtin function names and the values they return are as follows:

ONCHAR	Returns the character in ONSOURCE that caused the condition to be raised.
ONCODE	Returns an integer value indicating why the condition was raised. Refer to run-time diagnostics in appendix B.
ONFILE	Returns the name of the file constant that caused the condition to be raised; the name is returned even if the file reference specified a file variable.
ONKEY	Returns a character string containing the value of the KEY or KEYFROM expression.
ONLOC	Returns the name of the procedure containing the statement that caused the condition to be raised.
ONSOURCE	Returns the character string that caused CONVERSION to be raised.

When a condition is raised, the system determines condition builtin function values and maintains the values as part of the information unique to the on-unit activation. The values are stacked and unstacked as on-units are activated and terminated. When a condition builtin function is referenced in an on-unit, the system returns the most recently established value of that condition builtin function.

Values are only established for condition builtin functions applicable to the condition raised. A reference to a condition builtin function that does not have an established value yields a default value.

Pseudovariables are available for ONCHAR and ONSOURCE. These can be used to modify the source data and allow program execution to continue.

SNAP OUTPUT

The SNAP option of the ON statement specifies that diagnostic information is to be written onto file SYSPRINT when the named condition is raised. SNAP output consists of the following information:

- Error message
- Name of the condition raised
- Values of relevant condition builtin functions
- Statement number near the source of the failure
- Names of procedure blocks that are dynamic predecessors of the on-unit

If the information cannot be transmitted to SYSPRINT, an abbreviated SNAP output is transmitted to the dayfile. This can occur under any of the following circumstances:

- The declaration of SYSPRINT in the block is not a file constant declaration.
- SYSPRINT is not open as a print file and cannot be opened as a print file.
- The condition being raised is UNDF(SYSPRINT) or TRANSMIT(SYSPRINT).
- SYSPRINT has insufficient record length specified.

An example of SNAP output is shown in figure 10-5.

SEQUENCE OF OPERATIONS

When a condition is raised during program execution, the system takes the following steps in the order given:

1. Determines whether the condition is currently enabled or disabled. If the condition is disabled, steps 2 and 3 are bypassed except as noted under specific conditions.
2. Outputs diagnostic information if the SNAP option is in effect for the current established on-unit.
3. Activates the current established programmed or system on-unit, and establishes for the new on-unit activation the value of each condition builtin function that is pertinent to the condition.
4. Continues execution. The results of continuing after normal termination of an on-unit, or when the condition is disabled, vary depending upon the condition that was raised and the manner in which it was raised.

CONDITION DESCRIPTIONS

Condition names appear in the following statements:

ON statement	An on-unit is established for a condition.
REVERT statement	The on-unit established for the condition in the current block activation is removed.
SIGNAL statement	The raising of a condition is simulated.

Condition name syntax consists of the condition name only with the following exceptions:

I/O conditions require a parenthesized reference to the file.

Programmer-named conditions require a parenthesized identifier.

Conditions are presented in alphabetic order in the following paragraphs. Each condition description includes the circumstances under which the condition is raised and the processing that occurs on normal termination of the on-unit.

```
+++++ ERROR CONDITION RAISED WITH ONCODE = 9

   *********************************
SNAP GENERATED UPON OCCURRENCE OF ERROR
          ONCODE = 9

 +++ ACTIVE PROCEDURES AND ON-UNITS +++
 STMT / EXT. PROC  / PROCEDURE  / ENTRY OR ON-UNIT

      / *SYSTEM    /            / ON-ERROR
    6 / BEGIN      / BEGIN      / BEGIN
   *********************************

RPV - PREVIOUS ERROR CONDITIONS RESET.
```

Figure 10-5. Sample SNAP Output

Any of the conditions can be simulated by execution of a SIGNAL statement. On normal termination of the on-unit, program execution continues with the statement following the SIGNAL statement.

AREA CONDITION

The AREA condition is raised when an area is not large enough to contain a generation being allocated for a based variable, or is not large enough to contain an area value being assigned to the area variable. In the following example, the ALLOCATE statement raises the AREA condition because each generation allocated within an area requires one extra word for system control information.

```
DCL A AREA(100);
DCL B(100) FIXED DEC BASED(P);
ALLOC B IN(A); /*RAISES AREA*/
```

The condition is raised during the execution of the following statements:

ALLOCATE	A based variable is being allocated within an area and the referenced area does not have enough space to contain the generation. Existing generations cannot be repositioned to obtain space because offset values must be preserved.
Assignment	An area value is being assigned to an area variable and the target area is not large enough to contain the area value. Existing generations cannot be repositioned to obtain space because offset values in the target must match those in the source area.

On normal termination of the on-unit during ALLOCATE statement processing, the IN option is reevaluated and the allocation is attempted again. This can cause the AREA condition to be raised repeatedly unless the on-unit frees sufficient space in area storage. The on-unit activation must not be terminated by a nonlocal GOTO if the AREA condition was raised during ALLOCATE statement processing.

On normal termination of the on-unit during assignment statement execution, the result of the assignment operation is unpredictable. If there are remaining assignment targets, execution continues with the next one; otherwise, program execution continues with the statement following the assignment statement.

New values are established for condition builtin functions ONCODE and ONLOC when the AREA on-unit is activated.

The system AREA on-unit writes an informative message on SYSPRINT and raises the ERROR condition.

CONDITION/COND CONDITION

A CONDITION condition is a programmer-defined condition that is raised only by SIGNAL statement execution. The condition is referenced by keyword CONDITION or COND followed by an identifier in parentheses. Each unique identifier declares a separate condition. The identifier is contextually declared as a condition with the EXTERNAL attribute. For example:

```
ON CONDITION(MYNAME)  BEGIN;
                      :
                      :
                      END;
:
:
SIGNAL CONDITION(MYNAME);
```

On normal termination of the on-unit, program execution continues with the statement following the SIGNAL statement.

New values are established for condition builtin functions ONCODE and ONLOC when any CONDITION on-unit is activated.

The system CONDITION on-unit for any programmer-named condition writes an informative message on SYSPRINT and returns control to the point of interrupt.

CONVERSION/CONV CONDITION

The CONVERSION condition is raised when a character string value is invalid for conversion to an arithmetic, bit string, or pictured target data type. Such conversions occur during stream I/O, expression evaluation, and assignment. For example:

```
DCL A FIXED DEC;
A='123A4'; /*RAISES CONV*/
```

Character string conversions are executed character by character, from left to right. When an invalid character is encountered, the conversion attempt stops and the CONVERSION condition is raised. The value of the target is undefined, and remains undefined until the conversion is successfully completed or some subsequent statement execution assigns it a value.

On normal termination of the on-unit, the entire conversion is attempted again using the original source string. The user should have modified this string by assigning a value to the ONCHAR or ONSOURCE pseudovariable; if the user does not modify the string, the program might loop indefinitely. If the condition occurs during I/O statement execution, the on-unit must not have closed the file being used by the statement.

If the CONVERSION condition is disabled, execution continues and program results are unpredictable. In some cases ERROR will be raised; the ONCODE will indicate the CONVERSION error. If CONVERSION is disabled, SIGNAL CONVERSION acts like a null statement.

New values are established for condition builtin functions ONCODE, ONLOC, ONSOURCE, ONCHAR, and possibly ONFILE and ONKEY when the CONVERSION on-unit is activated. ONFILE is assigned a file name if the conversion is being performed during an I/O statement execution. The ONCHAR value is the character that caused the CONVERSION condition to be raised. The ONSOURCE value is the entire source string.

The system CONVERSION on-unit writes an informative message on SYSPRINT and raises the ERROR condition.

ENDFILE CONDITION

An ENDFILE condition is raised when an input statement attempts to read or skip past the end of a CRM file. The condition only occurs during execution of a GET statement or during execution of a READ statement without a KEY option. The ENDFILE condition is raised on each subsequent attempt to read data from the CRM file until the file associated with it is closed. Each of the following statements can raise an ENDFILE condition:

```
GET FILE(F) EDIT(A)(A);
READ FILE(G) SET(P);
READ FILE(G) SET(P) KEYTO(K);
```

On normal termination of the on-unit, program execution continues with the statement following the GET or READ statement.

New values are established for condition builtin functions ONCODE, ONLOC, and ONFILE when any ENDFILE on-unit is activated.

The system ENDFILE on-unit for any file writes an informative message on SYSPRINT and raises the ERROR condition.

ENDPAGE CONDITION

An ENDPAGE condition is raised when the program attempts to begin a line that would be beyond the page size defined for a file. The condition can occur during execution of a PUT statement or during execution of a GET statement with a COPY option while transmitting data to the copy file. The condition only occurs for files with the attributes STREAM OUTPUT PRINT. For example:

```
PUT LINE(30). . . ;
PUT LINE(20). . . ; /*RAISES ENDPAGE*/
```

On normal termination of the on-unit, the action taken depends on the circumstances under which the condition was raised.

If the condition was raised during an attempt to write data (including writing blank data for an X format or COLUMN format), the data not yet transmitted is written on return from the on-unit.

If the condition was raised during execution of a LINE format, LINE option, SKIP format, or SKIP option, the format item or option is considered to be completed.

In either case, processing of the PUT or GET statement continues. The file must not have been closed in the meantime.

When the condition is raised, the current line number has the value page size + 1, but no data has yet been transmitted to that line. If the on-unit does not cause the line number to be reset by starting a new page, line number will be incremented beyond page size by subsequent data transmission until a new page is established. In effect, there will be no page ejection. ENDPAGE for that file will not be raised again until the line number has been set to a value less than or equal to page size and subsequently exceeds page size. Line number can be set to 1 by any of the following:

- Execution of a PUT statement with a PAGE option
- Execution of a PUT statement with a LINE option with no expression or with an expression that has a value of 1
- Use of a PAGE format item
- Use of a LINE format item with no expression or with an expression that has a value of 1

New values are established for condition builtin functions ONCODE, ONLOC, and ONFILE when any ENDPAGE on-unit is activated.

The system ENDPAGE on-unit for any file begins a new page. If the condition was raised by SIGNAL statement execution and the file does not have the attribute PRINT, the ERROR condition is raised.

ERROR CONDITION

The ERROR condition is raised under the following circumstances:

- As standard action for most system on-units. No new condition builtin function values are established.
- As standard action for certain programming errors. These programming errors include mathematical builtin function errors, exponentiation errors during expression evaluation, and failure to open a file within an UNDEFINEDFILE on-unit. New values are established for condition builtin functions ONCODE and ONLOC. For example:

```
X=SQRT(-5); /*RAISES ERROR*/
```

Normal termination of the on-unit is prohibited. An attempted normal termination causes program execution to be aborted. Abnormal termination of the on-unit is permitted. An abnormal termination allows program execution to continue.

The system ERROR on-unit writes an informative message on SYSPRINT and aborts the program. The FINISH condition is not raised.

FINISH CONDITION

The FINISH condition is raised under the following circumstances:

- Control reaches a RETURN statement or END statement that would cause the first activation of the main procedure to be terminated.
- A STOP statement is executed.

When the FINISH condition is raised during execution of a RETURN or END statement, the first activation of the main procedure is the only one still active. A programmed FINISH on-unit is present only if one was established by an ON statement execution in that activation.

When the FINISH condition is raised by STOP statement execution, all the blocks that were active at that time are still active.

On normal termination of the on-unit, control is returned to the point of interrupt, program termination procedures are completed, and control is returned to the operating system. Program termination can be avoided by abnormal termination of the FINISH on-unit.

If the FINISH condition was raised by SIGNAL statement execution, the program is not terminated. The on-unit is executed and control is returned to the statement following the SIGNAL statement.

New values are established for condition builtin functions ONCODE and ONLOC when the FINISH on-unit is activated.

The system FINISH on-unit returns control to the point of interrupt.

FIXEDOVERFLOW/FOFL CONDITION

The FIXEDOVERFLOW condition is raised when the result of a fixed point calculation exceeds the maximum possible precision. The maximum precision for a binary fixed point value is 48 binary digits; the maximum precision for a decimal fixed point value is 14 decimal digits. The condition can occur during intermediate calculations.

FIXEDOVERFLOW should not be confused with the SIZE condition. The FIXEDOVERFLOW condition is raised when a value that is calculated during expression evaluation exceeds the maximum possible precision. SIZE is raised when a value being converted or assigned is too large for the declared or default precision of the target. For example:

```
DCL A(2) FIXED DEC(10,0) INIT(123456,1234567890);
DCL B FIXED DEC(10,0);
B=A(2)*A(2); /*RAISES FOFL DURING MULTIPLY*/
B=A(1)*A(1);/*RAISES SIZE DURING ASSIGNMENT*/
```

Normal termination of the on-unit is prohibited. An attempted normal termination causes the ERROR condition to be raised. Abnormal termination of the on-unit is permitted. An abnormal termination allows program execution to continue.

If the FIXEDOVERFLOW condition is disabled, execution continues from the point of interrupt and program results are unpredictable. If FOFL is disabled, SIGNAL FOFL acts like a null statement.

New values are established for condition builtin functions ONCODE and ONLOC when the FIXEDOVERFLOW on-unit is activated.

The system FIXEDOVERFLOW on-unit writes an informative message on SYSPRINT and raises the ERROR condition.

KEY CONDITION

A KEY condition is raised under one of the following circumstances:

- The value of the KEY option does not match any existing key value in the CRM file. This can be detected during execution of a READ, DELETE, or REWRITE statement.
- The value of the KEYFROM option duplicates an existing key value in the CRM file. This can be detected during execution of a WRITE or LOCATE statement, or during explicit or implicit file closing. For example:

```
WRITE FILE(F) KEYFROM(K) FROM(G);
WRITE FILE(F) KEYFROM(K) FROM(H);
    /*RAISES KEY*/
```

- The key value is not compatible with the selected CYBER Record Manager file organization.

On normal termination of the on-unit, program execution continues with the statement following the I/O statement.

New values are established for condition builtin functions ONCODE, ONLOC, ONKEY, and ONFILE when any KEY on-unit is activated.

The system KEY on-unit for any file writes an informative message on SYSPRINT and raises the ERROR condition.

OVERFLOW/OFL CONDITION

The OVERFLOW condition is raised when the result of a floating point calculation yields an exponent that exceeds the maximum allowed by the hardware. The condition can occur during intermediate calculations. For example:

```
DCL A FLOAT DEC;
A=1.E200**2; /*RAISES OFL*/
```

Normal termination of the on-unit is prohibited. An attempted normal termination causes the ERROR condition to be raised. Abnormal termination of the on-unit is permitted. An abnormal termination allows program execution to continue.

If the OVERFLOW condition is disabled, execution continues from the point of interrupt and program results are unpredictable. If OFL is disabled, SIGNAL OFL acts like a null statement.

New values are established for condition builtin functions ONCODE and ONLOC when the OVERFLOW on-unit is activated.

The system OVERFLOW on-unit writes an informative message on SYSPRINT and raises the ERROR condition.

RECORD CONDITION

A RECORD condition is raised during execution of the following:

- READ statement

  The size of the generation identified by the INTO option is different from the length of the record read from the CRM file.

- WRITE or REWRITE statement

  The CRM file has W, Z, or S type records and the size of the generation identified by the FROM option is larger than the maximum record length for the CRM file; or the CRM file has F type records and the size of the generation is different from the record length for the CRM file. In the following example, the WRITE statement raises the RECORD condition because the variable A requires 10 additional characters for overhead:

```
DCL F RECORD OUTPUT ENV('RT=F,FL=100');
DCL A CHAR(100) VARYING;
WRITE FILE(F) FROM(A); /*RAISES RECORD*/
```

- WRITE or LOCATE statement

  A record being written from an allocated buffer is longer than the maximum record length for the CRM file.

- Explicit or implicit file closing

  A record being written from an allocated buffer is longer than the maximum record length for the CRM file.

When the condition is raised during READ statement execution, the value of the target generation identified by the INTO option is as follows:

If the record is shorter than the target generation, the entire record is contained in the first part of the generation; the remainder of the generation has an undefined value.

If the record is longer than the target generation, only the first part of the record is contained in the generation; the excess portion is not transferred to the target generation.

When the condition is raised during output statement execution, the record has not been written when the on-unit is activated.

On normal termination of the on-unit activated during execution of a READ statement, program execution continues with the statement following the READ statement. On normal termination of the on-unit activated during execution of a REWRITE statement, statement processing continues from the point of interrupt; the file must not have been closed. On normal termination of the on-unit activated during execution of a WRITE or LOCATE statement or an explicit/implicit file closing, the ERROR condition is raised.

New values are established for condition builtin functions ONCODE, ONLOC, and ONFILE when any RECORD on-unit is activated. A new value is established for condition builtin function ONKEY if a KEY option is associated with the statement.

The system RECORD on-unit for any file writes an informative message on SYSPRINT and raises the ERROR condition.

SIZE CONDITION

The SIZE condition is raised during conversion of a value to a fixed point or pictured numeric target type when the converted value is too large for the precision of the target; that is, loss of significant digits would occur. For example:

```
DCL A(2) FIXED DEC(10,0) INIT(123456,1234567890);
DCL B FIXED DEC(10,0);
B=A(1)*A(1); /*RAISES SIZE DURING ASSIGNMENT*/
B=A(2)*A(2); /*RAISES FOFL DURING MULTIPLY*/
```

The SIZE condition can be raised under the following circumstances:

- Conversion to fixed point

  During assignment or during execution of an E or F format item by a PUT statement, SIZE is raised if the number of digits available to the left of the decimal or binary point is not large enough to contain the integer part of the converted value.

- Conversion to fixed point during intermediate calculations

  Conversion from FLOAT to BIT requires intermediate conversion to FIXED BINARY. SIZE can be raised during conversion to FIXED BINARY.

  Conversion from CHARACTER to arithmetic during expression evaluation such as (1.E20+'2.E20') requires intermediate conversion to FIXED DECIMAL (14,0) even if the type of the entire expression is FLOAT. SIZE can be raised during this intermediate conversion.

- Conversion to pictured numeric during edit through picture

  During assignment or during execution of a P format by a PUT statement, SIZE is raised when the picture is not large enough to contain the integer part of the value; or when the value is negative and the picture does not contain any of the sign specification codes S + - T I R CR or DB.

  During execution of an E or F format item by a PUT statement, SIZE is raised if the implied picture is not large enough to contain the integer part of the converted value.

  During execution of an E format item by a PUT statement, SIZE is raised if the implied picture is not large enough to contain the exponent, signs, decimal point, and E.

Normal termination of the on-unit is prohibited. An attempted normal termination causes the ERROR condition to be raised. Abnormal termination of the on-unit is permitted. An abnormal termination allows program execution to continue.

If the SIZE condition is disabled, execution continues from the point of interrupt and program results are unpredictable. In some cases ERROR will be raised; the ONCODE will indicate the SIZE error. If SIZE is disabled, SIGNAL SIZE acts like a null statement.

New values are established for condition builtin functions ONCODE and ONLOC when the SIZE on-unit is activated.

The system SIZE on-unit writes an informative message on SYSPRINT and raises the ERROR condition.

STORAGE CONDITION

The STORAGE condition is raised when an attempt is made to allocate storage and the required amount is not available. The condition can be raised during allocation of storage for variables of any storage type, for block activations, for dynamic loading of system routines, and so forth. The condition is not raised during allocation of based variables in areas; in this case the AREA condition is raised. The most common causes of the STORAGE condition are incorrect extents, insufficient field length, uncontrolled looping involving recursive procedure invocations, and failure to free based variables that are not allocated in areas. For example:

```
DCL A(10) FIXED BASED;
DO WHILE('1'B); /*LOOPS FOREVER*/
ALLOC A SET(P); /*RAISES STORAGE*/
END;
```

On normal termination of the on-unit, allocation is attempted again. If the on-unit has freed sufficient storage, the allocation is made and execution continues normally. If the on-unit has not freed sufficient storage, the STORAGE condition is raised again. If the allocation cannot be made after the STORAGE condition has been raised twice, the ERROR condition is raised.

Abnormal termination of the STORAGE on-unit is prohibited.

If the STORAGE condition is signaled, both normal and abnormal termination of the on-unit are permitted. For normal termination of the on-unit, program execution continues with the statement following the SIGNAL statement.

The system STORAGE on-unit writes an informative message on SYSPRINT and raises the ERROR condition.

NOTE

The raising of the STORAGE condition requires storage allocation. If insufficient storage is available for this, or if insufficient storage is available for any allocation while a STORAGE on-unit is active, the ERROR condition will be raised.

STRINGRANGE/STRG CONDITION

The STRINGRANGE condition is raised during evaluation of a SUBSTR builtin function or pseudovariable when the indicated substring is not entirely contained in the string specified. For example:

```
DCL(A,B) CHAR(10);
B=SUBSTR(A,5,8); /*RAISES STRG*/
B=SUBSTR(A,10,2); /*RAISES STRG*/
```

If the substring has a length of zero, it is legal to reference n+1 where n is the last character in the string. For example:

```
DCL(A,B) CHAR(10);
B=SUBSTR(A,11,0); /*DOES NOT RAISE STRG*/
B=SUBSTR(A,12,0); /*RAISES STRG*/
```

Normal termination of the on-unit is prohibited. An attempted normal termination causes the ERROR condition to be raised. Abnormal termination of the on-unit is permitted. An abnormal termination allows program execution to continue.

The STRINGRANGE condition is raised in the same way, regardless of whether it is enabled or disabled. If STRG is disabled, SIGNAL STRG acts like a null statement.

New values are established for condition builtin functions ONCODE and ONLOC when the STRINGRANGE on-unit is activated.

The system on-unit writes an informative message on SYSPRINT and raises the ERROR condition.

SUBSCRIPTRANGE/SUBRG CONDITION

The SUBSCRIPTRANGE condition is raised when an array subscript is evaluated and is found to be outside the bounds of the corresponding dimension. For example:

```
DCL A(5);
A(6)=3; /*RAISES SUBRG*/
```

Normal termination of the on-unit is prohibited. An attempted normal termination causes the ERROR condition to be raised. Abnormal termination of the on-unit is permitted. An abnormal termination allows program execution to continue.

If the SUBSCRIPTRANGE condition is disabled, execution continues from the point of interrupt and program results are unpredictable. If SUBRG is disabled, SIGNAL SUBRG acts like a null statement.

New values are established for condition builtin functions ONCODE and ONLOC when the SUBSCRIPTRANGE on-unit is activated.

The system on-unit writes an informative message on SYSPRINT and raises the ERROR condition.

TRANSMIT CONDITION

A TRANSMIT condition is raised when data cannot be properly transmitted to or from a CRM file; that is, when parity error status is returned by the CRM I/O processors. The condition can be raised during the following operations:

I/O statement execution

Implicit file closing during program termination

When the condition is raised, the value of the data that was being transmitted is undefined at the target.

On normal termination of the on-unit, program execution continues with the next statement.

New values are established for condition builtin functions ONCODE, ONLOC, and ONFILE when any TRANSMIT on-unit is activated. A new value is established for condition builtin function ONKEY if a KEY or KEYFROM option is associated with the statement or implicit file closing being executed.

The system TRANSMIT on-unit for any file writes an informative message on SYSPRINT and raises the ERROR condition.

UNDEFINEDFILE/UNDF CONDITION

An UNDEFINEDFILE condition is raised when a file opening fails. The condition can occur during an explicit file opening (OPEN statement) or an implicit file opening when the FILE or COPY option of an I/O statement refers to the unopened file. The most common causes of file opening failure are conflict of file attributes and inability to associate the specified CRM file with the file constant. When the on-unit is activated, the file is not open. For example:

```
DCL F RECORD;
PUT FILE(F) LIST(A); /*RAISES UNDF*/
```

On normal termination of the on-unit during implicit file opening, the ERROR condition is raised if the file is still not open. If the file is open, processing continues normally.

On normal termination of the on-unit during OPEN statement execution, statement processing continues with the next file to be opened if additional files are specified in the statement. If additional files are not specified, program execution continues with the statement following the OPEN statement.

New values are established for condition builtin functions ONCODE, ONLOC, and ONFILE when any UNDEFINEDFILE on-unit is activated.

The system UNDEFINEDFILE on-unit for any file writes an informative message on SYSPRINT and raises the ERROR condition.

UNDERFLOW/UFL CONDITION

The UNDERFLOW condition is raised when the result of a floating point calculation yields an exponent that is smaller than the minimum allowed by the hardware. For example:

```
A=1.5E-200**2; /*RAISES UFL*/
```

If termination of the on-unit is normal or if the condition is disabled, processing continues from the point of interrupt. The result of the operation that raised UNDERFLOW is taken to be zero.

New values are established for condition builtin functions ONCODE and ONLOC when the UNDERFLOW on-unit is activated.

The system UNDERFLOW on-unit writes an informative message on SYSPRINT and returns control to the point of interrupt.

ZERODIVIDE/ZDIV CONDITION

The ZERODIVIDE condition is raised when an attempt is made to divide by zero. The condition can occur for fixed point and floating point division operations. When the on-unit is activated, the value of the quotient is undefined. For example:

```
ZERO=0.;
A=1./ZERO; /*RAISES ZDIV*/
```

Normal termination of the on-unit is prohibited. An attempted normal termination causes the ERROR condition to be raised. Abnormal termination of the on-unit is permitted. An abnormal termination allows program execution to continue.

If the ZERODIVIDE condition is disabled, execution continues from the point of interrupt and program results are unpredictable. If ZDIV is disabled, SIGNAL ZDIV acts like a null statement.

New values are established for condition builtin functions ONCODE and ONLOC when the ZERODIVIDE on-unit is activated.

The system ZERODIVIDE on-unit writes an informative message on SYSPRINT and raises the ERROR condition.

PL/I supports a number of functions, referred to as builtin functions, which are provided as an integral part of the language. Each is identified by a builtin function name. When the name is appropriately referenced in a PL/I program, the operation indicated by the name is performed on the arguments passed.

An identifier is contextually declared with the BUILTIN attribute whenever the builtin function name appears as a function reference with an argument list. A builtin function usually does not require an explicit DECLARE statement declaration. If the builtin function name is used in a containing block for some other purpose, however, an explicit declaration must be made to reestablish the name as the name of the builtin function.

Builtin function arguments are enclosed in parentheses and follow the builtin function name. For example:

```
V = HBOUND(ARRAY1,2);
```

HBOUND is an array manipulation builtin function that determines the upper bound of an array dimension.

The first argument, ARRAY1, is the name of the array.

The second argument, 2, specifies the dimension.

V is the target variable designated to receive the result.

Some builtin functions do not require arguments; these are represented by the builtin function name followed by a set of empty parentheses. The empty parentheses can be omitted; if they are omitted from every reference to a particular builtin function within any external procedure, that builtin function name must be explicitly declared with the BUILTIN attribute.

BUILTIN FUNCTION CLASSIFICATION

Builtin functions can be grouped into nine functional categories. The categories and their respective functions are listed in table 11-1. A complete summary of mathematical builtin functions is included in table 11-2.

ARITHMETIC BUILTIN FUNCTIONS

Arithmetic builtin functions are used to facilitate arithmetic calculations. Arguments must be expressions that yield scalar computational values. Each argument is converted to arithmetic as described in section 7, Data Manipulation. The value returned by each of these functions is arithmetic.

ARRAY MANIPULATION BUILTIN FUNCTIONS

Array manipulation builtin functions are used to acquire information about arrays. The value returned by each of these functions is a fixed binary integer.

CONDITION BUILTIN FUNCTIONS

Condition builtin functions are used to acquire information about the circumstances under which a condition has been raised. Condition builtin function values are established when applicable conditions are raised. A stack of values is maintained for each condition builtin function. The values are stacked and unstacked as on-units are activated and terminated.

A reference to a condition builtin function returns the value associated with the most recent applicable on-unit activation in the dynamic stack. A reference to a condition builtin function that does not have an established value yields a default value as indicated in the individual descriptions.

DATE AND TIME BUILTIN FUNCTIONS

Date and time builtin functions are used to obtain the current date and time. The value returned by each of these functions is a character string.

MATHEMATICAL BUILTIN FUNCTIONS

Mathematical builtin functions are standard mathematical functions. Arguments must be expressions that yield scalar computational values. Each argument is converted to floating point; conversion to floating point arithmetic is as described in section 7, Data Manipulation. The value returned by each of these functions is a floating point value with mode, base, and precision the same as those of the converted argument.

For information on the accuracy of mathematical builtin functions, see the FORTRAN Common Library Mathematical Routines Reference Manual.

PICTURE HANDLING BUILTIN FUNCTION

The picture handling builtin function is used to perform validity checking on a pictured item. The function determines whether or not the value of an item is valid according to its numeric or character picture. The value returned by this function is a bit string.

STORAGE CONTROL BUILTIN FUNCTIONS

Storage control builtin functions are used to assist the programmer in manipulating controlled variables, based variables, and areas.

STREAM I/O BUILTIN FUNCTIONS

Stream I/O builtin functions are used to obtain the current line and page number of stream print files. The value returned by each of these functions is a fixed binary integer.

TABLE 11-1. CLASSIFICATION OF BUILTIN FUNCTIONS

Functional Category	Builtin Function Name	Value Returned
Arithmetic	ABS	Absolute value of an argument.
	ADD	Sum of two arguments with specified precision.
	BINARY	Argument value converted to binary with specified precision.
	CEIL	Smallest integer that is greater than or equal to an argument value.
	DECIMAL	Argument value converted to decimal with specified precision.
	DIVIDE	Quotient of two arguments with specified precision.
	FIXED	Argument value converted to fixed-point scale with specified precision.
	FLOAT	Argument value converted to floating-point scale with specified precision.
	FLOOR	Largest integer that does not exceed an argument value.
	MAX	Value of the argument with the largest value.
	MIN	Value of the argument with the smallest value.
	MOD	First argument modulo second argument.
	MULTIPLY	Product of two arguments with specified precision.
	PRECISION	Argument value converted to a given precision.
	ROUND	Value of an argument rounded to a specified number of digits.
	SIGN	Indication of whether an argument is positive, negative, or zero.
	SUBTRACT	Difference between two arguments with specified precision.
	TRUNC	Argument value truncated to an integer.
Array Manipulation	DIMENSION	Current extent of an array dimension.
	HBOUND	Upper bound of an array dimension.
	LBOUND	Lower bound of an array dimension.
Condition	ONCHAR	Character that caused the CONVERSION condition to be raised.
	ONCODE	Code that identifies a condition and circumstances under which it was raised.
	ONFILE	Name of the file for which an I/O condition was raised.
	ONKEY	Key value for which an I/O condition was raised.
	ONLOC	Name of the entry point used to invoke the procedure in which a condition was raised.
	ONSOURCE	Character-string value that caused the CONVERSION condition to be raised.
Date and Time	DATE	Current date.
	TIME	Current time.
Mathematical	ACOS	Arccosine in radians.
	ASIN	Arcsine in radians.
	ATAN	Arctangent in radians.
	ATAND	Arctangent in degrees.
	ATANH	Inverse hyperbolic tangent.
	COS	Trigonometric cosine of an argument in radians.
	COSD	Trigonometric cosine of an argument in degrees.
	COSH	Hyperbolic cosine of an argument.
	ERF	Error function.

TABLE 11-1. CLASSIFICATION OF BUILTIN FUNCTIONS (Cont'd)

Functional Category	Builtin Function Name	Value Returned
Mathematical (continued)	ERFC	Complementary error function.
	EXP	Value of e raised to a given power.
	LOG	Natural logarithm (base e).
	LOG10	Common logarithm (base 10).
	LOG2	Base 2 logarithm.
	SIN	Trigonometric sine of an argument in radians.
	SIND	Trigonometric sine of an argument in degrees.
	SINH	Hyperbolic sine of an argument.
	SQRT	Square root.
	TAN	Tangent of an argument in radians.
	TAND	Tangent of an argument in degrees.
	TANH	Hyperbolic tangent of an argument.
Picture Handling	VALID	Indication of the validity of a pictured item.
Storage Control	ADDR	Pointer value identifying the generation of storage for a variable.
	ALLOCATION	Value indicating the number of generations currently allocated for a controlled variable.
	EMPTY	Empty area value which, if assigned, frees all allocations in an area.
	NULL	Null pointer value.
	OFFSET	Pointer argument value converted to offset.
	POINTER	Offset argument value converted to pointer.
Stream I/O	LINENO	Current line number of a stream print file.
	PAGENO	Current page number of a stream print file.
String Handling	AFTER	Portion of a string that appears to the right of a substring.
	BEFORE	Portion of a string that appears to the left of a substring.
	BIT	Argument value converted to bit string.
	BOOL	Bit-string result of a specified boolean operation.
	CHARACTER	Argument value converted to character string.
	COLLATE	Collating sequence in use.
	COPY	String formed by repeated concatenation.
	DECAT	String composed of portions of a given string.
	HIGH	String consisting of copies of the highest character in the collating sequence.
	INDEX	Position of a substring within a string.
	LENGTH	Current length of a character or bit string or a pictured variable.
	LOW	String consisting of copies of the lowest character in the collating sequence.
	REVERSE	Argument string with characters positioned in reverse order.
	SUBSTR	Specified substring of an argument string.
	TRANSLATE	Character string on which specified character substitutions have been performed.
	UNSPEC	Internal representation of an argument value in bit-string form.
	VERIFY	Position of the first character in one string that does not appear in a second string.

TABLE 11-2. SUMMARY OF MATHEMATICAL BUILTIN FUNCTIONS

Syntax	Name	Domain	Definition	Range
ACOS(y)	arccosine (result in radians)	$\|y\| \leq 1$	$\cos^{-1}(y)$	$0 \leq \text{ACOS}(y) \leq \pi$
ASIN(y)	arcsine (result in radians)	$\|y\| \leq 1$	$\sin^{-1}(y)$	$-\pi/2 \leq \text{ASIN}(y) \leq \pi/2$
ATAN(y)	arctangent (result in radians)		$\tan^{-1}(y)$	$-\pi/2 \leq \text{ATAN}(y) \leq \pi/2$
ATAN(y,x)	arctangent (result in radians)	$x<0$ & $y<0$	$-\pi + \tan^{-1}(y/x)$	$-\pi < \text{ATAN}(y,x) < -\pi/2$
		$x=0$ & $y<0$	$-\pi/2$	
		$x>0$	$\tan^{-1}(y/x)$	$-\pi/2 < \text{ATAN}(y,x) < \pi/2$
		$x=0$ & $y>0$	$\pi/2$	
		$x<0$ & $y\geq 0$	$\pi + \tan^{-1}(y/x)$	$\pi/2 < \text{ATAN}(y,x) \leq \pi$
		$x=0$ & $y=0$	error	
ATAND(y)	arctangent (result in degrees)		$(180/\pi)*\text{ATAN}(y)$	$-90 \leq \text{ATAND}(y) \leq 90$
ATAND(y,x)	arctangent (result in degrees)	$\neg(x=0 \,\&\, y=0)$	$(180/\pi)*\text{ATAN}(y,x)$	$-180 < \text{ATAND}(y,x) \leq 180$
		$x=0$ & $y=0$	error	
ATANH(y)	inverse hyperbolic tangent	$\|y\| < 1$	$\tanh^{-1}(y)$	$-17.32 < \text{ATANH}(y) < 17.32$
COS(x)	trigonometric cosine (argument in radians)	$\|x\| \leq \pi * 2^{46}$	cos(x)	$-1 \leq \text{COS}(x) \leq 1$
COSD(x)	trigonometric cosine (argument in degrees)	$\|x\| < 2^{47}$	cos(x)	$-1 \leq \text{COSD}(x) \leq 1$
COSH(x)	hyperbolic cosine	$\|x\| \leq 742.36$	cosh(x)	$1 \leq \text{COSH}(x)$
ERF(x)	error function		$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$	$-1 \leq \text{ERF}(x) \leq 1$
ERFC(x)	complementary error function	$x < 25.923$	$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt = 1 - \text{erf}(x)$	$0 < \text{ERFC}(x) \leq 2$
EXP(x)	exponential	$-675.84 \leq x \leq 741.67$	e^x	$0 < \text{EXP}(x)$
LOG(x)	natural logarithm	$x>0$	$\log_e(x)$	
LOG10(x)	common logarithm (base 10)	$x>0$	$\log_{10}(x)$	
LOG2(x)	logarithm (base 2)	$x>0$	$\log_2(x)$	
SIN(x)	trigonometric sine (argument in radians)	$\|x\| \leq \pi * 2^{46}$	sin(x)	$-1 \leq \text{SIN}(x) \leq 1$
SIND(x)	trigonometric sine (argument in degrees)	$\|x\| < 2^{47}$	sin(x)	$-1 \leq \text{SIND}(x) \leq 1$
SINH(x)	hyperbolic sine	$\|x\| \leq 742.36$	sinh(x)	
SQRT(x)	square root	$x \geq 0$	$x^{0.5}$	$\text{SQRT}(x) \geq 0$
TAN(x)	trigonometric tangent (argument in radians)	$\|x\| \leq \pi * 2^{46}$	tan(x)	
TAND(x)	trigonometric tangent (argument in degrees)	$\|x\| < 2^{47}$	tan(x)	
TANH(x)	hyperbolic tangent		tanh(x)	$-1 \leq \text{TANH}(x) \leq 1$

STRING HANDLING BUILTIN FUNCTIONS

String handling builtin functions are used for manipulating bit and character strings. The arguments must be expressions that have scalar computational values. The value returned by each of these functions is a character or bit string, or a fixed binary integer.

PSEUDOVARIABLES

PL/I supports five pseudovariables, which are counterparts of the following builtin functions:

ONCHAR
ONSOURCE
PAGENO
SUBSTR
UNSPEC

A pseudovariable is a builtin function name that is used as the target of an assignment operation. Pseudovariables can be used in the following ways:

- As an assignment target in an assignment statement
- As an index in a DO statement or in an embedded-do in a GET or PUT statement
- As a key target in a KEYTO option of a READ statement
- As an output target in the STRING option of a PUT statement
- As an input target in the LIST or EDIT option of a GET statement

Arguments for a pseudovariable correspond to those specified for the associated builtin function. Arguments are enclosed in parentheses and follow the pseudovariable name. For example:

```
PAGENO(OUTFILE) = 1;
```

PAGENO is a stream I/O pseudovariable that assigns a value to the current page number for a stream print file.

The argument, OUTFILE, specifies the name of the file.

The expression, 1, is the value that is to be assigned to the current page number of OUTFILE.

Some pseudovariables do not require arguments; they are represented by the pseudovariable name followed by a set of empty parentheses. The empty parentheses can be omitted; if they are omitted from every reference to a particular pseudovariable, that pseudovariable name must be explicitly declared with the BUILTIN attribute.

BUILTIN FUNCTION AND PSEUDOVARIABLE DESCRIPTIONS

Builtin functions and corresponding pseudovariables, where appropriate, appear in alphabetic order. The descriptions of pseudovariables contain brief explanations of the differences in usage between the pseudovariables and the corresponding builtin functions. The builtin function description contains more specific information.

ABS BUILTIN FUNCTION

The ABS builtin function returns the absolute value of an argument. ABS syntax is shown in figure 11-1.

```
ABS(x)
```

Figure 11-1. ABS Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. The argument is evaluated and converted to an arithmetic value. The result of the function is the positive real value of the converted argument. The base, scale, and precision of the result are those of the converted argument.

ABS builtin function examples are shown in figure 11-2.

```
X                ABS(X)

319              319
-3.4500          3.4500
2.3E+013         2.3E+013
-5.13E-070       5.13E-070
```

Figure 11-2. ABS Builtin Function Examples

ACOS BUILTIN FUNCTION

The ACOS builtin function returns the arccosine of a value; the result is expressed in radians. ACOS syntax is shown in figure 11-3.

```
ACOS(y)
```

Figure 11-3. ACOS Builtin Function Syntax

Argument y can be an expression; the value must be scalar and computational. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

ACOS builtin function examples are shown in figure 11-4.

```
Y                ACOS(Y)

1.00000E+000     0.00000E+000
5.00000E-001     1.04720E+000
0.00000E+000     1.57080E+000
-5.00000E-001    2.09440E+000
-1.00000E+000    3.14159E+000
-1.00000B        3.1E+000
-1               3E+000
```

Figure 11-4. ACOS Builtin Function Examples

ADD BUILTIN FUNCTION

The ADD builtin function returns the sum of two values with a specified precision. ADD syntax is shown in figure 11-5.

```
ADD(x,y,p[,q])
```

Figure 11-5. ADD Builtin Function Syntax

Arguments x and y specify the two values to be added; each can be an expression and must be scalar and computational. Arguments p and q specify the precision of the result; they must be decimal integers. Argument q can be signed.

Arguments x and y are evaluated and converted to arithmetic values of the common base and scale. If the common scale is fixed point, q can be specified and must be within the range -255<=q and q<=255; if q is not specified, it is assumed to be zero. If the common scale is floating point, q cannot be specified. In either case, p must be greater than zero and must not exceed maximum precision (14 for decimal, 48 for binary).

The result of the function is the sum of the converted arguments x and y. The base and scale of the result are those of the converted arguments. The precision of the result is (p) or (p,q).

ADD builtin function examples are shown in figure 11-6.

```
     X              Y       P   Q   ADD(X,Y,P,Q)

   3               4        5   3      7.000
  13.45          2E+000     7  ---  1.545000E+001
  0.142857        17.356    8   3       17.498
  0.142857        17.356    6  ---        17
 1.42857E-001     17.356    6  ---  1.74989E+001
1.01B           1.10001B    8   3      2.8
```

Figure 11-6. ADD Builtin Function Examples

ADDR BUILTIN FUNCTION

The ADDR builtin function returns a scalar pointer value that identifies the generation of storage associated with the argument. ADDR syntax is shown in figure 11-7.

```
ADDR(x)
```

Figure 11-7. ADDR Builtin Function Syntax

Argument x must specify a variable with a generation that has connected storage. The variable can have any data type and storage type.

If x is a controlled variable that has no generations of storage, the result of the function is a null pointer. If x is not a controlled variable, it must have a generation of storage.

ADDR builtin function examples are shown in figure 11-8.

```
DCL 1 A CONTROLLED,
      2 B CHAR(5),
      2 C CHAR(5),
      2 D(12) CHAR(10);
DCL 1 X BASED(P) LIKE A;
DCL Z(12) CHAR(10) BASED(R);
ALLOCATE A;
P=ADDR(A);
R=ADDR(X.D);
```

Figure 11-8. ADDR Builtin Function Examples

AFTER BUILTIN FUNCTION

The AFTER builtin function examines two strings. If the second string occurs as a substring of the first, AFTER returns the portion of the first string that appears to the right of the first instance of the substring. AFTER syntax is shown in figure 11-9.

```
AFTER(string1,string2)
```

Figure 11-9. AFTER Builtin Function Syntax

Each argument can be an expression; the values must be scalar and computational. If arguments string1 and string2 are both bit strings, the result is a bit string; any other combination results in a character string. The arguments are first converted to the result type.

The result of the function is the portion of string1 that occurs to the right of the first instance of substring. If string2 is a null string, the result of the function is string1.

The result of the function is a null string under any of the following circumstances:

String1 is a null string.

String2 is not a substring of string1.

String2 appears once as a substring of string1, and no characters or bits appear in string2 to the right of the substring.

AFTER builtin function examples are shown in figure 11-10.

```
                                  AFTER
STRING1          STRING2          (STRING1,STRING2)

↑ABCDEFG↑        ↑CD↑             ↑EFG↑
↑ABCDEFCDG↑      ↑CD↑             ↑EFCDG↑
↑ABCDEFG↑        ↑DC↑             ↑↑
↑↑               ↑CD↑             ↑↑
↑ABCDEFG↑        ↑↑               ↑ABCDEFG↑
↑ABCDEFG↑        ↑FG↑             ↑↑
↑101101↑B        ↑01↑B            ↑101↑B
↑101101↑B        ↑01↑             ↑101↑
```

Figure 11-10. AFTER Builtin Function Examples

ALLOCATION BUILTIN FUNCTION

The ALLOCATION builtin function determines whether or not storage has been allocated for the controlled variable specified by the argument, and returns the number of generations currently allocated. ALLOCATION syntax is shown in figure 11-11.

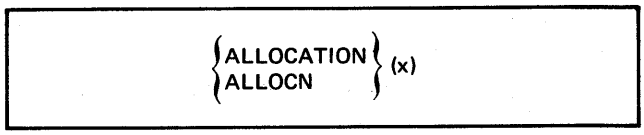


Figure 11-11. ALLOCATION Builtin Function Syntax

Argument x must specify a controlled variable that is unsubscripted and not a structure member.

If no generations are currently allocated for the controlled variable, the result of the function is zero; otherwise, the result is the number of generations currently allocated for the variable. The returned value is a fixed binary integer.

ASIN BUILTIN FUNCTION

The ASIN builtin function returns the arcsine of a value; the result is expressed in radians. ASIN syntax is shown in figure 11-12.

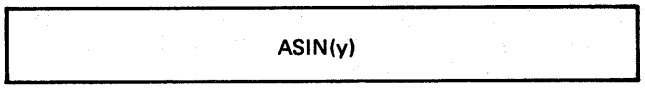


Figure 11-12. ASIN Builtin Function Syntax

Argument y can be an expression; the value must be scalar and computational. Argument y is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

ASIN builtin function examples are shown in figure 11-13.

```
          Y                    ASIN(Y)

 -1.00000E+000        -1.57080E+000
 -5.00000E-001        -5.23599E-001
  0.00000E+000         0.00000E+000
  5.00000E-001         5.23599E-001
  1.00000E+000         1.57080E+000
```

Figure 11-13. ASIN Builtin Function Examples

ATAN BUILTIN FUNCTION

The ATAN builtin function returns the arctangent of a value; the result is expressed in radians. ATAN syntax is shown in figure 11-14.

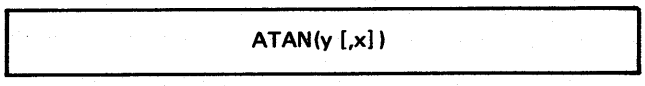


Figure 11-14. ATAN Builtin Function Syntax

Each argument can be an expression; the values must be scalar and computational. Arguments y and x are evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted arguments.

ATAN(y) returns the arctangent of the value of an expression y. The range of the result is

$-(\pi/2) < ATAN(y)$ and $ATAN(y) < (\pi/2)$

ATAN(y,x) treats its two arguments as the Cartesian coordinates of a point in the plane. The function returns the angle between the positive x-axis and the line joining the origin to the point (x,y). It is an error if x and y are both zero. The range of the result is

$-\pi < ATAN(y,x)$ and $ATAN(y,x) <= \pi$

ATAN(y,x) satisfies the identity

ATAN(SIN(z),COS(z))=z for all z within the range $-\pi < z$ and $z <= \pi$

Conversion from rectangular coordinates (x,y) to polar coordinates (r,theta) can be accomplished by

```
r=SQRT(x**2+y**2);
theta=ATAN(y,x);
```

ATAN builtin function examples are shown in figure 11-15.

```
          Y                    ATAN(Y)

 -1.00000E+320        -1.57080E+000
 -1.00000E+000        -7.85398E-001
  0.00000E+000         0.00000E+000
  1.00000E+000         7.85398E-001
  1.00000E+320         1.57080E+000



          Y                    X                    ATAN(Y,X)

 -1.00000E+000        -1.00000E+000        -2.35619E+000
 -1.00000E+000         0.00000E+000        -1.57080E+000
 -1.00000E+000         1.00000E+000        -7.85398E-001
  0.00000E+000         1.00000E+000         0.00000E+000
  1.00000E+000         1.00000E+000         7.85398E-001
  1.00000E+000         0.00000E+000         1.57080E+000
  1.00000E+000        -1.00000E+000         2.35619E+000
  0.00000E+000        -1.00000E+000         3.14159E+000
```

Figure 11-15. ATAN Builtin Function Examples

ATAND BUILTIN FUNCTION

The ATAND builtin function returns the arctangent of a value; the result is expressed in degrees. ATAND syntax is shown in figure 11-16.

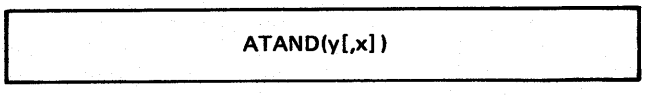


Figure 11-16. ATAND Builtin Function Syntax

Each argument can be an expression; the values must be scalar and computational. Arguments y and x are evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted arguments.

ATAND(y) returns the arctangent of the value of an expression y. The range of the result is

-90<ATAND(y) and ATAND(y)<90

ATAND(y,x) is identical to ATAN(y,x), except that ATAND returns an angle expressed in degrees. The range of the result is

-180<ATAND(y,x) and ATAND(y,x)<=180

The definition of ATAND is

ATAND(y[,x])=(180/π)*ATAN(y[,x])

ATAND builtin function examples are shown in figure 11-17.

```
      Y             ATAND(Y)

-1.00000E+320   -9.00000E+001
-1.00000E+000   -4.50000E+001
 0.00000E+000    0.00000E+000
 1.00000E+000    4.50000E+001
 1.00000E+320    9.00000E+001



      Y              X            ATAND(Y,X)

-1.00000E+000  -1.00000E+000  -1.35000E+002
-1.00000E+000   0.00000E+000  -9.00000E+001
-1.00000E+000   1.00000E+000  -4.50000E+001
 0.00000E+000   1.00000E+000   0.00000E+000
 1.00000E+000   1.00000E+000   4.50000E+001
 1.00000E+000   0.00000E+000   9.00000E+001
 1.00000E+000  -1.00000E+000   1.35000E+002
 0.00000E+000  -1.00000E+000   1.80000E+002
```

Figure 11-17. ATAND Builtin Function Examples

ATANH BUILTIN FUNCTION

The ATANH builtin function returns the inverse hyperbolic tangent of a value. ATANH syntax is shown in figure 11-18.

```
ATANH(y)
```

Figure 11-18. ATANH Builtin Function Syntax

Argument y can be an expression; the value must be scalar and computational. Argument y must have an absolute value of less than 1. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

ATANH builtin function examples are shown in figure 11-19.

```
          X                    ATANH(X)

-9.99999E-001              -7.25433E+000
-5.00000E-001              -5.49306E-001
 0.00000E+000               0.00000E+000
 5.00000E-001               5.49306E-001
 9.99990E-001               6.10303E+000
```

Figure 11-19. ATANH Builtin Function Examples

BEFORE BUILTIN FUNCTION

The BEFORE builtin function examines two strings. If the second string occurs as a substring of the first, BEFORE returns the portion of the first string that appears to the left of the first instance of the substring. BEFORE syntax is shown in figure 11-20.

```
BEFORE(string1,string2)
```

Figure 11-20. BEFORE Builtin Function Syntax

Each argument can be an expression; the values must be scalar and computational. If arguments string1 and string2 are both bit strings, the result is a bit string; any other combination results in a character string. The arguments are first converted to the result type.

The result of the function is the portion of string1 that occurs to the left of the first character or bit of the substring.

The result of the function is a null string under any of the following circumstances:

String1 is a null string.

String2 is a null string.

String2 appears as a substring of string1, beginning with the first character of string1.

BEFORE builtin function examples are shown in figure 11-21.

```
                                BEFORE
STRING1          STRING2        (STRING1,STRING2)

↑ABCDEFG↑        ↑CD↑           ↑AB↑
↑ABCDEFCDG↑      ↑CD↑           ↑AB↑
↑ABCDEFG↑        ↑DC↑           ↑ABCDEFG↑
↑↑               ↑CD↑           ↑↑
↑ABCDEFG↑        ↑↑             ↑↑
↑ABCDEFG↑        ↑AB↑           ↑↑
↑101101↑B        ↑01↑B          ↑1↑B
↑101101↑B        ↑01↑           ↑1↑
```

Figure 11-21. BEFORE Builtin Function Examples

BINARY BUILTIN FUNCTION

The BINARY builtin function converts a value to binary base with a specified precision and returns the binary value. BINARY syntax is shown in figure 11-22.

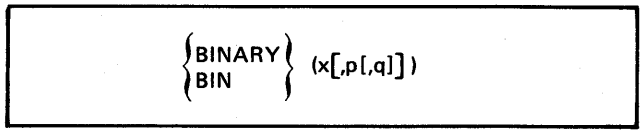


Figure 11-22. BINARY Builtin Function Syntax

Argument x specifies the value to be converted to binary; the argument can be an expression and the value must be scalar and computational. Arguments p and q specify the precision of the result. Argument p must be an unsigned decimal integer within the range 1<=p and p<=48. Argument q must be a decimal integer within the range -255<=q and q<=255. If the converted argument is floating point, q cannot be specified.

The result of the function is the binary value of the converted argument. The result precision is (p) or (p,q) if p and q are specified. If neither p nor q is specified, precision is determined by the standard conversion rules.

BINARY builtin function examples are shown in figure 11-23.

X	P	Q	BIN(X,P,Q)	DECIMAL EQUIVALENT
0.1	4	4	.0001B	0.06
0.1	8	8	.00011001B	0.098
1E-001	9	---	1.10011001E-4B	1.00E-001
↑1101↑B	8	3	01101.000B	13.0

Figure 11-23. BINARY Builtin Function Examples

BIT BUILTIN FUNCTION

The BIT builtin function converts an argument to a bit string of a specified length and returns the bit string value. BIT syntax is shown in figure 11-24.

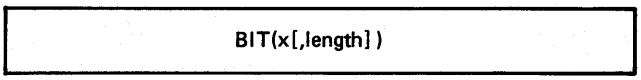


Figure 11-24. BIT Builtin Function Syntax

Arguments x and length can be expressions; the values must be scalar and computational. Argument x specifies the value to be converted. Argument length specifies the length of the result string.

If the length argument is specified, length is converted to a fixed binary integer that must not be negative. The value of x is converted to a bit string of the specified length.

If the length argument is not specified, the expression x is evaluated and converted to a bit string with a length determined according to standard conversion rules.

BIT builtin function examples are shown in figure 11-25.

DATATYPE OF X	X	LENGTH	BIT(X,LENGTH)
CHAR(4)	↑1101↑	6	↑110100↑B
CHAR(4)	↑1101↑	2	↑11↑B
FIXED BIN(4,0)	1101B	6	↑110100↑B
FIXED BIN(10,3)	-0001101.011B	6	↑000110↑B
FIXED DEC(4,3)	5.998	---	↑0101↑B
FIXED DEC(4,3)	-5.998	---	↑0101↑B
FIXED DEC(4,3)	5.998	9	↑010100000↑B
CHAR(6)	↑ 5.998↑	---	RAISES CONV

Figure 11-25. BIT Builtin Function Examples

BOOL BUILTIN FUNCTION

The BOOL builtin function performs a specified boolean operation on two bit strings and returns a bit string value. BOOL syntax is shown in figure 11-26.

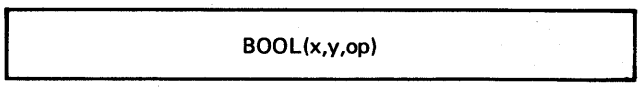


Figure 11-26. BOOL Builtin Function Syntax

Arguments x, y, and op can be expressions; the values must be scalar and computational. Arguments x and y specify the strings. Argument op specifies the boolean operation to be performed. All arguments are converted to bit strings before the boolean operation is performed.

If strings x and y are different in length, the shorter one is extended on the right with zero bits until it is the same length as the longer one. A boolean operation is performed on each pair of corresponding bits in strings x and y to create the result bit string.

Argument op is converted to a bit string of length four. Each bit specifies a boolean operation as shown in table 11-3; bits are referenced from left to right as n1, n2, n3, and n4. For example, if both x and y are '0'B and n1 is '1'B, the result bit is '1'B; if x is '0'B and y is '1'B and n2 is '1'B, the result bit is '1'B.

TABLE 11-3. BOOL OPERATIONS

Value of Bit x_i	Value of Bit y_i	Result Bit Determined By
0	0	n1
0	1	n2
1	0	n3
1	1	n4

The boolean functions corresponding to the possible values of op are listed in table 11-4. BOOL builtin function examples are shown in figure 11-27.

TABLE 11-4. BOOLEAN FUNCTION/OP VALUE CORRESPONDENCE

op	Meaning	PL/I Equivalent
'0000'B	false	x&¬x
'0001'B	x AND y	x&y
'0010'B	¬(x implies y)	x&¬y
'0011'B	x	x
'0100'B	¬(y implies x)	y&¬x
'0101'B	y	y
'0110'B	exclusive or (x,y)	(x|y)&¬(x&y)
'0111'B	x OR y	x|y
'1000'B	x NOR y	¬(x|y)
'1001'B	x ≡ y	(x&y) | (¬x&¬y)
'1010'B	¬y	¬y
'1011'B	y implies x	x|¬y
'1100'B	¬x	¬x
'1101'B	x implies y	y|¬x
'1110'B	x NAND y	¬(x&y)
'1111'B	true	x|¬x

Note: The PL/I equivalent column represents expressions that can be substituted for BOOL (x,y,op) only if strings x and y have the same length.

```
X          Y          OP         BOOL(X,Y,OP)
↑11101↑B   ↑00111↑B   ↑0110↑B    ↑11010↑B
↑11100↑B   ↑0↑B       ↑1101↑B    ↑00011↑B
↑11100↑B   ↑1↑B       ↑1101↑B    ↑10011↑B
```

Figure 11-27. BOOL Builtin Function Examples

CEIL BUILTIN FUNCTION

The CEIL builtin function returns the smallest integer that is greater than or equal to the value of an argument. CEIL syntax is shown in figure 11-28.

```
CEIL(x)
```

Figure 11-28. CEIL Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to an arithmetic value. The result of the function is a real value with the base and scale of the converted argument. If the result is a floating point value, the result precision is that of the argument. If the result is a fixed point value, the result precision is

(min(N,max(p-q+1,1)), 0)

where N is the maximum precision (p) for the given base (14 for decimal, 48 for binary), and (p,q) is the precision of the converted argument.

CEIL builtin function examples are shown in figure 11-29.

```
X                  CEIL(X)
  3.2                 4
 -3.8                -3
 -3.875              -3
-2.50E+000          -2.00E+000
-1.26583E+002       -1.26000E+002
```

Figure 11-29. CEIL Builtin Function Examples

CHARACTER BUILTIN FUNCTION

The CHARACTER builtin function converts an argument to a character string of a specified length and returns the character string value. CHARACTER syntax is shown in figure 11-30.

```
{CHARACTER}
{CHAR     } (x[,length])
```

Figure 11-30. CHARACTER Builtin Function Syntax

Arguments x and length can be expressions; the values must be scalar and computational. Argument x specifies the value to be converted. Argument length specifies the length of the result string.

If the length argument is specified, length is converted to a fixed binary integer that must not be negative. The value of x is converted to a character string of the specified length.

If the length argument is not specified, the expression x is evaluated and converted to a character string with a length determined according to standard conversion rules.

CHARACTER builtin function examples are shown in figure 11-31.

```
DATATYPE OF X        X              LENGTH   CHAR(X,LENGTH)
BIT(5)               ↑01011↑B         10     ↑01011     ↑
FIXED BIN(4,0)       11011B           ---    ↑    13↑
FIXED BIN(10,4)      -001101.1010B    ---    ↑ -13.62↑
FIXED DEC(4,3)         5.998          ---    ↑  5.998↑
FIXED DEC(4,3)        -5.998          ---    ↑ -5.998↑
PIC↑Z9V.99↑          ↑ 5.98↑          ---    ↑ 5.98↑
```

Figure 11-31. CHARACTER Builtin Function Examples

COLLATE BUILTIN FUNCTION

The COLLATE builtin function returns a character string containing, in order, all characters in the collating sequence. COLLATE syntax is shown in figure 11-32.

```
COLLATE[( )]
```

Figure 11-32. COLLATE Builtin Function Syntax

The collating sequence is determined by the character set in use. Collating sequences are listed in appendix A.

The collating sequence is used for the following purposes:

- Comparing character string values in an expression
- Determining the sequential order of records in a KEYED SEQUENTIAL file
- Translating a character string when the TRANSLATE builtin function is referenced with only two arguments

COPY BUILTIN FUNCTION

The COPY builtin function concatenates a given string a specified number of times and returns the new string. COPY syntax is shown in figure 11-33.

```
COPY(string,count)
```

Figure 11-33. COPY Builtin Function Syntax

Arguments string and count can be expressions; the values must be scalar and computational. Argument string specifies the source string; argument count specifies the number of copies of that string. If argument string is a bit string, the result of the function is a bit string; otherwise, the result is a character string.

Argument string is first converted to a string of the result type. Argument count is converted to an integer that must not be negative.

The result is calculated as follows:

count = 0	Result is a null string.
count = 1	Result is the converted argument.
count = n	Result is a string formed by concatenating n copies of the converted argument.

COPY builtin function examples are shown in figure 11-34.

```
STRING      COUNT     COPY(STRING,COUNT)

↑ABC↑         0       ↑↑
↑ABC↑         1       ↑ABC↑
↑ABC↑         3       ↑ABCABCABC↑
↑101↑B        3       ↑101101101↑B
  27          3       ↑   27   27   27↑
```

Figure 11-34. COPY Builtin Function Examples

COS BUILTIN FUNCTION

The COS builtin function returns the trigonometric cosine of a value; the argument is assumed to be expressed in radians. COS syntax is shown in figure 11-35.

```
COS(x)
```

Figure 11-35. COS Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

COS builtin function examples are shown in figure 11-36.

```
          X                    COS(X)

-3.14159E+000          -1.00000E+000
-1.57080E+000           5.90290E-012
 0.00000E+000           1.00000E+000
 1.57080E+000           5.90290E-012
 3.14159E+000          -1.00000E+000
 6.28319E+000           1.00000E+000
```

Figure 11-36. COS Builtin Function Examples

COSD BUILTIN FUNCTION

The COSD builtin function returns the trigonometric cosine of a value; the argument is assumed to be expressed in degrees. COSD syntax is shown in figure 11-37.

```
COSD(x)
```

Figure 11-37. COSD Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

COSD builtin function examples are shown in figure 11-38.

```
          X                    COSD(X)

-1.80000E+002          -1.00000E+000
-9.00000E+001           0.00000E+000
 0.00000E+000           1.00000E+000
 9.00000E+001           0.00000E+000
 1.80000E+002          -1.00000E+000
 3.60000E+002           1.00000E+000
```

Figure 11-38. COSD Builtin Function Examples

COSH BUILTIN FUNCTION

The COSH builtin function returns the hyperbolic cosine of a value. COSH syntax is shown in figure 11-39.

```
COSH(x)
```

Figure 11-39. COSH Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

COSH builtin function examples are shown in figure 11-40.

```
          X                    COSH(X)

 -5.00000E+000          7.42099E+001
 -2.00000E+000          3.76220E+000
 -1.00000E+000          1.54308E+000
  0.00000E+000          1.00000E+000
  1.00000E+000          1.54308E+000
  2.00000E+000          3.76220E+000
  1.00000E+002          1.34406E+043
```

Figure 11-40. COSH Builtin Function Examples

DATE BUILTIN FUNCTION

The DATE builtin function returns a string of six characters representing the current date. DATE syntax is shown in figure 11-41.

```
DATE[( )]
```

Figure 11-41. DATE Builtin Function Syntax

The returned value is in the form 'yymmdd' where yy is the year, mm is the month, and dd is the day. For example:

```
DCL TODAY PIC '99/99/99' INIT(DATE( ));
```

DATE builtin function examples are shown in figure 11-42.

```
POSSIBLE VALUES
OF DATE() INCLUDE:

    ↑771225↑
    ↑780315↑
    ↑840704↑
```

Figure 11-42. DATE Builtin Function Examples

DECAT BUILTIN FUNCTION

The DECAT builtin function creates a string that is formed of portions of a given string and returns the new string. The function has the combined capabilities of the BEFORE and AFTER builtin functions. DECAT syntax is shown in figure 11-43.

```
DECAT(string1,string2,op)
```

Figure 11-43. DECAT Builtin function Syntax

Arguments string1, string2, and op can be expressions; the values must be scalar and computational. Argument string1 specifies the string from which the new string is to be formed. If string2 occurs as a substring of string1, the portions of string1 used to form the result are determined by the position of string2 in string1. Argument op specifies the type of operation to be performed.

If string1 and string2 are bit strings, the result of the function is a bit string; any other combination results in a character string. If string1 is a null string, the result is a null string.

Arguments string1 and string2 are first converted to the result type. Argument op is converted to a bit string of length three. The argument can specify a number of operations as listed in table 11-5.

TABLE 11-5. DECAT OPERATIONS

op	Result		
	s2 Is a Substring of s1	s2 Is Null	s2 Is Not Substring of s1
'000'B	null	null	null
'001'B	AFTER(s1,s2)	s1	null
'010'B	s2	null	null
'011'B	s2||AFTER(s1,s2)	s1	null
'100'B	BEFORE(s1,s2)	null	s1
'101'B	BEFORE(s1,s2) ||AFTER(s1,s2)	s1	s1
'110'B	BEFORE(s1,s2)||s2	null	s1
'111'B	s1	s1	s1

DECAT builtin function examples are shown in figure 11-44.

```
STRING1          STRING2     OP          DECAT(STRING1,
                                         STRING2,OP)
↑TENDENCY↑       ↑DEN↑       ↑101↑B      ↑TENCY↑
↑TENDENCY↑       ↑EN↑        ↑101↑B      ↑TDENCY↑
↑TENDENCY↑       ↑TEN↑       ↑100↑B      ↑↑
↑TENDENCY↑       ↑TEN↑       ↑110↑B      ↑TEN↑
↑TENDENCY↑       ↑DEN↑       ↑010↑B      ↑DEN↑
↑TENDENCY↑       ↑CEN↑       ↑011↑B      ↑↑
↑TENDENCY↑       ↑↑          ↑110↑B      ↑↑
```

Figure 11-44. DECAT Builtin Function Examples

DECIMAL BUILTIN FUNCTION

The DECIMAL builtin function converts a value to decimal base with a specified precision and returns the decimal value. DECIMAL syntax is shown in figure 11-45.

```
{DECIMAL} (x[,p[,q]])
{DEC    }
```

Figure 11-45. DECIMAL Builtin Function Syntax

Argument x specifies the value to be converted to decimal; the argument can be an expression and the value must be scalar and computational. Arguments p and q specify the precision of the result. Argument p must be an unsigned decimal integer within the range 1<=p and p<=14. Argument q must be a decimal integer within the range -255<=q and q<=255. If the converted argument is floating point, q cannot be specified.

The result of the function is the decimal value of the converted argument. The result precision is (p) or (p,q) if p and q are specified. If neither p nor q is specified, precision is determined by the standard conversion rules.

DECIMAL builtin function examples are shown in figure 11-46.

```
        X          P     Q      DEC(X,P,Q)

    .00018         5     3         0.062
    .00018         5     4        0.0625
    .0001E08       5     ---   6.2500E-002
```

Figure 11-46. DECIMAL Builtin Function Examples

DIMENSION BUILTIN FUNCTION

The DIMENSION builtin function returns the span of the specified dimension of an array. DIMENSION syntax is shown in figure 11-47.

```
{DIMENSION}
{DIM      } (x,n)
```

Figure 11-47. DIMENSION Builtin Function Syntax

Argument x specifies the array, and n specifies the dimension to be examined. Argument n can be an expression; the value must be scalar and computational. If n is not an integer, it is converted to a fixed binary integer before the operation is performed.

The result of the function is a fixed binary integer that gives the span of the specified dimension of the array; that is, DIM(x,n)= HBOUND(x,n)-LBOUND(x,n)+1.

DIMENSION builtin function examples are shown in figure 11-48.

```
DECLARATIONS                    X          N    DIMENSION(X,N)

DCL A(5) CHAR(8);               A          1          5

DCL A(2:8) CHAR;                A          1          7

DCL A(-5:5) CHAR;               A          1         11

DCL A(3,4,0:5);                 A          3          6

DCL A(8,3:12);
DCL B(3:12) DEF(B(3,*));        B          1         10

DCL A(3,4,5);                   A(2,*,*)   2          5

      DCL B(5,3:8)
      CALL P(B);
P:    PROC(X);
        DCL X(*,*);             X          2          6
```

Figure 11-48. DIMENSION Builtin Function Examples

DIVIDE BUILTIN FUNCTION

The DIVIDE builtin function divides one argument by another and returns the quotient with a specified precision. DIVIDE syntax is shown in figure 11-49.

```
DIVIDE(x,y,p[,q])
```

Figure 11-49. DIVIDE Builtin Function Syntax

Argument x specifies the dividend, and y specifies the divisor; each can be an expression and must be scalar and computational. Arguments p and q specify the precision of the result; they must be decimal integers. Argument q can be signed.

Arguments x and y are evaluated and converted to arithmetic values of the common base and scale. If the common scale is fixed, q can be specified and must be within the range -255<=q and q<=255; if q is not specified, it is assumed to be zero. If the common scale is floating point, q cannot be specified. In either case, p must be greater than zero and must not exceed maximum precision (14 for decimal, 48 for binary).

The result of the function is the quotient of the converted arguments x and y, which represent the dividend and divisor, respectively. The base and scale of the result are those of the converted arguments. The precision of the result is (p) or (p,q). If y is zero, the ZERODIVIDE condition is raised.

DIVIDE builtin function examples are shown in figure 11-50.

```
        X              Y          P     Q     DIVIDE(X,Y,P,Q)

        3              4          5     3          0.750
    13.45          2E+000         7     ---   6.725000E+000
 0.142857          17.356         8     3          0.008
 0.142857          17.356         6     ---            0
1.42857E-001       17.356         6     ---    8.23099E-003
.1018              1.18           12    10   00.01101010108
```

Figure 11-50. DIVIDE Builtin Function Examples

EMPTY BUILTIN FUNCTION

The EMPTY builtin function returns an empty area value. If this value is assigned to an area variable, all based generations contained in the area are freed. EMPTY syntax is shown in figure 11-51. EMPTY builtin function examples are shown in figure 11-52.

```
EMPTY[( )]
```

Figure 11-51. EMPTY Builtin Function Syntax

```
      DECLARE C AREA(400) CONTROLLED;
      DECLARE A AREA(200);
      ALLOCATE C;
      .
      .
      .

      A=C=EMPTY();
/* FREES ALL BASED GENERATIONS ALLOCATED */
/* IN THE MOST RECENTLY ALLOCATED        */
/* GENERATIONS OF AREA VARIABLES A AND C */
```

Figure 11-52. EMPTY Builtin Function Examples

ERF BUILTIN FUNCTION

The ERF builtin function returns the error function of a value. ERF syntax is shown in figure 11-53.

```
ERF(x)
```

Figure 11-53. ERF Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

The error function is defined by

$$ERF(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

ERF builtin function examples are shown in figure 11-54.

X	ERF(X)
-1.00000E+320	-1.00000E+000
-2.00000E+000	-9.95322E-001
-1.00000E+000	-8.42701E-001
0.00000E+000	0.00000E+000
1.00000E+000	8.42701E-001
2.00000E+000	9.95322E-001
1.00000E+300	1.00000E+000

Figure 11-54. ERF Builtin Function Examples

ERFC BUILTIN FUNCTION

The ERFC builtin function returns the complementary error function of a value. ERFC syntax is shown in figure 11-55.

```
ERFC(x)
```

Figure 11-55. ERFC Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

The complementary error function is defined by

$$ERFC(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$
$$= 1 - ERF(x)$$

ERFC builtin function examples are shown in figure 11-56.

X	ERFC(X)
-1.00000E+320	2.00000E+000
-2.00000E+000	1.99532E+000
-1.00000E+000	1.84270E+000
0.00000E+000	1.00000E+000
1.00000E+000	1.57299E-001
2.00000E+000	4.67773E-003
2.50000E+001	8.30017E-274

Figure 11-56. ERFC Builtin Function Examples

EXP BUILTIN FUNCTION

The EXP builtin function returns the value of e raised to a given power, where e is the base of the natural logarithms. EXP syntax is shown in figure 11-57.

```
EXP(x)
```

Figure 11-57. EXP Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

EXP builtin function examples are shown in figure 11-58.

X	EXP(X)
-6.00000E+002	2.65040E-261
-1.00000E+001	4.53999E-005
-2.00000E+000	1.35335E-001
-1.00000E+000	3.67879E-001
0.00000E+000	1.00000E+000
1.00000E+000	2.71828E+000
2.00000E+000	7.38906E+000
7.50000E+001	3.73324E+032

Figure 11-58. EXP Builtin Function Examples

FIXED BUILTIN FUNCTION

The FIXED builtin function converts a value to fixed point scale with a specified precision and returns the converted value. FIXED syntax is shown in figure 11-59.

```
FIXED(x,p[,q])
```

Figure 11-59. FIXED Builtin Function Syntax

Argument x specifies the value to be converted to fixed-point scale; the argument can be an expression and the value must be scalar and computational. Arguments p and q specify the precision of the result. Argument p must be an unsigned decimal integer greater than zero and must not exceed maximum precision (14 for decimal, 48 for binary). Argument q must be a decimal integer within the range -255<=q and q<=255.

The result of the function is the fixed point value of the converted argument. The result precision is (p) or (p,q). If q is not specified, it is assumed to be zero.

FIXED builtin function examples are shown in figure 11-60.

```
        X              P      Q      FIXED(X,P,Q)

      2.34             5      3          2.340
      2.34             5     ---             2
-1.234E-003            7      5       -0.00123
-1.234E-003            5     ---             -0
 6.3E-002              5     ---              0
↑1101↑B                5      2          13.0
```

Figure 11-60. FIXED Builtin Function Examples

FLOAT BUILTIN FUNCTION

The FLOAT builtin function converts a value to floating point scale with a specified precision and returns the converted value. FLOAT syntax is shown in figure 11-61.

```
FLOAT(x,p)
```

Figure 11-61. FLOAT Builtin Function Syntax

Argument x specifies the value to be converted to floating point scale; the argument can be an expression and the value must be scalar and computational. Argument p specifies the precision of the result. Argument p must be an unsigned decimal integer greater than zero and must not exceed maximum precision (14 for decimal, 48 for binary).

The result of the function is the floating point value of the converted argument.

FLOAT builtin function examples are shown in figure 11-62.

```
        X              P      FLOAT(X,P)

      2.34             5      2.3400E+000
     12345.6           5      1.2346E+004
-1.234E-003            7     -1.234000E-003
-1.23456E+005          3     -1.23E+005
 6.3E-002              5      6.3E-002
↑1101↑B                5      1.3E+001
```

Figure 11-62. FLOAT Builtin Function Examples

FLOOR BUILTIN FUNCTION

The FLOOR builtin function returns the largest integer that does not exceed the value of the argument. FLOOR syntax is shown in figure 11-63.

```
FLOOR(x)
```

Figure 11-63. FLOOR Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to an arithmetic value. The result of the function is a real value with the base and scale of the converted argument. If the result is a floating point value, the result precision is that of the argument. If the result is a fixed point value, the result precision is

$$(\min(N,\max(p-q+1,1)),\ 0)$$

where N is the maximum precision (p) for the given base (14 for decimal, 48 for binary) and (p,q) is the precision of the converted argument.

FLOOR builtin function examples are shown in figure 11-64.

```
        X              FLOOR(X)

      3.2                  3
     -3.8                 -4
     -3.875               -4
  -2.50E+000           -3.00E+000
-1.26583E+002          -1.27000E+002
```

Figure 11-64. FLOOR Builtin Function Examples

HBOUND BUILTIN FUNCTION

The HBOUND builtin function returns the upper bound of the specified dimension of an array. HBOUND syntax is shown in figure 11-65.

```
HBOUND(x,n)
```

Figure 11-65. HBOUND Builtin Function Syntax

Argument x specifies the array, and n specifies the dimension to be examined. Argument n can be an expression; the value must be scalar and computational. If n is not an integer, it is converted to a fixed binary integer before the operation is performed.

The result of the function is a fixed binary integer that gives the upper bound of the specified dimension of the array.

HBOUND builtin function examples are shown in figure 11-66.

```
DECLARATIONS                    X          N    HBOUND(X,N)

DCL A(5) CHAR(8);               A          1        5

DCL A(2:8) CHAR;                A          1        8

DCL A(-5:5) CHAR;               A          1        5

DCL A(3,4,0:5);                 A          3        5

DCL A(8,3:12);
DCL B(3:12) DEF(B(3,*));        B          1       12

DCL A(3,4,5);                   A(2,*,*)   2        5

      DCL B(5,3:8)
      CALL P(B);
P:    PROC(X);
        DCL X(*,*);             X          2        8
```

Figure 11-66. HBOUND Builtin Function Examples

HIGH BUILTIN FUNCTION

The HIGH builtin function forms a string with a specified length of the highest character in the collating sequence and returns that string. HIGH syntax is shown in figure 11-67.

```
HIGH(length)
```

Figure 11-67. HIGH Builtin Function Syntax

Argument length can be an expression; the value must be scalar and computational. Argument length specifies the length of the string to be formed. The argument is converted to an integer that must not be negative.

If a CDC collating sequence is in use, the result of the function is a string consisting of copies of the character 9. If an ASCII collating sequence is in use, the result is a string consisting of copies of the underline character.

HIGH builtin function examples are shown in figure 11-68.

```
LENGTH                          HIGH(LENGTH)

   0                            ↑↑
   1                            ↑9↑
   4                            ↑9999↑

NOTE:  THIS RESULT DEPENDS UPON
       INSTALLATION PARAMETERS
```

Figure 11-68. HIGH Builtin Function Examples

INDEX BUILTIN FUNCTION

The INDEX builtin function returns a value that indicates the position of a substring within a string. Positions are numbered from the left, starting with 1 for the leftmost character. INDEX syntax is shown in figure 11-69.

```
INDEX(string1,string2)
```

Figure 11-69. INDEX Builtin Function Syntax

Arguments string1 and string2 can be expressions; the values must be scalar and computational. Argument string1 specifies the string; string2 specifies the substring.

The result of the function is a fixed binary integer indicating the position in string1 of the first bit or character of the substring. If the substring occurs more than once, the result indicates the position of the leftmost substring.

The result of the function is zero under any of the following circumstances:

String1 is a null string.

String2 is a null string.

String2 is not a substring of string1.

INDEX builtin function examples are shown in figure 11-70.

```
                                    INDEX
STRING1          STRING2    (STRING1,STRING2)

↑ABCDEFG↑        ↑CD↑              3
↑ABCDEFGCDHIJ↑   ↑CD↑              3
↑ABCDEFG↑        ↑DC↑              0
↑ABCDEFG↑        ↑↑                0
↑↑               ↑CD↑              0
```

Figure 11-70. INDEX Builtin Function Examples

LBOUND BUILTIN FUNCTION

The LBOUND builtin function returns the lower bound of the specified dimension of an array. LBOUND syntax is shown in figure 11-71.

```
LBOUND(x,n)
```

Figure 11-71. LBOUND Builtin Function Syntax

Argument x specifies the array, and n specifies the dimension to be examined. Argument n can be an expression; the value must be scalar and computational. If n is not an integer, it is converted to a fixed binary integer before the operation is performed.

The result of the function is a fixed binary integer that gives the lower bound of the specified dimension of the array.

LBOUND builtin function examples are shown in figure 11-72.

```
DECLARATIONS                    X          N    LBOUND(X,N)

DCL A(5) CHAR(8);               A          1        1

DCL A(2:8) CHAR;                A          1        2

DCL A(-5:5) CHAR;               A          1       -5

DCL A(3,4,0:5);                 A          3        0

DCL A(8,3:12);
DCL B(3:12) DEF(B(3,*));        B          1        3

DCL A(3,4,2:5);                 A(2,*,*)   2        2

      DCL B(5,3:8)
      CALL P(B);
P:    PROC(X);
        DCL X(*,*);             X          2        3
```

Figure 11-72. LBOUND Builtin Function Examples

LENGTH BUILTIN FUNCTION

The LENGTH builtin function determines the number of characters or bits in an argument and returns the count. LENGTH syntax is shown in figure 11-73.

```
LENGTH(string)
```

Figure 11-73. LENGTH Builtin Function Syntax

Argument string can be an expression; the value must be scalar and computational. If the argument is not a string, it is converted to a character string. The result of the function is a fixed binary integer that gives the current length of the character or bit string.

LENGTH builtin function examples are shown in figure 11-74.

```
DATATYPE OF X        VALUE OF X    LENGTH(X)

CHAR(5)              ↑ABCDE↑           5
CHAR(7)              ↑ABCDE  ↑         7
CHAR(7) VARYING      ↑ABCDE  ↑         7
CHAR(7) VARYING      ↑ABC↑             3
BIT(7)               ↑1011011↑B        7
BIT(7) VARYING       ↑1011011↑         7
BIT(7) VARYING       ↑101↑             3
FIXED DEC(5,2)         1.25            6
FIXED BIN(5,2)       1.01B             1
```

Figure 11-74. LENGTH Builtin Function Examples

LINENO BUILTIN FUNCTION

The LINENO builtin function returns the current line number of the specified stream print file. LINENO syntax is shown in figure 11-75.

```
LINENO(file-reference)
```

Figure 11-75. LINENO Builtin Function Syntax

Argument file-reference specifies the stream print file; the argument must be a scalar file value that identifies an open file with the attributes STREAM OUTPUT PRINT.

The result of the function is a fixed binary integer that gives the current line number for the file. The first line on a page of a print file is line number 1.

LINENO builtin function examples are shown in figure 11-76.

```
                                          VALUE
      PL/I STATEMENTS                     PRINTED

DO;
   PUT PAGE LIST(LINENO(SYSPRINT));         1
   PUT SKIP(5) LIST(LINENO(SYSPRINT));      6
   PUT EDIT(LINENO(SYSPRINT))(LINE,A);      1
END;
```

Figure 11-76. LINENO Builtin Function Examples

LOG BUILTIN FUNCTION

The LOG builtin function returns the natural logarithm (base e) of a value. LOG syntax is shown in figure 11-77.

```
LOG(x)
```

Figure 11-77. LOG Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x must specify a value greater than zero. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

LOG builtin function examples are shown in figure 11-78.

```
          X                    LOG(X)

1.00000E-290          -6.67750E+002
1.00000E-001          -2.30259E+000
1.00000E+000           0.00000E+000
2.71828E+000           1.00000E+000
1.00000E+001           2.30259E+000
2.718E+000             1.000E+000
```

Figure 11-78. LOG Builtin Function Examples

LOG10 BUILTIN FUNCTION

The LOG10 builtin function returns the common logarithm (base 10) of a value. LOG10 syntax is shown in figure 11-79.

```
LOG10(x)
```

Figure 11-79. LOG10 Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x must specify a value greater than zero. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

LOG10 builtin function examples are shown in figure 11-80.

```
          X                   LOG10(X)

1.00000E-290          -2.90000E+002
1.00000E-001          -1.00000E+000
5.00000E-001          -3.01030E-001
1.00000E+000           0.00000E+000
2.00000E+000           3.01030E-001
2.71828E+000           4.34294E-001
1.00000E+001           1.00000E+000
1.04858E+006           6.02060E+000
 10.0                  1.00E+000
```

Figure 11-80. LOG10 Builtin Function Examples

LOG2 BUILTIN FUNCTION

The LOG2 builtin function returns the base 2 logarithm of a value. LOG2 syntax is shown in figure 11-81.

```
LOG2(x)
```

Figure 11-81. LOG2 Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x must specify a value greater than zero. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

LOG2 builtin function examples are shown in figure 11-82.

```
          X                    LOG2(X)

  1.00000E-290          -9.63359E+002
  1.00000E-001          -3.32193E+000
  5.00000E-001          -1.00000E+000
  1.00000E+000           0.00000E+000
  2.00000E+000           1.00000E+000
  2.71828E+000           1.44270E+000
  1.00000E+001           3.32193E+000
  1.04858E+006           2.00000E+001
```

Figure 11-82. LOG2 Builtin Function Examples

LOW BUILTIN FUNCTION

The LOW builtin function forms a string with a specified length of the lowest character in the collating sequence and returns that string. LOW syntax is shown in figure 11-83.

```
LOW(length)
```

Figure 11-83. LOW Builtin Function Syntax

Argument length can be an expression; the value must be scalar and computational. Argument length specifies the length of the string to be formed. The argument is converted to an integer that must not be negative.

The result of the function is a string consisting of the number of blank characters specified by the length argument. The blank is the lowest character in all collating sequences.

LOW builtin function examples are shown in figure 11-84.

```
LENGTH                LOW(LENGTH)

  0                   ↑↑
  1                   ↑ ↑
  4                   ↑   ↑
```

Figure 11-84. LOW Builtin Function Examples

MAX BUILTIN FUNCTION

The MAX builtin function determines which argument has the largest value, and returns that value. MAX syntax is shown in figure 11-85.

```
MAX(xi , , , )
```

Figure 11-85. MAX Builtin Function Syntax

Arguments can be expressions; the values must be scalar and computational. Arguments are evaluated and converted to arithmetic values of the common base and scale. The result of the function is the largest value among the arguments. If the common scale is floating point, the result precision is

$(\max(p_i))$

If the common scale is fixed point, the precision is

$(\min(N,\max(p_i-q_i)+\max(q_i)),\ \max(q_i))$

where N is the maximum possible precision (p) for the base of the result (14 for decimal, 48 for binary), and (p_i,q_i) are the precisions of the converted arguments.

MAX builtin function examples are shown in figure 11-86.

```
                          VALUE              DECIMAL
   REFERENCE            RETURNED           EQUIVALENT

MAX(1, 2, -3)              2
MAX(1.0000, 2, -3E0)  2.0000E+000
MAX(-50, 11B)         00000011B                  3
MAX(16, 35, 28)            35
MAX(-16, -35, -28)        -16
```

Figure 11-86. MAX Builtin Function Examples

MIN BUILTIN FUNCTION

The MIN builtin function determines which argument has the smallest value, and returns that value. MIN syntax is shown in figure 11-87.

```
MIN(xi , , , )
```

Figure 11-87. MIN Builtin Function Syntax

Arguments can be expressions; the values must be scalar and computational. Arguments are evaluated and converted to arithmetic values of the common base and scale. The result of the function is the smallest value among the arguments. If the common scale is floating point, the result precision is

$(\max(p_i))$

If the common scale is fixed point, the result is

$(\min(N,\max(p_i-q_i)+\max(q_i)),\ \max(q_i))$

where N is the maximum possible precision (p) for the base of the result (14 for decimal, 48 for binary), and (p_i,q_i) are the precisions of the converted arguments.

MIN builtin function examples are shown in figure 11-88.

```
                          VALUE              DECIMAL
   REFERENCE            RETURNED           EQUIVALENT

MIN(1, 2, -3)              -3
MIN(1.0000, 2, -3E0) -3.0000E+000
MIN(-50, 11B)         00000011B                -50
MIN(16, 35, 28)            16
MIN(-16, -35, -28)        -35
```

Figure 11-88. MIN Builtin Function Examples

MOD BUILTIN FUNCTION

The MOD builtin function returns the value of an argument x (modulo y). MOD syntax is shown in figure 11-89.

```
MOD(x,y)
```

Figure 11-89. MOD Builtin Function Syntax

Arguments x and y can be expressions; the values must be scalar and computational. Arguments x and y are evaluated and converted to arithmetic values of the common base and scale. If the common scale is floating point, the result precision is

$(\max(p_1,p_2))$

If the common scale is fixed point, the result is

(min(N,p2-q2+max(q1,q2)), max(q1,q2))

where N is the maximum possible precision (p) for the base of the result (14 for decimal, 48 for binary), and (p1,q1) and (p2,q2) are precisions of x and y, respectively.

If y is equal to 0, the result is the value of x. If y is not 0, the result is

x-(y*FLOOR(x/y))

The value of this expression always has the same sign as y.

MOD builtin function examples are shown in figure 11-90.

X	Y	MOD(X,Y)
12	5	2
-12	5	3
12	-5	-3
-12	-5	-2
5	0	5
-5	0	-5
12	3.5	1.5
12.5	3	0.5

Figure 11-90. MOD Builtin Function Examples

MULTIPLY BUILTIN FUNCTION

The MULTIPLY builtin function returns the product of two values with a specified precision. MULTIPLY syntax is shown in figure 11-91.

```
MULTIPLY(x,y,p[,q])
```

Figure 11-91. MULTIPLY Builtin Function Syntax

Arguments x and y specify the two values to be multiplied; each can be an expression and must be scalar and computational. Arguments p and q specify the precision of the result; they must be decimal integers. Argument q can be signed.

Arguments x and y are evaluated and converted to arithmetic values of the common base and scale. If the common scale is fixed point, q can be specified and must be within the range -255<=q and q<=255; if q is not specified, it is assumed to be zero. If the common scale is floating point, q cannot be specified. In either case, p must be greater than zero and must not exceed maximum precision (14 for decimal, 48 for binary).

The result of the function is the product of the converted arguments x and y. The base and scale of the result are those of the converted arguments. The precision of the result is (p) or (p,q).

MULTIPLY builtin function examples are shown in figure 11-92.

X	Y	P	Q	MULTIPLY(X,Y,P,Q)
3	4	5	3	12.000
13.45	2E+000	7	---	2.690000E+001
0.142857	17.356	8	3	2.479
0.142857	17.356	6	---	2
1.42857E-001	17.356	6	---	2.47943E+000
0.6	1.5	12	10	0.3125

Figure 11-92. MULTIPLY Builtin Function Examples

NULL BUILTIN FUNCTION

The NULL builtin function returns a null pointer value. A null pointer value does not identify any generation. NULL syntax is shown in figure 11-93.

```
NULL[( )]
```

Figure 11-93. NULL Builtin Function Syntax

A null pointer value can be used to designate the end of a list during list processing operations as described under Based Storage in section 3, Data Elements.

NULL builtin function examples are shown in figure 11-94.

```
DCL P POINTER INITIAL(NULL());
DCL Q POINTER, R OFFSET(MYAREA);
Q,R=NULL();
```

Figure 11-94. NULL Builtin Function Examples

OFFSET BUILTIN FUNCTION

The OFFSET builtin function converts a pointer value to an offset value and returns the offset value. OFFSET syntax is shown in figure 11-95.

```
OFFSET(pointer,area)
```

Figure 11-95. OFFSET Builtin Function Syntax

Argument pointer specifies a scalar pointer value; argument area specifies a scalar area value. It is an error if pointer does not identify a generation in area.

The result of the function is an offset value that identifies the generation denoted by argument pointer.

ONCHAR BUILTIN FUNCTION

An ONCHAR value is established when the CONVERSION condition is raised; the value is the single character that caused the condition to be raised. This character is contained in the value of the ONSOURCE builtin function. The ONCHAR builtin function returns that character. ONCHAR syntax is shown in figure 11-96.

```
ONCHAR[( )]
```

Figure 11-96. ONCHAR Builtin Function Syntax

If the function is invoked outside a CONVERSION on-unit or a dynamic successor of a CONVERSION on-unit, a blank character is returned.

ONCHAR PSEUDOVARIABLE

ONCHAR operates as a pseudovariable when it is used as the target of an assignment operation. The ONCHAR pseudovariable replaces the character in ONSOURCE that caused the CONVERSION condition to be raised.

The syntax is the same as shown for the ONCHAR builtin function. The expression to be assigned is evaluated and converted to a character string of length 1. That character is then assigned to ONSOURCE, replacing the single character that caused the CONVERSION condition to be raised. ONCHAR modifies the original string that was being converted when CONVERSION was raised.

The ONCHAR pseudovariable can only be used in a CONVERSION on-unit or a dynamic successor of a CONVERSION on-unit. It is an error if it is used elsewhere.

ONCODE BUILTIN FUNCTION

An ONCODE value is established when any condition is raised; the value is a fixed binary integer that identifies the condition and the circumstances under which it was raised. The ONCODE builtin function returns that binary integer. ONCODE syntax is shown in figure 11-97.

```
ONCODE[( )]
```

Figure 11-97. ONCODE Builtin Function Syntax

If the function is invoked outside an on-unit or the dynamic successor of an on-unit, a zero is returned.

ONCODE values are listed under run-time diagnostics in appendix B.

ONFILE BUILTIN FUNCTION

An ONFILE value is established when any I/O condition is raised; an ONFILE value is also established when the CONVERSION condition is raised during an I/O operation. The value is a character string that contains the name of the file constant for which the condition was raised. The ONFILE builtin function returns that character string. ONFILE syntax is shown in figure 11-98.

```
ONFILE[( )]
```

Figure 11-98. ONFILE Builtin Function Syntax

If the function is invoked outside an applicable on-unit or a dynamic successor of an applicable on-unit, a null string is returned.

ONKEY BUILTIN FUNCTION

An ONKEY value is established when any I/O condition is raised during a keyed I/O operation; the value is a character string containing the key for the record that caused the condition to be raised. The ONKEY builtin function returns that character string. ONKEY syntax is shown in figure 11-99.

```
ONKEY[( )]
```

Figure 11-99. ONKEY Builtin Function Syntax

ONKEY is established for I/O conditions KEY, RECORD, and TRANSMIT.

If the function is invoked outside an applicable on-unit or a dynamic successor of an applicable on-unit, a null string is returned.

ONLOC BUILTIN FUNCTION

An ONLOC value is established when any condition is raised; the value is a character string that contains the name of the entry point used to enter the procedure in which the condition was raised. The ONLOC builtin function returns that character string. ONLOC syntax is shown in figure 11-100.

```
ONLOC[( )]
```

Figure 11-100. ONLOC Builtin Function Syntax

If the function is invoked outside an on-unit or the dynamic successor of an on-unit, a null string is returned.

ONSOURCE BUILTIN FUNCTION

An ONSOURCE value is established when the CONVERSION condition is raised; the value is a character string containing the source string that caused the condition to be raised. The

ONSOURCE builtin function returns that character string. ONSOURCE syntax is shown in figure 11-101.

```
ONSOURCE[( )]
```

Figure 11-101. ONSOURCE Builtin Function Syntax

If the function is invoked outside a CONVERSION on-unit or a dynamic successor of a CONVERSION on-unit, a null string is returned.

ONSOURCE PSEUDOVARIABLE

ONSOURCE operates as a pseudovariable when it is used as the target of an assignment operation. The ONSOURCE pseudovariable replaces the value of the current generation of ONSOURCE.

The syntax is the same as shown for the ONSOURCE builtin function. The expression to be assigned is evaluated and converted to a character string of the same length as that of the current generation of ONSOURCE. That string is then assigned to ONSOURCE, replacing the value of the current generation. ONSOURCE modifies the original string that was being converted when CONVERSION was raised. The new ONSOURCE value is used as the source for a subsequent conversion attempt if the CONVERSION on-unit terminates normally.

The ONSOURCE pseudovariable can only be used in a CONVERSION on-unit or a dynamic successor of a CONVERSION on-unit. It is an error if it is used elsewhere.

PAGENO BUILTIN FUNCTION

The PAGENO builtin function returns the current page number of the specified print file. PAGENO syntax is shown in figure 11-102.

```
PAGENO(file-reference)
```

Figure 11-102. PAGENO Builtin Function Syntax

Argument file-reference specifies the stream print file; the argument must be a scalar file value that identifies an open file with attributes STREAM OUTPUT PRINT.

The result of the function is a fixed binary integer that gives the current page number for the file.

PAGENO PSEUDOVARIABLE

PAGENO operates as a pseudovariable when it is used as the target of an assignment operation. The PAGENO pseudovariable replaces the current page number for a print file.

The syntax is the same as shown for the PAGENO builtin function. The expression to be assigned is evaluated and converted to a fixed binary integer, which must be positive. That integer value replaces the current page number for the file. The file identified by the argument must be an open file with attributes STREAM OUTPUT PRINT.

POINTER BUILTIN FUNCTION

The POINTER builtin function converts an offset value to a pointer value and returns the pointer value. POINTER syntax is shown in figure 11-103.

```
{ POINTER } (offset,area)
{ PTR     }
```

Figure 11-103. POINTER Builtin Function Syntax

Argument offset specifies a scalar offset value; argument area specifies an area value. It is an error if offset does not identify a generation in area.

The result of the function is a pointer value that identifies the generation denoted by argument area.

PRECISION BUILTIN FUNCTION

The PRECISION builtin function converts the value of an argument to a given precision and returns that value. PRECISION syntax is shown in figure 11-104.

```
{PRECISION} (x,p[,q])
{PREC     }
```

Figure 11-104. PRECISION Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x specifies the expression to be converted. Arguments p and q specify the precision of the result. Argument p must be an unsigned decimal integer greater than zero and must not exceed maximum precision (14 for decimal, 48 for binary). Argument q must be a decimal integer within the range -255<=q and q<=255. If x is floating point, q cannot be specified.

Argument x is evaluated and converted to an arithmetic value. The result of the function is a real value with the base and scale of the converted argument. The precision of the result is (p) or (p,q).

PRECISION builtin function examples are shown in figure 11-105.

```
X             P    Q     PRECISION(X,P,Q)
12.34         5    3        12.340
12.34         6    2         12.34
12.34         6    1          12.3
12.34         4    ---          12
1.234E+002    3    ---    1.23E+002
```

Figure 11-105. PRECISION Builtin Function Examples

REVERSE BUILTIN FUNCTION

The REVERSE builtin function reverses an argument string such that the first character or bit becomes the last, the second becomes the next to last, and so forth. The function returns that reversed string. REVERSE syntax is shown in figure 11-106.

```
REVERSE(string)
```

Figure 11-106. REVERSE Builtin Function Syntax

Argument string can be an expression; the value must be scalar and computational. If the argument is not a string, it is converted to a character string. The result string is the same type as the converted argument. If the argument string is null, the result is a null string; otherwise, the result is the reverse of the argument string.

REVERSE builtin function examples are shown in figure 11-107.

STRING	REVERSE(STRING)
↑ABC↑	↑CBA↑
↑↑	↑↑
↑1011↑B	↑1101↑B

Figure 11-107. REVERSE Builtin Function Examples

ROUND BUILTIN FUNCTION

The ROUND builtin function rounds the value of an argument at a specified digit and returns the rounded value. ROUND syntax is shown in figure 11-108.

```
ROUND(x,n)
```

Figure 11-108. ROUND Builtin Function Syntax

Argument x specifies the value to be rounded; the argument can be an expression and the value must be scalar and computational. Argument n specifies the digit at which rounding is to take place; the argument must be a decimal integer and can be signed. If x has a floating point value, n must be positive.

Argument x is evaluated and converted to an arithmetic value. The result of the function is a real value with the base and scale of the converted argument.

If the converted value is fixed point of precision (p,q), the result precision is

(max(1,min(p-q+1+n,N)), n)

where N is the maximum precision for the base (14 for decimal, 48 for binary).

If the converted value is floating point, the result precision is

(min(n,N))

If n is positive, rounding occurs at the specified digit to the right of the decimal or binary point. If n is negative, rounding occurs at the specified digit to the left of the decimal or binary point. The result value is

SIGN(x)*FLOOR(ABS(x)*(b**k)+.5)/(b**k)

where k=n if x is a fixed point value, or k=n-e if x is a floating point value with the exponent e; and where b=2 if x is binary, or b=10 if x is decimal.

ROUND builtin function examples are shown in figure 11-109.

X	N	ROUND(X,N)	DECIMAL EQUIVALENT
123.453	2	123.450	
123.456	2	123.460	
123.456	-2	100.000	
-123.456	2	-123.460	
11.1010111B	4	11.1010B	3.69

Figure 11-109. ROUND Builtin Function Examples

SIGN BUILTIN FUNCTION

The SIGN builtin function examines the value of an argument and returns an indication of whether the value is positive, negative, or zero. SIGN syntax is shown in figure 11-110.

```
SIGN(x)
```

Figure 11-110. SIGN Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to an arithmetic value. The result of the function is a fixed binary integer with a value that is determined as follows:

x > 0 result = 1

x = 0 result = 0

x < 0 result = -1

SIGN builtin function examples are shown in figure 11-111.

X	SIGN(X)
1.23	1
-2.79	-1
0.00E+000	0
-9.283E+005	-1
↑ -1.23↑	-1

Figure 11-111. SIGN Builtin Function Examples

SIN BUILTIN FUNCTION

The SIN builtin function returns the trigonometric sine of a value; the argument is assumed to be expressed in radians. SIN syntax is shown in figure 11-112.

```
SIN(x)
```

Figure 11-112. SIN Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

SIN builtin function examples are shown in figure 11-113.

X	SIN(X)
-3.14159E+000	-1.18058E-011
-1.57080E+000	-1.00000E+000
-7.85398E-001	-7.07107E-001
0.00000E+000	0.00000E+000
5.23599E-001	5.00000E-001
1.57080E+000	1.00000E+000
3.14159E+000	1.18058E-011

Figure 11-113. SIN Builtin Function Examples

SIND BUILTIN FUNCTION

The SIND builtin function returns the trigonometric sine of a value; the argument is assumed to be expressed in degrees. SIND syntax is shown in figure 11-114.

```
SIND(x)
```

Figure 11-114. SIND Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

SIND builtin function examples are shown in figure 11-115.

X	SIND(X)
-1.80000E+002	0.00000E+000
-9.00000E+001	-1.00000E+000
-4.50000E+001	-7.07107E-001
0.00000E+000	0.00000E+000
3.00000E+001	5.00000E-001
9.00000E+001	1.00000E+000
1.80000E+002	0.00000E+000

Figure 11-115. SIND Builtin Function Examples

SINH BUILTIN FUNCTION

The SINH builtin function returns the hyperbolic sine of a value. SINH syntax is shown in figure 11-116.

```
SINH(x)
```

Figure 11-116. SINH Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

SINH builtin function examples are shown in figure 11-117.

X	SINH(X)
-5.00000E+000	-7.42032E+001
-2.00000E+000	-3.62686E+000
-1.00000E+000	-1.17520E+000
0.00000E+000	0.00000E+000
5.00000E-001	5.21095E-001
1.00000E+000	1.17520E+000
2.00000E+000	3.62686E+000
5.00000E+000	7.42032E+001

Figure 11-117. SINH Builtin Function Examples

SQRT BUILTIN FUNCTION

The SQRT builtin function returns the square root of the value of an argument. SQRT syntax is shown in figure 11-118.

```
SQRT(x)
```

Figure 11-118. SQRT Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x must not be negative. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

SQRT builtin function examples are shown in figure 11-119.

X	SQRT(X)
0.00000E+000	0.00000E+000
5.00000E-001	7.07107E-001
1.00000E+000	1.00000E+000
2.00000E+000	1.41421E+000
5.00000E+000	2.23607E+000
2.0E+000	1.4E+000

Figure 11-119. SQRT Builtin Function Examples

SUBSTR BUILTIN FUNCTION

The SUBSTR builtin function extracts a substring of a specified string and returns the substring. SUBSTR syntax is shown in figure 11-120.

```
SUBSTR(string,position[,length])
```

Figure 11-120. SUBSTR Builtin Function Syntax

Arguments string, position, and length can be expressions; the values must be scalar and computational. Argument string specifies the source string from which the substring is to be extracted. Argument position specifies the starting position of the substring within the string. Argument length specifies the length of the substring to be extracted.

If argument string is arithmetic, it is converted to a character string. If argument string is pictured numeric, its character string value is used. Arguments position and length are converted to fixed binary integers.

The result of the function is a string of the same type as the converted string argument. If argument position is 1, the substring begins with the first character of the string. If argument length is not specified, the function returns the character or bit designated by argument position and all subsequent characters or bits of the converted string.

The substring specified by arguments position and length must be contained within the original string. Position and length have the following requirements:

1<=position and position<=LENGTH(string)+1

0<=length and length<=LENGTH(string)-position+1

The STRINGRANGE condition is raised if the requirements are not met.

SUBSTR builtin function examples are shown in figure 11-121.

STRING	POSITION	LENGTH	SUBSTR(STRING, POSITION,LENGTH)
'ABCDEFG'	3	1	'C'
'ABCDEFG'	3	---	'CDEFG'
'ABCDEFG'	3.5	2.8	'CD'
'ABCDEFG'	1	0	''
'ABCDEFG'	8	0	''
'101101'B	2	3	'011'B
'101101'B	2	---	'01101'B

Figure 11-121. SUBSTR Builtin Function Examples

SUBSTR PSEUDOVARIABLE

SUBSTR operates as a pseudovariable when it is used as the target of an assignment operation. The SUBSTR pseudovariable replaces a substring of the specified string. SUBSTR pseudovariable syntax is shown in figure 11-122.

```
SUBSTR(string-target,position[,length])
```

Figure 11-122. SUBSTR Pseudovariable Syntax

Argument string-target must be a character or bit variable. Arguments position and length can be expressions; the values must be scalar and computational. Argument position specifies the starting position of the substring within the string. Argument length specifies the length of the substring to be replaced.

Assignment of a source value to the specified substring of the variable leaves all other characters or bits of the string unmodified.

The requirements on arguments position and length are the same as shown for the SUBSTR builtin function. Note that this implies that the SUBSTR pseudovariable cannot be used to change the current length of a varying string.

SUBSTR pseudovariable examples are shown in figure 11-123.

DATATYPE OF STRING	ORIGINAL VALUE OF STRING	POSITION	LENGTH	VALUE ASSIGNED TO SUBSTR (STRING, POSITION, LENGTH)	RESULTING VALUE OF STRING
CHAR(7)	'.......'	3	1	'X'	'..X....'
BIT(7)	'0011001'B	2	3	'100'B	'0100001'B
CHAR(6) VAR	'ABCD'	3	2	'XY'	'ABXY'
CHAR(6) VAR	'ABCD'	5	0	'XY'	'ABCD'

Figure 11-123. SUBSTR Pseudovariable Examples

SUBTRACT BUILTIN FUNCTION

The SUBTRACT builtin function returns the difference of two values with a specified precision. SUBTRACT syntax is shown in figure 11-124.

```
SUBTRACT(x,y,p[,q])
```

Figure 11-124. SUBTRACT Builtin Function Syntax

Arguments x and y can be expressions; the values must be scalar and computational. Argument y specifies the value that is to be subtracted from argument x. Arguments p and q specify the precision of the result; they must be decimal integers. Argument q can be signed.

Arguments x and y are evaluated and converted to arithmetic values of the common base and scale. If the common scale is fixed point, q can be specified and must be within the range -255<=q and q<=255; if q is not specified, it is assumed to be zero. If the common scale is floating point, q cannot be specified. In either case, p must be greater than zero and must not exceed maximum precision (14 for decimal, 48 for binary).

The result of the function is x-y. The base and scale of the result are those of the converted arguments. The precision of the result is (p) or (p,q).

SUBTRACT builtin function examples are shown in figure 11-125.

TAN BUILTIN FUNCTION

The TAN builtin function returns the trigonometric tangent of a value; the argument is assumed to be expressed in radians. TAN syntax is shown in figure 11-126.

X	Y	P	Q	SUBTRACT(X,Y,P,Q)
3	4	5	3	-1.000
13.45	2E+000	7	---	1.145000E+001
0.142857	17.356	8	3	-17.213
0.142857	17.356	6	---	-17
1.42857E-001	17.356	6	---	-1.72131E+001
1.01B	1.10001B	8	3	-0.3

Figure 11-125. SUBTRACT Builtin Function Examples

```
TAN(x)
```

Figure 11-126. TAN Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x must not be an odd multiple of $\pi/2$. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

TAN builtin function examples are shown in figure 11-127.

X	TAN(X)
0.00000E+000	0.00000E+000
7.85398E-001	1.00000E+000
1.57000E+000	1.25577E+003
3.14159E+000	-1.18058E-011

Figure 11-127. TAN Builtin Function Examples

TAND BUILTIN FUNCTION

The TAND builtin function returns the trigonometric tangent of a value; the argument is assumed to be expressed in degrees. TAND syntax is shown in figure 11-128.

```
TAND(x)
```

Figure 11-128. TAND Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x must not be an odd multiple of 90. The argument is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

TAND builtin function examples are shown in figure 11-129.

X	TAND(X)
0.00000E+000	0.00000E+000
4.50000E+001	1.00000E+000
8.99000E+001	5.72957E+002
9.01000E+001	-5.72957E+002
1.80000E+002	0.00000E+000

Figure 11-129. TAND Builtin Function Examples

TANH BUILTIN FUNCTION

The TANH builtin function returns the hyperbolic tangent of a value. TANH syntax is shown in figure 11-130.

```
TANH(x)
```

Figure 11-130. TANH Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to floating point. The mode, base, scale, and precision of the result are those of the converted argument.

TANH builtin function examples are shown in figure 11-131.

X	TANH(X)
-1.00000E+320	-1.00000E+000
-2.00000E+000	-9.64028E-001
-1.00000E+000	-7.61594E-001
0.00000E+000	0.00000E+000
1.00000E+000	7.61594E-001
2.00000E+000	9.64028E-001
1.00000E+001	1.00000E+000

Figure 11-131. TANH Builtin Function Examples

TIME BUILTIN FUNCTION

The TIME builtin function returns a string of six characters representing the current time. TIME syntax is shown in figure 11-132.

```
TIME[( )]
```

Figure 11-132. TIME Builtin Function Syntax

The returned value is in the form 'hhmmss' where hh is hours, mm is minutes, and ss is seconds. For example:

```
DCL TIMEZ PIC '99.99.99' INIT(TIME( ));
```

TIME builtin function examples are shown in figure 11-133.

```
POSSIBLE VALUES
OF TIME() INCLUDE:

      ↑000001↑
      ↑085959↑
      ↑223344↑
```

Figure 11-133. TIME Builtin Function Examples

TRANSLATE BUILTIN FUNCTION

The TRANSLATE builtin function performs character substitution from one string to another and returns the result string. TRANSLATE syntax is shown in figure 11-134.

```
TRANSLATE(string,replacement[,pattern])
```

Figure 11-134. TRANSLATE Builtin Function Syntax

Argument string is the source string for the translation. Argument replacement is the replacement character string. Argument pattern is the string that determines which characters are to be replaced. If argument pattern is not specified, the character collating sequence is used. Argument string, replacement, and pattern can be expressions; their values must be scalar and computational.

All three arguments are converted to character strings. If argument replacement is shorter than argument pattern, it is extended on the right with blanks until the two strings are equal in length.

The result string is the same as the source string except for the following translation:

> For each character in argument string, a search is performed for the leftmost identical character in argument pattern.
>
> If such a character is found in pattern, the character in the same position in argument replacement is selected as the replacement character.
>
> This replacement character is then substituted for the original character in the result.

TRANSLATE builtin function examples are shown in figure 11-135.

STRING	REPLACEMENT	PATTERN	TRANSLATE(STRING, REPLACEMENT, PATTERN)
↑ 105↑	↑0↑	↑ ↑	↑000105↑
↑000105↑	↑ ↑	↑0↑	↑ 1 5↑
↑12.345.678.90↑	↑.↑	↑.↑	↑12.345.678.90↑
↑12.345↑	↑*.↑	↑.↑	↑12*345↑
↑AB CD EF↑	↑B↑	--	↑ B B ↑

Figure 11-135. TRANSLATE Builtin Function Examples

TRUNC BUILTIN FUNCTION

The TRUNC builtin function truncates the fractional part of an argument and returns the integer value. TRUNC syntax is shown in figure 11-136.

```
TRUNC(x)
```

Figure 11-136. TRUNC Builtin Function Syntax

Argument x can be an expression; the value must be scalar and computational. Argument x is evaluated and converted to an arithmetic value. The result of the function is a real value with the base and scale of the converted argument. If the result is a floating point value, the result precision is that of the converted argument. If the result is a fixed point value, the result precision is

$$(\min(N,\max(p-q+1,1)),\ 0)$$

where N is the maximum precision (p) for the given base (14 for decimal, 48 for binary), and (p,q) is the precision of the converted argument.

If x is greater than or equal to zero, the result value is FLOOR(x). If x is less than zero, the result value is CEIL(x).

TRUNC builtin function examples are shown in figure 11-137.

X	TRUNC(X)
3.2	3
-3.8	-3
-3.875	-3
-2.50E+000	-2.00E+000
1.26583E+002	1.26000E+002

Figure 11-137. TRUNC Builtin Function Examples

UNSPEC BUILTIN FUNCTION

The UNSPEC builtin function determines the internal representation of an argument value and returns the value in bit string form. UNSPEC syntax is shown in figure 11-138.

```
UNSPEC(x)
```

Figure 11-138. UNSPEC Builtin Function Syntax

Argument x must be scalar; it can have any data type except file or entry; it can be an expression, but cannot be a constant. The value and length of the bit string result depend on the data type and value of x.

UNSPEC PSEUDOVARIABLE

UNSPEC operates as a pseudovariable when it is used as the target of an assignment operation. The source value is converted to a bit string and is then assigned, without further conversion, to the internal representation of the variable identified by the argument. UNSPEC pseudovariable syntax is shown in figure 11-139.

```
UNSPEC(target-variable)
```

Figure 11-139. UNSPEC Pseudovariable Syntax

Argument target-variable must reference a scalar variable. The source value is converted to a bit string; its length depends on the data type of argument target-variable.

Any use of the UNSPEC pseudovariable requires detailed knowledge, on the part of the programmer, of the internal representation of PL/I data. The results are implementation-dependent and might not remain constant across system modifications.

VALID BUILTIN FUNCTION

The VALID builtin function determines whether or not the value of a pictured item is valid according to its picture, and returns a bit string indication of its validity. VALID syntax is shown in figure 11-140.

```
VALID(pictured-variable)
```

Figure 11-140. VALID Builtin Function Syntax

Argument pictured-variable must be scalar and must be a reference to a pictured variable; it can be either pictured numeric or pictured character. If the character string value of the argument is a possible legal result of assignment to that variable, the result of the function is '1'B; otherwise, the result is '0'B.

VALID builtin function examples are shown in figure 11-141.

```
DATATYPE          STRING
OF X              VALUE OF X      VALID(X)

PIC↑AAA99↑        ↑JAN05↑         ↑1↑B
PIC↑AAA99↑        ↑JAN05↑         ↑0↑B
PIC↑AAA99↑        ↑00105↑         ↑0↑B
PIC↑$$$9V.99↑     ↑ $33.25↑       ↑1↑B
PIC↑$$$9V.99↑     ↑  33.25↑       ↑0↑B
```

Figure 11-141. VALID Builtin Function Examples

VERIFY BUILTIN FUNCTION

The VERIFY builtin function returns a value that indicates the position in one character string of the first character that does not appear anywhere in a second character string. VERIFY syntax is shown in figure 11-142.

```
VERIFY(string1,string2)
```

Figure 11-142. VERIFY Builtin Function Syntax

Both arguments are first converted to character string. Each character in string1, taken in left-to-right order, is compared with all characters in string2 to see if string2 contains that character anywhere.

The result is a fixed binary integer indicating the position within the converted value of string1 of the leftmost character within string1 that has no match in string2.

The result is zero under any of the following circumstances:

String1 is a null string.

All string1 characters are found in string2.

VERIFY builtin function examples are shown in figure 11-143.

```
                                  VERIFY
STRING1      STRING2              (STRING1,STRING2)

↑↑           ↑ 0123456789↑              0
↑  123.78↑   ↑ 0123456789↑              6
↑  123.78↑   ↑ 0123456789.↑             0
↑  123.78↑   ↑↑                         1
```

Figure 11-143. VERIFY Builtin Function Examples

PL/I statements establish names to be used in the program, state the characteristics that are to be associated with the names, and describe the operations to be performed at run time. A statement consists of the language keywords in combination with programmer-supplied elements.

This section of the manual defines the syntax for each statement that can be used in the source program. Coding conventions and specific rules regarding the use and positioning of delimiters are detailed in section 1, PL/I Source Program.

STATEMENT CLASSIFICATION

Statements can be grouped into nine functional categories. The categories and their respective statements are listed in table 12-1. The categories are not significant to the language; they are presented for descriptive purposes only.

PREFIXES

Prefixes can be grouped into two general categories: condition and statement. Condition prefixes enable or

TABLE 12-1. CLASSIFICATION OF STATEMENTS

Functional Category	Statement	Statement Description
Declaration	DECLARE	Establishes explicit declarations for the identifiers.
Structural	BEGIN	Denotes the beginning of a begin block.
	DO	Denotes the beginning of a do group.
	END	Denotes the end of a procedure block, begin block, or a do group.
	ENTRY	Denotes a secondary point of a procedure.
	PROCEDURE	Denotes the beginning of a procedure block.
Storage Control	ALLOCATE	Reserves storage for controlled or based variables.
	FREE	Releases previously allocated storage for controlled or based variables.
Interrupt Handling	ON	Establishes the on-unit for a condition.
	REVERT	Removes the on-unit for a condition.
	SIGNAL	Simulates a condition.
Data Manipulation	Assignment	Assigns the value of an expression to one or more target items.
Program Control	CALL	Invokes a procedure as a subroutine.
	GOTO	Transfers control to a labeled statement.
Program Control (Cont'd)	IF	Allows the flow of control to take alternate paths depending on the value of an expression.
	Null	Indicates no action is to be taken.
	RETURN	Terminates a procedure activation and returns control to the point of invocation.
	STOP	Terminates program execution.
I/O Control	CLOSE	Closes a file and releases all storage associated with the file.
	OPEN	Opens a file for subsequent I/O operations.
Stream I/O	FORMAT	Specifies the formats and lengths of data item values transmitted during edit-directed stream I/O.
	GET	Specifies input operations on a stream file.
	PUT	Specifies output operations on a stream file.
Record I/O	DELETE	Deletes a record from a record file.
	LOCATE	Allocates a record buffer and transmits a previously located record to a record file.
	READ	Reads a record from a record file.
	REWRITE	Replaces a record in a record file.
	WRITE	Transmits a record to a record file.

disable computational conditions that can cause program interrupts. Statement prefixes identify and provide a means of referencing statements or blocks; they are classified as entry, format, and label.

Prefixes precede the body of a statement. Each must be terminated by a colon. Any statement can have one or more prefixes with the following restrictions:

- Each ENTRY and PROCEDURE statement must have at least one entry prefix.
- Each FORMAT statement must have at least one format prefix.
- DECLARE, END, and ENTRY statements cannot have condition prefixes.
- The first or only statement of an on-unit cannot have a label prefix.

ENTRY PREFIX

An entry prefix is used only with the ENTRY and PROCEDURE statements. Either statement requires at least one. Entry prefix syntax is shown in figure 12-1.

Figure 12-1. Entry Prefix Syntax

An entry prefix cannot be subscripted. The entry-name is an identifier that represents an entry point at which the procedure can be invoked. Each entry-name is established as an entry constant by the compiler. Control cannot be transferred to an entry constant by a GOTO statement.

The entry-name on a PROCEDURE statement is the primary entry point of the procedure block. Multiple entry-names can be used to provide alternate methods of referring to the same statement. Any of the entry-names on the PROCEDURE statement can be used to invoke the procedure at the PROCEDURE statement.

The entry-name on an ENTRY statement is a secondary entry point of the procedure block in which the statement appears. Multiple entry-names can be used to provide alternate methods of referring to the same statement. Any of the entry-names on the ENTRY statement can be used to invoke the procedure at the ENTRY statement.

Examples of entry prefixes are shown in figure 12-2.

```
PAYROLL: PROCEDURE;
           .
           .
           .
SALARY:  ENTRY(P1,P2) RETURNS(FIXED);
           .
           .
           .
OVTIME:  ENTRY(P3,P4);
```

Figure 12-2. Sample Entry Prefixes

FORMAT PREFIX

A format prefix is used only with the FORMAT statement. Each FORMAT statement requires at least one. Format prefix syntax is shown in figure 12-3.

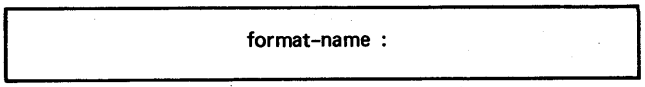


Figure 12-3. Format Prefix Syntax

A format prefix cannot be subscripted. The format-name is an identifier that is used to reference a FORMAT statement and cause it to be executed during edit-directed stream I/O. Multiple format-names can be used to provide alternate methods of referring to the same statement. Each format-name is established as a format constant by the compiler. Control cannot be transferred to a format constant by a GOTO statement.

Examples of format prefixes are shown in figure 12-4.

```
GET EDIT(A,B) (R(INP1));
PUT EDIT(HEADER) (R(OUT1));
  .
  .
  .
INP1: FORMAT(F(4),E(7,3));
OUT1: FORMAT(PAGE,A);
```

Figure 12-4. Sample Format Prefixes

LABEL PREFIX

A label prefix is used with any statement except ENTRY, PROCEDURE, and FORMAT. A label prefix cannot be used with the first statement of an on-unit. Label prefix syntax is shown in figure 12-5.

Figure 12-5. Label Prefix Syntax

The label is an identifier that is used to reference the associated statement. Multiple labels can be used to provide alternate methods of referring to the same statement. Each label is established as a label constant by the compiler. Control can be transferred to a label constant by a GOTO statement.

Each subscript must be an optionally signed decimal integer.

Examples of label prefixes are shown in figure 12-6.

CONDITION PREFIX

A condition prefix can be used with any statement except END, ENTRY, and DECLARE. The condition prefix enables or disables program interrupts that can result when computational conditions are raised. Condition prefix syntax is shown in figure 12-7.

```
TAG1:LOOP1: GET LIST(A,B);
TAG2(1,2): CALL TRANSAC;
.
.
.
B1: BEGIN;
.
.
.
FINAL: END;
```

Figure 12-6. Sample Label Prefixes

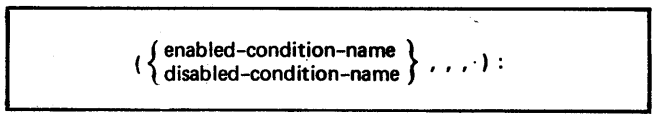


Figure 12-7. Condition Prefix Syntax

Each enabled-condition-name or disabled-condition-name can be any of the keywords listed in table 12-2. Condition prefixes must not contain conflicting names, such as SIZE and NOSIZE; redundant names are ignored. Refer to section 10 for additional information regarding condition prefix specification and handling.

TABLE 12-2. CONDITION PREFIX CONDITION NAMES

Enabled-Condition-Name	Disabled-Condition-Name
CONVERSION or CONV	NOCONVERSION or NOCONV
FIXEDOVERFLOW or FOFL	NOFIXEDOVERFLOW or NOFOFL
OVERFLOW or OFL	NOOVERFLOW or NOOFL
SIZE	NOSIZE
STRINGRANGE or STRG	NOSTRINGRANGE or NOSTRG
SUBSCRIPTRANGE or SUBRG	NOSUBSCRIPTRANGE or NOSUBRG
UNDERFLOW or UFL	NOUNDERFLOW or NOUFL
ZERODIVIDE or ZDIV	NOZERODIVIDE or NOZDIV

When an enabled-condition-name is specified, the condition is enabled and subject to interrupt. If the condition is raised during execution, a programmed or system action takes place. The action is called an on-unit and is detailed under ON Statement in this section. Enabling a condition ensures that errors will be detected.

When a disabled-condition-name is specified, the condition is disabled and not subject to interrupt. It is an error if the condition is raised during execution. Disabling a condition improves the execution speed of the program.

When condition prefixes are used in combination with entry, format, or label prefixes, no restrictions are imposed on the order in which they can appear. Examples of condition prefixes used alone and in combination with other prefixes are shown in figure 12-8.

```
(SIZE): SUM=A+B;

(OVERFLOW): SUM=A+B;

(SIZE,OVERFLOW): SUM=A+B;

(SIZE):(OFL): SUM=A+B;

(SIZE):P: SUM=A+B/C;

PROB1:(SIZE,OVERFLOW): SUM=A+B*C/D;

(SIZE,OFL):PROB1:PROBA: SUM=A+B*C/D;

(SIZE):PROB1:(OFL):PROBB: SUM=A+B*C/D;

(NOOFL):B1: BEGIN;
```

Figure 12-8. Sample Condition Prefixes

STATEMENT DESCRIPTIONS

Statement descriptions appear in alphabetic order. Each statement description includes the statement syntax, applicable rules, processing details, and examples.

Prefixes are indicated in the syntax as follows:

When any combination of label and condition prefixes is allowed, the notation [prefix]. . . is used.

In all other cases, prefix options are specifically indicated.

Although prefixes are shown in the syntax diagrams, a prefix is not considered to be contained in a statement.

ALLOCATE STATEMENT

The ALLOCATE statement allocates a generation of storage for controlled or based variables. One ALLOCATE statement can allocate both controlled and based variables. The syntax of the ALLOCATE statement is shown in figure 12-9.

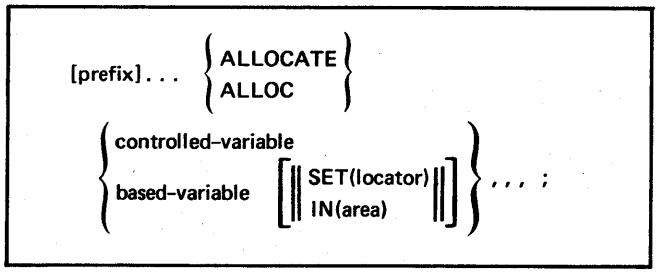


Figure 12-9. ALLOCATE Statement Syntax

The following rules apply:

- A condition prefix applies to execution of the statement, including the evaluation of dimension, INITIAL, or other attributes of a variable being allocated.

- The controlled-variable can be a scalar, an array, or a structure; it cannot be an array element or a structure member. The variable must be explicitly declared with the CONTROLLED attribute. It can have either the INTERNAL or EXTERNAL attribute.
- The based-variable can be a scalar, an array, or a structure; it cannot be an array element or a structure member. The variable must be explicitly declared with the BASED attribute.
- The IN and SET options can appear in either order.
- The locator specified by the SET option can be a scalar, an array element, or a structure member; it cannot be an array.
- The area specified by the IN option can be a scalar, an array element, or a structure member; it cannot be an array. The area must have a generation of storage.

Allocating a Controlled Variable

When a controlled variable is allocated, a new generation of storage is added as the top entry in the stack of generations currently associated with the variable. Extent expressions in the declaration of the controlled variable are evaluated when the variable is allocated.

Allocating a Based Variable

When a based variable is allocated, a locator value identifying the generation is created and assigned to a locator variable. The new generation can then be accessed through this locator variable. Extent expressions for the based variable are evaluated when the variable is allocated and each time the variable is referenced.

SET Option

The SET option specifies the locator (pointer or offset) variable. The value assigned to the locator by the ALLOCATE statement identifies the newly allocated based generation.

If a based variable is allocated without a SET option, an assumed SET option is constructed from the declaration of the based variable. If the locator in the specified or assumed SET option is an offset variable, the allocation is made within an area.

SET option processing is summarized in table 12-3.

TABLE 12-3. SET OPTION PROCESSING

Based Variable Declaration	SET Option Specified	SET Option Not Specified
BASED(locator-variable)	SET option locator is used; declared locator is ignored.	Declared locator is used.
BASED(locator-valued-function)	SET option locator is used.	Illegal.
BASED	SET option locator is used.	Illegal.

IN Option

The IN option specifies the area in which the based variable is to be allocated.

If the locator in the specified or assumed SET option is an offset variable, the allocation is made within an area.

If the locator in the specified or assumed SET option is an offset variable and the based variable is allocated without an IN option, an assumed IN option is constructed from the declaration of the offset variable. The declaration must contain OFFSET (area), where area is a scalar area variable that has a generation of storage.

If the locator in the specified or assumed SET option is a pointer variable, the allocation is made within an area only if the based variable is allocated with an explicit IN option.

IN option processing is summarized in table 12-4.

TABLE 12-4. IN OPTION PROCESSING

Locator-Variable Declaration	IN Option Specified	IN Option Not Specified
OFFSET(area-variable)	IN option area generation is used. IN option and declaration of offset variable must specify the same area generation.	Declared area variable is used.
OFFSET	IN option area generation is used.	Illegal.
POINTER	IN option area generation is used.	The based variable is allocated; it is not allocated within an area.

Statement Processing

Execution of the ALLOCATE statement causes a generation of storage to be allocated for each named variable. The variables to be allocated are processed in left-to-right order, and each variable is processed through the steps listed in table 12-5.

A nonlocal GOTO must not cause the termination of any block activation that has an immediate dynamic predecessor currently executing an ALLOCATE statement.

Examples of ALLOCATE statements are shown in figure 12-10.

Conditions

AREA, STORAGE, and computational conditions can be raised during ALLOCATE statement execution. If control returns from an AREA or STORAGE on-unit and sufficient storage has not been provided, the program will loop indefinitely.

TABLE 12-5. ALLOCATE STATEMENT PROCESSING

Step	Performed For	Action Taken
1.	A controlled or based variable.	Extent expressions are evaluated to determine sizes of arrays, areas, and strings; the data description is constructed from these results and from the attributes in the declaration of the based or controlled variable.
2.	A controlled variable.	The stack of generations currently associated with the controlled variable is pushed down; a new generation is allocated and placed at the top of the stack. If insufficient storage is available, the STORAGE condition is raised. On normal termination of the STORAGE on-unit, step 2 is repeated.
3.	A based variable with a specified or assumed IN option.	The based variable is allocated within the specified area. If insufficient storage is available, the AREA condition is raised. On normal termination of the AREA on-unit, steps 1 and 3 are repeated.
4.	A based variable with no specified or assumed IN option.	The based variable is allocated; it is not allocated within an area. If insufficient storage is available, the STORAGE condition is raised. On normal termination of the STORAGE on-unit, step 4 is repeated.
5.	Any based variable.	A locator value identifying the new generation is assigned to the locator variable specified in the SET option or in the declaration of the based variable.
6.	A based structure.	Appropriate values are assigned to all variables appearing in REFER options in the declaration of the based variable.
7.	A generation containing one or more areas.	EMPTY ( ) is assigned to all areas in the new generation.
8.	A controlled or based variable.	The newly allocated generation is initialized according to INITIAL attributes in its declaration.

```
ALLOCATE A,B;

ALLOCATE C SET(P) IN (X);

ALLOC D IN(Y),E SET(Q);
        .
        .
        .
DCL F AREA(1000)BASED(P),G AREA(300)BASED(Q);
ALLOCATE H,K IN (F);
        .
        .
        .
DCL L BASED(P),M CONTROLLED;
ALLOCATE M,L;
```

Figure 12-10. Sample ALLOCATE Statements

ASSIGNMENT STATEMENT

The assignment statement causes the value of the expression on the right of the assignment symbol (=) to be assigned to each target on the left of the assignment symbol. The syntax of the assignment statement is shown in figure 12-11.

```
[prefix]. . . assignment-target, , , = expression ;
```

Figure 12-11. Assignment Statement Syntax

The following rules apply:

- Each assignment-target must be a scalar variable or pseudovariable. The assignment-target can be an array element or a structure member, but cannot be an array or structure. It can be subscripted, but cannot have any asterisk subscripts.
- Assignment-target and expression data types must be compatible. They must all be in one of the four following classes: computational (arithmetic, string, or pictured); locator (pointer or offset); label; or area. They cannot be of type file, entry, or format.
- The expression must yield a scalar value.

The assignment statement is processed through the following steps in the order given. The program must not depend upon the order of evaluation of variable references, function references, and expressions within the assignment-target or the expression to the right of the assignment symbol (except as controlled by parentheses).

1. The expression on the right of the assignment symbol is evaluated first to produce a result value. The result value is used as the source value for assignment. The source value is not modified during subsequent processing of the statement.

2. Assignment-targets are processed in left-to-right order. For each assignment-target, the following steps are taken: the target reference is evaluated; the source value is converted, if necessary, to an intermediate value with the data type of the target; and the intermediate value is assigned to the target. When the data type of the source value is the same as that of the target, assignment consists of copying the source value to the target.

Expression evaluation, conversion, and assignment are detailed in section 7, Data Manipulation.

CONVERSION, SIZE, STRINGRANGE, and SUBSCRIPTRANGE conditions can be raised during the assignment action; computational conditions can be raised during evaluation of any of the expressions in the statement. If a condition is raised, the current established on-unit is activated.

Examples of assignment statements are shown in figure 12-12.

```
A = X;

A, B.C = 0;

A(I,J) = X*Y-1;

ONSOURCE() = ↑↑;

SUBSTR(A(2,J)) = ↑ABCDEF↑;

X,Y = Z;    ←——Equivalent to X=Z; Y=Z;

X = Y = Z;  ←——Y=Z is a comparison expression with
                a value of '1'B if true and '0'B if false.
                If Y is equal to Z, the value '1'B is
                assigned to X.

      An assignment can depend upon previous assignments
      in the same statement. For example:

I = 3;
I,B(I+1) = I+1; ←—— sets I to 4 and B(5) to 4
```

Figure 12-12. Sample Assignment Statements

BEGIN STATEMENT

The BEGIN statement, together with its associated END statement, delineates a sequence of statements called a begin block. A begin block is used primarily to specify the scope within which declared names are known, and to control the allocation and use of automatic variables. The syntax of a begin block is shown in figure 12-13. The syntax of the BEGIN statement is shown in figure 12-14.

The following rule applies:

- A condition prefix applies to all statements contained in the begin block. The only exception is when a contained statement or block has a condition prefix enabling or disabling the same condition.

Execution of a BEGIN statement causes the begin block to be activated. A BEGIN statement is executed when it is encountered during the normal sequential flow of control or when control is transferred to it by a GOTO statement.

A BEGIN statement label is declared as a label constant in the block that contains the begin block; therefore, the same name can be redeclared in the begin block.

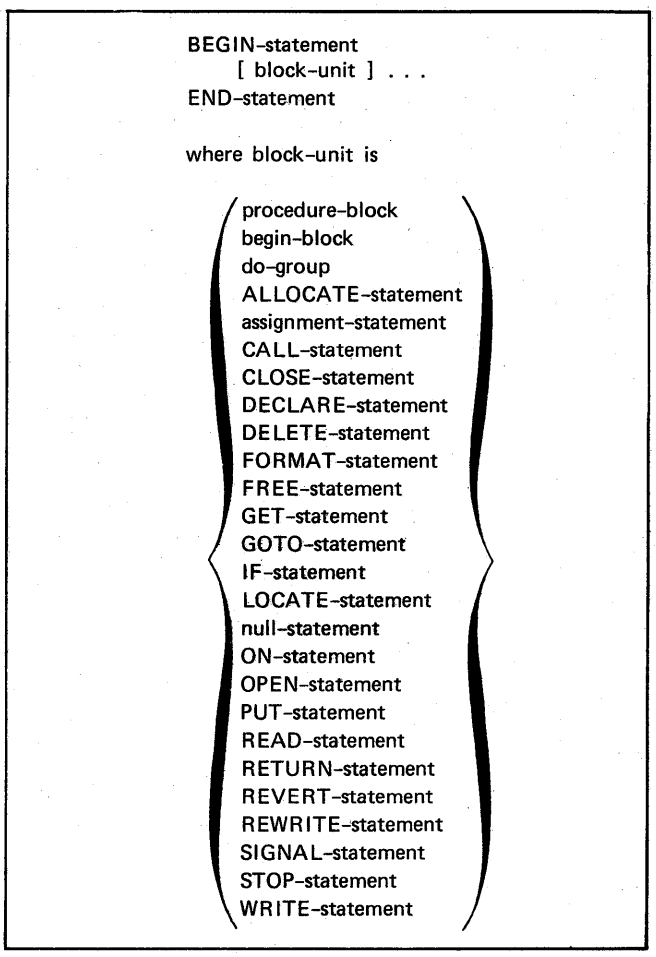


Figure 12-13. Begin Block Syntax

```
[prefix]. . . BEGIN ;
```

Figure 12-14. BEGIN Statement Syntax

Examples of BEGIN statements are shown in figure 12-15.

CALL STATEMENT

The CALL statement causes a procedure to be invoked as a subroutine. The syntax of the CALL statement is shown in figure 12-16.

The following rules apply:

- The subroutine-reference must be an entry constant or an entry parameter. An entry constant is the entry prefix on a PROCEDURE or ENTRY statement, or an external entry constant declared by DECLARE statement.
- The PROCEDURE or ENTRY statement identified by subroutine-reference must not have a RETURNS attribute.
- The subroutine-reference must not reference a function entry point.

```
A: BEGIN;
   .
   .
   .
   END A;


ON AREA BEGIN;
        .
        .
        .
        END;


(SIZE): B: BEGIN;
             .
             .
             .
C:D:           BEGIN;
               .
               .
               .
               END C;
             END B;
```

Figure 12-15. Sample BEGIN Statements

```
[prefix]... CALL subroutine-reference [([argument, , ,])]  ;
```

Figure 12-16. CALL Statement Syntax

- The number of arguments in the CALL statement must equal the number of parameters listed in the PROCEDURE or ENTRY statement identified by the subroutine-reference. If the subroutine-reference is an external entry constant, the number of arguments must equal the number of parameters declared for the entry constant.
- Each argument is an expression that can have a scalar or aggregate value.
- Arguments must be compatible with the corresponding parameters. Each argument and the corresponding parameter must both be in one of the following six classes: computational (arithmetic, string, or pictured); locator (pointer or offset); label; area; entry; or file. An argument cannot be a format value.
- The data type, aggregate type, and alignment of an aggregate argument must match those of the corresponding parameter as defined in section 6, References.

When the CALL statement is executed, the referenced procedure is invoked as a subroutine. Arguments in the CALL statement are passed to corresponding parameters in the PROCEDURE or ENTRY statement of the invoked procedure. Refer to section 6 for details regarding argument passing.

Argument expressions are evaluated in the block activation in which the CALL statement is executed; consequently, the expressions cannot depend on values established by the new block activation. Any on-units activated as a result of argument expression evaluation are activated before the new block activation occurs.

The block containing the entry point to be invoked is activated, and control is transferred to the designated entry point in that block. If the block activation to which control is transferred terminates by execution of a RETURN or END statement, control returns to the statement following the CALL statement.

Examples of CALL statements are shown in figure 12-17.

```
CALL A;

CALL A();

CALL B(CODE,I+J,TIME());

CALL C('NONE');
```

Figure 12-17. Sample CALL Statements

CLOSE STATEMENT

The CLOSE statement closes each referenced file. The syntax of the CLOSE statement is shown in figure 12-18.

The following rules apply:

- Each file-reference must name a file constant or a file variable.
- The ENVIRONMENT option specified for a file can precede or follow the FILE option.

ENVIRONMENT Option

The ENVIRONMENT option specifies CRM file processing options. Options represented by the ENVIRONMENT option expression augment the CYBER Record Manager File Information Table (FIT) for the file.

Some of the options permitted on the CYBER Record Manager FILE control statement can be included. The only useful parameter for a close operation is the close flag, CF, which provides for file positioning after the close operation. Refer to section 9, CYBER Record Manager Interface, and to the CYBER Record Manager reference manuals for additional details.

```
[prefix]... CLOSE { FILE(file-reference) [ {ENVIRONMENT} (expression) ] } ... ;
                                          {ENV        }
```

Figure 12-18. CLOSE Statement Syntax

Statement Processing

The files to be closed are processed in left-to-right order. If a file is not open, no action is taken for that file. Each open file is processed through the following steps:

1. The ENVIRONMENT option expression is evaluated and converted to character string.
2. If the file is an output record file and an allocated buffer created by a previous LOCATE statement execution exists, the buffer is written as a record. If a key value is associated with the buffer, the value is used as the record key. The KEY and RECORD conditions can be raised during this step.
3. If the file has a buffer allocated by the previous execution of a LOCATE statement or a READ statement with a SET option, that buffer is freed.
4. The ENVIRONMENT character string is used to establish CRM FIT fields before the CRM file is closed. The status of the file is set to closed, and the CRM file is closed by CYBER Record Manager.
5. All file status information added during file opening is discarded.

When all file-references have been processed, control passes to the statement following the CLOSE statement. A closed file can subsequently be reopened. On each opening, the file can have a different completed set of file description attributes and can be associated with a different CRM file.

Examples of CLOSE statements are shown in figure 12-19.

```
CLOSE FILE(F);

CLOSE FILE(A),FILE(B);

ON ENDFILE(F) CLOSE FILE(F);

CLOSE FILE(F) ENVIRONMENT('CF=N');

DCL CLOSEFG CHAR(5) INIT('R');
CLOSE FILE(A) ENV('CF=' || CLOSEFG);
```

Figure 12-19. Sample CLOSE Statements

Conditions

Conditions that can be raised when an allocated buffer is being written during CLOSE statement execution include: KEY, RECORD, and TRANSMIT.

If the key value associated with the allocated buffer duplicates an existing key in the CRM file, the KEY condition is raised and the record is not written. If control returns from the KEY on-unit, execution continues with the next file to be closed.

If the length of the allocated buffer conflicts with the record length specified for the CRM file, the RECORD condition is raised and the record is not written. If control returns from the RECORD on-unit, the ERROR condition is raised.

The TRANSMIT condition is raised if parity errors are encountered. If control returns from the on-unit, execution continues with the next file to be closed.

Computational conditions can be raised during expression evaluation, conversion, and assignment.

DECLARE STATEMENT

The DECLARE statement is used during compilation to establish explicit declarations for identifiers. The syntax of the DECLARE statement is shown in figure 12-20.

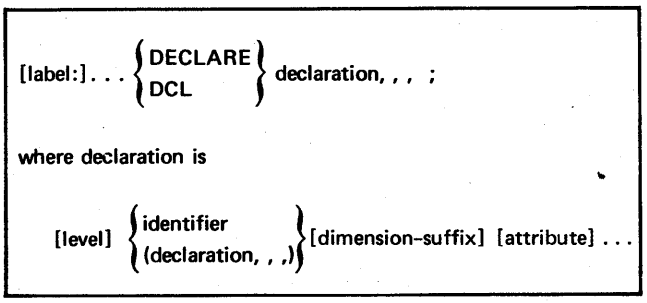


Figure 12-20. DECLARE Statement Syntax

The following rules apply:

- The statement cannot have a condition prefix.
- Each level is an unsigned decimal integer.

The DECLARE statement establishes declarations for identifiers and can specify attributes that are to be associated with the identifiers. An identifier can have only one meaning in a given block; therefore, only one explicit declaration for an identifier can be made in one block.

The use of level numbers creates a structure declaration. Refer to the structure attribute in section 4.

The dimension-suffix specifies the dimension attribute. Refer to the dimension attribute in section 4.

Declarations can be factored if identifiers have the same attributes or level numbers. This convention reduces the need for repeated specification of the same attribute or level number.

Attributes are factored by enclosing the identifiers having the same attributes in parentheses and following the parenthesized list with the set of applicable attributes. For example:

```
DECLARE (A,B,C) FIXED DECIMAL (5,2);
```

is equivalent to

```
DECLARE A FIXED DECIMAL (5,2),
        B FIXED DECIMAL (5,2),
        C FIXED DECIMAL (5,2);
```

Structure level numbers are factored by enclosing the identifiers and attributes in parentheses and preceding the parenthesized list with the single level number. For example:

```
DECLARE 1A, 2(X FIXED, Z AREA(100));
```

is equivalent to

```
DECLARE 1 A,
          2 X FIXED,
          2 Z  AREA(100);
```

Examples of factored and defactored DECLARE statements are shown in figure 12-21.

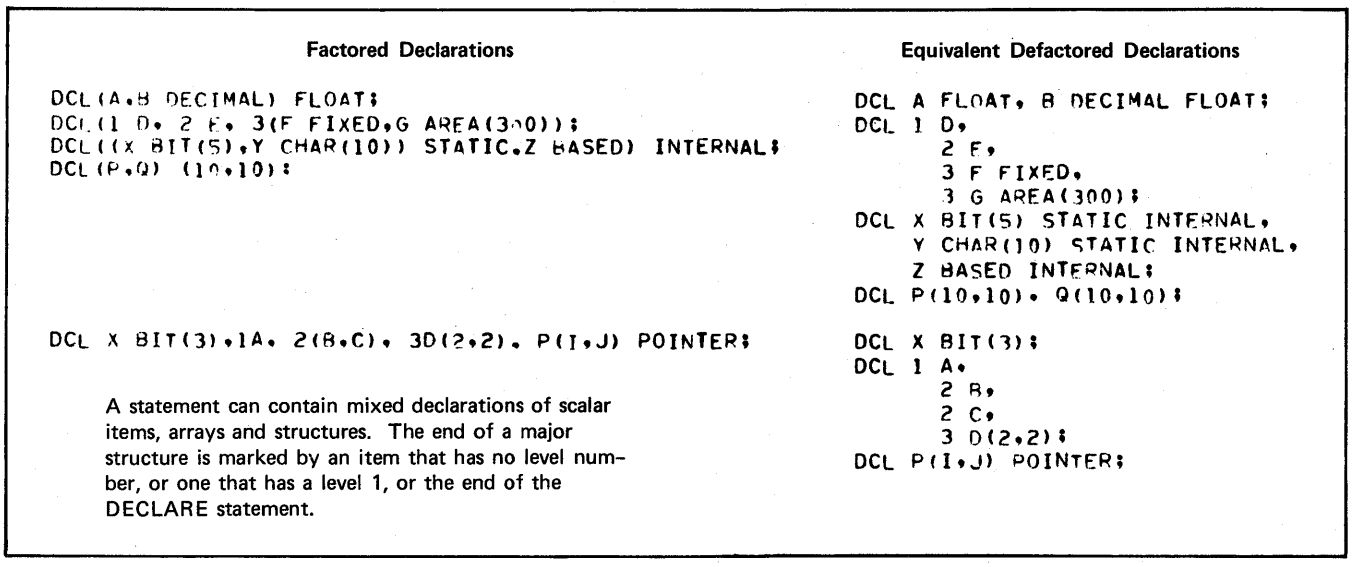


Figure 12-21. Sample DECLARE Statements

Refer to section 5, Declarations, for detailed information regarding explicit declarations and section 4, Attributes, for detailed descriptions of attributes.

DELETE STATEMENT

The DELETE statement deletes a record from the CRM file associated with the referenced file. The syntax of the DELETE statement is shown in figure 12-22.

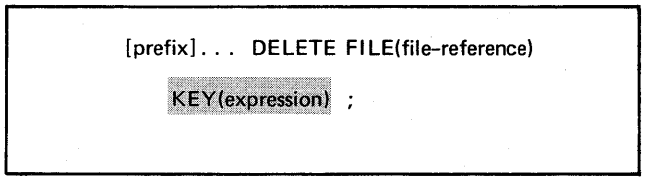


Figure 12-22. DELETE Statement Syntax

The following rules apply:

- The FILE and KEY options can appear in either order after the DELETE keyword.
- The file-reference must be a file constant or a file variable.
- The DELETE statement must have a KEY option.
- The KEY expression must be scalar and computational.

Statement Processing

The DELETE statement is processed through the following steps:

1. If the file is not open, an implicit file opening is executed. Implied attributes RECORD and UPDATE are supplied for the file opening.
2. The file must now have the RECORD, UPDATE, and KEYED attributes.
3. The KEY option expression is evaluated and converted to a character string. The record specified by the KEY option is deleted. If the file is SEQUENTIAL, the current-record designator is set to designate the record preceding the deleted record.
4. If the file has an allocated buffer from previous execution of a READ statement with a SET option, that allocated buffer is freed.

Examples of DELETE statements are shown in figure 12-23.

```
DELETE FILE(A) KEY(143);

DELETE FILE(PAY) KEY(↑A↑vvID);
```

Figure 12-23. Sample DELETE Statements

Conditions

Conditions that can be raised during DELETE statement execution include: UNDEFINEDFILE, KEY, and TRANSMIT.

The UNDEFINEDFILE condition is raised when an implicit file opening fails. If control returns from the UNDEFINEDFILE on-unit and the file is still not open, the ERROR condition is raised; if the file is open, processing of the statement continues.

The KEY condition is raised when the value of the KEY option does not identify a record in the CRM file. If control returns from the KEY on-unit, program execution continues at the statement following the DELETE statement.

The TRANSMIT condition is raised if parity errors are encountered. If control returns from the TRANSMIT on-unit, execution continues with the statement following the DELETE statement.

Computational conditions can be raised during expression evaluation, conversion, and assignment.

DO STATEMENT

The DO statement, together with its associated END statement, delineates a sequence of statements called a do group. The syntax of a do group is shown in figure 12-24. The DO statement controls the number of times the do group is executed, and it can cause a value to be assigned to an index variable each time the do group is executed. The syntax of the DO statement is shown in figure 12-25. The syntax reflects three types of operations: noniterative DO, DO WHILE, and indexed DO. Notice that the WHILE expression (but not the TO and BY expressions) must always be enclosed in parentheses.

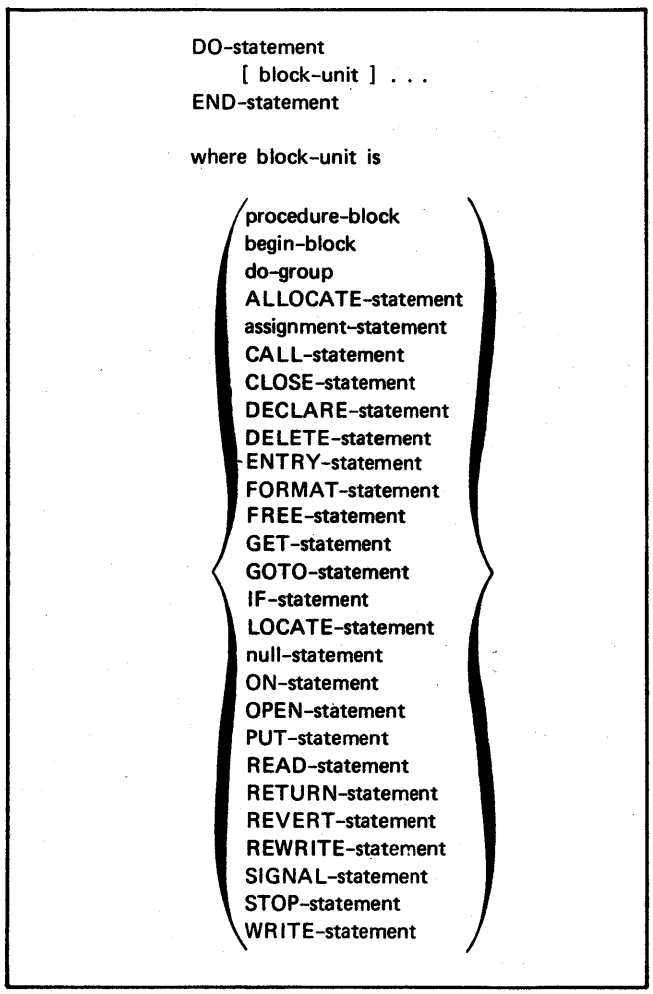


Figure 12-24. Do Group Syntax

The following rules apply:

- A condition prefix on the DO statement refers only to the DO statement and not to other statements within the do group.
- Each expression must yield a scalar value.
- The data type and value of each WHILE expression must permit conversion to a bit string.
- The index must be a scalar variable or pseudovariable.
- If the data type of the index is noncomputational, the TO and BY options are not permitted.
- The data type and value of each start-expression must permit assignment to the index.
- The TO and BY options can appear in either order.
- The data type and value of a TO or BY expression must permit conversion to arithmetic.
- If a do-specification contains a TO option but not a BY option, BY 1 is assumed.
- The block that immediately contains an ENTRY statement must be a procedure block; an ENTRY statement cannot be contained in an iterative do group that is contained in that block.

The three forms of the DO statement are described in the following paragraphs.

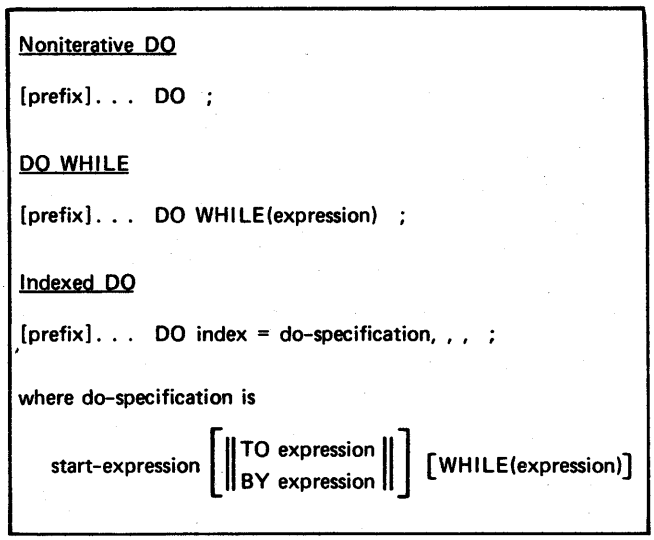


Figure 12-25. DO Statement Syntax

Noniterative DO

A noniterative DO is executed once. The DO statement consists of the single word DO followed by a semicolon.

When the DO statement is encountered, control passes to the statement following the DO statement. When the END statement of the group is encountered, control passes to the statement following the END statement.

Control can be transferred from a point outside a noniterative do group to a statement or block contained in the group unless the statement or block is within a nested block, a nested DO WHILE, or a nested indexed DO.

A noniterative DO is commonly used to delineate a THEN or ELSE clause in an IF statement. This approach provides grouping without block activation overhead.

Examples of noniterative DO operations are shown in figure 12-26.

```
IF X = Y THEN DO;
                •
                •
                •
              END;
         ELSE DO;
                •
                •
                •
              END;
```

Figure 12-26. Noniterative DO Operations

DO WHILE

A DO WHILE is executed repetitively. The WHILE expression controls the number of times the group is executed.

When the DO statement is executed, the WHILE expression is evaluated and converted to a bit string. Comparison operators yield '1'B or '0'B for true and false, respectively. If the string contains any one-bits ('1'B), control passes to the statement following the DO statement. If the string does not contain any one-bits ('1'B), control passes to the statement following the END statement of the do group.

Each time control reaches the END statement of the group, the WHILE expression is again evaluated, converted, and tested to determine the next point of control.

Control must not be transferred by a statement outside a DO WHILE to a statement or block contained in the group.

Examples of DO WHILE operations are shown in figure 12-27.

```
X = 5;
•
•
•
DO WHILE(X>0);
•
•
•
X = X-1;
END;
```

Figure 12-27. DO WHILE Operations

Indexed DO

An indexed DO is executed repetitively. The list of do-specifications determines the number of times the group is executed.

When the DO statement is executed, each do-specification causes the index to be set to an initial value for the first execution of the do group under control of that do-specification, and incremented for each succeeding execution of the group. Do-specifications are interpreted in left-to-right order. When the last execution of the do group is completed, control is transferred to the statement following the END statement of the group.

Control must not be transferred by a statement outside an indexed DO to a statement or block contained in the group.

If the index is a controlled or based variable, it must not be allocated or freed while the do group is active. If the index is a based variable, the value of its locator must not be changed while the do group is active. Either of these situations leads to unpredictable results.

Examples of indexed DO operations are shown in figure 12-28.

```
DO I = 1 TO J;
•
•
•
END;

DO I = 2 TO 10, 12 TO J BY 2 WHILE(A>100);
•
•
•
END;

DECLARE D(5) FIXED;
DO D(5) = J WHILE(K<=L);
•
•
•
END;

DO I = A, B, C/J+1;
•
•
•
END;

DECLARE I CHAR(10);
DO I = 'ABC', 1 TO 5, 'LAST';
•
•
•
END;

DECLARE L LABEL;
DO L = LABEL1,LABEL2,LABEL3;
•
•
•
END;
```

Figure 12-28. Indexed DO Operations

Do-Specification Processing

Each do-specification is processed through the following steps:

1. The start-expression, TO expression, and BY expression are evaluated. These expressions are not reevaluated while under control of this execution of the DO statement. If TO is specified and BY is not specified, BY 1 is assumed.
2. The value of the start-expression is assigned to the index. The values of the TO expression and the BY expression are preserved for use in tests during interpretation of the do-specification.
3. The do-specification is interpreted to determine whether or not the do group should be executed. Interpretation is performed through TO and WHILE

testing as described in a later paragraph. If the do group is to be executed, control is transferred to the statement following the DO statement. If the do group is not to be executed, control is transferred to the next do-specification or, if this is the last do-specification, to the statement following the group END statement.

Each time control reaches the END statement, the do-specification is interpreted to determine whether the index should be incremented and whether the group should be executed again under control of this do-specification.

The value of the do-specification is always assigned to the index before the TO and WHILE tests are performed.

TO and WHILE Testing

Two basic tests are used in the interpretation of a do-specification: TO-test and WHILE-test. The TO-test determines whether the value of the index is beyond the limit specified by the TO expression. The WHILE-test determines the value of the WHILE expression.

The TO-test is performed as follows:

- If the value of the index is greater than the value of the TO expression and the value of the BY expression is positive or zero, control under this do-specification is terminated.
- If the value of the index is less than the value of the TO expression and the value of the BY expression is negative, control under this do-specification is terminated.
- If neither of the above cases applies, control under this do-specification continues.

The WHILE-test is performed as follows:

- The WHILE expression is evaluated and converted to a bit string. Comparison operators yield '1'B and '0'B for true and false, respectively.
- If the string contains any one-bits ('1'B), control under this do-specification continues.
- If the string does not contain any one-bits ('1'B), control under this do-specification is terminated.

Do-Specification Options

Eight combinations of options can appear in a do-specification and cause six different actions to take place. These options and the resulting actions are summarized in table 12-6.

TABLE 12-6. DO-SPECIFICATION OPTIONS

DO index =	Initial Action	END Statement Action
start-expression	Control unconditionally passes to the statement following the DO statement.	Control under this specification is terminated.
start-expression WHILE(expression)	The WHILE-test is performed. If the test does not terminate control under this do-specification, control passes to the statement following the DO statement.	Control under this specification is terminated.
start-expression BY expression	Control unconditionally passes to the statement following the DO statement.	The index is incremented by the value of the BY expression. Control passes to the statement following the DO statement. This form of the do-specification does not provide a test to terminate execution under its control.
start-expression BY expression WHILE(expression)	The WHILE-test is performed. If the test does not terminate the do-specification, control passes to the statement following the DO statement.	The index is incremented by the value of the BY expression. The WHILE-test is performed. If the test does not terminate the do-specification, control passes to the statement following the DO statement.
start-expression TO expression BY expression start-expression TO expression	The TO-test is performed. If the test does not terminate the do-specification, control passes to the statement following the DO statement.	The index is incremented by the value of the BY expression. (If the BY option is not included, BY 1 is assumed by default.) The TO-test is performed. If the test does not terminate the do-specification, control passes to the statement following the DO statement.
start-expression TO expression BY expression WHILE(expression) start-expression TO expression WHILE(expression)	The TO-test is performed. If the test does not terminate the do-specification, the WHILE-test is performed. If this test does not terminate the do-specification, control passes to the statement following the DO statement.	The index is incremented by the value of the BY expression. (If the BY option is not included, BY 1 is assumed by default.) The TO-test is performed. If the test does not terminate the do-specification, the WHILE-test is performed. If this test does not terminate the do-specification, control passes to the statement following the DO statement.

END STATEMENT

The END statement denotes the end of a procedure block, begin block, or do group. The syntax of the END statement is shown in figure 12-29.

```
[label:]... END [closure-name] ;
```

Figure 12-29. END Statement Syntax

The following rules apply:

- The statement cannot have a condition prefix.
- The closure-name must match a preceding BEGIN or DO statement label prefix or a preceding PROCEDURE statement entry prefix.
- The closure-name cannot be subscripted.

Execution of an END statement that denotes the end of a begin block terminates the current block activation. The dynamic predecessor of the current block activation becomes the current block activation. Control passes to the statement following the END statement.

Execution of an END statement that denotes the end of an on-unit begin block terminates the on-unit activation; this is illegal for some conditions. Refer to section 10, Conditions, for additional information.

Execution of an END statement that denotes the end of a do group causes tests to be made to determine whether or not the group is to be executed again. The decision is dependent upon the form of the DO statement that heads the group.

Execution of an END statement that denotes the end of a procedure block terminates the current block activation. This produces the same effect as the execution of a RETURN statement with no return value. Execution of the END statement is illegal if the procedure was invoked as a function. If the current block activation is the first or only activation of the main procedure, execution of the END statement terminates program execution.

If an END statement denotes multiple closure, the statement is equivalent to several END statements, one for each block or group it closes. If the END statement has a label prefix, the label applies to the END statement of the outermost block or group closed; that is, it applies to the block or group referenced by the closure-name.

Refer to section 2, Dynamic Program Structure, for additional details regarding block and program termination.

Examples of END statements are shown in figure 12-30.

```
P: PROC;
   .
   .
   .
   B: BEGIN;
   .
   .
   .
      DO I = 1 TO J;
      .
      .
      .
      END;     /*END OF DO GROUP*/
      END B;   /*END OF BEGIN BLOCK*/
   END P;      /*END OF PROCEDURE BLOCK*/

P: PROC;
   .
   .
   .
   B: BEGIN;
   .
   .
   .
      DO I = 1 TO J;
   .
   .
   .
   END P;      /*MULTIPLE CLOSURE*/
```

Figure 12-30. Sample END Statements

```
{entry-name:}... ENTRY [(parameter, , ,)]
    [RETURNS [(returns-descriptor)]] ;
```

Figure 12-31. ENTRY Statement Syntax

ENTRY STATEMENT

The ENTRY statement denotes a secondary entry point of the procedure block that immediately contains the ENTRY statement. It can specify parameters for the entry point and the attributes of the value to be returned by the procedure when invoked at this entry point by a function reference. The syntax of the ENTRY statement is shown in figure 12-31.

The following rules apply:

- The statement cannot have a condition prefix.
- The block that immediately contains an ENTRY statement must be a procedure block; an ENTRY statement cannot be contained in a begin block, a DO WHILE, an indexed DO, or an on-unit. The statement can be contained in a noniterative DO.
- At least one entry-name must be included. The entry-name must not be subscripted.
- A variable cannot appear more than once in the same parameter list.
- Each parameter can be a scalar, an array, or a structure; it cannot be an array element, an array slice, or a structure member.
- Data types of corresponding arguments and parameters must be compatible. The number of parameters must equal the number of arguments passed when the entry point is invoked.
- If the procedure is invoked by CALL statement execution, the RETURNS option must not be included; if the procedure is invoked by function reference, the RETURNS option must be included.

- RETURNS( ) is illegal.
- Each extent (string length or area size) appearing in a returns-descriptor must be an expression consisting only of literal constants and the operators + - * / and ||. Array bounds cannot appear in a returns-descriptor.

Entry-Name

Each entry-name of an internal procedure is explicitly declared by the compiler as an internal entry constant in the block that immediately contains the procedure. An internal entry-name cannot be declared by DECLARE statement. The entry-name is known in the block that immediately contains the procedure and in all blocks contained in that block. Refer to the PROCEDURE statement for additional details regarding the scope of internal entry-names.

Each entry-name of an external procedure is explicitly declared by the compiler as an external entry constant outside of any block. Unless there is an overriding declaration, the entry-name is known in the external procedure and in all blocks contained in that external procedure.

A reference to an external entry constant from a point outside the external procedure that contains the ENTRY statement requires a DECLARE statement declaration as described under the ENTRY attribute in section 4, Attributes.

Each external name that is longer than seven characters is modified before external references are resolved by the loader. Only the first four and last three characters of the original identifier are used. Care should be taken to avoid accidental duplication of external names that are modified according to this convention. External names should not include any of the characters $ @ # and _ ; external names containing these characters might conflict with names used by the system.

An entry-name on an ENTRY statement can be referenced in a CALL statement or function reference that invokes the procedure, and in an argument list. It cannot be referenced in any other context.

An ENTRY statement with multiple entry-names creates one entry point that can be referenced by any of its names. For example:

```
E: X: ENTRY(PAR1) RETURNS(FIXED BIN);
```

All entry-names of a single ENTRY statement are equivalent with the following exception:

If INRULE processing is specified on the PLI control statement when the procedure is compiled, the first character of an entry-name can determine the default attributes supplied for a returns-descriptor; therefore, a single entry point with multiple names can have two different returns-descriptors.

Two consecutive ENTRY statements (even with no intervening executable statements) create two separate entry points. The entry points can have different parameters and RETURNS options. For example:

```
E: ENTRY(PAR1) RETURNS(FIXED BIN);
X: ENTRY(PAR2) RETURNS(FLOAT);
```

Parameters and Arguments

Each parameter in an ENTRY statement is a variable with which an argument will be associated when the procedure is invoked at that entry point. The appearance of a variable in the parameter list constitutes an explicit declaration of the variable as a parameter in that procedure. The variable can also be declared by a DECLARE statement immediately contained in the same procedure block.

A parameter can be declared with any consistent set of the following attributes: ALIGNED, AREA, BINARY, BIT, CHARACTER, DECIMAL, dimension, ENTRY, FILE, FIXED, FLOAT, INTERNAL, LABEL, LIKE, OFFSET, PICTURE, POINTER, precision, REAL, RETURNS, structure, UNALIGNED, and VARYING. FORMAT is not permitted.

Arguments passed to the procedure are associated with corresponding parameters of the entry point. The first argument is passed to the first parameter, the second argument to the second parameter, and so on. The various entry points of a procedure can have identical or different parameters. When a procedure is invoked, only those parameters associated with the invoked entry point should be referenced during that activation. Refer to section 6, References, for additional information.

Procedure Invocation

An entry point with the RETURNS option is a function entry; one without the RETURNS option is a subroutine entry. A procedure can have both function and subroutine entries.

A procedure invoked as a function must be invoked at a function entry by a function reference. The procedure activation must be terminated by a RETURN statement with a return-value, a STOP statement, or a GOTO statement.

A procedure invoked as a subroutine must be invoked at a subroutine entry by a CALL statement. The procedure activation must be terminated by a RETURN statement with no return-value, an END statement, a STOP statement, or a GOTO statement.

Returns-Descriptor

The returns-descriptor is a list of attributes. It specifies the attributes of the value that is to be returned to the point of invocation by a RETURN statement with a return-value. When the RETURN statement is executed, the value of the expression is converted to the data type specified by the returns-descriptor of the entry point through which the procedure was activated. The returned value must be a scalar value.

The returns-descriptor can include any consistent set of the following attributes: ALIGNED, AREA, BINARY, BIT, CHARACTER, DECIMAL, FIXED, FLOAT, OFFSET, PICTURE, POINTER, precision, REAL, UNALIGNED, and VARYING. Dimension, ENTRY, FILE, FORMAT, LABEL, member, and structure are not permitted.

If INRULE processing is specified on the PLI control statement when the procedure is compiled, the default attributes for RETURNS with no explicit returns-descriptor depend upon the initial character of the entry-name. If the character is one of the letters I through N, RETURNS is equivalent to RETURNS(FIXED BINARY(15,0)); otherwise, RETURNS is equivalent to RETURNS(FLOAT DECIMAL(14)).

If standard arithmetic defaults are used when the procedure is compiled, RETURNS with no parentheses or returns-descriptor is equivalent to RETURNS(REAL FIXED BINARY(15,0)).

Statement Processing

When control is transferred to the entry point, the procedure that immediately contains the entry point is activated. Each argument passed to the procedure is associated with the corresponding parameter of the entry point. A generation of storage is allocated for each automatic variable declared in the procedure; if the variable has the INITIAL attribute, it is assigned an initial value. Each defined variable declared in the procedure is associated with a generation of storage. Control then passes to the statement following the ENTRY statement.

An ENTRY statement has no effect when it is encountered in the sequential flow of control; control passes to the statement following the ENTRY statement.

A procedure is recursive if its PROCEDURE statement has the RECURSIVE option. A recursive procedure can be invoked at any of its entry points while it is already active. Recursive invocations of a procedure can be made at the same entry point or at different entry points.

Examples of ENTRY statements are shown in figure 12-32.

```
P: PROC;
     .
     .
     .
E: ENTRY(PM1,PM2,PM3) RETURNS(CHAR(10));
     .
     .
     .
END P;
```

E is a function entry point. It has three parameters and, therefore, requires three arguments. The result of an invocation of E is a character string value.

```
Q: PROC(B);
     .
     .
     .
G: ENTRY(A,B);
     .
     .
     .
END Q;
```

G is a subroutine entry point. It has two parameters and, therefore, requires two arguments.

Figure 12-32. Sample ENTRY Statements

Conditions

The STORAGE condition is raised during block activation if storage cannot be obtained for generations of storage associated with the block activation. If the STORAGE on-unit frees sufficient space, the allocation is made and execution continues normally. If the on-unit does not free sufficient space, the ERROR condition is raised.

Computational conditions can be raised during evaluation of extent expressions for parameters and for automatic and defined variables.

FORMAT STATEMENT

The FORMAT statement is used during edit-directed I/O to control the format of the data being transmitted. The statement is designated for use during a GET or PUT statement execution when the format-name is referenced by a remote format item R. The syntax of the FORMAT statement is shown in figure 12-33.

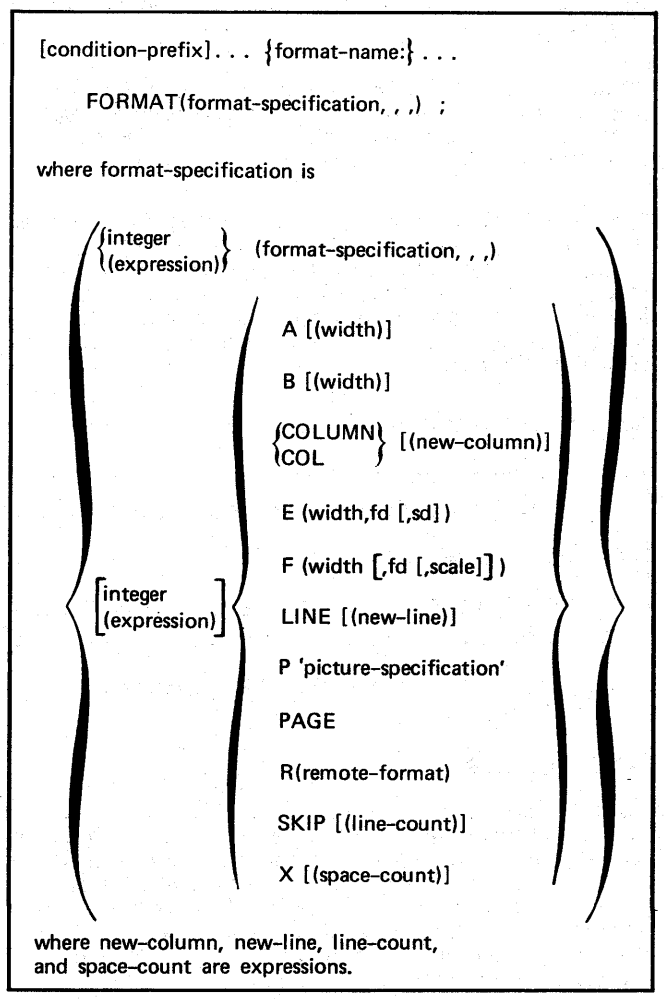


Figure 12-33. FORMAT Statement Syntax

The following rules apply:

- At least one format-name must be included; the format-name must not be subscripted.
- Every expression must yield a scalar computational (arithmetic, string, or pictured) value.
- The FORMAT statement must be immediately contained in the same block as the GET or PUT statement being executed.
- When a character or a bit format item (A or B) is used during GET statement execution, the item must have width specified.
- A GET or PUT statement with a STRING option cannot use a LINE, SKIP, COLUMN, or PAGE format item.
- A PAGE or LINE format item can only be used for a PUT statement that references a print file.

Each format-name is explicitly declared by the compiler as a format constant in the block that immediately contains the FORMAT statement.

A FORMAT statement is bypassed in the normal sequential flow of the program. The statement is executed only when the format-name is referenced by a remote format item during execution of a GET or PUT statement. Control cannot be transferred to a FORMAT statement by a GOTO statement.

When control passes to a FORMAT statement during edit-directed I/O, execution of the format-specification list begins with the leftmost format item as described under Edit-Directed I/O in section 8, Input/Output.

The integer or expression that precedes the list of format items is an iteration factor, which is the number of times a format item or an entire parenthesized list of items is to be used in succession. When an iteration factor is included, it is converted to integer; its value must be positive or zero. If the iteration factor is zero, the format item or parenthesized list of items is ignored.

Examples of FORMAT statements are shown in figure 12-34.

```
FM1: FORMAT(SKIP(1),F(5,2), COL(20),A(8));

FM2: FORMAT (PAGE, 2 A(12));

FM3: FORMAT(3(2(B(3),X(2)),E(12,3)),R(FM2));
```

Figure 12-34. Sample FORMAT Statements

FREE STATEMENT

The FREE statement frees a generation of storage belonging to a controlled or based variable. Only generations allocated by an ALLOCATE statement can be freed by a FREE statement. Once a generation is freed, the storage previously occupied by the generation is immediately available for other uses. Freeing an area implicitly frees all based generations allocated within the area. The syntax of the FREE statement is shown in figure 12-35.

```
[prefix] . . . FREE

    { controlled-variable                                    }
    { [locator-reference->] based-variable [IN(area)]        } ,,, ;
```

Figure 12-35. FREE Statement Syntax

The following rules apply:

- A condition prefix applies to execution of the statement, including the evaluation of dimension or other attributes of a variable being freed.
- The controlled-variable can be a scalar, an array, or a structure; it cannot be an array element or a structure member. The variable must be explicitly declared with the CONTROLLED attribute. It can have either the INTERNAL or EXTERNAL attribute.
- The based-variable can be a scalar, an array, or a structure; it cannot be an array element or a structure member. The variable must be explicitly declared with the BASED attribute.
- The locator-reference must be a scalar locator (pointer or offset) variable or function. If it is a variable, it can be a scalar, an array element, or a structure member; it cannot be an array.
- The area variable specified by the IN option must be scalar. The area must have a generation of storage.

Freeing a Controlled Variable

When a controlled variable is freed, the most recently allocated generation is removed from the stack of generations currently associated with the variable. The dynamic predecessor becomes the top generation in the stack. If the controlled variable has no generations associated with it, the freeing is ignored.

Freeing a Based Variable

When a based variable is freed, the allocated generation identified by a locator value is released. The locator value can be specified either by an explicit locator-reference or by an assumed locator. The based variable used to free the generation must have the same attributes as the one used to allocate it as described under Based Storage in section 3, Data Elements.

It is an error if there is no allocated generation. The allocated generation must have been allocated for a based variable.

Locator-Reference

The based reference can be locator-qualified. The locator-reference specifies the locator (pointer or offset) value that identifies the generation to be freed. If the locator specified by the locator-reference is an offset value, the freeing is performed within an area.

If a based variable is freed without a locator-reference, an assumed locator is constructed from the declaration of the based variable.

Locator-reference processing is summarized in table 12-7.

TABLE 12-7. LOCATOR-REFERENCE PROCESSING

Based Variable Declaration	Locator-Reference Specified	Locator-Reference Not Specified
BASED(locator-variable)	Specified locator-reference is used; declared locator is ignored.	Declared locator is used.
BASED(locator-valued-function)	Specified locator-reference is used; declared locator is ignored.	Declared locator is used.
BASED	Specified locator-reference is used.	Illegal

IN Option

The IN option specifies the area that contains the generation of storage to be freed.

If the specified locator-reference or the assumed locator is an offset value, the freeing is performed within an area.

If the specified locator-reference or the assumed locator is an offset value and the based variable is freed without an IN option, an assumed IN option is constructed from the declaration of the offset variable; in this case the declaration must contain OFFSET(area), where area is a scalar area variable that has a generation of storage. The based generation must be immediately contained in the specified area.

If the specified locator-reference or the assumed locator is a pointer value, the freeing is performed within an area only if the allocation contains an explicit IN option.

IN option processing is summarized in table 12-8.

TABLE 12-8. IN OPTION PROCESSING

Locator-Variable Declaration	IN Option Specified	IN Option Not Specified
OFFSET(area-variable)	IN option area generation is used.	Declared area variable is used.
OFFSET	IN option area generation is used.	Illegal.
POINTER	IN option area generation is used.	The based variable is freed; the generation being freed must not be contained in an area.

Statement Processing

Execution of the FREE statement causes a generation of storage to be freed for each named variable. The variables to be freed are processed in left-to-right order, and each variable is processed through the steps listed in table 12-9.

Examples of FREE statements are shown in figure 12-36.

GET STATEMENT

The GET statement is a stream I/O statement that performs the following operations:

The statement can derive values from a continuous stream of characters, called the source stream, located in PL/I storage or on a CRM file and assign those values to one or more target variables.

The statement can position a stream input file forward before transmitting data, between data transmissions, or without transmitting data.

The statement can copy the input data without alteration to a stream output file.

The syntax of the GET statement is shown in figure 12-37.

TABLE 12-9. FREE STATEMENT PROCESSING

Step	Performed For	Action Taken
1.	A controlled variable.	The latest generation of the controlled variable, which is at the top of the stack, is freed. The dynamic predecessor becomes the top generation in the stack. If the variable has no generations of storage, no action is taken.
2.	A based variable.	The size of the generation to be freed is determined from attributes (aggregate type, data types, and alignment) in the declaration of the based variable. Extents of arrays, strings, and areas are evaluated at this time. The size of the generation must be identical to that used when the variable was originally allocated.
3.	A based variable with a specified or assumed IN option.	The based generation is located and freed. The generation must be immediately contained in the specified area generation. The area variable must have a generation of storage associated with it. If this is the only generation contained in the area, the area is set to EMPTY( ).
4.	A based variable with no specified or assumed IN option.	The based generation is located and freed. The generation must not be contained in an area generation.

```
FREE A,B;

FREE P->B IN(X);


DCL A AREA(1000) BASED(P);
DCL B AREA(300) BASED(Q);
ALLOCATE A, B IN(A);
FREE A; /* FREES BOTH A AND B */


DCL A(5) AREA(300);
DCL B(10) CHAR(20) BASED;
DCL OFF3 OFFSET(A(3));
ALLOC B IN(A(3)) SET(OFF3);
FREE OFF3->B IN A(3));
```

Figure 12-36. Sample FREE Statements

File Positioning

```
[prefix]. . .  GET [COPY [(file-reference)]]  [FILE(file-reference)] SKIP [(line-count)]  ;
```

Stream Input

```
[prefix]. . . GET [ COPY [(file-reference)]] [ [FILE(file-reference)]  [SKIP [(line-count)]] ]
                                             [ STRING(input-source)                          ]

{ LIST(input-target , , ,)                                                                  }
{ EDIT {(input-target , , ,) (format-specification , , ,)}. . .                             } ;
```

where input-target is

```
{ target-variable                    }
{ pseudovariable                     }
{ (input-target , , ,  embedded-do)  }
```

where embedded-do is

```
DO index = { start-expression [ [TO expression] ]  [WHILE(expression)] } , , ,
                               [ [BY expression] ]
```

where format-specification is

```
{ {integer}  (format-specification , , ,)                                                }
{ {(expression)}                                                                         }
{                                                                                        }
{ [integer]    { A (width)                                                               }
{ [(expression)] { B (width)                                                             }
{              { {COLUMN} [(new-column)]                                                 }
{              { {COL   }                                                                }
{              { E (width,fd [,sd])                                                      }
{              { F (width [,fd [,scale]])                                                }
{              { P 'picture-specification'                                               }
{              { R(remote-format)                                                        }
{              { SKIP [(line-count)]                                                     }
{              { X [(space-count)]                                                       }
```

where new-column, line-count, new-line, and space-count are expressions.

Figure 12-37. GET Statement Syntax

The following rules apply:

- The options COPY, FILE, STRING, SKIP, LIST, and EDIT can appear in any order. The order of option keywords has no significance; the order of items within an option is significant.
- The file-reference specified in the COPY option must be a file constant or a file variable.
- The file-reference specified in the FILE option must be a file constant or a file variable.
- The STRING option input-source can be an expression. It must be scalar and computational.
- The SKIP option line-count can be an expression. It must be scalar and computational.
- Each target-variable must be a scalar computational variable.

LIST and EDIT Options

The LIST or EDIT option specifies the mode of stream input transmission. Each LIST or EDIT option contains a list of input-targets to which values are to be assigned. Each EDIT option includes format instructions for deriving source values from the source stream. A GET statement can contain one LIST option or one EDIT option, but not both.

The source stream for list-directed input is interpreted as a list of source values. The source stream for edit-directed input is interpreted according to the format-specification list and any FORMAT statements referenced by that list.

Input-targets are processed from left to right. If the EDIT option contains multiple input-target lists, the lists are processed from left to right until values have been assigned to all input-targets; control then passes to the next statement.

Each input-target specifies one or more scalar items to which a value is assigned. A target-variable can be an array element, but not an unsubscripted array name. It cannot have any asterisk subscripts. It can be a structure member, but cannot be a structure.

An input-target list can specify iteration by means of an embedded-do. The following rules apply:

- Parentheses enclose one or more variables and the embedded-do.
- Each level of embedded-do requires its own set of parentheses.
- The LIST or EDIT option has its own required parentheses, which are distinct from any parentheses required for the embedded-do.
- No comma appears between the last (or only) input-target in the list and the embedded-do.

Detailed information concerning list-directed input and edit-directed input is included in section 8, Input/Output.

FILE and STRING Options

The FILE or STRING option specifies the location of the source stream. The source stream can be located on a CRM file (FILE option) or can be a PL/I generation (STRING option).

A GET statement can contain one FILE option or one STRING option, but not both. If the GET statement contains neither a FILE option nor a STRING option, FILE(SYSIN) is assumed.

FILE Option

The FILE option specifies that the source stream is located on a file. If the referenced file is open, it must be a STREAM INPUT file. If the file is not open, an attempt will be made to open it implicitly with the attributes STREAM and INPUT.

STRING Option

The STRING option specifies that the input-source represents the source stream. The expression is evaluated and converted to character before data transmission begins. The evaluated and converted expression becomes the source stream. Each time the GET statement is executed, the initial position of the stream is the first character of the evaluated result. If the expression is simply a variable reference, it must not reference the same generation of storage as any input-target.

SKIP Option

The SKIP option specifies that the CRM file is to be positioned forward by some number of lines before data transmission begins, or is to be simply positioned forward with no subsequent data transmission. Skipping is performed before data transmission, even if the SKIP option appears to the right of the LIST or EDIT option.

The expression is evaluated and converted to a fixed binary integer; its value must be greater than zero. If the SKIP option does not include an expression to indicate the actual number of lines, SKIP(1) is assumed. The initial position of a file immediately after it is opened is the first character of the first line.

Skipped data is copied if the COPY option is included.

COPY Option

The COPY option specifies that all source data skipped or transmitted is also to be copied to an output file. If the referenced file is open, it must be a STREAM OUTPUT or a STREAM OUTPUT PRINT file. If the file is not open, an attempt will be made to open it implicitly with the attributes STREAM and OUTPUT.

If the COPY option does not include a file-reference, COPY(SYSPRINT) is assumed.

Statement Processing

Execution of the GET statement is performed in the order shown in table 12-10. This information is important for the following reasons:

- If a condition is raised during execution of the GET statement, not all of the operations will have been performed.

- If an on-unit does not return to the point at which the condition was raised, not all of the operations will have been performed.
- If any option contains a function reference, the effect of the GET statement can depend on the order in which operations are performed.

Examples of GET statements are shown in figure 12-38.

TABLE 12-10. GET STATEMENT PROCESSING

Step	GET Statement Option	Action
1.	STRING	The expression is evaluated and converted to CHARACTER(n) nonvarying, where n is the number of characters in the evaluated string.
2.	SKIP	The expression is evaluated and converted to a fixed binary integer.
3.	FILE or assumed file option	If the file is not open, it is opened implicitly with attributes STREAM and INPUT.
4.	COPY	If the file is not open, it is opened implicitly with attributes STREAM and OUTPUT.
5.	SKIP	The source stream is moved forward the number of lines specified and positioned at column 1. The bypassed data is copied to the copy-file if the COPY option is included.
6.	LIST or EDIT	Data is transmitted as described in section 8, Input/Output. When values have been assigned to all input-targets, the operation is complete and control passes to the next statement.

Conditions

Conditions that can be raised during GET statement execution include: UNDEFINEDFILE, ENDFILE, ENDPAGE, ERROR, and TRANSMIT.

The UNDEFINEDFILE condition is raised when an implicit file opening fails. If control returns from the UNDEFINEDFILE on-unit and the file is still not open, the ERROR condition is raised.

The ENDFILE condition is raised when end-of-file is reached while processing the SKIP option or an input-target. If control returns from the ENDFILE on-unit, execution continues with the statement following the GET statement.

The ENDPAGE condition is raised only for a print output file referenced by the COPY option when the GET statement causes page size to be exceeded. If control returns from the ENDPAGE on-unit, statement processing continues.

The ERROR condition is raised when the string specified by the STRING option does not contain enough characters to satisfy all the input-targets.

The TRANSMIT condition is raised if parity errors are encountered. If control returns from the TRANSMIT on-unit, execution continues with the statement following the GET statement.

Computational conditions can be raised during expression evaluation, conversion, and assignment.

GOTO STATEMENT

The GOTO statement transfers control to the statement identified by label-reference. If the label-reference is a label variable, the GOTO statement can transfer control to a different statement each time it is executed. The syntax of the GOTO statement is shown in figure 12-39.

The following rules apply:

- The label-reference must be a label constant or a label variable. It cannot be a function reference.

```
GET LIST(A,B);
```

The statement specifies list-directed input of two values from the CRM file (normally INPUT) associated with the file SYSIN. The values are to be assigned to the variables A and B.

```
GET FILE(INFILE) SKIP EDIT(P,Q,R)(3(F(6,2),X(4)));
```

The statement specifies edit-directed input from the CRM file associated with INFILE. The values are assigned to P, Q, and R. The total number of characters read from INFILE is 26. The X format item is not executed after the last input-target is executed.

```
GET STRING(S) COPY EDIT (SUBSTR(x,3),Y)  (A(10), X(3),P↑AA999↑);
```

The statement specifies edit-directed input from the character string value of S. Values are assigned to X, using the SUBSTR pseudovariable, and to Y. The length of S must be at least 18, the total length specified by the three format items. A copy of the first 18 characters of S is written on file SYSPRINT.

```
GET FILE(SYSIN) SKIP LIST((A(I) DO I=1 TO J));
```

The statement specifies list-directed input of J values to be assigned to A(1), . . . ,A(J). Note the double parentheses.

Figure 12-38. Sample GET Statements

- The label-reference cannot identify an ENTRY, FORMAT, or PROCEDURE statement; the prefixes of these statements are not label constants.
- The block activation that contains the statement identified by label-reference must be active when the GOTO statement is executed; that is, control can only be transferred to a statement in the current block activation or in a dynamic predecessor of the current block activation.
- The statement must not be used to transfer control into a DO WHILE or an indexed DO from outside the do group.

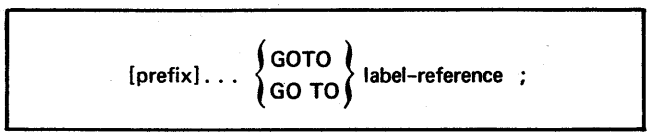


Figure 12-39. GOTO Statement Syntax

The label value of the label-reference identifies not only a statement, but a particular activation of the block that contains the statement.

If the label value designates a statement in the current block activation, control is transferred to the statement it identifies. This process is called a local GOTO.

If the label value designates a statement contained in a dynamic predecessor of the current block activation, the current block activation is terminated abnormally as are all of its predecessors up to, but not including, the activation containing the referenced statement. Control is then transferred to the designated statement in that block activation. It could be another activation of the block containing the GOTO statement. This process is called a nonlocal GOTO.

Examples of a local GOTO and nonlocal GOTO are illustrated in figure 12-40. The following points should be considered:

- If a GOTO statement contained in a begin block transfers control to the BEGIN statement that heads the block, the transfer is a nonlocal GOTO.
- If a GOTO statement in one block transfers control to a statement in a containing block, the transfer is a nonlocal GOTO.
- A nonlocal GOTO with a label-reference that is a label variable can transfer control to a statement in any active block.
- A nonlocal GOTO must not cause the termination of any block activation that has an immediate dynamic predecessor currently performing any of the following operations:

  Evaluation of extents of automatic or defined variables.

  Allocation of storage for automatic variables.

  Initialization of automatic variables.

  Execution of an ALLOCATE statement.

  Allocation of the based-variable in a LOCATE statement.

Examples of GOTO statements are shown in figure 12-41.

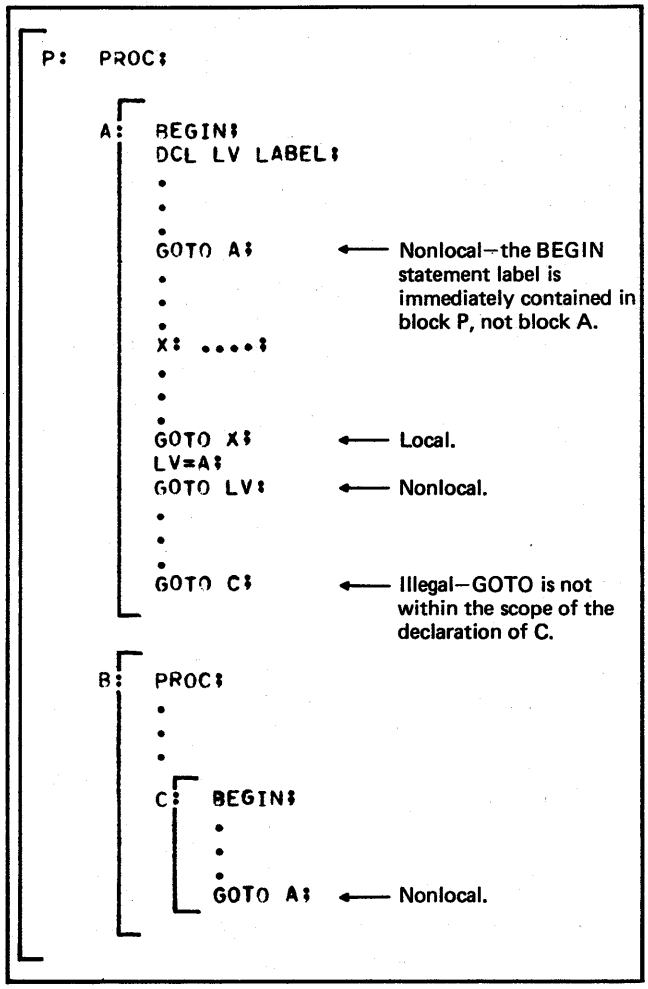


Figure 12-40. Local/Nonlocal GOTO

IF STATEMENT

The IF statement allows the flow of control to take one of two alternative paths, depending on the value of a specified expression. The syntax of the IF statement is shown in figure 12-42.

The following rules apply:

- A condition prefix applies only to the evaluation and conversion of the expression; the condition prefix does not apply to any statements in the executable-units.
- The expression must be scalar and computational.
- Statements in the executable-units can have prefixes.
- If the ELSE clause is included, executable-unit-1 cannot be an IF statement with no ELSE clause.

When the IF statement is executed, the expression is evaluated, converted to a bit string, and tested. Comparison operators yield '1'B if the comparison is true and '0'B if the comparison is false. Testing proceeds as follows:

If the string contains any one-bits ('1'B), control passes to executable-unit-1.

```
    IF X<Y THEN GO TO L;
    .
    .
    .
L:  X = Z - 1;
    .
    .
    .


P:  PROC(X);
    DCL X FIXED;      /*PARAMETER*/
    DCL LV(3) LABEL INITIAL(L1,L2,ERR);
    I = MIN(X,3);
    .
    .
    .
    GOTO LV(I);
L1:
    .
    .
    .
    RETURN;
L2:
    .
    .
    .
    RETURN;
ERR:
    .
    .
    .
    END P;
```

Figure 12-41. Sample GOTO Statements

[prefix] . . . IF expression

THEN executable-unit-1

[ELSE executable-unit-2]

where each executable-unit is

- begin-block
- do-group
- ALLOCATE-statement
- assignment-statement
- CALL-statement
- CLOSE-statement
- DELETE-statement
- FREE-statement
- GET-statement
- GOTO-statement
- IF-statement
- LOCATE-statement
- null-statement
- ON-statement
- OPEN-statement
- PUT-statement
- READ-statement
- RETURN-statement
- REVERT-statement
- REWRITE-statement
- SIGNAL-statement
- STOP-statement
- WRITE-statement

Figure 12-42. IF Statement Syntax

If the string does not contain any one-bits ('1'B) and the IF statement contains an ELSE clause, control passes to executable-unit-2.

If the string does not contain any one-bits ('1'B) and the IF statement does not contain an ELSE clause, control passes to the statement following the IF statement.

If control reaches the end of an executable-unit (that is, control is not transferred out of the executable-unit by a GOTO or RETURN statement), control passes to the statement following the IF statement.

Examples of IF statements are shown in figure 12-43.

```
IF A=B
   THEN IF P>=Q
           THEN CALL APROC;
           ELSE RETURN;
```

The ELSE clause in the above statement is paired with the THEN clause of the inner IF statement. If it is intended for the outer IF statement, the statement must be balanced by using an ELSE clause with a null statement as its executable unit. For example:

```
IF A=B
   THEN IF P<=Q
           THEN CALL APROC;
           ELSE;
   ELSE RETURN;


IF A THEN GO TO THEEND;

IF A<B
   THEN IF C>0
           THEN IF X¬=Y THEN RETURN(X);

IF B1=(B2|B3)
   THEN E: CALL EQ(B1);
   ELSE N: CALL NE(B1,B2,B3);
```

Figure 12-43. Sample IF Statements

LOCATE STATEMENT

The LOCATE statement is a record I/O statement that allocates a based variable for use as an output buffer. The statement can also transmit a record to a CRM file. The syntax of the LOCATE statement is shown in figure 12-44.

[prefix]. . . LOCATE based-variable

FILE(file-reference) [SET(pointer) | KEYFROM(expression)] ;

Figure 12-44. LOCATE Statement Syntax

The following rules apply:

- A condition prefix applies to execution of the statement, including the evaluation of dimension, INITIAL, or other attributes of the based variable being allocated.

- The based-variable must be unsubscripted and must not have the member attribute.
- The FILE, SET, and KEYFROM options can appear in any order after the based-variable reference.
- The file-reference must be a file constant or a file variable.
- Either the SET option must be present, or the declaration of the based variable must have a BASED attribute with a locator variable. If the SET option is not specified, the declared locator variable is used as an assumed SET option.
- The specified or assumed SET option must reference a scalar pointer variable.
- The specified or assumed SET option must not reference an offset variable.
- The KEYFROM option must be included for a file that has the KEYED attribute.
- The KEYFROM expression must be scalar and computational.

Execution of the LOCATE statement causes a record buffer to be allocated. The record buffer is subsequently transmitted as a record to the CRM file identified by file-reference. Transmission occurs when the next I/O statement for the file is executed or when the file is closed. The program can assign values to the based generation after buffer allocation and before transmission.

Any open record output file can have one allocated buffer associated with it. Any subsequent I/O statement that can be executed for such a file (LOCATE, WRITE, or CLOSE) causes the allocated buffer to be written as a record and then freed. The pointer value that addresses the buffer is invalid after the buffer is freed by the system.

Statement Processing

The LOCATE statement is processed through the following steps in the order given:

1. The specified or assumed SET option determines the pointer variable to be used. The value of the pointer variable is set to the start of the allocated buffer. If the KEYFROM option is present, its expression is evaluated and converted to a character string if necessary. This option specifies the key value that is used to identify the record when it is written. The value is maintained by the system until the allocated buffer is written by a subsequent I/O statement execution.
2. If the file is open, it must have the attributes RECORD and OUTPUT. If the file is not open, an implicit file opening is executed. Implied attributes RECORD and OUTPUT are supplied for the file opening.
3. If an allocated buffer created by a previous LOCATE statement exists, the buffer is written as a record to the CRM file. If the output file is direct, the key value specified in the KEYFROM option is associated with the record. If the output file is sequential, the record is written at the position immediately following the one indicated by the current-record designator. The current-record designator is changed to identify the new record. The buffer is then freed.
4. The based variable is allocated and initialized normally as if by ALLOCATE statement execution. A pointer value is assigned to the pointer variable identified by the specified or assumed SET option. A nonlocal GOTO must not cause the termination of any block activation that has an immediate dynamic predecessor currently performing this operation.
5. The newly allocated generation is established as an allocated buffer associated with the file. If the LOCATE statement has a KEYFROM option, the evaluated key value is associated with the allocated buffer.

Examples of LOCATE statements are shown in figure 12-45.

```
DCL B(5,5) BASED(P) CHAR(100);
.
.
.
LOCATE B FILE(F);
.
.
.
P->B(I,J) = X;
.
.
.
LOCATE B SET(Q) FILE(F);
.
.
.
Q->B(3,I) = Y;
.
.
.
CLOSE FILE(F);
```

Two records are written to the CRM file associated with the file F. The first record contains the value assigned to it from X, and it is written during execution of the second LOCATE statement. The second record contains the value of Y, and it is written during the execution of the CLOSE statement. Both P and Q are pointer variables.

Figure 12-45. Sample LOCATE Statements

Conditions

Conditions that can be raised during LOCATE statement execution include: UNDEFINEDFILE, KEY, RECORD, and TRANSMIT.

The UNDEFINEDFILE condition is raised when an implicit file opening fails. If control returns from the UNDEFINEDFILE on-unit and the file is still not open, the ERROR condition is raised.

The KEY condition is raised during the buffer write operation when the key value duplicates an existing key in the CRM file. If control returns from the KEY on-unit, control passes to the next statement with LOCATE statement execution incomplete.

The RECORD condition is raised during the buffer write operation when the record size is unacceptable for the CRM file. If control returns from the RECORD on-unit, the ERROR condition is raised.

The TRANSMIT condition is raised if parity errors are encountered. If control returns from the TRANSMIT on-unit, execution continues with the statement following the LOCATE statement.

Computational conditions can be raised during expression evaluation, conversion, and assignment.

NULL STATEMENT

The null statement has no effect on the execution of the program. The statement is represented by a semicolon. The syntax of the null statement is shown in figure 12-46.

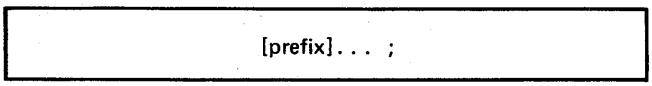


Figure 12-46. Null Statement Syntax

Execution of the null statement passes control to the next statement. A null statement is used principally in an IF or ON statement. An IF statement with the construction ELSE; can be used to balance a nested IF statement. An ON statement with the construction ON condition; establishes a null on-unit, which causes system action to be bypassed for a condition that cannot be disabled.

Examples of null statements are shown in figure 12-47.

```
IF I=J
   THEN IF P<Q
           THEN GOTO OUT;
           ELSE;   /*NULL ACTION*/
   ELSE GOTO CONTINUE;

ON SIZE;              /*NULL ON-UNIT*/

START:;
```

Figure 12-47. Sample Null Statements

ON STATEMENT

The ON statement establishes an on-unit for the named condition. The on-unit specifies the action to be taken when the named condition is raised. The syntax of the ON statement is shown in figure 12-48.

The following rules apply:

- A condition prefix applies to execution of the statement; it does not apply to the on-unit. A condition prefix on the first or only statement of the on-unit applies to the on-unit.
- The condition must be one of the names listed in the table of conditions in section 10 (table 10-1).
- The first or only statement of the on-unit cannot have a label prefix.

The ON statement is executed when it is encountered during the normal sequential flow of control. This establishes the specified on-unit in the current block activation. The on-unit is not executed until the condition is subsequently raised and then only if the condition is enabled.

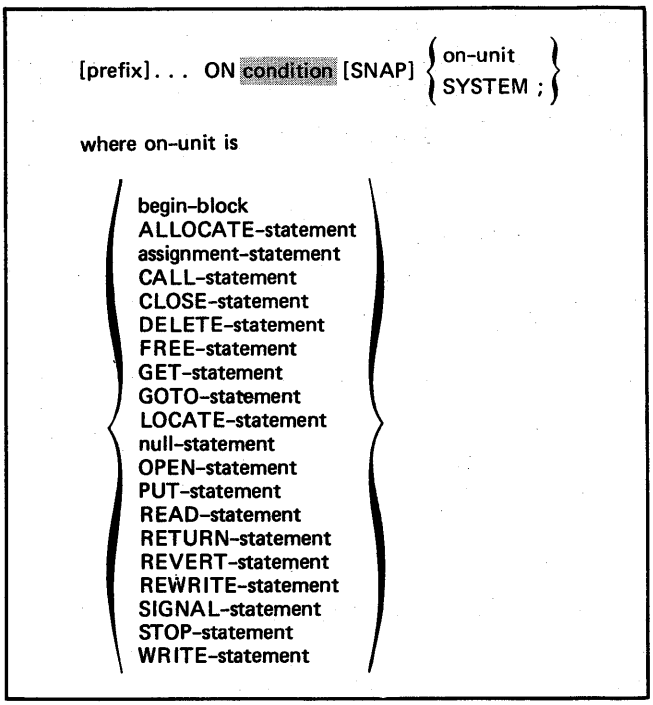


Figure 12-48. ON Statement Syntax

The SNAP option specifies that diagnostic information is to be written to the system error file when the named condition is raised. This occurs immediately before the on-unit is activated, and then only if the condition is enabled.

The SYSTEM option specifies that a system on-unit (standard system action) for the named condition is to be established in the current block activation. A system on-unit is automatically in effect for each condition that has no programmed on-unit.

When the ON statement is executed, the on-unit is placed at the top of the stack and becomes the current established on-unit for the condition. The on-unit established in a block activation for any condition remains in effect until a subsequent ON or REVERT statement for the same condition is executed in the same block activation or until the block activation is terminated.

A null on-unit can be specified to bypass system action or to obtain only SNAP output. For example:

```
ON RECORD(FILE6);
```

Examples of ON statements are shown in figure 12-49.

OPEN STATEMENT

The OPEN statement opens each referenced file for subsequent I/O operations. Any I/O statement other than CLOSE can perform an implicit file opening and supply additional attributes for a file. The OPEN statement performs an explicit file opening and allows additional attributes and options to be specified for the file. The syntax of the OPEN statement is shown in figure 12-50.

```
ON ENDFILE(SYSIN) GOTO FINAL;

ON ENDFILE(INP) BEGIN;
                  .
                  .
                  .
                END;

ON ENDFILE(INP) (SIZE); BEGIN;
                          .
                          .
                          .
                        END;

ON ENDFILE(INPUTF) CLOSE FILE(INPUTF);

ON CONDITION(SEQERR)
   CALL ERRS(ONLOC(), ONCODE(), ^SEQ^);

ON CONVERSION BEGIN;
                .
                .
                .
                ONSOURCE() = ...;
              END;
```

Figure 12-49. Sample ON Statements

The following rules apply:

- The options specified for a file (including the FILE option that names the file) can appear in any order.
- Each file-reference must be a file constant or a file variable.
- The TITLE option expression must be scalar and computational.
- The PAGESIZE option can only be used for a STREAM OUTPUT PRINT file.
- A LINESIZE or PAGESIZE expression must be scalar and computational.
- Any specified file description options must be consistent with attributes declared for the file constant.

TITLE Option

The TITLE option specifies the name of the CRM file that is to be associated with the file being opened. The CRM file name, and not the PL/I file name, is the one known to CYBER Record Manager and to the operating system. The name can only contain letters A through Z and digits 0 through 9, and the name must begin with a letter. If the option is not included, the name of the PL/I file constant is used. If the name is longer than seven characters, the local file name is formed by concatenating the first four and last three characters of the PL/I name.

ENVIRONMENT Option

The ENVIRONMENT option specifies CRM file processing options. Options represented by the ENVIRONMENT option expression augment the CYBER Record Manager file information table (FIT) for the file and override values already established by DECLARE statement. All constant values must be decimal; that is, no B (bit) or W (word) suffixes are permitted.

Some of the options permitted on the CYBER Record Manager FILE control statement can be included. Refer to Environment Processing and Defaults in section 9, CYBER Record Manager Interface, for additional details.

ENVIRONMENT options remain in effect until the file is closed. A FILE control statement overrides ENVIRONMENT options, and ENVIRONMENT options override values specified with the ENVIRONMENT attribute in a DECLARE statement. Embedded blanks are allowed in the ENVIRONMENT expression.

Stream Input Option

The LINESIZE option specifies the maximum number of characters per line of user data. If the option is not included, system default is 80 characters.

Stream Output Options

The PRINT option indicates that the file can be used to write data formatted for printing.

The LINESIZE option specifies the maximum number of characters per line of user data. If the option is not included, system default is 80 characters for STREAM OUTPUT and 136 characters for STREAM OUTPUT PRINT.

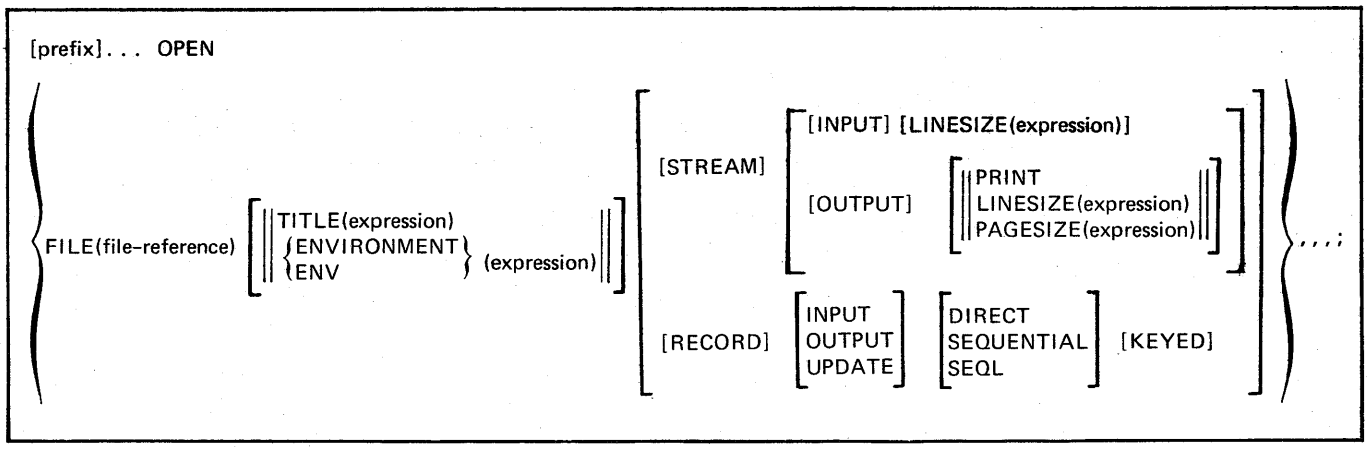


Figure 12-50. OPEN Statement Syntax

The PAGESIZE option defines the maximum number of lines that can be written before the ENDPAGE condition is raised. If the option is not included, system default is 60 lines.

Statement Processing

The files to be opened are processed in left-to-right order. Each file is processed through the following steps:

1. If a TITLE option is specified for the file, its expression is evaluated and converted to a character string. If the character string is longer than seven characters, the local file name is formed by concatenating the first four and the last three characters.

2. If the LINESIZE or PAGESIZE option is present, its expression is evaluated and converted to a fixed binary integer. The value of this integer must be greater than zero. If the ENVIRONMENT option is present, its expression is evaluated and converted to character string.

3. File opening is performed as described in section 8, Input/Output. If the file is already open, processing continues with the next file to be opened.

4. If the file opening fails, the UNDEFINEDFILE condition is raised. If control returns from the UNDEFINEDFILE on-unit, processing continues with the next file to be opened.

Examples of OPEN statements are shown in figure 12-51.

```
OPEN FILE(FA) PRINT;


DCL TRANS RECORD FILE;
DCL MRL FIXED DEC(5,0) INIT(300);
DCL CYCLE CHAR(4) INITIAL('0001');
OPEN FILE(TRANS) TITLE('TR' || CYCLE) KEYED
   UPDATE ENV('FO=IS,RT=W,MRL=' || MRL);
```

Figure 12-51. Sample OPEN Statements

PROCEDURE STATEMENT

The PROCEDURE statement, together with its associated END statement, delineates a sequence of statements called a procedure block. The syntax of a procedure block is shown in figure 12-52. The PROCEDURE statement denotes the primary entry point of the procedure. It can specify parameters for the entry point and the attributes of the value to be returned by the procedure when invoked at this entry point by a function reference. The syntax of the PROCEDURE statement is shown in figure 12-53.

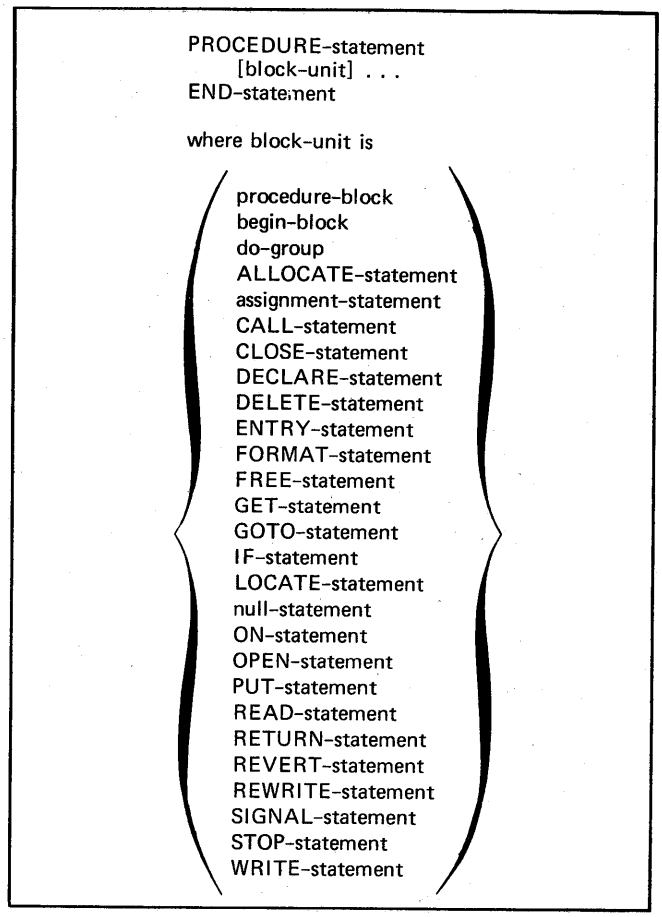


Figure 12-52. Procedure Block Syntax

The following rules apply:

- A condition prefix applies to all statements contained in the procedure block and in its contained blocks. The only exception is when a contained statement or block has a condition prefix enabling or disabling the same condition.

- At least one entry-name must be included. The entry-name must not be subscripted.

- A variable cannot appear more than once in the same parameter list.

- Each parameter can be a scalar, an array, or a structure; it cannot be an array element, an array slice, or a structure member.

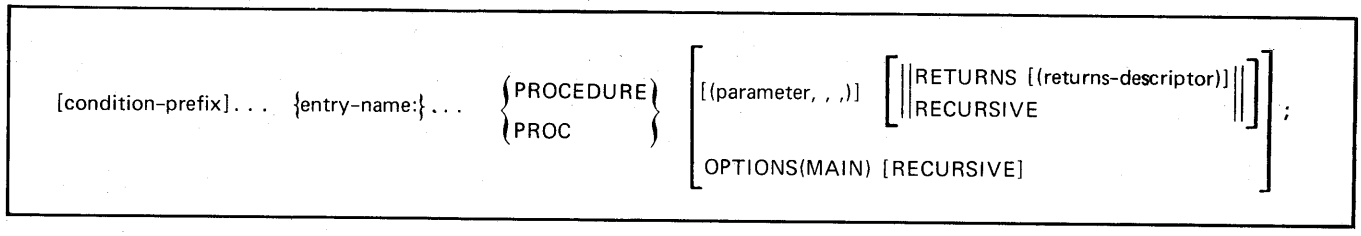
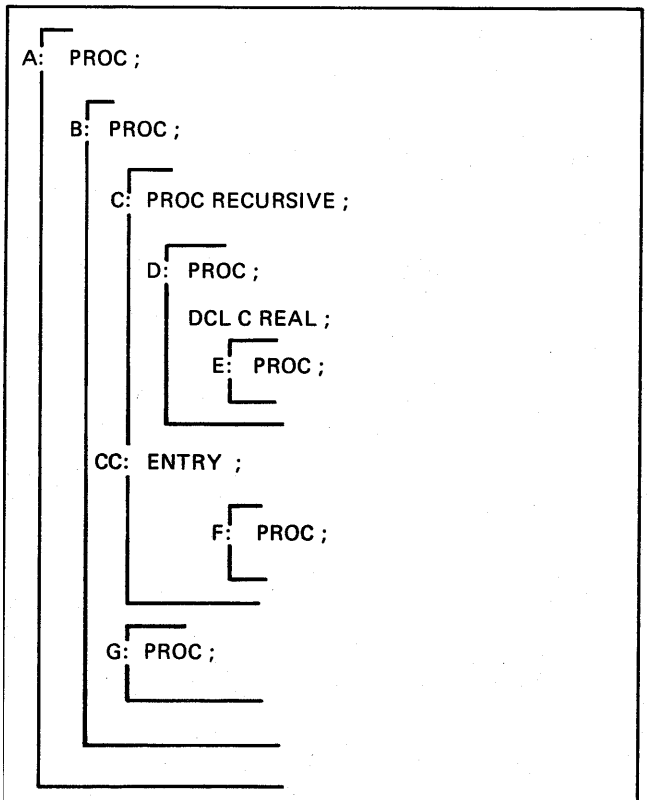


Figure 12-53. PROCEDURE Statement Syntax

- Data types of corresponding parameters and arguments must be compatible. The number of parameters must equal the number of arguments passed when the entry point is invoked.
- If the procedure is invoked by CALL statement execution, the RETURNS option must not be included; if the procedure is invoked by function reference, the RETURNS option must be included.
- RETURNS( ) is illegal.
- Each extent (string length or area size) appearing in a returns-descriptor must be an expression consisting only of literal constants and the operators + - * / and ||. Array bounds cannot appear in a returns-descriptor.
- OPTIONS(MAIN) is permitted only on an external procedure.
- RECURSIVE can precede or follow OPTIONS(MAIN).

Entry-Name

Each entry-name of an internal procedure is explicitly declared by the compiler as an internal entry constant in the block that immediately contains the procedure. An internal entry-name cannot be declared by DECLARE statement. The entry-name is known in the block that immediately contains the procedure and in all blocks contained in that block. The scope of internal procedure entry-names is illustrated in figure 12-54.



The statement CALL C; can be immediately contained in block B, C, F, or G. The statement CALL CC; can be immediately contained in block B, C, D, E, F, or G.

Figure 12-54. Scope of Internal Procedure Entry-Names

Each entry-name of an external procedure is explicitly declared by the compiler as an external entry constant outside of any block. Unless there is an overriding declaration, the entry-name is known in the external procedure and in all blocks contained in that external procedure.

A reference to an external entry constant from a point outside the external procedure that contains the PROCEDURE statement requires a DECLARE statement declaration as described under the ENTRY attribute in section 4, Attributes.

Each external name that is longer than seven characters is modified before external references are resolved by the loader. Only the first four and last three characters of the original identifier are used. Care should be taken to avoid accidental duplication of external names that are modified according to this convention. External names should not include any of the characters $ @ # and _; external names containing these characters might conflict with names used by the system.

An entry-name on a PROCEDURE statement can be used as the closure-name on the END statement associated with the procedure block. The entry-name can be referenced in a CALL statement or function reference that invokes the procedure, and in an argument list. It cannot be referenced in any other context.

A PROCEDURE statement with multiple entry-names creates one entry point that can be referenced by any of its names. For example:

```
ENTER:LOOP: PROC(P1,P2) RETURNS(BIT(1));
```

All entry-names of a single PROCEDURE statement are equivalent with the following exception:

If INRULE processing is specified on the PLI control statement when the procedure is compiled, the first character of an entry-name can determine the default attributes supplied for a returns-descriptor; therefore, a single entry point with multiple names can have two different returns descriptors.

If a PROCEDURE statement is immediately followed by an ENTRY statement (even with no intervening executable statements), two separate entry points are created. The entry points can have different parameters and RETURNS options. For example:

```
ENTER: PROC(P1,P2) RETURNS(BIT(1));

LOOP: ENTRY(P3,P2) RETURNS(FLOAT);
```

Parameters and Arguments

Each parameter in a PROCEDURE statement is a variable with which an argument will be associated when the procedure is invoked at that entry point. The appearance of a variable in the parameter list constitutes an explicit declaration of the variable as a parameter in that procedure. The variable can also be declared by a DECLARE statement immediately contained in the same procedure block.

A parameter can be declared with any consistent set of the following attributes: ALIGNED, AREA, BINARY, BIT, CHARACTER, DECIMAL, dimension, ENTRY, FILE, FIXED, FLOAT, INTERNAL, LABEL, LIKE, OFFSET, PICTURE, POINTER, precision, REAL, RETURNS, structure, UNALIGNED, and VARYING. FORMAT is not permitted.

Arguments passed to the procedure are associated with corresponding parameters of the entry point. The first argument is passed to the first parameter, the second argument to the second parameter, and so forth. The various entry points of a procedure can have identical or different parameters. When a procedure is invoked, only those parameters associated with the invoked entry point should be referenced during that activation. Refer to section 6, References, for additional information.

Procedure Invocation

An entry point with the RETURNS option is a function entry; one without the RETURNS option is a subroutine entry. A procedure can have both function and subroutine entries.

A procedure invoked as a function must be invoked at a function entry by a function reference. The procedure activation must be terminated by a RETURN statement with a return-value, a STOP statement, or a GOTO statement.

A procedure invoked as a subroutine must be invoked at a subroutine entry by a CALL statement. The procedure activation must be terminated by a RETURN statement with no return-value, an END statement, a STOP statement, or a GOTO statement.

Returns-Descriptor

The returns-descriptor is a list of attributes. It specifies the attributes of the value that is to be returned to the point of invocation by a RETURN statement with a return-value. When the RETURN statement is executed, the value of the expression is converted to the data type specified by the returns-descriptor of the entry point through which the procedure was activated. The returned value must be a scalar value.

The returns-descriptor can include any consistent set of the following attributes: ALIGNED, AREA, BINARY, BIT, CHARACTER, DECIMAL, FIXED, FLOAT, OFFSET, PICTURE, POINTER, precision, REAL, UNALIGNED, and VARYING. Dimension, ENTRY, FILE, FORMAT, LABEL, member, and structure are not permitted.

If INRULE processing is specified on the PLI control statement when the procedure is compiled, the default attributes for RETURNS with no explicit returns-descriptor depend upon the initial character of the entry-name. If the character is one of the letters I through N, RETURNS is equivalent to RETURNS(FIXED BINARY(15,0)); otherwise, RETURNS is equivalent to RETURNS(FLOAT DECIMAL(14)).

If standard arithmetic defaults are used when the procedure is compiled, RETURNS with no parentheses or returns-descriptor is equivalent to RETURNS(REAL FIXED BINARY(15,0)).

RECURSIVE Option

The RECURSIVE option specifies that the procedure can be activated recursively; that is, the procedure can be activated while it is already active. A procedure can be invoked recursively at the same entry point or at different entry points.

OPTIONS(MAIN)

OPTIONS (MAIN) specifies that the external procedure is a main procedure. A PL/I program must have exactly one main procedure. When program execution is initiated by an operating system control statement, the main procedure is activated at its primary entry point.

OPTIONS(MAIN) is permitted only in the PROCEDURE statement that heads an external procedure.

Statement Processing

When control is transferred to the entry point, the procedure that immediately contains the entry point is activated. Each argument passed to the procedure is associated with the corresponding parameter of the entry point. A generation of storage is allocated for each automatic variable declared in the procedure; if the variable has the INITIAL attribute, it is assigned an initial value. Each defined variable declared in the procedure is associated with a generation of storage. Control then passes to the statement following the PROCEDURE statement.

A PROCEDURE statement has no effect when it is encountered in the sequential flow of control. The entire procedure is bypassed and control passes to the statement following the END statement of the procedure.

Examples of PROCEDURE statements are shown in figure 12-55.

```
P: PROC OPTIONS(MAIN);
   DCL B ENTRY EXTERNAL;
   .
   .
   .
   X = Y*INNER(A)+1;
   .
   .
   .

INNER: PROCEDURE(PM) RETURNS(FIXED);
       DCL PM FLOAT;
       .
       .
       .
       RETURN(Q);
       END INNER;

   .
   .
   .
   CALL B;
   .
   .
   .
   END P;


B: PROC;
   .
   .
   .
   END B;
```

P and B are external procedures; INNER is an internal procedure. INNER is invoked as a function with a parameter list (PM); the function returns a fixed point value to be used in the evaluation of the expression Y*INNER(A) + 1. B is invoked by the statement CALL B, and it does not return a value.

Figure 12-55. Sample PROCEDURE Statements

Conditions

The STORAGE condition is raised during block activation if storage cannot be obtained for generations of storage associated with the block activation. If the STORAGE on-unit frees sufficient space, the allocation is made and execution continues normally. If the on-unit does not free sufficient space, the ERROR condition is raised.

Computational conditions can be raised during evaluation of extent expressions for parameters and for automatic and defined variables.

PUT STATEMENT

The PUT statement is a stream I/O statement that performs the following operations:

The statement can construct a continuous stream of characters from values located in PL/I generations of storage and transmit that character stream to a PL/I storage generation or to a CRM file.

The statement can position a stream output file forward before transmitting data, between data transmissions, or without transmitting data.

The syntax of the PUT statement is shown in figure 12-56.

The following rules apply:

- The options FILE, STRING, SKIP, LINE, PAGE, LIST, and EDIT can appear in any order. The order of option keywords has no significance; the order of items within an option is significant.
- In the FILE option, file-reference must be a file constant or a file variable.
- In the STRING option, output-target must be a scalar character variable or pseudovariable.
- In the SKIP option, line-count can be an expression. It must be scalar and computational.
- In the LINE option, new-line can be an expression. It must be scalar and computational.
- When the PAGE or LINE option is executed, the file must have the attribute PRINT.

LIST and EDIT Options

The LIST or EDIT option specifies the mode of stream output transmission. Each option contains a list of source expressions with values that are to be transmitted, and includes instructions for constructing a target stream from the source values. A PUT statement can contain one LIST option or one EDIT option, but not both.

The target stream for list-directed output is created as a list of target values. The target stream for edit-directed output is created according to the format-specification list and any FORMAT statements referenced by that list.

Each output-source specifies either a source expression, or one or more source expressions that are under control of an embedded-do. Output-sources are processed from left to right. If the EDIT option contains multiple output-source lists, the lists are processed from left to right until values have been assigned to all output-sources; control then passes to the next statement.

Each output-source expression yields a scalar value to be used in constructing the target stream. The source expression can be an array element, but not an unsubscripted array name. It cannot have asterisk subscripts. It can be a structure member, but cannot be a structure.

An output-source list can specify iteration by means of an embedded-do. The following rules apply:

- Parentheses enclose one or more output-source expressions and the embedded-do.
- Each level of embedded-do requires its own set of parentheses.
- The LIST or EDIT option has its own required parentheses, which are distinct from any parentheses required for the output-source expressions and embedded-do.
- No comma appears between the last (or only) output-source in the list and the embedded-do.

FILE and STRING Options

The FILE or STRING option specifies the location of the target stream. The target stream can be located on a CRM file (FILE option) or can be a PL/I generation (STRING) option.

The target stream is the stream of characters constructed from the source values. If the target stream is located on a CRM file, the PUT statement treats the target stream as a continuous stream of characters with the following structure:

- The target stream has a current position so that transmitted data can be appended to the end of the existing stream.
- The target stream is divided into lines so that it is possible to skip to the beginning of a line.
- Each character has a column position within a line.
- If the CRM file has the PRINT attribute, the target stream is divided into pages so that it is possible to skip to the beginning of a page or to the beginning of a specified line on a page.

A PUT statement can contain one FILE option or one STRING option, but not both. If the PUT statement contains neither a FILE option nor a STRING option, FILE(SYSPRINT) is assumed.

FILE Option

The FILE option specifies that the target stream is located on a file. If the referenced file is open, it must be a STREAM OUTPUT or STREAM OUTPUT PRINT file. If the file is not open, an attempt will be made to open it implicitly with the attributes STREAM and OUTPUT.

STRING Option

The STRING option specifies that the target stream is a scalar character variable or one of the pseudovariables ONCHAR, ONSOURCE, or SUBSTR. Each time the PUT statement is executed, the initial position of the stream is the first character in the character variable or pseudovariable. The character variable or pseudovariable must not reference the same generation of storage as any output-source item.

File Positioning

```
[prefix]. . .  PUT  [FILE(file-reference)] { SKIP [(line-count)]
                                           { LINE [(new-line)]
                                           { PAGE [LINE  [(new-line)]] } ;
```

Stream Output

```
[prefix]. . .  PUT [ [FILE(file-reference)] [ SKIP [(line-count)]
                   [                        [ PAGE] [LINE [(new-line)]] ] ]
                   [STRING(output-target)                                 ]

{ LIST(output-source , , ,)                                                }
{ EDIT{(output-source , , ,) (format-specification , , ,)}. . .            } ;
```

where output-source is

```
{ expression                          }
{ (output-source , , , embedded-do)   }
```

where embedded-do is

```
DO index = { start-expression [ [TO expression] ] [WHILE(expression)] }, , ,
                              [ [BY expression] ]
```

where format-specification is

```
{ {integer   }  (format-specification , , ,)                                  }
{ {(expression)}                                                              }
{                                                                             }
{ [integer   ]  { A [(width)]                                                 }
{ [(expression)]{ B [(width)]                                                 }
{               { {COLUMN } [(new-column)]                                    }
{               { {COL    }                                                   }
{               { E (width,fd [,sd])                                          }
{               { F (width [,fd [,scale]])                                    }
{               { LINE [(new-line)]                                           }
{               { P 'picture-specification'                                   }
{               { PAGE                                                        }
{               { R(remote-format)                                            }
{               { SKIP [(line-count)]                                         }
{               { X [(space-count)]                                           }
```

where line-count, new-line, new-column, and space-count are expressions.

Figure 12-56. PUT Statement Syntax

SKIP Option

The SKIP option specifies that the CRM file is to be positioned forward by some number of lines before data transmission begins, or is to be simply positioned forward with no subsequent data transmission. Skipping is performed before data transmission, even if the SKIP option appears to the right of the LIST or EDIT option.

The expression is evaluated and converted to an integer. If the SKIP option does not include an expression to indicate the actual number of lines, SKIP(1) is assumed. SKIP(0) is legal only when the file is opened with the PRINT attribute; this option causes the next line to be overprinted on the preceding line. The initial position of a new output file immediately after it is opened is the first character of the first line. For preexisting files, the initial position is the top of page 1, but not the beginning-of-information. A print file is positioned at the first page.

LINE Option

The LINE option specifies vertical spacing for a print file. The operation is performed before data transmission, even if the LINE option appears to the right of the LIST or EDIT option.

The expression is evaluated and converted to an integer. When the LINE option does not include an expression, LINE(1) is assumed. LINE(n) is processed as follows:

LINE(n) specifies the current line.	No effect if line n is empty. If it is not empty, it skips to a new page.
LINE(n) specifies a line beyond current line, but not beyond PAGESIZE.	Skip to specified line.
LINE(n) specifies a line before current line or beyond PAGESIZE.	Skip remainder of page and raise ENDPAGE condition.

PAGE Option

The PAGE option specifies positioning of a print file at the beginning of a new page. Positioning is at column 1, line 1 of the following page. The operation is performed before data transmission, even if the PAGE option appears to the right of the LIST or EDIT option.

Statement Processing

Execution of the PUT statement is performed in the order shown in table 12-11. This information is important for the following reasons:

- If a condition is raised during execution of the PUT statement, not all of the operations will have been performed.
- If an on-unit does not return to the point at which the condition was raised, not all of the operations will have been performed.
- If any option contains a function reference, the effect of the PUT statement can depend on the order in which operations are performed.

Examples of PUT statements are shown in figure 12-57.

TABLE 12-11. PUT STATEMENT PROCESSING

Step	PUT Statement Option	Action Taken
1.	STRING	The length of the target generation is evaluated. (The length of the ONCHAR pseudovariable is 1.)
2.	SKIP	The expression is evaluated and converted to a fixed binary integer.
3.	LINE	The expression is evaluated and converted to a fixed binary integer.
4.	FILE or assumed file option	If the file is not open, it is opened implicitly with attributes STREAM and OUTPUT.
5.	PAGE	The file must have the PRINT attribute. The file is positioned at the top of the next page.
6.	LINE	The file must have the PRINT attribute. The line value must be greater than zero. The file is positioned at the required line. This step can raise ENDPAGE.
7.	SKIP	If the file has the PRINT attribute, the skip value must be greater than or equal to zero; if the file does not have the PRINT attribute, the skip value must be greater than zero. The file is positioned forward by the specified number of lines; the next character transmitted to the file will be in column 1. Execution of the SKIP option can raise ENDPAGE.
8.	LIST or EDIT	Data is transmitted as described in section 8, Input/Output.
9.	STRING	The output stream is assigned to the string target.

Conditions

Conditions that can be raised during PUT statement execution include: UNDEFINEDFILE, ENDPAGE, ERROR, and TRANSMIT.

The UNDEFINEDFILE condition is raised when an implicit file opening fails. If control returns from the UNDEFINEDFILE on-unit and the file is still not open, the ERROR condition is raised.

```
PUT STRING(ONSOURCE()) EDIT(P,Q)(A(4),A(6));
```

The statement specifies edit-directed transmission of the values of P and Q to the ONSOURCE pseudovariable.

```
PUT FILE(OUTFILE) LINE(35) LIST(A.B.C, P, Q);
```

The statement specifies list-directed output of the values of A.B.C, P, and Q to the CRM file associated with the file OUTFILE. Before transmission, the CRM file is positioned forward to the line specified by the LINE option.

```
PUT FILE(F) SKIP(3) LIST(A,
   (B(I) DO I = 3 TO J BY 2));
```

The statement specifies list-directed output of the value of A and the values of zero or more elements of the array B. Before transmission, the CRM file is positioned forward to the third line relative to the current line.

Figure 12-57. Sample PUT Statements

The ENDPAGE condition is raised when the current line becomes larger than the page size. If control returns from the ENDPAGE on-unit, statement processing continues.

The ERROR condition is raised when the string specified by the STRING option is too small to contain the transmitted characters.

The TRANSMIT condition is raised if parity errors are encountered. If control returns from the TRANSMIT on-unit, execution continues with the statement following the PUT statement.

Computational conditions can be raised during expression evaluation, conversion, and assignment.

READ STATEMENT

The READ statement is a record I/O statement that reads a record from a CRM file into a generation of storage or reads forward for record positioning purposes. The syntax of the READ statement is shown in figure 12-58.

```
[prefix]... READ FILE(file-reference)
    { INTO(read-target)      }   [ KEY(expression)    ]
    { SET(pointer)           }   [ KEYTO(key-target)  ]   ;
    { IGNORE(expression)     }
```

Figure 12-58. READ Statement Syntax

The following rules apply:

- Keywords can appear in any order after the READ keyword.
- The file-reference must be a file constant or a file variable.
- A direct file requires the KEY option and either the INTO or SET option.
- The INTO option read-target must be an unsubscripted reference to a variable that does not have the member attribute. The storage type of the variable must be STATIC, AUTOMATIC, CONTROLLED, or BASED.
- The SET option pointer must be an allocated scalar pointer variable.
- The IGNORE option expression must be scalar and computational. The option can only be used for a file that has the SEQUENTIAL attribute.
- The KEY option expression must be scalar and computational. The option can only be used for a file with the KEYED attribute.
- The KEYTO option key-target must be an allocated scalar character variable, or an appropriate pseudovariable. The option can only be used for a file with the KEYED attribute.

INTO, SET, and IGNORE Options

A READ statement can include only one of the three options INTO, SET, and IGNORE. Each option is identified as follows:

The INTO option identifies a variable. The record is read into the generation of storage for the variable.

The SET option specifies that a buffer is to be allocated and a record is to be read into the buffer. The address of the allocated buffer is assigned to the pointer variable.

The IGNORE option specifies that the CRM file is to be positioned forward by zero or more records and the skipped records are to be ignored.

KEY and KEYTO Options

A READ statement can include only one of the two options KEY and KEYTO. Each option is identified as follows:

The KEY option specifies the key of the record to be read. If both the KEY and IGNORE options are included, the KEY option specifies the key of the record to which the CRM file is to be positioned before skipping begins.

The KEYTO option specifies that the key of the record is to be obtained and assigned to the key-target. If both the KEYTO and IGNORE options are included, the IGNORE option specifies forward positioning and the KEYTO option specifies that the key of the last record ignored is to be obtained and assigned.

Statement Processing

The READ statement is processed through the following steps in the order given:

1. The INTO, SET, or IGNORE option is evaluated. The IGNORE option expression is converted to a fixed binary integer; its value must be greater than or equal to zero. The KEY or KEYTO option is evaluated. The value of the KEY option expression is converted to a character string.

2. If the file is open, it must have the RECORD attribute and must have either the INPUT or the UPDATE attribute. If the file is not open, an implicit file opening is executed. The RECORD attribute is supplied for the file opening. If the file is not declared with the UPDATE attribute, the INPUT attribute is also supplied.

3. If the file has an allocated buffer from previous execution of a READ statement with a SET option, that allocated buffer is freed.

4. KEY option processing is performed as follows:

KEY option is included	The current-record designator is set to identify the record.
KEY option is not included	The record denoted by the current-record designator is the previous record read. If an INTO or SET option is present, the designator is reset to identify the next record in the CRM file.

5. The IGNORE option is processed. The current-record designator is set to identify the record that is the specified number of lines beyond the current record. If, for example, the current-record designator is positioned at record 1, IGNORE(2) causes the designator to be reset to identify record 3. If the IGNORE option value is zero, no action is taken.

6. The KEYTO option is processed. The value of the key associated with the current record is assigned to the key-target.

7. The INTO option is processed. The current record is copied into the generation associated with the read-target.

8. The SET option is processed. A generation is allocated and associated with the file as a record buffer. The length of the buffer is established as the current record length. The current record is copied into the allocated buffer, and a value is assigned to the pointer variable specified by the SET option. The generation will be freed by a subsequent READ, DELETE, or REWRITE statement, or when the file is closed.

Examples of READ statements are shown in figure 12-59.

```
READ FILE(A) INTO(INREC);

READ FILE(B) IGNORE(I);

READ FILE(C) SET(CRECPTR) KEY(J+1);

READ FILE(D) INTO(DREC) KEYTO(CURRKEY);

READ FILE(E) KEY(K) IGNORE(2);
```

Figure 12-59. Sample READ Statements

Conditions

Conditions that can be raised during READ statement execution include: UNDEFINEDFILE, KEY, ENDFILE, RECORD, and TRANSMIT.

The UNDEFINEDFILE condition is raised when an implicit file opening fails. If control returns from the UNDEFINEDFILE on-unit and the file is still not open, the ERROR condition is raised.

The KEY condition is raised when the key identified by the KEY option does not identify a record in the CRM file. The current-record designator is set to undefined. If control returns from the KEY on-unit, program execution continues at the statement following the READ statement.

The ENDFILE condition is raised when end-of-file is reached during IGNORE, INTO, and SET option processing. If control returns from the ENDFILE on-unit, execution continues with the statement following the READ statement.

The RECORD condition is raised when the record designated by the current-record designator is not the same length as the buffer allocated during SET option processing. If control returns from the RECORD on-unit, statement processing continues. If the record is shorter than the buffer, the excess area in the buffer has an undefined value. If the record is longer than the buffer, the excess part of the record is ignored.

The TRANSMIT condition is raised if parity errors are encountered. If control returns from the TRANSMIT on-unit, execution continues with the statement following the READ statement.

Computational conditions can be raised during expression evaluation, conversion, and assignment.

RETURN STATEMENT

The RETURN statement terminates a procedure or on-unit activation and returns control to the point at which the procedure was invoked or to the block in which the condition was raised. The RETURN statement can specify a value to be returned by a procedure. The syntax of the RETURN statement is shown in figure 12-60.

```
[prefix]... RETURN [(return-value)] ;
```

Figure 12-60. RETURN Statement Syntax

The following rules apply:

- The return-value must be scalar.
- The return-value can be a computational, locator, or area value. The return-value cannot be a file, format, entry, or label value.

A particular RETURN statement and a particular entry point of a procedure are said to be compatible if one of the following is true:

The RETURN statement has no return-value, and the entry point has no RETURNS option.

The RETURN statement has a return-value, the entry point has a RETURNS option, and the data type of the return-value can be converted to the data type specified by the RETURNS option.

A RETURN statement with no return-value is compatible with an on-unit.

When the RETURN statement is executed, the following actions are taken in the order shown:

1. If the RETURN statement has a return-value, the return-value is evaluated.

2. If the current block activation is an activation of a begin block, the activation is terminated abnormally; this step is repeated until the current block activation is either a procedure or an on-unit activation.

3. The RETURN statement is checked against the entry point used to invoke the procedure or against the on-unit; it must be compatible with the entry point or on-unit.

4. If the current block activation is now a procedure activated at a subroutine entry point, the current block activation is terminated normally; and control passes to the statement following the CALL statement that invoked the procedure.

   If the current block activation is now a procedure activated at a function entry point, the return-value is converted to the type specified by the RETURNS option of the entry point. The current block activation is terminated normally, control passes to the statement containing the function reference that invoked the procedure, and the converted return-value is used in that statement execution.

   If the current block activation is now an on-unit activation, the current on-unit activation is terminated normally; and control passes either to the statement in which the condition was raised, or to the following statement. (Note that on-units for some conditions cannot be terminated normally.) The precise point to which control is returned depends upon the particular condition and the manner in which it was raised. Refer to section 10, Conditions, for details.

Examples of RETURN statements are shown in figure 12-61.

```
A: PROC RETURNS(CHAR(5));
     .
     .
     .
     IF I<J THEN RETURN(ISTRING);
            ELSE RETURN ('FAIL');
   END A;

B: PROC RECURSIVE;
     DCL (P,Q) FLOAT;
     .
     .
     .
     X = E(I,J)+K;
     .
     .
     .
     RETURN;

E: ENTRY(P,Q) RETURNS(FIXED);
     .
     .
     .
INNER: BEGIN;
         .
         .
         .
         RETURN(Y**2);
       END;
   END B;
```

Figure 12-61. Sample RETURN Statements

```
[prefix]... REVERT condition ;
```

Figure 12-62. REVERT Statement Syntax

REVERT STATEMENT

The REVERT statement removes the on-unit established in the current block activation for the specified condition. The on-unit that was in effect for the specified condition when the current block was activated is restored. The syntax of the REVERT statement is shown in figure 12-62.

The following rules apply:

- The condition must be one of the names listed in the table of conditions in section 10 (table 10-1).

- Only one condition can be referenced in the statement.

The REVERT statement only removes an on-unit that was established in the current block activation. If the REVERT statement references a condition for which there is no established on-unit in the current block activation, no action is taken.

Examples of REVERT statements are shown in figure 12-63.

```
P: PROC;
     .
     .
     .
     ON RECORD(FILEA) CALL MSGPROC;
     ON RECORD(FILEB) SYSTEM;
     .
     .
     .
     REVERT RECORD(FILEA);  <-- Removes programmed on-unit for RECORD(FILEA).
     .
     .
     .
     REVERT ZERODIVIDE;     <-- No action taken; ZERODIVIDE on-unit was not established in this block activation.
     .
     .
     .
   END;
```

Figure 12-63. Sample REVERT Statements

REWRITE STATEMENT

The REWRITE statement is a record I/O statement that replaces an existing record in a CRM file. The syntax of the REWRITE statement is shown in figure 12-64.

```
[prefix]... REWRITE FILE(file-reference)

[FROM(source) [KEY(expression)]] ;
```

Figure 12-64. REWRITE Statement Syntax

The following rules apply:

- Keywords can appear in any order after the REWRITE keyword.
- The file-reference must name a file constant or a file variable.
- The FROM option source must be an unsubscripted reference to a variable that does not have the member attribute. The storage type of the variable must be STATIC, AUTOMATIC, CONTROLLED, or BASED.
- The KEY option expression must be scalar and computational.

FROM and KEY Options

The FROM option identifies the variable with a generation that is to be written as the replacement record. If the FROM option is not included, an allocated buffer established by a READ statement with a SET option must be associated with the file; the buffer contains the replacement record.

The KEY option supplies the key value that is used to identify the record when it is written.

Statement Processing

The REWRITE statement is processed through the following steps in the order given:

1. The FROM option is evaluated to determine the generation to be written. The KEY option is evaluated and the expression is converted to a character string.
2. If the file is open, it must have the attributes RECORD and UPDATE. If the file is not open, an implicit file opening is executed. Implied attributes RECORD and UPDATE are supplied for the file opening.
3. If the KEY option is present, the file must have the KEYED attribute; if the KEY option is not present, the file must not have the KEYED attribute. KEY option processing is performed as follows:

KEY option is included.	The current-record designator is set to identify the record specified by the KEY option.
KEY option is not included.	The record denoted by the current-record designator is the record to be replaced.

4. FROM option processing is performed as follows:

FROM option is included.	The generation identified by the FROM option is written as a replacement record.
FROM option is not included.	The allocated buffer previously established by a READ statement with a SET option is written as a replacement record. The buffer is then freed.

Examples of REWRITE statements are shown in figure 12-65.

```
REWRITE FILE(A) FROM(X(2));

REWRITE FILE(B);

REWRITE FILE(C) KEY(CKEY+1) FROM(Y);
```

Figure 12-65. Sample REWRITE Statements

Conditions

Conditions that can be raised during REWRITE statement execution include: UNDEFINEDFILE, KEY, RECORD, and TRANSMIT.

The UNDEFINEDFILE condition is raised when an implicit file opening fails. If control returns from the UNDEFINEDFILE on-unit and the file is still not open, the ERROR condition is raised.

The KEY condition is raised when the key identified by the KEY option does not identify a record in the CRM file. The current-record designator is set to undefined. If control returns from the KEY on-unit, program execution continues at the statement following the REWRITE statement.

The RECORD condition is raised when the record designated by the current-record designator is not the same length as the FROM option source. If control returns from the RECORD on-unit, statement processing continues.

The TRANSMIT condition is raised if parity errors are encountered. If control returns from the TRANSMIT on-unit, execution continues with the statement following the REWRITE statement.

Computational conditions can be raised during expression evaluation, conversion, and assignment.

SIGNAL STATEMENT

The SIGNAL statement simulates a condition. The syntax of the SIGNAL statement is shown in figure 12-66.

```
[prefix]... SIGNAL condition ;
```

Figure 12-66. SIGNAL Statement Syntax

The following rule applies:

- The condition must be one of the names listed in the table of conditions in section 10 (table 10-1).

If the condition is enabled, execution of the SIGNAL statement simulates the condition. Values are assigned to the appropriate condition builtin functions, and the current established on-unit for that condition is activated. If control returns from the on-unit, control passes to the statement following the SIGNAL statement.

When the SIGNAL statement is executed, the following condition builtin function assignments are made:

ONCODE and ONLOC	are set to new current values..
ONFILE	is set to the named file if an I/O condition is signaled.
ONKEY	is set to a null character string if the KEY condition is signaled.
ONCHAR	is set to a null character string if the CONVERSION condition is signaled.
ONSOURCE	is set to a null character string if the CONVERSION condition is signaled.

Examples of SIGNAL statements are shown in figure 12-67.

```
SIGNAL CONVERSION;
SIGNAL TRANSMIT(FILEA);
SIGNAL AREA;
SIGNAL CONDITION(MINE);
```

Figure 12-67. Sample SIGNAL Statements

STOP STATEMENT

The STOP statement raises the FINISH condition. On normal termination of the FINISH on-unit, the program is terminated. The syntax of the STOP statement is shown in figure 12-68.

```
[prefix]... STOP ;
```

Figure 12-68. STOP Statement Syntax

The FINISH condition is raised before program execution is terminated. All active blocks remain active during the execution of the FINISH on-unit. If control returns from the on-unit to the point of interrupt, all active blocks are terminated, all open files are closed, and control is returned to the operating system.

A programmed FINISH on-unit can be written to perform special termination processes or cause program execution to continue. Program execution continues if the FINISH on-unit is terminated by a nonlocal GOTO.

The system FINISH on-unit returns control to the point of interrupt.

Examples of STOP statements are shown in figure 12-69.

```
IF COUNT > LIMIT THEN STOP;
ON ENDFILE(F) SNAP STOP;
```

Figure 12-69. Sample STOP Statements

WRITE STATEMENT

The WRITE statement is a record I/O statement that transmits a record to a CRM file. The statement can be used to create CRM files and to append records to existing CRM files. The syntax of the WRITE statement is shown in figure 12-70.

```
[prefix]... WRITE FILE(file-reference)
    FROM(source) [KEYFROM(expression)] ;
```

Figure 12-70. WRITE Statement Syntax

The following rules apply:

- Option keywords can appear in any order after the WRITE keyword.
- The file-reference must name a file constant or a file variable.
- The FROM option source must be an unsubscripted reference to a variable that does not have the member attribute. The storage type of the variable must be STATIC, AUTOMATIC, CONTROLLED, or BASED.
- The KEYFROM expression must be scalar and computational.

FROM and KEYFROM Options

The FROM option identifies the variable with the generation that is to be written as the new record.

The KEYFROM option supplies the key value that is used to identify the record when it is written.

Statement Processing

The WRITE statement is processed through the following steps in the order given:

1. The FROM option is evaluated to determine the generation to be written. If the KEYFROM option is present, its expression is evaluated and converted to a character string.
2. If the file is not open, an implicit file opening is executed. The RECORD attribute is supplied for the file opening. If the file is not declared with the UPDATE attribute, the OUTPUT attribute is also supplied.

3. The file must now have the RECORD attribute; it must also have either the OUTPUT attribute or the UPDATE and DIRECT attributes.

4. If the KEYFROM option is present, the file must have the KEYED attribute. If the KEYFROM option is not present, the file must not have the KEYED attribute.

5. If an output record buffer created by a previous LOCATE statement exists, the buffer is written as a record to the CRM file. If the output file is DIRECT, the key value specified in the KEYFROM option is associated with the record. If the output file is SEQUENTIAL, the record is written at the position immediately following the one indicated by the current-record designator. The current-record designator is changed to identify the new record. The buffer is then freed.

6. If the file has a buffer allocated by the previous execution of a READ statement with a SET option, that buffer is freed.

7. The generation identified by the FROM option is written to the CRM file. If a KEYFROM option is included, its value is associated with the new record and determines where the record is written. If a KEYFROM option is not included, the record is written at the position immediately following the one indicated by the current-record designator. The current-record designator is changed to identify the new record.

Examples of WRITE statements are shown in figure 12-71.

```
WRITE FILE(A) FROM(PAYROLL) KEYFROM(EMPNO);
WRITE FROM(SNAPSHOT) FILE(B);
```

Figure 12-71. Sample WRITE Statements

Conditions

Conditions that can be raised during WRITE statement execution include: UNDEFINEDFILE, KEY, RECORD, and TRANSMIT.

The UNDEFINEDFILE condition is raised when an implicit file opening fails. If control returns from the UNDEFINEDFILE on-unit and the file is still not open, the ERROR condition is raised.

The KEY condition is raised during the buffer write operation or the record write operation when the key value duplicates an existing key in the CRM file. If control returns from the KEY on-unit, program execution continues at the statement following the WRITE statement.

The RECORD condition is raised during buffer write operations when the record size is unacceptable for the CRM file. If control returns from the RECORD on-unit, the ERROR condition is raised.

The TRANSMIT condition is raised if parity errors are encountered. If control returns from the TRANSMIT on-unit, execution continues with the statement following the WRITE statement.

Computational conditions can be raised during expression evaluation, conversion, and assignment.

The PL/I compiler is called with a control statement that conforms to operating system syntax. The control statement can be continued under the NOS/BE operating system; it cannot be continued under the NOS operating system.

More than one external procedure can be compiled by a single call to the compiler. The procedures must follow each other without any intervening end-of-record or end-of-file.

PLI CONTROL STATEMENT

The name to be used on the control statement is PLI, beginning in column 1. The PLI control statement can appear in one of the following forms:

PLI(p1,p2, . . . ,pn) comments

PLI,p1,p2, . . . ,pn. comments

PLI. comments

Parameters (p_i) can be written in any order and must be separated by commas. It is an error if any parameter appears more than once. Comments are optional. They are ignored by the compiler but are printed on the dayfile.

All parameters are optional. The defaults that result from omission of a parameter might be changed at an installation. Parameters are given in alphabetic order in the following paragraphs.

B BINARY OUTPUT FILE NAME

omitted	Binary output from compilation is written to file LGO.
B	Binary output from compilation is written to file BIN.
B=lfn	Binary output from compilation is written to file lfn, where lfn is a name consisting of letters A through Z and digits 0 through 9. The name must be one to seven characters in length and must begin with a letter.
B=0	Binary output from compilation is suppressed.

BL BURSTABLE LISTING

omitted	The error list and the attribute list can begin anywhere on a page.
BL	Page ejection occurs before the error directory and before the attribute list.

COL SOURCE COLUMNS

omitted	Source text on the input (I) file is in columns 1 to 72, inclusive; the system supplies standard carriage control to the source listing. Columns outside 1 through 72 are ignored.
COL	Source text on the input (I) file is in columns 2 to 72, inclusive; a carriage control character is found in column 1. Columns outside 2 through 72 are ignored.
COL=m/n[/p]	Source text on the input (I) file is in columns m to n, inclusive; a carriage control character is found in column p. All other columns are ignored. The allowable ranges of m, n, and p are $1 \leq n \leq 100$ $1 \leq m \leq n$ $0 \leq p \leq 100$ The value of p must be less than m or greater than n. If p is 0 or is not specified, the system supplies standard carriage control to the source listing. Carriage control characters are used to print the source listing on the L file if the LO=S option is selected. Characters in column p have the following meanings: blank — Single space 0 — Double space - — Triple space 1 — Page ejection Any other characters appearing in column p are ignored.

DB DEBUGGING OPTIONS

omitted	Equivalent to DB=0.
DB	Equivalent to DB=B.
DB=B	Loadable binary output is produced regardless of errors. Execution of a statement with an error of severity C or F will raise the ERROR condition.
DB=0	Loadable binary output is produced for external procedures that do not contain any errors of severity C or F. The binary output produced for an external procedure that does contain an error of severity C or F is not loadable; attempting to load it produces a fatal loader error.

E ERROR FILE NAME

omitted	Error information specified by the EL parameter is written to file OUTPUT.

E — Error information specified by the EL parameter is written to file ERRS.

E=lfn — Error information specified by the EL parameter is written to file lfn, where lfn is a name consisting of letters A through Z and digits 0 through 9. The name must be one to seven characters in length and must begin with a letter.

E=0 — Error file output is suppressed, despite any options specified by the EL parameter.

Diagnostics written to the file specified by the E option are also written to the file specified by the L option. If the two files are the same, only one copy is written.

EL ERROR LEVEL TO BE REPORTED

omitted — Equivalent to EL=W.

EL — Equivalent to EL=F.

EL=I — Informational diagnostics, plus all errors of levels T, W, F, and C are listed.

Level I diagnostics provide information to help the user improve the program.

EL=T — Trivial errors, plus all errors of levels W, F, and C are listed.

Level T diagnostics indicate a minor violation of implementation limits (the source program has been adjusted by the compiler); implementation-dependent usage; or potentially troublesome constructs in PL/I. The syntax is correct, and the binary output is good.

EL=W — Warning errors, plus all errors of levels F and C are listed.

Level W diagnostics indicate correct syntax but questionable usage. The binary output is good, but execution of the diagnosed statement might raise the ERROR condition.

EL=F — Fatal errors, plus all errors of level C are listed.

Level F diagnostics indicate incorrect syntax or unresolvable semantic errors. Loadable binary output is produced only if DB=B is specified.

EL=C — Compiler errors only are listed.

Level C diagnostics indicate that the limits of the compiler have been exceeded or the compiler has malfunctioned. Some level C errors cause immediate termination of compilation.

Diagnostics are listed on the files specified by the E and L parameters.

ET ERROR TERMINATION

omitted — Equivalent to ET=0.

ET — Equivalent to ET=F.

ET=I — Upon completion of all compilations, control passes to the next appropriate control statement if the compiler has issued any diagnostics of levels I, T, W, F, or C.

ET=T — Upon completion of all compilations, control passes to the next appropriate control statement if the compiler has issued any diagnostics of levels T, W, F, or C.

ET=W — Upon completion of all compilations, control passes to the next appropriate control statement if the compiler has issued any diagnostics of levels W, F, or C.

ET=F — Upon completion of all compilations, control passes to the next appropriate control statement if the compiler has issued any diagnostics of levels F or C.

ET=C — Upon completion of all compilations, control passes to the next appropriate control statement if the compiler has issued any diagnostics of level C.

ET=0 — Upon completion of all compilations, control passes to the next appropriate control statement despite any errors diagnosed during compilation.

Error termination does not depend on options specified for the E, L, and EL parameters.

GO COMPILE AND EXECUTE

omitted — Equivalent to GO=0.

GO — The binary output produced by the compiler is loaded and executed at the end of compilation. If GO is specified without DB=B and there are any diagnostics of level F or C, control passes to the next appropriate EXIT control statement without invoking the loader.

GO=0 — The binary output is not loaded and executed by the PLI control statement.

I INPUT FILE NAME

omitted — Source code for the program to be compiled resides on file INPUT.

I — Source code for the program to be compiled resides on file COMPILE.

I=lfn — Source code for the program to be compiled resides on file lfn, where lfn is a name consisting of letters A through Z and digits 0 through 9. The name must be one to seven characters in length and must begin with a letter.

INRULE I-THROUGH-N RULE

omitted	Equivalent to INRULE=0.
INRULE	The compiler uses nonstandard default attributes for arithmetic variables, parameter descriptors, and returns descriptors as described under Default Attributes in section 5.
INRULE=0	The compiler uses the standard default attributes for all identifiers and descriptors.

L LISTING FILE NAME

omitted	Diagnostics and the information selected by the LO parameter are written to file OUTPUT.
L	Diagnostics and the information selected by the LO parameter are written to file LIST.
L=lfn	Diagnostics and the information selected by the LO parameter are written to file lfn, where lfn is a name consisting of letters A through Z and digits 0 through 9. The name must be one to seven characters in length and must begin with a letter.
L=0	No listing is produced, despite any options specified by the LO parameter.

Diagnostics written to the file specified by the E option are also written to the file specified by the L option. If the two files are the same, only one copy is written.

LO LISTING OPTIONS

omitted	Equivalent to LO=S/A.
LO	Equivalent to LO=S/A/R.
LO=A	A complete set of attributes for each declared identifier is listed.
LO=O	Generated object code with pseudo COMPASS mnemonics is listed.
LO=R	A reference list is generated. For each identifier used in the program, a list of statement numbers of the statements referencing the identifier is generated.
LO=S	Source program is listed. All columns are listed, despite any options specified by the COL parameter.
LO=0	None of the information that can be selected by A, O, S, or R is listed. Only error messages are listed.

Listing options are listed on the file specified by the L option.

Multiple options for the LO parameter can be selected by separating the options with slashes. Options can appear in any order.

PD PRINT DENSITY

omitted	Equivalent to PD=6.
PD	Equivalent to PD=8.
PD=6	Listing specified by L and E parameters is spaced at six lines per inch.
PD=8	Listing specified by L and E parameters is spaced at eight lines per inch.

Any option specified by this parameter must be supported by the printer on which the files are output. The PD parameter is ignored for interactive terminals.

PS PAGE SIZE

omitted	Number of lines on a printed output page is 60 if PD is omitted or PD=6. Number of lines is 80 if PD=8.
PS	Illegal.
PS=n	Number of lines on a printed output page is n; the value of n cannot be less than four. The value of n does not include space for top and bottom margins.

COMPILATION LISTINGS

The LO parameter of the compiler call selects the listings that are produced during compilation. The listings appear on the standard job print file, OUTPUT, unless another file is selected by the L parameter of the PLI control statement.

Listings that can be selected by the LO parameter are

S	Source listing
A	Attribute list
R	Reference list
O	Object code

Another listing that appears is the error directory. Information in the error directory is duplicated on the error file, which is selectable by the E parameter of the compiler call.

Output listings appear in the following order: source listing, error directory, attribute and reference list, object code.

SOURCE PROGRAM LISTING

The source listing shows the source program compiled. Each source line appears as it exists on the file identified by the I parameter on the PLI control statement. Statements are numbered for correlation with any diagnostic messages.

An example of a source listing is shown in figure 13-1. The listing is divided into four parts, which contain the following information:

STMT	Indicates the number of the first source statement that begins on the line.
BLK	Indicates the level of block nesting for the statement.
DO	Indicates the level of do group nesting for the statement.
SOURCE	Indicates the source line.

If the source statement is unrecognizable, an asterisk appears below the statement at the approximate point where the error was detected. The word ERROR appears in the left margin, and the message SYNTAX ERROR appears in the error directory.

ERROR DIRECTORY

Diagnostics listed during compilation are controlled by the EL parameter on the PLI control statement. An example of the error directory is shown in figure 13-2.

The listing shows the following column headings:

ERROR NUMBER	Is a 6-character sequence containing a 3-character error code and a 3-digit number. The 3-character code can be one of the following:

BIF	Builtin function error
CMP	Compiler error
CNV	Conversion error
DCL	Declaration error
EXP	Expression error
MSC	Miscellaneous error
REF	Reference error
SYN	Syntax error

SEVERITY	Indicates severity of error as follows:

I	Informational
T	Trivial
W	Warning
F	Fatal

STMT NO	Indicates the source statement number in which the error was detected.
ERROR TEXT	Provides the full text of the error message.

ATTRIBUTE AND REFERENCE LISTS

The attribute and reference lists provide pertinent information concerning each identifier used in the source program. The attribute list supplies a complete set of attributes for the identifier, and the reference list indicates the source statements in which the identifier is referenced. Either or both lists can be selected for printing.

An example of a combined attribute and reference list is shown in figure 13-3. The listing shows the following column headings:

DCL AT	The number of the statement that makes the explicit declaration of the identifier. The character X indicates the identifier has the EXTERNAL attribute. The character – indicates the identifier has an implicit or contextual declaration.
IDENTIFIER	The identifier used in the program or assumed by default for stream I/O (SYSIN and SYSPRINT).
IN PROC	The name of the innermost internal procedure that contains the declaration. If no name appears, the declaration is not contained in an internal procedure.
ATTRIBUTE AND REFERENCE LIST	The attribute list includes the complete set of attributes for the identifier. The reference list includes the number of each statement in which the identifier is referenced.

Possible attributes and their meanings are listed in table 13-1.

OBJECT CODE

The object listing shows the generated object code instructions. With few exceptions, the actual instruction for executable code appears. Data is generated through the use of macros as well as pseudo instructions; when macros are used, their names and parameters are listed along with the first word of generated code. An example of an object code listing is shown in figure 13-4.

```
EXT. PROC PASCAL      73/74
 PASCAL:
  STMT BLK   DO     SOURCE

    1    0          PASCAL:
                    PROCEDURE OPTIONS(MAIN);
                    /* THIS PROGRAM PRODUCES A SMALL PASCAL TRIANGLE AS OUTPUT. */
                    /* ONE INPUT VALUE IS READ TO DETERMINE THE SIZE OF THE TRIANGLE. */

                       /* DECLARATIONS. */
    2    1             DECLARE ROW(15) FIXED DECIMAL(5,0) INITIAL((15)0);

    3    1             GET LIST(SIZE);
    4    1             IF SIZE > 15 THEN SIZE = 15;

                       /* HEADER IS PRODUCED. */
    6    1             PUT SKIP LIST('PASCAL''S TRIANGLE SHOWING THE FIRST',SIZE,' ROWS');
                       /* EXTRA LINE IS SKIPPED BETWEEN HEADER AND TRIANGLE ROWS. */
    7    1             PUT SKIP;

    8    1          ROW_LOOP:
                       /* ROW_LOOP GENERATES AND PRINTS EACH ROW OF THE TRIANGLE. */
                       DO ITEMS = 1 TO SIZE;
                          /* PERFORM ADDITIONS. FOR ONLY ONE ENTRY, LOOP IS SKIPPED. */
    9    1    1            DO ENTRY = ITEMS TO 2 BY -1;
   10    1    2               ROW(ENTRY) = ROW(ENTRY) + ROW(ENTRY-1);
   11    1    2            END;
                          /* SET THE FIRST ENTRY TO 1. */
   12    1    1            ROW(1) = 1;
                          /* FIND NUMBER OF SPACES TO SKIP SO TRIANGLE LINES UP. */
   13    1    1            SLIDE_OVER = (SIZE - ITEMS) * 3;
   14    1    1            PUT EDIT ((ROW(ENTRY) DO ENTRY = 1 TO ITEMS))
                                    (SKIP, (SLIDE_OVER)X(1), (15)F(6,0));
   15    1    1       END ROW_LOOP;

   16    1          END PASCAL;
```

Figure 13-1. Sample Source Listing

```
        EXT. PROC PASCAL      73/74
         PASCAL:
ERROR NUMBER / SEVERITY / STMT NO /   ERROR TEXT

        DCL024   F         3          SCALAR VARIABLE -FFF- CANNOT HAVE MULTIPLE INITIAL VALUES
        SYN001   F         8          SYNTAX ERROR
        REF002   F        12          ASSIGNMENT-TARGET MUST BE SCALAR
        SYN001   F        19          SYNTAX ERROR
```

Figure 13-2. Sample Error Directory

```
        EXT. PROC PASCAL      73/74
         PASCAL:
DCL AT /IDENTIFIER/IN PROC /ATTRIBUTE AND REFERENCE LIST
     -  ENTRY                FIXED,    BINARY    (15,   0),AUTOMATIC,ALIGNED
                                9   10   10   10
     -  ITEMS                FIXED,    BINARY    (15,   0),AUTOMATIC,ALIGNED
                                8    9   13
     1X PASCAL               ENTRY,    CONSTANT, OPTS MAIN
     2  ROW                  FIXED,    DECIMAL   ( 5,   0),INITIAL,  ARRAY,    AUTOMATIC,ALIGNED
                               10   10   10   12
     8  ROW_LOOP             LABEL,    CONSTANT
                                8
     -  SIZE                 FIXED,    BINARY    (15,   0),AUTOMATIC,ALIGNED
                                3    4    5    6    8   13
     -  SLIDE_OVER           FIXED,    BINARY    (15,   0),AUTOMATIC,ALIGNED
                               13
     -X SYSIN                FILE,     INPUT,    STREAM
                                3
     -X SYSPRINT             FILE,     OUTPUT,   PRINT,    STREAM
                                6    7
```

Figure 13-3. Sample Attribute and Reference List

TABLE 13-1. LIST OF ATTRIBUTES

Attribute	Definition
ALIGNED	The identifier is a variable with the ALIGNED attribute.
AREA (size)†	The identifier is an area variable.
ARRAY†	The identifier is an array; it has the dimension attribute.
AUTOMATIC	The identifier is a variable with the automatic storage type.
BASED	The identifier is a variable with the based storage type.
BINARY (p[,q])	The identifier is an arithmetic variable with a binary base; precision and scale factor are p and q.
BIT (length)†	The identifier is a bit string variable.
BUILTIN	The identifier is the name of a builtin function.
CHARACTER (length)†	The identifier is a character string variable.
CONDITION	The identifier is a programmer-named condition.
CONSTANT	The identifier is a named constant.
CTL	The identifier is a variable with the controlled storage type.
DECIMAL (p[,q])	The identifier is an arithmetic variable with a decimal base; precision and scale factor are p and q.
DEFINED ON††	The identifier is a variable with the defined storage type; the name of the host variable and the position (if one was specified) will follow.
DIRECT	The identifier is a direct access file.
ENTRY	The identifier is an entry constant or parameter.
ENV	The identifier has an ENVIRONMENT attribute.
FILE	The identifier is a file constant or parameter.
FIXED	The identifier is an arithmetic variable of fixed point scale.
FLOAT	The identifier is an arithmetic variable of floating point scale.
FORMAT	The identifier is a format constant.
INITIAL	The identifier has an initial value specified.
INPUT	The identifier is an input file.
KEYED	The identifier is a keyed file.
LABEL	The identifier is a label constant or variable.
MEMBER††	The identifier is a member of a structure; the names of up to 20 containing structures will follow.
OFFSET	The identifier is an offset variable.
OPTIONS (MAIN)	The identifier is an entry name of a PROCEDURE statement with OPTIONS (MAIN).
OUTPUT	The identifier is an output file.
PARAMETER	The identifier is a variable with the parameter storage type.
PICTURE	The identifier is a pictured variable.
POINTER	The identifier is a pointer variable.

TABLE 13-1. LIST OF ATTRIBUTES (Cont'd)

Attribute	Definition
POS	See DEFINED ON.
PRINT	The identifier is a print file.
RECORD	The identifier is a record file.
RETURNS	The identifier is a function entry constant or parameter.
SEQL	The identifier is a sequential file.
STATIC	The identifier is a variable with the static storage type.
STREAM	The identifier is a stream file.
STRUCTURE	The identifier is a structure variable; if it is a substructure, it also has the member attribute.
UNALIGNED	The identifier is a variable with the UNALIGNED attribute.
UPDATE	The identifier is an update file.
VARYING	The identifier is a string of varying length.
*ERROR	The declaration of the identifier is in error.

†The extent associated with an AREA, BIT, or CHARACTER attribute is listed in parentheses following that attribute. If the extent was specified as an unsigned decimal integer, it will be listed; otherwise, the extent cannot be determined at compile time, and will be listed as a minus sign. Array extents do not appear in the attribute list.

††If the name of a containing structure or a host variable is longer than 8 characters, the printed name is modified to show the first 4 characters, a hyphen, and the last 3 characters. For example, VERYLONGNAME becomes VERY-AME.

```
131 7070000052                                  SX7    A0+52B
              7060000023                        SX6    A0+23B
132 5110000062                          +       SA1    S100
              53670                             SA6    X7
133 0100000000                  <EXT>           RJ     PLLDIO+
134                                                                LINE 3
134 0400000000                  <EXT>           EQ     PLLDXX+
135                                                                LINE 3
135                                     GL.11   BSS    0
135                                                                LINE 4
135 7170000004                                  SX7    4
              5070000016                        SA7    A0+16B
136 43000                                       MX0    0
         5050000023                             SA5    A0+23B
137 5140000014                          +       SA4    I14
              37745                             IX7    X4-X5
                   36607                        IX6    X0+X7
140 0326000143                          +       PL     X6,GL.12
                                                                   LINE 5
              7170000005                        SX7    5
141 5070000016                                  SA7    A0+16B
              5150000014                +       SA5    I14
142 10755                                       BX7    X5
         5070000023                             SA7    A0+23B
143                                     GL.12   BSS    0
```

Figure 13-4. Sample Object Code Listing

This section provides sample programs that are complete PL/I programs. Deck structure is shown for a PL/I program submitted on cards. Each sample program is explained, and the input cards and output listings for each sample program are shown.

DECK STRUCTURE

A job deck submitted for execution through a card reader begins with a job statement and ends with an end-of-information (EOI) statement that has a 6/7/8/9 multiple punch in column 1. Between the job statement and the EOI statement, the job deck is divided into sections separated by cards with a 7/8/9 multiple punch in column 1.

If the job deck is submitted under NOS, the job deck has the following basic structure:

```
job statement.
USER control statement.
CHARGE control statement. (if required)
PLI control statement.
LGO control statement.
7/8/9
PL/I source program statements
.
.
.
7/8/9
Input data
.
.
.
6/7/8/9
```

If the job deck is submitted under NOS/BE, the job deck has the following basic structure:

```
job statement.
ACCOUNT control statement. (if required)
PLI control statement.
LGO control statement.
7/8/9
PL/I source program statements
.
.
.
7/8/9
Input data
.
.
.
6/7/8/9
```

The appropriate operating system reference manuals describe the control statements necessary to direct other operations, in particular the use of permanent files or magnetic tape files. In the deck structures just shown, the input data is included in the job deck and is known by the local file name INPUT. Any program output written to the local file OUTPUT is printed at job termination unless a control statement indicates another disposition for the program output.

SAMPLE PROGRAM PASCAL

Program PASCAL creates a small Pascal's triangle that illustrates the numbers used in polynomial expansion. The size of the triangle is set by the single value read from the input data. Arbitrarily, the maximum size of the triangle is kept at 15 rows.

The program consists of one main external procedure and no internal procedures. The program listing is shown in figure 14-1.

Line 2 shows the only necessary DECLARE statement declaration. The elements in array ROW are initialized to zero. Various other variables used as counters in the program default to FIXED BINARY (15,0).

Line 3 shows the input statement that reads the input value for the size of the triangle. Program input is shown in figure 14-2. Since GET LIST is used, the input value can appear at any position on the input card. The FILE(SYSIN) option is supplied by the compiler, and the GET LIST statement implicitly opens the file, using local file name INPUT.

Lines 6 and 7 write an output header to the output file and skip an extra line after the header. Program output is shown in figure 14-3. The FILE(SYSPRINT) option is supplied by the compiler in both statements, and line 6 opens the output file implicitly with local file name OUTPUT.

Lines 8 through 15 contain a do group that loops to produce each line of the triangle. Line 9 shows a nested do group where the increment on the index runs negative. Additionally, the nested do group is set up so that the entire nested do group is skipped on the first pass through the containing do group.

Line 13 shows the calculation used to produce the triangular shape of the output. The calculated value is used as the expression controlling the X format item in line 14. Note also that line 14 shows an embedded-do in the EDIT option of the PUT statement.

SAMPLE PROGRAM TBINT

Program TBINT calculates and prints amortization schedules from supplied input data. For each schedule, the amount borrowed, the annual percentage rate, the monthly payment amount, and the date of the first payment are supplied. Each line in the schedule lists the number of the month, the date of the payment, the amount to interest and total to interest, the amount to principal and total to principal, and the remaining balance.

The program consists of one main external procedure and one internal procedure named REPORT. The program listing is shown in figure 14-4.

Lines 2 through 6 shows declarations for the main external procedure. Note that START is declared as a structure. The members MON, DAY, and YEAR default to FIXED BINARY (15,0).

Line 7 shows an ON statement that establishes the action to be taken when the ENDFILE condition is raised for file SYSIN, automatically associated with local file name INPUT. The action is to go to label QUIT on the END statement for the main external procedure.

Line 9 reads the next set of input values. Program input is shown in figure 14-5. Since GET LIST is used, the input values can appear at any positions on the input card but must appear in the correct order. The first execution of line 9 implicitly opens the input file SYSIN with local file name INPUT.

```
PASCAL:
 STMT BLK   DO     SOURCE

   1    0          PASCAL:
                   PROCEDURE OPTIONS(MAIN);
                   /* THIS PROGRAM PRODUCES A SMALL PASCAL TRIANGLE AS OUTPUT. */
                   /* ONE INPUT VALUE IS READ TO DETERMINE THE SIZE OF THE TRIANGLE. */

                      /* DECLARATIONS. */
   2    1             DECLARE ROW(15) FIXED DECIMAL(5,0) INITIAL((15)0);

   3    1             GET LIST(SIZE);
   4    1             IF SIZE > 15 THEN SIZE = 15;

                      /* HEADER IS PRODUCED. */
   6    1             PUT SKIP LIST('PASCAL''S TRIANGLE SHOWING THE FIRST',SIZE,' ROWS');
                      /* EXTRA LINE IS SKIPPED BETWEEN HEADER AND TRIANGLE ROWS. */
   7    1             PUT SKIP;

   8    1          ROW#LOOP:
                      /* ROW#LOOP GENERATES AND PRINTS EACH ROW OF THE TRIANGLE. */
                      DO ITEMS = 1 TO SIZE;
                         /* PERFORM ADDITIONS. FOR ONLY ONE ENTRY, LOOP IS SKIPPED. */
   9    1    1           DO ENTRY = ITEMS TO 2 BY -1;
  10    1    2              ROW(ENTRY) = ROW(ENTRY) + ROW(ENTRY-1);
  11    1    2           END;
                         /* SET THE FIRST ENTRY TO 1. */
  12    1    1           ROW(1) = 1;
                         /* FIND NUMBER OF SPACES TO SKIP SO TRIANGLE LINES UP. */
  13    1    1           SLIDE#OVER = (SIZE - ITEMS) * 3;
  14    1    1           PUT EDIT ((ROW(ENTRY) DO ENTRY = 1 TO ITEMS))
                                 (SKIP, (SLIDE#OVER)X(1), (15)F(6,0));
  15    1    1        END ROW#LOOP;

  16    1          END PASCAL;
```

Figure 14-1. Program PASCAL Listing

```
13
```

Figure 14-2. Program PASCAL Input

```
PASCAL'S TRIANGLE SHOWING THE FIRST          13          ROWS

                                1
                             1     1
                          1     2     1
                       1     3     3     1
                    1     4     6     4     1
                 1     5    10    10     5     1
              1     6    15    20    15     6     1
           1     7    21    35    35    21     7     1
        1     8    28    56    70    56    28     8     1
     1     9    36    84   126   126    84    36     9     1
  1    10    45   120   210   252   210   120    45    10     1
1    11    55   165   330   462   462   330   165    55    11     1
1    12    66   220   495   792   924   792   495   220    66    12     1
```

Figure 14-3. Program PASCAL Output

```
TBINT:
 STMT BLK   DO     SOURCE

   1    0          TBINT:
                   PROCEDURE OPTIONS(MAIN);
                   /* THIS PROGRAM GENERATES AND PRINTS AMORTIZATION SCHEDULES FROM */
                   /* INFORMATION SUPPLIED ON EACH INPUT RECORD. */

                      /* DECLARATIONS. */
   2    1              DECLARE AMOUNT FIXED DECIMAL(7,2);
   3    1              DECLARE RATE_APR FIXED DECIMAL(5,5);
   4    1              DECLARE RATE_MONTHLY FIXED DECIMAL(5,5);
   5    1              DECLARE PAYMENT FIXED DECIMAL(5,2);
   6    1              DECLARE 1 START,
                                 2 MON,
                                 2 DAY,
                                 2 YEAR;

                      /* ENDFILE FOR THE INPUT FILE IS TAKEN CARE OF. */
   7    1              ON ENDFILE(SYSIN) GOTO QUIT;

   9    1          NEXT_RECORD:
                      /* GET NEXT INPUT RECORD. */
                      GET LIST (AMOUNT, RATE_APR, PAYMENT, MON, DAY, YEAR);
                      /* VERIFY INPUT DATA READ. */
  10    1              PUT EDIT ('AMOUNT    ', AMOUNT, 'RATE      ', RATE_APR,
                                 'PAYMENT   ', PAYMENT,
                                 'DATE      ', MON, '/', DAY, '/', YEAR)
                                (PAGE, A, P'$$$,$$9V.99', SKIP, A, F(10,5),
                                 SKIP, A, P'$$$9V.99', SKIP, A, F(2), A, F(2), A, F(2));
                      /* CHECK FOR LOAN AMOUNT GREATER THAN ZERO. */
  11    1              IF AMOUNT <= 0.00 THEN GOTO BAD_RECORD;
                      /* CALCULATE MONTHLY INTEREST RATE. */
  13    1              RATE_MONTHLY = RATE_APR / 12.0000;
                      /* CHECK FOR SUFFICIENT PAYMENT TO COVER THE INTEREST. */
  14    1              IF PAYMENT <= AMOUNT * RATE_MONTHLY THEN GOTO BAD_RECORD;
                      /* CALL REPORT PROCEDURE. */
  16    1              GOTO PRODUCE_REPORT;

  17    1          BAD_RECORD:
                      /* PRODUCE A MESSAGE FOR THE BAD INPUT RECORD. */
                      PUT SKIP(5) LIST ('BAD INPUT RECORD DETECTED',
                                        'AMORTIZATION SCHEDULE NOT PRODUCED');
  18    1              GOTO NEXT_RECORD;

  19    1          PRODUCE_REPORT:
                      CALL REPORT (AMOUNT, RATE_MONTHLY, PAYMENT, START);
  20    1              GOTO NEXT_RECORD;

                   /*****************************************/
  21    1          REPORT:
                      PROCEDURE (BALANCE, RATE, MONTHLY, DATE);
                      /* THIS INTERNAL PROCEDURE CALCULATES AND PRINTS THE */
                      /* AMORTIZATION SCHEDULE FROM THE SUPPLIED INPUT DATA. */
                      /* RATE HAS BEEN CHANGED TO MONTHLY INTEREST RATE. */

                         /* DECLARATIONS. */
  22    2                 DECLARE BALANCE FIXED DECIMAL(7,2);
  23    2                 DECLARE RATE FIXED DECIMAL(5,5);
  24    2                 DECLARE MONTHLY FIXED DECIMAL(5,2);
  25    2                 DECLARE 1 DATE,
                                    2 MM,
                                    2 DD,
                                    2 YY;
  26    2                 DECLARE TO_INTEREST FIXED DECIMAL(5,2) INIT(0);
  27    2                 DECLARE TOTAL_INTEREST FIXED DECIMAL(7,2) INIT(0);
  28    2                 DECLARE TO_PRINCIPAL FIXED DECIMAL(5,2) INIT(0);
  29    2                 DECLARE TOTAL_PRINCIPAL FIXED DECIMAL(7,2) INIT(0);
  30    2                 DECLARE MONTH INITIAL(0);
```

Figure 14-4. Program TBINT Listing (Sheet 1 of 2)

```
                        /* PRINT AMORTIZATION SCHEDULE HEADERS. */
 31     2                 PUT SKIP(2) EDIT ('TOTAL', 'TOTAL', 'MONTHS', 'DUE DATE',
                                         'INTEREST', 'INTEREST', 'PRINCIPAL',
                                         'PRINCIPAL', 'BALANCE')
                                        (COL(36),A,COL(60),A,SKIP,A,COL(12),A,COL(24),
                                         A,COL(36),A,COL(48),A,COL(60),A,COL(72),A);
 32     2           NEXT_MONTH:
                        /* CALCULATES THE NEXT ROW OF THE SCHEDULE. */
                        DO WHILE ((BALANCE > 0.00) & (MONTH <= 360));

                           /* INCREMENT MONTH COUNTER. */
 33     2     1           MONTH = MONTH +1;
                           /* INCREMENT CURRENT DATE. */
 34     2     1           IF MONTH > 1
 35     2     1              THEN DO;
 36     2     2                   DATE.MM = DATE.MM + 1;
 37     2     2                   IF DATE.MM > 12
 38     2     2                      THEN DO;
 39     2     3                           DATE.YY = DATE.YY + 1;
 40     2     3                           DATE.MM = 1;
 41     2     3                           END;
 42     2     2                   END;

                           /* CALCULATE INTEREST DUE ON CURRENT BALANCE. ACCUMULATE. */
 43     2     1           TO_INTEREST = BALANCE * RATE;
 44     2     1           TOTAL_INTEREST = TOTAL_INTEREST + TO_INTEREST;
                           /* CALCULATE AMOUNT OF PAYMENT AGAINST PRINCIPAL. */
 45     2     1           TO_PRINCIPAL = MONTHLY - TO_INTEREST;
                           /* CHECK WHETHER PAYMENT IS SUFFICIENT TO CANCEL BALANCE. */
 46     2     1           IF TO_PRINCIPAL < BALANCE
 47     2     1              THEN DO;
                                 /* ACCUMULATE TOTAL PRINCIPAL AND ADJUST BALANCE. */
 48     2     2                   TOTAL_PRINCIPAL = TOTAL_PRINCIPAL + TO_PRINCIPAL;
 49     2     2                   BALANCE = BALANCE - TO_PRINCIPAL;
 50     2     2                   END;
 51     2     1              ELSE DO;
                                 /* ADJUST LAST PAYMENT TO COVER REMAINING BALANCE */
                                 /* AND INTEREST ON THE REMAINING BALANCE. */
 52     2     2                   TO_PRINCIPAL = BALANCE;
 53     2     2                   MONTHLY = TO_PRINCIPAL + TO_INTEREST;
                                 /* WRITE MESSAGE FOR LAST PAYMENT. */
 54     2     2                   PUT EDIT ('LAST PAYMENT CALCULATED AS ',MONTHLY)
                                          (SKIP, A, P'$$$9V.99');
                                 /* ACCUMULATE TOTAL PRINCIPAL AND ZERO BALANCE. */
 55     2     2                   TOTAL_PRINCIPAL = TOTAL_PRINCIPAL + TO_PRINCIPAL;
 56     2     2                   BALANCE = 0.00;
 57     2     2                   END;
                           /* DISPLAY ROW AS CALCULATED. */
 58     2     1           PUT EDIT (MONTH, MM, '/', DD, '/', YY, TO_INTEREST,
                                  TOTAL_INTEREST, TO_PRINCIPAL, TOTAL_PRINCIPAL,
                                  BALANCE)
                                 (SKIP, F(3), COL(12), F(2), A, F(2), A, F(2),
                                  COL(24), P'$$$9V.99', COL(36), P'$$$,$$9V.99',
                                  COL(48), P'$$$9V.99', COL(60), P'$$$,$$9V.99',
                                  COL(72), P'$$$,$$9V.99');
 59     2     1              END NEXT_MONTH;
                           /* END INTERNAL PROCEDURE. */
 60     2                 END REPORT;
                        /*****************************************/

 61     1           QUIT:
                        /* END MAIN EXTERNAL PROCEDURE. */
                        END TBINT;
```

Figure 14-4. Program TBINT Listing (Sheet 2 of 2)

```
500.00  .18157  25.00  11 05 75
1500.00  .11000  10.00  2 02 74
3000.00  .07500  93.56  4 15 72
```

Figure 14-5. Program TBINT Input

Line 10 verifies the input values by producing a header at the top of the next page of output. Program output is shown in figure 14-6. All values read in are written out. Lines 11 through 14 then make some basic checks on the input values, either directing control to line 17 or line 19. If the input values are definitely invalid, a suitable message is written and control passes to line 9. If the input values seem valid, control passes to line 9 after the internal procedure has been called to produce the amortization schedule.

Lines 21 through 60 contain the internal procedure REPORT.

Lines 22 through 30 show the declarations for the REPORT procedure. The members MM, DD, and YY, and the variable MONTH default to FIXED BINARY (15,0). Lines 22 through 25 declare the parameters passed to REPORT. Lines 26 through 30 declare variables used only in the REPORT procedure.

Lines 32 through 59 show a do group that loops to produce each line of the schedule. The do group is controlled by a WHILE option that terminates execution of the do group if the balance drops to zero or if the schedule has listed monthly calculations for a 30-year period, whichever comes first.

Lines 33 through 42 show a sequence of nested but unbalanced IF statements used to increment the current date. The nested IF statements are unbalanced in the sense that no ELSE operations are shown.

```
AMOUNT       $500.00
RATE          0.18157
PAYMENT     $25.00
DATE        11/ 5/75

                                        TOTAL                          TOTAL
MONTHS      DUE DATE     INTEREST      INTEREST     PRINCIPAL      PRINCIPAL     BALANCE
  1         11/ 5/75       $7.56         $7.56       $17.44         $17.44       $482.56
  2         12/ 5/75       $7.30        $14.86       $17.70         $35.14       $464.86
  3          1/ 5/76       $7.03        $21.89       $17.97         $53.11       $446.89
  4          2/ 5/76       $6.76        $28.65       $18.24         $71.35       $428.65
  5          3/ 5/76       $6.48        $35.13       $18.52         $89.87       $410.13
  6          4/ 5/76       $6.20        $41.33       $18.80        $108.67       $391.33
  7          5/ 5/76       $5.92        $47.25       $19.08        $127.75       $372.25
  8          6/ 5/76       $5.63        $52.88       $19.37        $147.12       $352.88
  9          7/ 5/76       $5.33        $58.21       $19.67        $166.79       $333.21
 10          8/ 5/76       $5.04        $63.25       $19.96        $186.75       $313.25
 11          9/ 5/76       $4.73        $67.98       $20.27        $207.02       $292.98
 12         10/ 5/76       $4.43        $72.41       $20.57        $227.59       $272.41
 13         11/ 5/76       $4.12        $76.53       $20.88        $248.47       $251.53
 14         12/ 5/76       $3.80        $80.33       $21.20        $269.67       $230.33
 15          1/ 5/77       $3.48        $83.81       $21.52        $291.19       $208.81
 16          2/ 5/77       $3.15        $86.96       $21.85        $313.04       $186.96
 17          3/ 5/77       $2.82        $89.78       $22.18        $335.22       $164.78
 18          4/ 5/77       $2.49        $92.27       $22.51        $357.73       $142.27
 19          5/ 5/77       $2.15        $94.42       $22.85        $380.58       $119.42
 20          6/ 5/77       $1.80        $96.22       $23.20        $403.78        $96.22
 21          7/ 5/77       $1.45        $97.67       $23.55        $427.33        $72.67
 22          8/ 5/77       $1.09        $98.76       $23.91        $451.24        $48.76
 23          9/ 5/77       $0.73        $99.49       $24.27        $475.51        $24.49
LAST PAYMENT CALCULATED AS  $24.86
 24         10/ 5/77       $0.37        $99.86       $24.49        $500.00         $0.00


AMOUNT       $1,500.00
RATE          0.11000
PAYMENT     $10.00
DATE        2/ 2/74


BAD INPUT RECORD DETECTED                    AMORTIZATION SCHEDULE NOT PRODUCED
```

Figure 14-6. Program TBINT Output (Sheet 1 of 2)

```
AMOUNT      $3,000.00
RATE          0.07500
PAYMENT     $93.56
DATE        4/15/72

                                    TOTAL                          TOTAL
MONTHS    DUE DATE     INTEREST     INTEREST     PRINCIPAL        PRINCIPAL     BALANCE
   1       4/15/72      $18.75       $18.75       $74.81           $74.81      $2,925.19
   2       5/15/72      $18.28       $37.03       $75.28          $150.09      $2,849.91
   3       6/15/72      $17.81       $54.84       $75.75          $225.84      $2,774.16
   4       7/15/72      $17.33       $72.17       $76.23          $302.07      $2,697.93
   5       8/15/72      $16.86       $89.03       $76.70          $378.77      $2,621.23
   6       9/15/72      $16.38      $105.41       $77.18          $455.95      $2,544.05
   7      10/15/72      $15.90      $121.31       $77.66          $533.61      $2,466.39
   8      11/15/72      $15.41      $136.72       $78.15          $611.76      $2,388.24
   9      12/15/72      $14.92      $151.64       $78.64          $690.40      $2,309.60
  10       1/15/73      $14.43      $166.07       $79.13          $769.53      $2,230.47
  11       2/15/73      $13.94      $180.01       $79.62          $849.15      $2,150.85
  12       3/15/73      $13.44      $193.45       $80.12          $929.27      $2,070.73
  13       4/15/73      $12.94      $206.39       $80.62        $1,009.89      $1,990.11
  14       5/15/73      $12.43      $218.82       $81.13        $1,091.02      $1,908.98
  15       6/15/73      $11.93      $230.75       $81.63        $1,172.65      $1,827.35
  16       7/15/73      $11.42      $242.17       $82.14        $1,254.79      $1,745.21
  17       8/15/73      $10.90      $253.07       $82.66        $1,337.45      $1,662.55
  18       9/15/73      $10.39      $263.46       $83.17        $1,420.62      $1,579.38
  19      10/15/73       $9.87      $273.33       $83.69        $1,504.31      $1,495.69
  20      11/15/73       $9.34      $282.67       $84.22        $1,588.53      $1,411.47
  21      12/15/73       $8.82      $291.49       $84.74        $1,673.27      $1,326.73
  22       1/15/74       $8.29      $299.78       $85.27        $1,758.54      $1,241.46
  23       2/15/74       $7.75      $307.53       $85.81        $1,844.35      $1,155.65
  24       3/15/74       $7.22      $314.75       $86.34        $1,930.69      $1,069.31
  25       4/15/74       $6.68      $321.43       $86.88        $2,017.57        $982.43
  26       5/15/74       $6.14      $327.57       $87.42        $2,104.99        $895.01
  27       6/15/74       $5.59      $333.16       $87.97        $2,192.96        $807.04
  28       7/15/74       $5.04      $338.20       $88.52        $2,281.48        $718.52
  29       8/15/74       $4.49      $342.69       $89.07        $2,370.55        $629.45
  30       9/15/74       $3.93      $346.62       $89.63        $2,460.18        $539.82
  31      10/15/74       $3.37      $349.99       $90.19        $2,550.37        $449.63
  32      11/15/74       $2.81      $352.80       $90.75        $2,641.12        $358.88
  33      12/15/74       $2.24      $355.04       $91.32        $2,732.44        $267.56
  34       1/15/75       $1.67      $356.71       $91.89        $2,824.33        $175.67
  35       2/15/75       $1.09      $357.80       $92.47        $2,916.80         $83.20
LAST PAYMENT CALCULATED AS  $83.72
  36       3/15/75       $0.52      $358.32       $83.20        $3,000.00          $0.00
```

Figure 14-6. Program TBINT Output (Sheet 2 of 2)

Lines 46 through 57 show a balanced IF statement. Lines 47 through 50 are the THEN operation that is performed for every month except the last month in the schedule. Lines 51 through 57 are the ELSE operation that is performed for the last month.

Line 58 prints the amortization schedule values just calculated for the current line of the schedule. Control then passes back to line 32 for generation of the next line of the schedule, if any.

Line 60 ends the internal procedure REPORT. Control returns to line 20 of the main external procedure. Line 20 directs control to line 9 for a read of the next set of input values, if any.

STANDARD CHARACTER SETS

A

CONTROL DATA operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

The set in use at a particular installation is specified when the operating system is installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect throughout the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card. Under NOS, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1; 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Graphic character representation on a terminal or printer depends on the installation character set and the terminal type. Characters shown in the CDC Graphic column of the standard character set table are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

PL/I	Display Code (octal)	CDC			ASCII		
		Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
: (colon)	00†	: (colon)††	8-2	00	: (colon) ††	8-2	072
A	01	A	12-1	61	A	12-1	101
B	02	B	12-2	62	B	12-2	102
C	03	C	12-3	63	C	12-3	103
D	04	D	12-4	64	D	12-4	104
E	05	E	12-5	65	E	12-5	105
F	06	F	12-6	66	F	12-6	106
G	07	G	12-7	67	G	12-7	107
H	10	H	12-8	70	H	12-8	110
I	11	I	12-9	71	I	12-9	111
J	12	J	11-1	41	J	11-1	112
K	13	K	11-2	42	K	11-2	113
L	14	L	11-3	43	L	11-3	114
M	15	M	11-4	44	M	11-4	115
N	16	N	11-5	45	N	11-5	116
O	17	O	11-6	46	O	11-6	117
P	20	P	11-7	47	P	11-7	120
Q	21	Q	11-8	50	Q	11-8	121
R	22	R	11-9	51	R	11-9	122
S	23	S	0-2	22	S	0-2	123
T	24	T	0-3	23	T	0-3	124
U	25	U	0-4	24	U	0-4	125
V	26	V	0-5	25	V	0-5	126
W	27	W	0-6	26	W	0-6	127
X	30	X	0-7	27	X	0-7	130
Y	31	Y	0-8	30	Y	0-8	131
Z	32	Z	0-9	31	Z	0-9	132
0	33	0	0	12	0	0	060
1	34	1	1	01	1	1	061
2	35	2	2	02	2	2	062
3	36	3	3	03	3	3	063
4	37	4	4	04	4	4	064
5	40	5	5	05	5	5	065
6	41	6	6	06	6	6	066
7	42	7	7	07	7	7	067
8	43	8	8	10	8	8	070
9	44	9	9	11	9	9	071
+	45	+	12	60	+	12-8-6	053
-	46	-	11	40	-	11	055
*	47	*	11-8-4	54	*	11-8-4	052
/	50	/	0-1	21	/	0-1	057
(	51	(	0-8-4	34	(	12-8-5	050
)	52	)	12-8-4	74	)	11-8-5	051
$	53	$	11-8-3	53	$	11-8-3	044
=	54	=	8-3	13	=	8-6	075
blank	55	blank	no punch	20	blank	no punch	040
, (comma)	56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
. (period)	57	. (period)	12-8-3	73	. (period)	12-8-3	056
#	60	≡	0-8-6	36	#	8-3	043
[	61	[	8-7	17	[	12-8-2	133
]	62	]	0-8-2	32	]	11-8-2	135
%	63	% ††	8-6	16	% ††	0-8-4	045
" (quote)	64	≠	8-4	14	" (quote)	8-7	042
_ (underline)	65	←	0-8-5	35	_ (underline)	0-8-5	137
|	66	v	11-0 or 11-8-2†††	52	!	12-8-7 or 11-0†††	041
&	67	∧	0-8-7	37	&	12	046
' (apostrophe)	70	↑	11-8-5	55	' (apostrophe)	8-5	047
?	71	↓	11-8-6	56	?	0-8-7	077
<	72	<	12-0 or 12-8-2†††	72	<	12-8-4 or 12-0†††	074
>	73	>	11-8-7	57	>	0-8-6	076
@	74	≤	8-5	15	@	8-4	100
\	75	≥	12-8-5	75	\	0-8-2	134
¬	76	¬	12-8-6	76	~ (circumflex)	11-8-7	136
; (semicolon)	77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

†Twelve zero bits at the end of a 60-bit word in a zero byte record are an end of record mark rather than two colons.
††In installations using a 63-graphic set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55_8).
†††The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

CDC CHARACTER SET COLLATING SEQUENCE									
Collating Sequence Decimal	Collating Sequence Octal	CDC Graphic	Display Code	External BCD	Collating Sequence Decimal	Collating Sequence Octal	CDC Graphic	Display Code	External BCD
00	00	blank	55	20	32	40	H	10	70
01	01	≤	74	15	33	41	I	11	71
02	02	%†	63 †	16 †	34	42	∨	66	52
03	03	[	61	17	35	43	J	12	41
04	04	↱	65	35	36	44	K	13	42
05	05	≡	60	36	37	45	L	14	43
06	06	∧	67	37	38	46	M	15	44
07	07	↑	70	55	39	47	N	16	45
08	10	↓	71	56	40	50	O	17	46
09	11	>	73	57	41	51	P	20	47
10	12	≥	75	75	42	52	Q	21	50
11	13	⌐	76	76	43	53	R	22	51
12	14	.	57	73	44	54	]	62	32
13	15	)	52	74	45	55	S	23	22
14	16	;	77	77	46	56	T	24	23
15	17	+	45	60	47	57	U	25	24
16	20	$	53	53	48	60	V	26	25
17	21	*	47	54	49	61	W	27	26
18	22	-	46	40	50	62	X	30	27
19	23	/	50	21	51	63	Y	31	30
20	24	,	56	33	52	64	Z	32	31
21	25	(	51	34	53	65	:	00 †	none†
22	26	=	54	13	54	66	0	33	12
23	27	≠	64	14	55	67	1	34	01
24	30	<	72	72	56	70	2	35	02
25	31	A	01	61	57	71	3	36	03
26	32	B	02	62	58	72	4	37	04
27	33	C	03	63	59	73	5	40	05
28	34	D	04	64	60	74	6	41	06
29	35	E	05	65	61	75	7	42	07
30	36	F	06	66	62	76	8	43	10
31	37	G	07	67	63	77	9	44	11

†In installations using the 63-graphic set, the % graphic does not exist and position 02 is skipped in the sequence. The : graphic is display code 63, External BCD code 16.

ASCII CHARACTER SET COLLATING SEQUENCE

Collating Sequence Decimal	Collating Sequence Octal	ASCII Graphic Subset	Display Code	ASCII Code	Collating Sequence Decimal	Collating Sequence Octal	ASCII Graphic Subset	Display Code	ASCII Code
00	00	blank	55	20	32	40	@	74	40
01	01	!	66	21	33	41	A	01	41
02	02	"	64	22	34	42	B	02	42
03	03	#	60	23	35	43	C	03	43
04	04	$	53	24	36	44	D	04	44
05	05	%†	63†	25	37	45	E	05	45
06	06	&	67	26	38	46	F	06	46
07	07	'	70	27	39	47	G	07	47
08	10	(	51	28	40	50	H	10	48
09	11	)	52	29	41	51	I	11	49
10	12	*	47	2A	42	52	J	12	4A
11	13	+	45	2B	43	53	K	13	4B
12	14	,	56	2C	44	54	L	14	4C
13	15	-	46	2D	45	55	M	15	4D
14	16	.	57	2E	46	56	N	16	4E
15	17	/	50	2F	47	57	O	17	4F
16	20	0	33	30	48	60	P	20	50
17	21	1	34	31	49	61	Q	21	51
18	22	2	35	32	50	62	R	22	52
19	23	3	36	33	51	63	S	23	53
20	24	4	37	34	52	64	T	24	54
21	25	5	40	35	53	65	U	25	55
22	26	6	41	36	54	66	V	26	56
23	27	7	42	37	55	67	W	27	57
24	30	8	43	38	56	70	X	30	58
25	31	9	44	39	57	71	Y	31	59
26	32	:	00†	3A	58	72	Z	32	5A
27	33	;	77	3B	59	73	[	61	5B
28	34	<	72	3C	60	74	\	75	5C
29	35	=	54	3D	61	75	]	62	5D
30	36	>	73	3E	62	76	^	76	5E
31	37	?	71	3F	63	77	_	65	5F

†In installations using a 63-graphic set, the % graphic does not exist and position 05 is skipped in the sequence. The : graphic is display code 63.

Diagnostic messages are messages that are intended to provide information, usually about a program error. The following types of diagnostic messages are produced as appropriate:

- Dayfile messages, including diagnostic messages for control statement errors
- Compile-time messages produced when program errors are diagnosed during program compilation
- Run-time messages produced when program errors are diagnosed during program execution

DAYFILE MESSAGES

The dayfile messages are written to the user dayfile for the job. The user dayfile provides a record of activity for the job and is printed at the end of the job listing. The messages that can be written to the dayfile are:

- Messages indicating counts of fatal and nonfatal compile-time diagnostics for each external procedure.
- Messages from CYBER Record Manager (CRM) indicating CRM errors.
- Messages issued at run time if the file SYSPRINT is not available for run-time diagnostics.
- Compiler error message issued if the compiler is probably unable to continue compilation:

  PASS n COMPILER ERROR

  Any compiler error message should be reported to a systems analyst. Compile-time diagnostics (type CMP) are also produced.
- Message indicating that a particular part of the compiler cannot be loaded:

  LDV ERROR 0004
  CANT LOAD PLI1n.

  If this message appears, increase field length and retry. If the message still appears, report the problem to a systems analyst.
- Message indicating a memory overflow:

  MEMORY OVERFLOW IN CODE TRANSFORMER.
  MEMORY OVERFLOW IN END PROCESSOR.
  MEMORY OVERFLOW IN INTERNAL ASSEMBLER.

  If one of these messages appears, increase field length and retry. If the message still appears, report to a systems analyst.
- Message indicating that the input file containing the program to be compiled cannot be found. A message of this type indicates that the input file does not exist or must be rewound:

  INPUT FILE EMPTY OR MISPOSITIONED.
- Message indicating insufficient field length for the compiler:

  PLI NEEDS AT LEAST 56000B CM FL.

  Increase the field length as indicated.
- Message indicating a compiler time limit or a mode error, either during compilation or execution.
- Message produced by the system on-unit for the ERROR condition:

  +++++ ERROR CONDITION RAISED WITH ONCODE = nnnn

  Note that many other system on-units take action to raise the ERROR condition.
- Message produced as a result of abnormal program termination, either because of STOP statement execution or a program abort:

  TERMINATED AT STMT nnn IN procedure
- Message indicating central processor (CP) time and central memory (CM) used for program compilation.
- Message indicating CP time and CM used for program execution.
- Message indicating LGO (or the name of the file specified by the B parameter) if the GO parameter is specified and the binary output file is executed.
- Messages indicating control statement errors. Messages are produced for unrecognized parameters and for incorrectly specified options. The control statement diagnostic messages are listed in table B-1.

COMPILE-TIME DIAGNOSTIC MESSAGES

The compile-time diagnostic messages are issued for program errors and for potential program errors. Compile-time diagnostics are written to the files specified by the E and L parameters of the PLI control statement, as described in section 13, Compilation.

Each compile-time diagnostic has an identifying prefix that includes a 3-character category and a 3-digit number. The 3-digit number uniquely identifies a message in a particular category. The categories are:

BIF	Builtin function errors
CMP	Compiler errors
CNV	Conversion errors
DCL	Declaration errors
EXP	Expression errors
FNI	Feature not implemented errors
MSC	Miscellaneous errors
REF	Reference errors
SYN	Syntax errors

TABLE B-1. CONTROL STATEMENT DIAGNOSTICS

Message	Significance	Programmer Action
parameter IS UNRECOGNIZABLE--IGNORED	Unknown specification is ignored.	Correct as appropriate.
parameter MAY NOT BE EQUIVALENCED	Any option is illegal for the parameter. Compilation is terminated and control passes to the next appropriate EXIT control statement.	Correct error.
parameter MAY ONLY BE EQUIVALENCE TO 0--IGNORED	Any option other than 0 is illegal. The parameter is considered unspecified.	Correct as appropriate.
lfn IS AN ILLEGAL FILE NAME	Probably an illegal character in the file name. Compilation is terminated and control passes to the next appropriate EXIT control statement.	Correct error.
lfn IS MORE THAN 7 CHARACTERS--EXCESS IGNORED	First seven characters specified are used.	Correct as appropriate.
**NO COMPILATION.	Compilation is terminated. Control passes to the next appropriate EXIT control statement.	See other messages.
**PLI CONTROL CARD ERRORS.	One or more control statement errors have been detected.	See other messages.
=WITH NO FOLLOWING VALUE--IGNORED	The parameter is considered unspecified.	Correct as appropriate.
B=0 AND GO IS INCONSISTENT	The specified parameters conflict. Compilation is terminated and control passes to the next appropriate EXIT control statement.	Correct error.
COL MUST BE COL=NN/MMM	At least the nn and mmm options must be present and separated with a slash. The parameter is considered unspecified.	Correct as appropriate.
EL=0 ILLEGAL--IGNORED	Illegal option for the EL parameter. The parameter is considered unspecified.	Correct as appropriate.
ERROR IN COL LIMITS, IGNORED	The specified nn and mmm options are not reasonable. The parameter is considered unspecified.	Correct as appropriate.
ILLEGAL DB OPTION--IGNORED	Illegal option for the DB parameter. The parameter is considered unspecified.	Correct as appropriate.
ILLEGAL EL LEVEL--IGNORED	Illegal option for the EL parameter. The parameter is considered unspecified.	Correct as appropriate.
ILLEGAL ET LEVEL--IGNORED	Illegal option for the ET parameter. The parameter is considered unspecified.	Correct as appropriate.
ILLEGAL LO OPTION--IGNORED	Illegal option for the LO parameter. The option is considered unspecified. Any other specified options are used. For example, LO=A/G/S is treated as LO=A/S.	Correct as appropriate.
INPUT FILE MAY NOT BE SUPPRESSED--IGNORED	Illegal option for the I parameter. The parameter is treated as unspecified.	Correct as appropriate.
INVALID NUMERIC FIELD IN PS ARGUMENT--IGNORED	Illegal option for the PS parameter. The parameter is treated as unspecified.	Correct as appropriate.

TABLE B-1. CONTROL STATEMENT DIAGNOSTICS (Contd)

Message	Significance	Programmer Action
MISSING NUMERIC FIELD IN PS ARGUMENT--IGNORED	Unsupplied option for the PS parameter. The parameter is considered unspecified.	Correct as appropriate.
PD ARGUMENT NOT 6 OR 8--IGNORED	Illegal value for the PD parameter. The parameter is considered unspecified.	Correct as appropriate.
PS.LT.4--IGNORED	Illegal value for the PS parameter. The parameter is considered unspecified.	Correct as appropriate.
THIRD COL VALUE OUT OF RANGE, IGNORED	Third value for the COL parameter is not valid. Third value is considered unspecified.	Correct as appropriate.
THIRD COL VALUE WITHIN THE FIRST TWO--IGNORED	Third value is not outside the nn and mmm limits. Third value is considered unspecified.	Correct as appropriate.
WARNING--TERMINATOR MISSING	The . or) is missing at the end of the control statement.	Correct error.

A compile-time diagnostic is issued for each detected error in the source program. Note that an error can cause other errors to remain undetected. Note also that an error can cause additional diagnostics to be issued for subsequent statements that would otherwise be correct.

Some errors in declarations are diagnosed only when the improperly declared identifiers are referenced. There might not be a message for the erroneous DECLARE statement.

The compile-time diagnostic messages are listed in table B-2. The appearance of -identifier- in a message in the table indicates that the identifier is inserted into the message if the identifier is known when the message is produced.

RUN-TIME DIAGNOSTIC MESSAGES

The run-time diagnostic messages are issued for program errors and for unusual conditions that occur during program execution. Run-time diagnostic messages are written to a file or to the dayfile. File SYSPRINT is used if there is an external file constant named SYSPRINT which is open, or can be opened for stream output, and its line size is at least 45.

The CRM file OUTPUT is used if SYSPRINT cannot be used; if it is not already being used by the PL/I program; it is open, or can be opened for stream output; and its line size is at least 45. If OUTPUT is used for run-time diagnostics, it is no longer available to the PL/I program.

If neither SYSPRINT nor OUTPUT can be used for run-time diagnostics, the dayfile is used.

For each error or unusual condition that occurs during program execution, a condition is raised with a specific ONCODE value. The ONCODE values associated with each condition are:

AREA	360-362
CONDITION	500
CONVERSION	600-625
ENDFILE	70-72
ENDPAGE	90
ERROR	3, 9, 1000-9999
FINISH	4
FIXEDOVERFLOW	310-313
KEY	50-57
OVERFLOW	300-307
RECORD	20-25
SIZE	340-343
STORAGE	370-379
STRINGRANGE	350-352
SUBSCRIPTRANGE	520-523
TRANSMIT	40-44
UNDEFINEDFILE	80-89, 91-92
UNDERFLOW	330-337
ZERODIVIDE	320-321

ON statements in the program can specify whether the system on-unit or a programmed on-unit is to be executed when a specific condition is raised. The production of snap output is also specified by the ON statements in the program. The production of the run-time diagnostic message and the snap output is determined by the current established on-unit in the following way:

- System on-unit without SNAP option: the diagnostic is produced, unless the condition is ENDPAGE or FINISH, in which case no diagnostic is produced.
- System on-unit with SNAP option: the diagnostic and snap output are produced, unless the condition is ENDPAGE or FINISH, in which case only the snap output is produced.
- Programmed on-unit without SNAP option: no diagnostic or snap output is produced.
- Programmed on-unit with SNAP option: no diagnostic is produced, but the snap output is produced.
- No on-unit established by ON statement execution: system on-unit without SNAP option, except for the ERROR condition. For the ERROR condition, the system on-unit produces snap output. Many other system on-units take action to raise the ERROR condition, in which case ERROR is raised with the ONCODE value belonging to the original condition that was raised.

The run-time diagnostic messages are listed by ONCODE value in table B-3. The appearance of -number- in a message in the table indicates that the number is inserted into the message.

TABLE B-2. COMPILE-TIME DIAGNOSTICS

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
BIF001	F	WRONG NUMBER OF ARGUMENTS TO -identifier- BUILTIN FUNCTION	As stated.	Correct error.
BIF003	F	IMPROPER ARGUMENT -identifier- TO -identifer- BUILTIN FUNCTION	Argument has wrong data type or wrong aggregate type.	Correct error.
BIF005	F	ARGUMENT -identifier- TO -identifier- MATH BUILTIN FUNCTION MUST BE SCALAR	As stated.	Correct error.
BIF006	F	ARGUMENT -identifier- TO -identifier- MATH BUILTIN FUNCTION MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
BIF009	F	ARGUMENT -identifier- TO VALID BUILTIN FUNCTION MUST BE PICTURED	As stated.	Correct error.
BIF011	F	PRECISION P IN -identifier- BUILTIN FUNCTION CANNOT BE SIGNED OR ZERO	As stated.	Correct error.
BIF012	F	PRECISION P OR SCALE FACTOR Q IN -identifier- BUILTIN FUNCTION MUST BE AN INTEGER	As stated.	Correct error.
BIF015	F	-identifier- BUILTIN FUNCTION CANNOT BE USED AS SUBROUTINE-REFERENCE IN CALL STATEMENT	As stated.	Correct error.
CMPnnn	C	Any message text	Compiler error. The text of the message is not necessarily meaningful to the user. An error occurred in the compiler, and the compiler is probably unable to continue compilation of the source program.	If the program has other compilation errors, fix those errors and then retry. If the message persists, report the problem to a systems analyst.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
CNV001	F	OPERAND -identifier- MUST BE COMPUTATIONAL TYPE	All operands in an expression must be computational, except that locator values can be compared for equality.	Correct error.
CNV002	F	VALUE ASSIGNED TO PICTURED VARIABLE -identifier- MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV003	F	VALUE ASSIGNED TO ARITHMETIC VARIABLE -identifier- MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV004	F	VALUE ASSIGNED TO BIT VARIABLE -identifier- MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV005	F	VALUE ASSIGNED TO CHARACTER VARIABLE -identifier- MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV006	F	VALUE ASSIGNED TO AREA VARIABLE -identifier- MUST BE AN AREA VALUE	As stated.	Correct error.
CNV007	F	VALUE ASSIGNED TO LOCATOR VARIABLE -identifier- MUST BE A LOCATOR VALUE	As stated.	Correct error.
CNV008	F	VALUE ASSIGNED TO LABEL VARIABLE -identifier- MUST BE A LABEL VALUE	As stated.	Correct error.
CNV009	F	SUBSCRIPT OR EXTENT -identifier- MUST BE SCALAR AND OF COMPUTATIONAL TYPE	The statement contains a noncomputational or nonscalar subscript in a reference or expression, or the statement contains a reference to a controlled, based, or defined variable whose declaration contains a noncomputational or nonscalar extent or subscript.	Correct error.
CNV010	F	DECLARATION OF -identifier-: REFER OPTION VARIABLE -identifier- MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV011	F	WHILE-EXPRESSION MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV012	F	TO-EXPRESSION MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV013	F	BY-EXPRESSION MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV014	F	EXPRESSION IN IF STATEMENT MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV015	F	ARGUMENT -identifier- PASSED TO ENTRY PARAMETER MUST BE AN ENTRY VALUE	As stated. Argument might be a structure that has no data type.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
CNV016	F	ARGUMENT -identifier- PASSED TO ARITHMETIC PARAMETER MUST BE OF COMPUTATIONAL TYPE	As stated. Argument might be a structure that has no data type.	Correct error.
CNV017	F	ARGUMENT -identifier- PASSED TO CHARACTER PARAMETER MUST BE OF COMPUTATIONAL TYPE	As stated. Argument might be a structure that has no data type.	Correct error.
CNV018	F	ARGUMENT -identifier- PASSED TO BIT PARAMETER MUST BE OF COMPUTATIONAL TYPE	As stated. Argument might be a structure that has no data type.	Correct error.
CNV019	F	ARGUMENT -identifier- PASSED TO LABEL PARAMETER MUST BE A LABEL VALUE	As stated. Argument might be a structure that has no data type.	Correct error.
CNV020	F	ARGUMENT -identifier- PASSED TO LOCATOR PARAMETER MUST BE A LOCATOR VALUE	As stated. Argument might be a structure that has no data type.	Correct error.
CNV021	F	ARGUMENT -identifier- PASSED TO AREA PARAMETER MUST BE AN AREA VALUE	As stated. Argument might be a structure that has no data type.	Correct error.
CNV022	F	STRUCTURE OF ARGUMENT -identifier- MUST MATCH STRUCTURE OF PARAMETER	Argument and parameter must be structures with the same aggregate type, and with corresponding alignment and data types for the members.	Correct error.
CNV023	F	ARGUMENT -identifier- PASSED TO SCALAR PARAMETER MUST BE SCALAR	Argument cannot be an array or structure.	Correct error.
CNV024	F	RETURN-VALUE CANNOT BE CONVERTED TO TYPE SPECIFIED BY ANY RETURNS-DESCRIPTOR	The innermost containing procedure block does not have any entry point with a returns descriptor that is compatible with the data type of the return-value in the RETURN statement.	Correct the RETURN statement, or modify at least one returns descriptor in the immediately containing procedure.
CNV025	W	RETURN-VALUE CANNOT BE CONVERTED TO TYPE SPECIFIED BY RETURNS-DESCRIPTOR OF AT LEAST ONE ENTRY	Unless the return-value in the RETURN statement is returned during an activation invoked through a compatible entry point, the ERROR condition is raised.	Correct as appropriate.
CNV026	W	CONSTANT IS COMPILED AS ZERO	As stated.	Correct as appropriate.
CNV027	W	CONSTANT IS COMPILED AS INFINITE	As stated.	Correct as appropriate.
CNV028	F	I/O EXPRESSION OR REFERENCE -identifier- MUST BE OF COMPUTATIONAL TYPE	The error is in a statement option or format specification.	Correct error.
CNV029	F	ARGUMENT -identifier- PASSED TO FILE PARAMETER MUST BE A FILE VALUE	As stated. Argument might be a structure that has no data type.	Correct error.
CNV030	F	DECLARATION OF -identifier-: EXTENT -identifier- MUST BE SCALAR AND OF COMPUTATIONAL TYPE	As stated.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
CNV031	F	VALUE OF LOCATOR MUST BE SCALAR AND LOCATOR	In a reference to a based variable, the explicit locator or implicit locator must be scalar. Also, arrays of locators cannot be compared for equality.	Correct error.
CNV032	F	ARITHMETIC OPERATION OR CONVERSION GENERATES ILLEGAL PRECISION	The operation or assignment requires creation of an intermediate value that will raise SIZE or FIXEDOVERFLOW, or will create an illegal scale factor for a fixed point intermediate value.	Correct error.
CNV033	F	ARGUMENT -identifier- PASSED TO PICTURED PARAMETER MUST BE OF COMPUTATIONAL TYPE	As stated. Argument might be a structure that has no data type.	Correct error.
CNV034	F	EXPRESSION IN KEY OR KEYFROM OPTION MUST BE SCALAR AND OF COMPUTATIONAL TYPE	As stated.	Correct error.
CNV035	F	EXPRESSION IN IGNORE OPTION MUST BE SCALAR AND OF COMPUTATIONAL TYPE	As stated.	Correct error.
DCL001	F	ENVIRONMENT ATTRIBUTE IN DECLARATION OF -identifier- MUST CONTAIN A CHARACTER CONSTANT	The ENVIRONMENT attribute in a DECLARE statement must specify a character constant. The ENVIRONMENT option in an OPEN or CLOSE statement can specify a character variable or any other computational expression.	Correct error.
DCL002	F	DECLARATION OF -identifier-: VARIABLE -identifier- IN REFER OPTION MUST BE A PRECEDING MEMBER OF SAME STRUCTURE	The variable in a REFER option must be an earlier member of the same based structure as the variable whose extent contains the REFER option.	Correct error.
DCL003	F	DECLARATION OF -identifier-: REFER OPTION IS ALLOWED ONLY IN EXTENT OF MEMBER OF BASED STRUCTURE	The REFER option cannot be used for a structure with a storage type other than BASED.	Correct error.
DCL004	F	DECLARATION OF -identifier-: VARIABLE -identifier- IN REFER OPTION MUST BE SCALAR	The refer-variable in the extent must be scalar.	Correct error.
DCL005	F	DECLARATION OF -identifier-: ASTERISK SUBSCRIPT ILLEGAL IN ISUB DEFINING	The host-reference in a DEFINED attribute cannot contain both an asterisk subscript and an iSUB. The asterisk extent is not permitted in iSUB defining.	Correct error.
DCL006	F	DECLARATION OF -identifier-: CONTAINS AN ILLEGAL USE OF ISUB	An iSUB cannot be used in any function reference contained in the host-reference.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
DCL007	F	DECLARATION OF -identifier-: I IN ISUB EXCEEDS NUMBER OF DIMENSIONS OF DEFINED VARIABLE	As stated.	Correct error.
DCL008	F	DECLARATION OF -identifier-: POSITION AND ISUB ARE INCOMPATIBLE	Declaration of a defined variable cannot contain both an iSUB in the host-reference and a POSITION attribute. The iSUB is permitted only in iSUB defining; the POSITION attribute is permitted only in string overlay defining.	Correct error.
DCL009	F	HOST AND DEFINED VARIABLES ARE NOT COMPATIBLE FOR ISUB DEFINING	Both the host variable and the defined variable must be arrays of scalars. Structures are not permitted.	Correct error.
DCL010	F	DECLARATION OF -identifier-: HOST VARIABLE -identifier- HAS ILLEGAL TYPE FOR DEFINING	The host variable cannot be a label constant, entry constant, file constant, condition, or builtin function.	Correct error.
DCL011	F	DECLARATION OF -identifier-: HOST VARIABLE -identifier- CANNOT BE BASED OR DEFINED	As stated.	Correct error.
DCL012	F	DECLARATION OF -identifier-: HOST VARIABLE -identifier- CANNOT BE OR CONTAIN A VARYING STRING	As stated.	Correct error.
DCL014	F	DECLARATION OF -identifier-: HOST VARIABLE -identifier-: CANNOT BE A MEMBER OF AN ARRAY OF STRUCTURES	As stated.	Correct error.
DCL015	F	DECLARATION OF -identifier: HOST REFERENCE -identifier- HAS WRONG NUMBER OF DIMENSIONS	The host variable in simple defining must have the same number of dimensions as the defined variable. If the host reference is subscripted, then it must have as many asterisk subscripts as the defined variable has dimensions.	Correct error.
DCL016	F	DECLARATION OF -identifier-: SIMPLE DEFINING IS NOT APPLICABLE AND DATATYPE IS ILLEGAL FOR STRING OVERLAY DEFINING	As stated. The defined variable is not defined in one of the three possible ways.	Correct error.
DCL018	F	DECLARATION OF -identifier-: SIMPLE DEFINING IS NOT APPLICABLE AND ALIGNED ITEMS CANNOT BE STRING OVERLAY DEFINED	As stated. The defined variable is not defined in one of the three possible ways.	Correct error.
DCL019	F	DECLARATION OF -identifier-: SIMPLE DEFINING IS NOT APPLICABLE AND THIS COMBINATION OF DATATYPES IS ILLEGAL IN STRING OVERLAY DEFINING	As stated. The defined variable is not defined in one of the three possible ways.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
DCL020	F	ASTERISK CANNOT BE USED IN EXTENTS OF BASED VARIABLE -identifier-	As stated. Check the declaration.	Correct error.
DCL021	F	DECLARATION OF -identifier-: DEFINED VARIABLE CANNOT BE OR CONTAIN A VARYING STRING	As stated.	Correct error.
DCL022	F	DECLARATION OF -identifier-: ASTERISK EXTENT IS PERMITTED ONLY FOR PARAMETERS	As stated.	Correct error.
DCL023	F	STATIC LABEL VARIABLE -identifier- CANNOT BE INITIALIZED	A label variable with storage type STATIC cannot have an INITIAL attribute, because a label value contains information about a particular block activation, and no block activations exist at the time when static variables are initialized.	Remove INITIAL attribute or change the storage type.
DCL024	F	SCALAR VARIABLE -identifier- CANNOT HAVE MULTIPLE INITIAL VALUES	An INITIAL attribute with an iteration factor or with a list of initial values separated by commas can only be used for an array.	Correct error.
DCL025	F	DECLARATION OF -identifier-: INITIAL CANNOT BE USED WITH DEFINED OR PARAMETER	Variables with storage type DEFINED or parameter cannot have the INITIAL attribute because they share storage and are not allocated.	Remove INITIAL attribute.
DCL026	F	ATTRIBUTE CONFLICT IN DECLARATION OF -identifier-	Two conflicting or identical attributes have been explicitly declared for the diagnosed variable. Most conflicts involve illegal keyword combinations within a DECLARE statement declaration. Repeated attribute keywords are illegal in the DECLARE statement.	Remove conflict, or remove duplicate.
DCL027	F	IDENTIFIER -identifier- HAS MULTIPLE DECLARATIONS IN THE SAME BLOCK	As stated.	Remove additional declarations.
DCL028	W	EXTERNAL FILE CONSTANT -identifier- HAS MULTIPLE DECLARATIONS IN THE SAME EXTERNAL PROCEDURE	As stated. One of the declarations might be contextual. File description attributes in the diagnosed declaration have been ignored; those in the declaration not diagnosed have been used instead.	Move the explicit declaration to the outermost block.
DCL029	F	DECLARATION OF -identifier-: ITERATION FACTOR IN INITIAL ATTRIBUTE MUST BE OF COMPUTATIONAL TYPE	As stated.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
DCL030	F	DECLARATION OF -identifier-: ITERATION FACTOR IN INITIAL ATTRIBUTE MUST BE SCALAR	As stated.	Correct error.
DCL031	F	DECLARATION OF -identifier-: REFERENCE OR EXPRESSION IN INITIAL ATTRIBUTE MUST BE SCALAR	As stated, even for initialization of array elements.	Correct error.
DCL032	F	DECLARATION OF -identifier-: REFERENCE OR EXPRESSION IN INITIAL ATTRIBUTE CANNOT BE OF TYPE ENTRY, FILE, FORMAT, OR CONDITION	As stated.	Correct error.
DCL033	W	PRECISION OF -identifier- EXCEEDS FLOAT BINARY MAXIMUM OF 48	The diagnosed variable has been declared FIXED BINARY with a precision p>48. The compiler has reduced the precision to 48, leaving the scale factor q unchanged.	Check the program to ensure that SIZE will not be raised during execution.
DCL034	T	PRECISION OF -identifier- EXCEEDS FLOAT BINARY MAXIMUM OF 48	The diagnosed variable has been declared FLOAT BINARY with a precision p>48. The compiler has reduced the precision to 48.	Check the program to ensure that 48 digits of precision are sufficient.
DCL035	W	PRECISION OF -identifier- EXCEEDS FIXED DECIMAL MAXIMUM OF 14	The diagnosed variable has been declared FIXED DECIMAL with a precision p>14. The compiler has reduced the precision to 14.	Check the program to ensure that SIZE will not be raised during execution.
DCL036	T	PRECISION OF -identifier- EXCEEDS FLOAT DECIMAL MAXIMUM OF 14	The diagnosed variable has been declared FLOAT DECIMAL with a precision p>14. The compiler has reduced the precision to 14.	Check the program to ensure that 14 digits of precision are sufficient.
DCL037	F	SCALE FACTOR OF -identifier- OUTSIDE ALLOWABLE RANGE OF +/-255	As stated.	Correct error.
DCL038	F	ABSOLUTE VALUE OF ARRAY BOUND * STRIDE IS TOO LARGE	The restriction applies to every array whose data type is BIT nonvarying, CHARACTER nonvarying, or PICTURE. The stride is a measure of the distance between array elements in storage.	Change the lower bound, the corresponding upper bound, and subscripts in all references to the array.
DCL040	F	DECLARATION OF -identifier- CONTAINS CIRCULAR DEPENDENCIES	The diagnosed declaration references itself, directly or indirectly.	Correct error.
DCL041	F	EXTERNAL DECLARATIONS OF -identifier- CONTAIN CONFLICTING ATTRIBUTES	The external procedure currently being compiled contains two declarations of the same identifier with the EXTERNAL attribute and with conflicting attributes. The declarations are not in the same block.	Remove conflict.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
DCL042	F	NUMBER OF DIMENSIONS OF -identifier- EXCEEDS 32	An identifier cannot have more than 32 dimensions, including dimensions inherited from containing structures.	Correct error.
DCL043	F	SUBSCRIPTED LABEL CONSTANT -identifier- MUST BE DECLARED AS LABEL ARRAY BY DECLARE STATEMENT IN SAME BLOCK	A subscripted label constant appears in the block, and there must be a corresponding DECLARE statement declaration of the label constant array.	Correct error.
DCL044	F	LIKE ATTRIBUTE CANNOT REFERENCE A STRUCTURE DECLARED WITH LIKE	As stated.	Correct error.
DCL045	F	LIKE ATTRIBUTE MUST REFERENCE A STRUCTURE	As stated.	Correct error.
DCL046	F	PARAMETER -identifier- OCCURS MORE THAN ONCE IN PARAMETER LIST	As stated.	Correct error.
DCL047	T	DECLARATION OF -identifier- DEPENDS ON AN AUTOMATIC OR DEFINED VARIABLE DECLARED IN THE SAME BLOCK	The diagnosed identifier is declared with storage type AUTOMATIC or DEFINED and the declaration contains a reference to another automatic or defined variable declared in the same block.	None, unless strict conformance to the ANSI standard is intended.
DCL048	W	STATIC EXTERNAL VARIABLE -identifier- CAN ONLY BE INITIALIZED IN OUTERMOST BLOCK OF MAIN PROCEDURE	As stated. The INITIAL attribute has been ignored by by the compiler.	Move the initialization to the outermost block of the main external procedure.
DCL049	F	ABSOLUTE VALUE OF ARRAY BOUND GREATER THAN 131071	As stated.	Correct error.
DCL050	F	DECLARATION OF -identifier-: PARAMETER EXTENT MUST BE LITERAL CONSTANT OR ASTERISK	As stated.	Correct error.
DCL051	T	STRUCTURE MUST HAVE AT LEAST ONE MEMBER	As stated.	Correct error.
DCL052	F	DECLARATION OF -identifier- DEPENDS ON AN AUTOMATIC OR DEFINED VARIABLE	As stated.	Correct error.
EXP001	F	AGGREGATE VALUE CANNOT BE ASSIGNED TO SCALAR VARIABLE	As stated.	Correct error.
EXP002	F	RETURN-VALUE CANNOT BE OF TYPE LABEL, ENTRY, FILE, FORMAT, OR CONDITION	As stated.	Correct error.
EXP003	F	START-EXPRESSION IN DO-SPECIFICATION MUST BE SCALAR	As stated.	Correct error.
EXP004	F	BY-EXPRESSION IN DO-SPECIFICATION MUST BE SCALAR	As stated.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
EXP005	F	TO-EXPRESSION IN DO-SPECIFICATION MUST BE SCALAR	As stated.	Correct error.
EXP006	F	WHILE-EXPRESSION IN DO-SPECIFICATION MUST BE SCALAR	As stated.	Correct error.
EXP007	F	EXPRESSION IN IF STATEMENT MUST BE SCALAR	As stated.	Correct error.
EXP008	T	PRECISION P FOR CONSTANT HAS BEEN REDUCED TO LEGAL MAXIMUM	The diagnosed constant has more than 14 decimal digits or 48 binary digits. The 14 or 48 high order digits have been kept; all others have been discarded by the compiler.	Correct as appropriate.
EXP009	F	I/O EXPRESSION OR REFERENCE MUST BE SCALAR	As stated. The error is in a statement option or format specification.	Correct error.
EXP010	F	SCALE FACTOR Q OUT OF RANGE	As stated.	Correct error.
EXP011	F	PRECISION P FOR FIXED POINT CONSTANT EXCEEDS LEGAL MAXIMUM	As stated.	Correct error.
FNI001	F	RETURN-VALUE MUST BE SCALAR	Aggregate return values are not supported.	Correct error.
FNI002	F	DATA DIRECTED I/O NOT SUPPORTED	As stated.	Correct error.
FNI003	F	BY NAME ASSIGNMENT NOT SUPPORTED	As stated.	Correct error.
FNI004	F	AGGREGATE ARGUMENT -identifier- CANNOT BE PASSED TO UNLIKE PARAMETER	An aggregate argument cannot be passed except by reference. Creation of aggregate dummy arguments is not supported.	Correct error.
FNI005	F	SCALAR ARGUMENT -identifier- CANNOT BE PASSED TO ARRAY PARAMETER	Promotion of arguments is not supported.	Correct error.
FNI006	T	DECLARATION OF -identifier-: UNALIGNED ATTRIBUTE DOES NOT CAUSE PACKING OF VALUES FOR THIS DATATYPE	The UNALIGNED attribute causes packing of values in storage only for the data types CHARACTER nonvarying, BIT nonvarying, and PICTURE. The packing of values is not supported for UNALIGNED values with other data types.	None.
FNI007	F	MEMBER OF ARRAY OF STRUCTURES CANNOT BE INITIALIZED	Initialization of unconnected storage is not supported.	Recognize the aggregate or use assignment statements to do the initialization. The INITIAL attribute can be used for a dimensioned member of a structure; INITIAL only conflicts with inherited dimensions.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
FNI008	F	INPUT-TARGET VARIABLE OR OUTPUT-SOURCE EXPRESSION MUST BE SCALAR	Stream I/O of aggregates is not supported.	Replace each array reference with an embedded do-specification. Replace each structure reference with a list of the members.
MSC001	F	ASSIGNMENT-TARGET CANNOT BE OF TYPE ENTRY, FILE, FORMAT, OR CONDITION, AND CANNOT BE NAMED CONSTANT	As stated.	Correct error.
MSC002	F	EXPRESSION IN ASSIGNMENT STATEMENT CANNOT BE OF TYPE ENTRY, FILE, FORMAT, OR CONDITION	As stated.	Correct error.
MSC003	F	OPERATOR IN COMPARISON OF LOCATOR VALUES MUST BE = OR ¬ =	As stated.	Correct error.
MSC004	F	I/O OPERATION INCOMPATIBLE WITH ATTRIBUTES DECLARED FOR FILE	The I/O statement is not compatible with the file description attributes for the file.	Remove conflict by changing the declaration or changing the I/O statement.
MSC005	F	REFERENCE -identifier- IN REMOTE FORMAT ITEM MUST BE A LOCAL REFERENCE TO A FORMAT CONSTANT	As stated.	Correct error.
MSC006	F	INVALID USE OF OPTIONS(MAIN)	OPTIONS(MAIN) is permitted only in the PROCEDURE statement that heads the main external procedure.	Correct error.
MSC007	F	CONDITION -identifier- CANNOT BE PASSED AS ARGUMENT	As stated.	Correct error.
MSC010	F	TOO MANY IDENTIFIERS IN PROGRAM - IDENTIFIER TABLE OVERFLOW	The number of identifiers in this external procedure exceeds the maximum allowed.	Reorganize and separate this external procedure into two more external procedures.
MSC011	F	BAD LABEL OR FORMAT PREFIX -identifier-	The label or format name is explicitly declared with conflicting attributes in the same block.	Correct error.
MSC012	F	UNDECLARED ARRAY OR EXTERNAL FUNCTION -identifier	An identifier for which no explicit declaration exists, and which is not the name of a builtin function, has been referenced with arguments or subscripts.	Declare the identifier according to the intended use.
MSC013	F	SUBROUTINE-REFERENCE -identifier- IN CALL STATEMENT MUST BE OF TYPE ENTRY	As stated.	Remove conflicting declaration, or supply a DECLARE statement declaration for the external entry constant.
MSC014	F	PRECISION P OF VARIABLE -identifier- MUST BE UNSIGNED NON-ZERO INTEGER	As stated.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
MSC016	F	ARGUMENT IN FUNCTION REFERENCE -identifier- CANNOT BE AN ASTERISK	As stated.	Correct error.
MSC020	F	DECLARATION OF LABEL CONSTANT ARRAY -identifier- : EXTENTS MUST BE INTEGER CONSTANTS	All upper and lower bounds for dimensions of the label constant array must be specified as optionally signed decimal integer constants.	Correct error.
MSC021	F	ILLEGAL I/O OPTION COMBINATION	The combination of options specified in the I/O statement is illegal. There is no type of file upon which the specified operation can be performed.	Correct error.
MSC022	F	RECORD I/O STATEMENT MUST HAVE FILE-OPTION	As stated.	Correct error.
MSC023	F	ILLEGAL FILE-DESCRIPTION ATTRIBUTE COMBINATION IN OPEN STATEMENT	As stated.	Correct error.
MSC024	F	CHARACTER CONFLICT IN PICTURE	The picture specification is not legal.	Correct error.
MSC025	F	NUMBER OF DIGITS IN PICTURE EXPONENT MUST BE ONE, TWO, OR THREE	As stated for a pictured numeric floating point specification.	Correct error.
MSC026	F	NO DIGITS IN PICTURE MANTISSA	As stated for a pictured numeric floating point specification.	Correct error.
MSC027	F	CONFLICTING CONDITION PREFIXES	The condition prefixes on a statement cannot contain an enabled condition name and a disabled condition name for the same condition.	Correct error.
MSC028	F	DIVISION BY ZERO ILLEGAL	As stated.	Correct error.
MSC030	F	CONSTANT EXTENTS REQUIRED	As stated.	Correct error.
MSC031	F	SUBSCRIPT OF REFERENCE -identifier- IS OUT OF RANGE	As stated. If the diagnosed statement is a DECLARE statement, and -identifier- is an array of label constants, then the dimension attribute is inconsistent with a label prefix for the same array.	Correct error.
MSC032	F	ARRAY -identifier- UPPER BOUND LESS THAN LOWER BOUND	As stated.	Correct error.
MSC033	F	AREA -identifier- SIZE MUST BE BETWEEN 0 AND 131071	As stated.	Correct error.
MSC034	F	STRING -identifier- LENGTH MUST BE BETWEEN 0 AND 131071	As stated.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
MSC035	F	ILLEGAL FLOATING POINT CONSTANT	The statement contains a float decimal or float binary literal constant with a value outside the legal range.	Correct error.
MSC036	F	THIS STATEMENT CANNOT BE COMPILED BECAUSE OF UNRELATED ERROR IN SOME PREVIOUS SOURCE STATEMENT	As stated.	Correct the other diagnosed error.
MSC037	F	PROCEDURE CONTAINS TOO MANY AUTOMATIC VARIABLES	Total storage requirements of automatic variables and temporaries in a procedure and its contained begin-blocks exceeds 120 000 words.	Correct error.
MSC038	F	ENTRY CONSTANT OR PARAMETER -identifier- CAN ONLY BE USED FOR PROCEDURE INVOCATION AND AS ARGUMENT PASSED TO ENTRY PARAMETER	As stated. Note that a function reference must have the parentheses for the argument list even if no arguments are passed.	Correct error.
MSC039	F	SUBROUTINE-REFERENCE -identifier- IN CALL STATEMENT CANNOT BE AN ENTRY POINT WITH A RETURNS-DESCRIPTOR	As stated. A CALL statement can invoke a procedure at a subroutine entry point, not a function entry point.	Correct error.
MSC040	F	FUNCTION-REFERENCE -identifier- MUST BE AN ENTRY POINT WITH A RETURNS-DESCRIPTOR	As stated. A function reference can invoke a procedure at a function entry point, not a subroutine entry point.	Correct error.
MSC041	F	FORM OF RETURN STATEMENT IS INCOMPATIBLE WITH ALL ENTRY POINTS	A RETURN statement with a return-value can only be used to return from a procedure invoked at a function entry point. A RETURN statement without a return-value can only be used to return from a procedure invoked at a subroutine entry point. The diagnosed RETURN statement is incompatible with all entry points.	Correct error.
MSC042	W	FORM OF RETURN STATEMENT IS INCOMPATIBLE WITH AT LEAST ONE ENTRY	A RETURN statement with a return-value can only be used to return from a procedure invoked at a function entry point. A RETURN statement without a return value can only be used to return from a procedure invoked at a subroutine entry point. The diagnosed RETURN statement is incompatible with at least one entry point, but not all entry points, of the innermost containing procedure. If the RETURN statement is executed during an activation for which it is not appropriate, the ERROR condition is raised.	Correct as appropriate.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
MSC043	F	FUNCTION -identifier DOES NOT CONTAIN SCALAR RETURN STATEMENT	A procedure which has one or more function entry points must contain a RETURN statement with a return-value.	Correct error.
MSC044	F	PROCEDURE WITH OPTIONS MAIN CANNOT HAVE PARAMETERS OR RETURNS DESCRIPTOR	The main external procedure cannot receive parameters or return a value.	Correct error.
MSC045	F	THE MAXIMUM NUMBER OF CHARACTERS IN A PICTURE EXPONENT IS 510	As stated for a pictured numeric floating point specification.	Correct error.
MSC046	W	EXTERNAL NAMES -identifier- -identifier- TRUNCATE TO SAME NAME	The two specified identifiers are both declared with the EXTERNAL attribute, and each is at least 7 characters long. Both begin with the same 4 characters and end with the same 3 characters. The operating system will treat them as identical.	Correct as appropriate.
MSC047	W	EXTERNAL NAME -identifier- CONTAINS SPECIAL CHARACTER	External names should not contain the special character $ # @ or _. Use of a special character can cause conflict with system names or raise the UNDEFINEDFILE condition.	Correct as appropriate.
MSC048	W	LEADING SPECIAL CHARACTER IN EXTERNAL NAME -identifier- CHANGED TO A -3-	External names should not contain the special characters $ # @ or _. Special character changed to a 3.	Correct error.
MSC049	F	RETURN FROM AN ON-UNIT MUST NOT HAVE A RETURN VALUE	As stated.	Correct error.
MSC050	F	LENGTH OF THE PICTURE-SPECIFICATION IN P FORMAT ITEM CANNOT EXCEED 1000 CHARACTERS	As stated.	Correct error.
MSC052	C	COMPILATION ABORTED BECAUSE OF COMPILER ERROR	Error in compiler. May have been caused by other compile-time errors.	Correct any other errors and recompile. If error persists, see systems analyst.
REF001	F	ASSIGNMENT-TARGET MUST BE VARIABLE OR PSEUDOVARIABLE	As stated.	Correct error.
REF002	F	ASSIGNMENT-TARGET MUST BE SCALAR	As stated.	Correct error.
REF003	F	INDEX -identifier- IN DO MUST BE VARIABLE OR PSEUDOVARIABLE	As stated.	Correct error.
REF004	F	INDEX -identifier- IN DO MUST BE SCALAR	As stated.	Correct error.
REF006	F	WRONG NUMBER OF ARGUMENTS IN FUNCTION REFERENCE -identifier-	As stated.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
REF007	F	SUBROUTINE-REFERENCE -identifier- HAS WRONG NUMBER OF ARGUMENTS	As stated.	Correct error.
REF008	F	ARGUMENT -identifier- AND PARAMETER MUST HAVE SAME NUMBER OF DIMENSIONS	As stated for a subscripted reference used for an argument.	Correct error.
REF009	F	REFERENCE TO VARIABLE -identifier- HAS WRONG NUMBER OF SUBSCRIPTS	As stated.	Correct error.
REF010	F	BASED REFERENCE -identifier- REQUIRES A LOCATOR-QUALIFIER OR AN IMPLICIT LOCATOR	As stated.	Correct error by supplying a locator-qualifier or specifying an implicit locator in the declaration of the based variable.
REF011	F	REFERENCE TO A NON-BASED VARIABLE -identifier- CANNOT BE LOCATOR-QUALIFIED	As stated.	Correct error.
REF012	F	SCALE-FACTOR Q ILLEGAL IN -identifier- BUILTIN FUNCTION REFERENCE BECAUSE RESULT IS FLOAT	Since all arguments to the builtin function are not FIXED, the result is FLOAT and a scale factor is not appropriate.	Correct error by removing the scale factor or changing the data types of the arguments.
REF013	F	VARIABLE -identifier- BEING FREED MUST BE UNSUBSCRIPTED	As stated.	Correct error.
REF014	F	VARIABLE -identifier- BEING FREED MUST BE BASED OR CONTROLLED	As stated.	Correct error.
REF015	F	VARIABLE -identifier- BEING FREED CANNOT BE A MEMBER OF A STRUCTURE	As stated.	Correct error.
REF016	F	VARIABLE -identifier- BEING ALLOCATED CANNOT BE A MEMBER OF A STRUCTURE	As stated.	Correct error.
REF017	F	VARIABLE -identifier- BEING ALLOCATED MUST BE BASED OR CONTROLLED	As stated.	Correct error.
REF018	F	IN-OPTION IN FREE STATEMENT MUST BE SCALAR	As stated.	Correct error.
REF019	F	IN-OPTION IN FREE STATEMENT MUST REFERENCE AN AREA VARIABLE	As stated.	Correct error.
REF020	F	IN-OPTION IS ILLEGAL IN FREEING A CONTROLLED VARIABLE	As stated.	Correct error.
REF021	F	IN-OPTION IN ALLOCATE STATEMENT MUST REFERENCE AN AREA VARIABLE	As stated.	Correct error.
REF022	F	IN-OPTION IN ALLOCATE STATEMENT MUST BE SCALAR	As stated.	Correct error.
REF023	F	IN-OPTION MUST REFERENCE A VARIABLE	As stated.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
REF024	F	SET-OPTION IN READ OR LOCATE STATEMENT MUST REFERENCE A SCALAR POINTER VARIABLE	As stated.	Correct error.
REF025	F	SET-OPTION MUST REFERENCE A VARIABLE	As stated.	Correct error.
REF026	F	SET-OPTION IN ALLOCATE STATEMENT MUST REFERENCE A SCALAR VARIABLE	As stated.	Correct error.
REF027	F	SET-OPTION IN ALLOCATE STATEMENT MUST REFERENCE A LOCATOR VARIABLE	As stated.	Correct error.
REF028	F	GOTO STATEMENT MUST REFERENCE A LABEL CONSTANT OR VARIABLE	As stated.	Correct error.
REF029	F	LABEL-REFERENCE IN GOTO STATEMENT MUST BE SCALAR	As stated.	Correct error.
REF030	F	IDENTIFIER IN PROGRAMMER-NAMED CONDITION MUST BE OF TYPE CONDITION	The identifier in a programmer-named condition cannot have an explicit declaration in the block in which the condition is used, or in any containing block.	Remove declaration.
REF033	F	OPERAND -identifier- MUST BE SCALAR	All operands of all operators must be scalar; arrays and structures are not permitted as operands.	Correct error.
REF034	F	REFERENCE IN I/O CONDITION MUST BE A FILE CONSTANT OR VARIABLE	As stated.	Correct error.
REF036	F	REFERENCE -identifier- IN FILE-OPTION MUST BE A FILE CONSTANT OR VARIABLE	As stated.	Correct error.
REF037	F	VARIABLE -identifier- IN LOCATE STATEMENT MUST BE BASED	The identifier following the keyword LOCATE must be a based variable. It cannot be subscripted and cannot be a member of a structure.	Correct error.
REF038	F	VARIABLE -identifier- IN FROM- OR INTO-OPTION MUST BE UNSUBSCRIPTED AND CANNOT BE A MEMBER OF A STRUCTURE	As stated.	Correct error.
REF039	F	LABEL REFERENCE -identifier- IN GOTO STATEMENT CANNOT BE A FORMAT CONSTANT	As stated.	Correct error.
REF040	F	REFERENCE -identifier- IN SET- OR KEYTO-OPTION MUST BE SCALAR	As stated.	Correct error.
REF041	F	REFERENCE IN SET- OR KEYTO-OPTION MUST BE A VARIABLE OR PSEUDOVARIABLE	As stated.	Correct error.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
REF042	F	ILLEGAL STRING OPTION	In a GET statement, the STRING option can be an expression that must be scalar and computational. In a PUT statement, the reference in the STRING option must be a scalar variable or pseudovariable with the data type CHARACTER nonvarying or CHARACTER VARYING.	Correct error.
REF043	F	VARIABLE IN KEYTO-OPTION MUST BE OF TYPE CHARACTER	As stated.	Correct error.
REF044	F	VARIABLE IN INTO- OR FROM-OPTION MUST BE STATIC, AUTOMATIC, CONTROLLED, OR BASED	The reference in an INTO or FROM option cannot be a variable that is DEFINED or parameter.	Either change to an appropriate storage type, or declare a based variable and overlay the based variable on the defined variable or parameter that contains the values to be transmitted.
REF045	F	OFFSET -identifier- REQUIRES REFERENCE TO SCALAR AREA	A based variable is referenced with a locator-qualifier or implicit locator that is an offset variable. The locator value must be declared with a reference to a scalar area variable.	Correct error.
REF046	F	REFERENCE -identifier- TO AN UNDECLARED STRUCTURE	The statement contains a structure-qualified reference whose structure-qualifier does not reference any known structure.	Correct error.
REF047	F	STRUCTURE-QUALIFIED REFERENCE -identifier- MUST BE PROPERLY QUALIFIED	The statement contains an ambiguous structure-qualified reference; that is, there are two or more known structures to which the entire structure-qualified reference might apply.	Correct error by resolving the ambiguity.
REF048	F	REFERENCE -identifier- MUST BE PROPERLY STRUCTURE-QUALIFIED	The statement contains a reference which has no structure-qualifier and which is ambiguous; that is, it does not reference any know identifier that is not a member, and there are two or more structure members to which the reference might apply.	Correct error by resolving the ambiguity.

TABLE B-2. COMPILE-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
Message Code	Severity	Message Text		
REF049	F	BASED ALLOCATION WHICH SETS OFFSET VARIABLE REQUIRES IN-OPTION IN ALLOCATION OR AREA-REFERENCE IN OFFSET ATTRIBUTE	As stated.	Correct error.
REF050	F	ALLOCATION OF BASED VARIABLE REQUIRES SET-OPTION OR ASSUMED SET-OPTION	As stated.	Correct error.
REF051	W	REMOTE FORMAT -identifier- IS NONLOCAL	As stated.	Correct error.
SYN001	F	SYNTAX ERROR	This message is issued for most of the errors that can be detected before all declarations have been processed. The compiler indicates the place at which the statement has ceased to conform to any legal syntax. Other messages (preceded by ***) can be in the source listing. These messages follow.	Correct error.
		*** END OF PROGRAM INSIDE A COMMENT	Compiler was expecting a delimiter to end a comment, but found the end-of-information.	Check for unmatched comment delimiter.
		*** FIRST STATEMENT NOT A LEGAL PROC. NO.PROC: PROCEDURE OPTIONS(MAIN); ASSUMED	The PROCEDURE statement is missing or invalid. The compiler generated a PROCEDURE statement for the program.	Correct error.
		*** PROGRAM MUST BE A COMPLETE EXTERNAL PROCEDURE. END CARDS SUPPLIED	END statement or statements missing. The compiler generated the necessary END statements.	Check for mismatched BEGIN-END blocks. Check for mismatched PROCEDURE-END statements.
		*** THE CLOSURE NAME CANNOT BE FOUND, STATEMENT IGNORED	The closure name specified in the END statement does not match an entry name on a PROCEDURE statement, a label on a BEGIN statement, or a label on a DO statement.	Correct error.
SYN002	F	ALLOCATION WITH SET- OR IN-OPTION IS ALLOWED ONLY FOR BASED VARIABLES	As stated.	Correct error.
SYN003	F	BY OR TO ILLEGAL WITH INDEX OF NON-COMPUTATIONAL TYPE	As stated.	Correct error.
SYN004	F	LARGEST INTEGER PERMITTED IN CERTAIN CONTEXTS IS 16383	The statement contains a decimal integer larger than 16383.	Correct error.

TABLE B-3. RUN-TIME DIAGNOSTICS

Message			Significance	Programmer Action
ONCODE Value	Condition	Message Text		
3	ERROR	COMPILATION ERROR AT STATEMENT -number-	A fatal error was diagnosed at compile time, or a warning was issued at compile time and the diagnosed statement was executed in an illegal manner.	Correct error and recompile.
4	FINISH	None	FINISH condition signaled, STOP statement executed, or main procedure terminated normally.	None.
9	ERROR	SIGNAL ERROR	ERROR condition signaled.	None.
20	RECORD	RECORD (SIGNALED)	RECORD condition signaled.	None.
21	RECORD	RECORD (RECORD LENGTH NOT EQUAL TO RECORD AREA LENGTH)	CRM error 143. Record length less than generation (allocation unit) size.	Correct error.
22	RECORD	RECORD (RECORD LENGTH NOT EQUAL TO RECORD AREA LENGTH)	CRM error 142 or 170. Record length exceeds generation (allocation unit) size.	Correct error.
23	RECORD	RECORD (RECORD LENGTH NOT EQUAL TO RECORD AREA LENGTH)	Record length exceeds size of allocated buffer.	Correct error.
24	RECORD	RECORD (RECORD LENGTH NOT EQUAL TO RECORD AREA LENGTH)	Record length is less than size of allocated buffer.	Correct error.
25	RECORD	RECORD (RECORD LENGTH NOT EQUAL TO RECORD AREA LENGTH)	Record length not equal to generation (allocation unit) size.	Correct error.
40	TRANSMIT	TRANSMIT (SIGNALED)	TRANSMIT condition signaled.	None.
41	TRANSMIT	TRANSMIT (OUTPUT)	Transmission error occurred during output. If the diagnosed statement is a GET statement with the COPY option, the problem occurred on the copy file.	Retry. See CRM manuals. Check CRM error status in the dayfile.
42	TRANSMIT	TRANSMIT (INPUT)	Transmission error occurred during input. CRM errors 130, 135, 136, 137, and 140 occurred. If the diagnosed statement is a GET statement with a COPY option, the problem occurred on the input file.	Retry. See CRM manuals.
43	TRANSMIT	TRANSMIT (OUTPUT) - SOME REC. MGR. ERROR. SEE DAYFILE	CRM error.	Retry. See CRM manuals.
44	TRANSMIT	TRANSMIT (INPUT) - SOME REC. MGR. ERROR. SEE DAYFILE	CRM error.	Retry. See CRM manuals.
50	KEY	KEY (SIGNALED)	KEY condition signaled.	None.
51	KEY	KEY (KEYED RECORD NOT FOUND)	CRM error 445. Record not found.	Correct error.

TABLE B-3. RUN-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
ONCODE Value	Condition	Message Text		
52	KEY	KEY (ATTEMPT TO ADD DUPLICATE KEY)	CRM error 446. Duplicate key.	Correct error.
53	KEY	KEY (KEY SEQUENCE ERROR)	CRM error. Bad key specification.	Correct error.
54	KEY	KEY (KEY CONVERSION ERROR)	Key on record file does not convert.	Correct error.
55	KEY	KEY (KEY SPECIFICATION ERROR)	Unacceptable key format.	Correct error.
57	KEY	KEY (NO SPACE AVAILABLE TO ADD KEYED RECORD)	Key conversion/length problem.	Correct error.
70	ENDFILE	ENDFILE	EOF encountered during GET COPY.	Correct error.
70	ENDFILE	ENDFILE	ENDFILE condition signaled.	None.
71	ENDFILE	ENDFILE - END OF DATA	End of file (EOF) encountered, end of information (EOI) encountered, or attempted read beyond end of file.	Close the file and open it again to read beyond end of file.
72	ENDFILE	ENDFILE - FILE AT BOI	End of file encountered at beginning of information (BOI). No data on file.	Correct error.
80	UNDEFINEDFILE	UNDEFINEDFILE (SIGNALED)	UNDEFINEDFILE condition signaled.	None.
81	UNDEFINEDFILE	UNDEFINEDFILE (ATTRIBUTE CONFLICT)	Error in attributes detected during opening/closing.	Correct error.
82	UNDEFINEDFILE	UNDEFINEDFILE (FILE TYPE NOT SUPPORTED)	Invalid option in OPEN statement, or invalid file description attribute.	Correct error.
83	UNDEFINEDFILE	UNDEFINEDFILE (OPEN STATEMENT PARAMETER ERROR)	File constant and CRM file incompatible. Error in ENVIRONMENT specification or FILE control statement, or CRM file nonexistent for input or update.	Correct error.
84	UNDEFINEDFILE	UNDEFINEDFILE (BAD OPTION INTRODUCED)	FILE control statement parameter unacceptable. MRL/FL option not specified.	Correct error.
85	UNDEFINEDFILE	UNDEFINEDFILE (DATASET ALREADY IN USE)	Attempt to open a file constant and associate it with a CRM file already in use.	Correct error.
86	UNDEFINEDFILE	UNDEFINEDFILE (SOME CRM ERROR. SEE DAYFILE)	CRM error 5, 6, 21, 25, 30, 31, 32, 33, 35, 36, 45, 50, 144, 150, 151, 154, 165, or 354.	See CRM manuals.
87	UNDEFINEDFILE	UNDEFINEDFILE (FILE TITLE CONTAINS AN INVALID CHARACTER OR HAS A LENGTH OF ZERO)	The title option has a length of zero, or contains a blank or special character; or there is no title option and the filename contains a $, @, #, or _.	Correct error.
89	UNDEFINEDFILE	UNDEFINEDFILE (CRM ERROR ON FILE OPENING)	CRM error during file opening.	See CRM manuals.

TABLE B-3. RUN-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
ONCODE Value	Condition	Message Text		
90	ENDPAGE	None	ENDPAGE condition raised during output on print file or during GET COPY.	Correct as appropriate.
90	ENDPAGE	None	ENDPAGE condition signaled.	None.
91	UNDEFINEDFILE	None	ENDPAGE condition signaled on nonexistent file.	Correct error.
92	UNDEFINEDFILE	UNDEFINED FILE (RECORD MANAGER, MODULE NOT LOADED)	File opening failed.	Correct error.
300	OVERFLOW	OVERFLOW	OVERFLOW condition signaled.	None.
301	OVERFLOW	OVERFLOW (IN EXECUTION OF IN-LINE CODE)	OVERFLOW condition raised.	Correct error.
302	OVERFLOW	OVERFLOW (EXP ARGUMENT GREATER THAN 741.67)	OVERFLOW condition raised during exponential evaluation.	Correct error.
303	OVERFLOW	OVERFLOW (SINH ARGUMENT GREATER THAN 741.67)	OVERFLOW condition raised in computation of SINH.	Correct error.
304	OVERFLOW	OVERFLOW (COSH ARGUMENT GREATER THAN 741.67)	OVERFLOW condition raised in computation of COSH.	Correct error.
307	OVERFLOW	OVERFLOW (OCCURRED DURING EXPONENTIATION)	OVERFLOW condition raised during exponentiation.	Correct error.
310	FIXEDOVERFLOW	FIXEDOVERFLOW	FIXEDOVERFLOW condition signaled.	None.
310	FIXEDOVERFLOW	FIXEDOVERFLOW (IN EXECUTION OF IN-LINE CODE)	FIXEDOVERFLOW condition raised.	Correct error.
320	ZERODIVIDE	ZERODIVIDE	ZERODIVIDE condition signaled.	None.
320	ZERODIVIDE	ZERODIVIDE (IN EXECUTION OF IN-LINE CODE)	ZERODIVIDE condition raised.	Correct error.
330	UNDERFLOW	UNDERFLOW	UNDERFLOW condition signaled.	None.
330	UNDERFLOW	UNDERFLOW (IN EXECUTION OF IN-LINE CODE)	UNDERFLOW condition raised.	Correct error.
332	UNDERFLOW	UNDERFLOW (EXP ARGUMENT LESS THAN -675.82)	UNDERFLOW condition raised during exponential evaluation.	Correct error.
337	UNDERFLOW	UNDERFLOW (OCCURRED DURING EXPONENTIATION)	UNDERFLOW condition raised during exponentiation.	Correct error.
340	SIZE	SIZE (NORMAL)	SIZE condition raised in an operation other than I/O.	Correct error.
341	SIZE	SIZE (I/O)	SIZE condition raised during conversion in stream I/O.	Correct error.
342	SIZE	SIZE (ARITHMETIC TO CHARACTER STRING CONVERSION)	SIZE condition raised in FLOAT arithmetic to character string conversion.	Correct error.
343	SIZE	SIZE (ARITHMETIC TO ARITHMETIC I/O CONVERSION)	SIZE condition raised.	Correct error.

TABLE B-3. RUN-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
ONCODE Value	Condition	Message Text		
350	STRINGRANGE	STRINGRANGE (SIGNALED)	STRINGRANGE condition in substring computation.	Correct error.
351	STRINGRANGE	STRINGRANGE (IN SUBSTR OF A CHARACTER STRING)	STRINGRANGE condition raised in SUBSTR builtin function or pseudovariable.	Correct error.
352	STRINGRANGE	STRINGRANGE (IN SUBSTR OF A BIT STRING)	STRINGRANGE condition raised in SUBSTR builtin function or pseudovariable.	Correct error.
355	STRINGRANGE	Message	STRINGRANGE condition raised in REVERSE builtin function.	Correct error.
360	AREA	AREA RAISED IN ALLOCATE STATEMENT	Allocation of nonstacked storage block has failed.	Correct error.
361	AREA	AREA RAISED IN ASSIGNMENT STATEMENT	Assignment of area to smaller area.	Correct error.
362	AREA	AREA (SIGNALED)	AREA condition signaled.	None.
370	STORAGE	STORAGE (SIGNALED)	STORAGE condition signaled.	None.
371	STORAGE	STORAGE (CMM IS UNABLE TO ALLOCATE SPACE FOR CRM OR OTHER ROUTINE OUTSIDE OF PL/I)	STORAGE condition raised, possibly while attempting to load a CRM capsule. Diagnostics and snap output will always appear on the dayfile.	Correct error.
372	STORAGE	STORAGE (UNABLE TO ALLOCATE A BASED VARIABLE)	STORAGE condition raised while attempting to allocate a based variable which is not in an area.	Correct error.
373	STORAGE	STORAGE (UNABLE TO ALLOCATE A CONTROLLED VARIABLE)	STORAGE condition raised while attempting to allocate a controlled variable.	Correct error.
374	STORAGE	STORAGE (UNABLE TO ALLOCATE AN AUTOMATIC VARIABLE OR A TEMPORARY)	STORAGE condition raised while attempting to allocate an automatic variable or a temporary.	Correct error.
375	STORAGE	STORAGE (UNABLE TO ALLOCATE A TEMPORARY)	STORAGE condition raised while attempting to allocate a temporary.	Correct error.
376	STORAGE	Message	STORAGE condition raised during allocation of a temporary such as a dummy argument.	Correct error.
377	STORAGE	Message	STORAGE condition raised with illegal oncode value.	Correct error.
500	CONDITION	A PROGRAMMER-NAMED CONDITION HAS BEEN SIGNALED.	CONDITION signaled for a programmer-defined condition.	None.
520	SUBSCRIPTRANGE	SUBSCRIPTRANGE	SUBSCRIPTRANGE condition signaled.	None.
521	SUBSCRIPTRANGE	SUBSCRIPTRANGE (IN EXECUTION OF IN-LINE CODE)	SUBSCRIPTRANGE condition raised.	Correct error.

TABLE B-3. RUN-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
ONCODE Value	Condition	Message Text		
522	SUBSCRIPTRANGE	SUBSCRIPTRANGE (HIGH)	SUBSCRIPTRANGE condition raised for subscript that exceeds upper bound.	Correct error.
523	SUBSCRIPTRANGE	SUBSCRIPTRANGE (LOW)	SUBSCRIPTRANGE condition raised for subscript smaller than lower bound.	Correct error.
600	CONVERSION	CONVERSION (INTERNAL) (SIGNALED)	CONVERSION condition signaled.	None.
601	CONVERSION	CONVERSION (I/O)	Conversion condition raised in character to character conversion in I/O.	Correct error.
604	CONVERSION	CONVERSION (ERROR IN F-FORMAT INPUT) (I/O)	CONVERSION condition raised as stated in I/O.	Correct error.
607	CONVERSION	CONVERSION (ERROR IN E-FORMAT INPUT) (I/O)	CONVERSION condition raised as stated in I/O.	Correct error.
610	CONVERSION	CONVERSION (ERROR IN B-FORMAT INPUT) (I/O)	CONVERSION condition raised as stated in I/O.	Correct error.
612	CONVERSION	CONVERSION (CHARACTER STRING TO ARITHMETIC)	CONVERSION condition raised (not in I/O) as stated.	Correct error.
613	CONVERSION	CONVERSION (CHARACTER STRING TO ARITHMETIC) (I/O)	CONVERSION condition raised as stated in I/O.	Correct error.
615	CONVERSION	CONVERSION (CHARACTER STRING TO BIT STRING)	CONVERSION condition raised (not in I/O) as stated.	Correct error.
616	CONVERSION	CONVERSION (CHARACTER STRING TO BIT STRING) (I/O)	CONVERSION condition raised as stated in I/O.	Correct error.
618	CONVERSION	CONVERSION (CHARACTER TO PICTURE)	CONVERSION condition raised (not in I/O) in picture verification.	Correct error.
619	CONVERSION	CONVERSION (CHARACTER TO PICTURE) (I/O)	CONVERSION condition raised as stated in I/O.	Correct error.
622	CONVERSION	CONVERSION (P-FORMAT INPUT - DECIMAL) (I/O)	CONVERSION condition raised as stated in I/O.	Correct error.
625	CONVERSION	CONVERSION (P-FORMAT INPUT - CHARACTER) (I/O)	CONVERSION condition raised as stated in I/O.	Correct error.
1000	ERROR	Message	ERROR condition raised during input.	Correct error.
1002	ERROR	STRING LENGTH ERROR or no message	String length error during input or output.	Correct error.
1003	ERROR	Message	RECORD condition raised during stream input. Actual line size on CRM file exceeds line size specified or declared during file opening.	Open file with sufficient line size specified in LINESIZE option.
1004	ERROR	None	PAGE or LINE requested on a file that is not a print file.	Correct error.

TABLE B-3. RUN-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
ONCODE Value	Condition	Message Text		
1005	ERROR	None	Reference to current length of uninitialized VARYING string.	Correct error.
1010	ERROR	None	REWRITE keyed operation on FO=WA, RT=F. Statement not legal on the file.	Correct error.
1011	ERROR	None	CRM error on file closing.	Correct error.
1012	ERROR	Message	Record length inaccessible, or partial record I/O not possible.	Correct error.
1013	ERROR	None	Key has length of zero, or CRM file has FO=WA and key cannot be converted to arithmetic.	Correct error.
1015	ERROR	Message	File undefined.	Correct error.
1016	ERROR	Message	File not open on normal return from UNDEFINEDFILE on-unit.	Correct error.
1020	ERROR	Message	Statement not executable, or statement illegal for CRM file.	Correct error.
1021	ERROR	Message	No allocated buffer.	Correct error.
1030	ERROR	None	Attempted format list recursion.	Correct error.
1031	ERROR	WIDTH MUST BE SPECIFIED IN AN A OR B FORMAT ITEM USED FOR INPUT	An A or B format item with no width has been executed by a GET statement.	Correct error.
1040	ERROR	Message	CRM error 355, or fatal CRM error.	See CRM manuals.
1100	ERROR	ILLEGAL ATTEMPTED NORMAL RETURN FROM ON-UNIT	Attempt to perform normal termination of an on-unit for FIXEDOVERFLOW, OVERFLOW, SIZE, STRINGRANGE, SUBSCRIPTRANGE, or ZERODIVIDE condition.	Correct error.
1102	ERROR	ERROR (NORMAL TERMINATION OF ERROR ON-UNIT)	Attempt to perform normal termination of an on-unit for an ERROR condition. This activates the system on-unit for ERROR; the current established on-unit is by-passed.	Correct error.
1500	ERROR	Message	Negative argument.	Correct error.
1504	ERROR	Message	Negative or zero argument.	Correct error.
1506	ERROR	Message	Argument too large for SIN.	Correct error.
1508	ERROR	Message	Argument too large.	Correct error.
1510	ERROR	Message	Both arguments are zero for ATAN.	Correct error.
1514	ERROR	Message	Absolute value of argument greater than or equal to 1.0.	Correct error.
1515	ERROR	Message	Intermediate result is out of range.	Correct error.

TABLE B-3. RUN-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
ONCODE Value	Condition	Message Text		
1516	ERROR	Message	Argument exceeds 1.0 in absolute value.	Correct error.
1518	ERROR	Message	Argument exceeds 1.0 in absolute value.	Correct error.
1520	ERROR	Message	Argument too large for SIND.	Correct error.
1522	ERROR	Message	Argument too large for COSD. Accuracy lost.	Correct error.
1524	ERROR	Message	Argument too large for COSD. Accuracy lost.	Correct error.
1526	ERROR	Message	Argument too large for TAND. Accuracy lost.	Correct error.
1528	ERROR	Message	Both arguments for ATAND are zero.	Correct error.
1532	ERROR	Message	Exponentiation with negative base. Logarithm (base 2) requested of nonpositive quantity.	Correct error.
1534	ERROR	Message	Common logarithm of nonpositive item requested.	Correct error.
1542	ERROR	Message	Both arguments zero in exponentiation.	Correct error.
1550	ERROR	Message	Exponentiation attempted with zero base and exponent not exceeding zero.	Correct error.
1552	ERROR	Message	Zero base raised to a nonpositive power.	Correct error.
1600	ERROR	Message	Unacceptable argument for SIN.	Correct error.
1604	ERROR	Message	Unacceptable argument for SIND.	Correct error.
1606	ERROR	Message	Unacceptable argument for COS.	Correct error.
1610	ERROR	Message	Unacceptable argument for COSD.	Correct error.
1612	ERROR	Message	Unacceptable argument for TAN.	Correct error.
1616	ERROR	Message	Unacceptable argument for TAND.	Correct error.
1618	ERROR	Message	Unacceptable argument for ATAN.	Correct error.
1622	ERROR	Message	Unacceptable argument for ATAND.	Correct error.
1624	ERROR	Message	Unacceptable argument for ATAN.	Correct error.
1626	ERROR	Message	Unacceptable argument for ATAN.	Correct error.
1628	ERROR	Message	Unacceptable argument for SINH.	Correct error.
1632	ERROR	Message	Unacceptable argument for COSH.	Correct error.
1636	ERROR	Message	Unacceptable argument for TANH.	Correct error.

TABLE B-3. RUN-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
ONCODE Value	Condition	Message Text		
1648	ERROR	Message	Unacceptable argument for EXP.	Correct error.
1652	ERROR	Message	Unacceptable argument for LOG.	Correct error.
1656	ERROR	Message	Unacceptable argument for LOG2.	Correct error.
1658	ERROR	Message	Unacceptable argument for LOG10.	Correct error.
1660	ERROR	Message	Unacceptable argument for SQRT.	Correct error.
1666	ERROR	Message	Unacceptable argument for ACOS.	Correct error.
1670	ERROR	Message	Unacceptable argument for ASIN.	Correct error.
1674	ERROR	Message	Argument for ATANH indefinite or out of range.	Correct error.
1678	ERROR	Message	Operand for exponentiation indefinite or out of range.	Correct error.
1680	ERROR	Message	Operand for exponentiation indefinite or out of range.	Correct error.
3000	ERROR	None	Illegal specification for an E format item (sd<0, sd>width, fd>sd, fd<0, or fd absent).	Correct error.
3001	ERROR	None	Illegal specification for an F format item (fd>width or fd absent).	Correct error.
3004	ERROR	Message	Width specification absent for format during GET statement execution.	Correct error.
3799	ERROR	FAILURE TO CORRECT ERROR FOR ONCODE = 601	Failure to correct previous error in character to character conversion.	Correct error.
3799	ERROR	FAILURE TO CORRECT ERROR FOR ONCODE = 612	Failure to correct previous conversion error (not in I/O).	Correct error.
3802	ERROR	ARRAY BOUND MAGNITUDE >131071	Absolute value of upper bound too large.	Correct error.
3804	ERROR	STRING LENGTH NEGATIVE, TOO LARGE, OR UNINITIALIZED	Attempt to allocate or reference a string with improper maximum or current length.	Correct error.
4051	ERROR	BLOCK NOT IN AREA	Request to free a storage block not in specified area.	Correct error.
4061	ERROR	BLOCK WRONG SIZE	Wrong size supplied in request to free an outer storage block.	Correct error.
4062	ERROR	BLOCK WRONG SIZE	Wrong size supplied in request to free a storage block in an area.	Correct error.
4063	ERROR	Message	Storage in an area has been overlaid and is now incorrect.	Correct error, or report to a systems analyst.

TABLE B-3. RUN-TIME DIAGNOSTICS (Contd)

Message			Significance	Programmer Action
ONCODE Value	Condition	Message Text		
4064	ERROR	Message	Storage in a stack has been overlaid and is now incorrect	Report to a systems analyst.
4065	ERROR	Message	Allocated temporary storage is now flawed.	Report to a systems analyst.
4066	ERROR	Message	Allocated temporary storage has been overlaid and is now incorrect.	Report to a systems analyst.
5000	ERROR	TOO MANY ARGUMENTS	Too many arguments in procedure or function reference.	Correct error.
5000	ERROR	TOO FEW ARGUMENTS	Too few arguments in procedure or function reference.	Correct error.
9000	ERROR	Message	Nonlocal GOTO during block activation or during allocation.	Correct error.
9001	ERROR	Message	Activation descriptor not found in GOTO label variable execution.	Correct error, or report to systems analyst.
9002	ERROR	Message	Block descriptor not found in GOTO label variable execution.	Correct error, or report to a systems analyst.
9020	ERROR	Message	CRM out of space. Field length must be increased.	Increase field length for program execution.

GLOSSARY C

ABNORMAL BLOCK TERMINATION – Termination of a block activation in a manner such that control does not return to the predecessor block activation in the normal manner. Execution of a nonlocal GOTO causes the abnormal termination of one or more block or on-unit activations. Execution of a RETURN statement when a begin block is currently active causes the abnormal termination of one or more begin block activations.

ABNORMAL ON-UNIT TERMINATION – Termination of an on-unit activation in a manner such that control does not return to the point of interrupt. Execution of a nonlocal GOTO causes the abnormal termination of one or more block or on-unit activations.

ACTIVATION – A specific execution of a block or on-unit; the process of initiating execution of a block or on-unit.

ACTIVE BLOCK – A block that is presently being executed.

ADVANCED ACCESS METHODS (AAM) – A file manager within CYBER Record Manager that processes indexed sequential file organizations and supports the Multiple Index Processor.

AGGREGATE – An organized collection of data elements.

AGGREGATE TYPE – The particular organization used to form an aggregate, including the simple case of a single scalar.

ALIGNED VALUE – A value positioned on a word boundary in storage.

ALLOCATED BUFFER – A generation specifically allocated for use as a record I/O buffer. Only one allocated buffer can exist for a file.

ALLOCATED VARIABLE – A variable for which storage has been reserved and not freed.

ALLOCATION – The process of reserving storage for a variable.

ALLOCATION UNIT – The basic unit for allocating and freeing the storage associated with a generation.

AMBIGUOUS REFERENCE – A reference that is not sufficiently qualified to identify one and only one structure member or substructure.

AREA – Storage that is reserved, upon allocation, for the subsequent allocation of based generations.

AREA VARIABLE – A variable that holds area values. An area variable can be of any storage type.

ARGUMENT – An expression that is specified in the argument list of a function reference or a CALL statement.

ARGUMENT LIST – A parenthesized list of one or more arguments separated by commas. Each argument in a procedure reference is passed to a corresponding parameter; each argument in a builtin function reference represents a value to be passed to the function.

ARGUMENT PASSED BY REFERENCE – An argument that shares a generation with the corresponding parameter.

ARGUMENT PASSED BY VALUE – An argument that has a temporary generation that is not shared with the corresponding parameter; a dummy argument.

ARITHMETIC CONSTANT – A constant arithmetic value with a mode, scale, base, and precision implied by the actual form of the constant.

ARITHMETIC OPERATOR – An operator used in arithmetic operations. Arithmetic operators are + (positive), - (negative), + (add), - (subtract), * (multiply), / (divide), and ** (exponentiate).

ARITHMETIC VARIABLE – A variable that has arithmetic values.

ARRAY – A collection of data elements where each element has the same name and the same set of attributes. Contrast with structure.

ARRAY BOUNDS – See Bounds.

ARRAY CROSS SECTION – A portion of an array formed by selecting certain elements. The elements are selected by assigning definite constant values to some of the subscript positions and allowing other subscript positions to range from their lower bounds to their upper bounds.

ARRAY SLICE – Synonymous with array cross section.

ASSIGNMENT – The process of storing a source value into a target item.

ATTRIBUTE – A descriptive and defining characteristic of data elements.

AUTOMATIC VARIABLE – A data item that has storage allocated automatically when a block is activated and has storage released automatically when that block is terminated.

BALANCED IF – An IF statement that has an ELSE clause.

BASE – A characteristic of arithmetic data; either the BINARY or DECIMAL attribute. The base controls the meaning of the precision attribute and does not control the internal representation of values.

BASED VARIABLE – A variable used to allocate and free based generations and to reference generations of any storage type.

BASIC ACCESS METHODS (BAM) – A file manager within CYBER Record Manager that processes sequential and word addressable file organizations.

BEGIN BLOCK – A collection of statements headed by a BEGIN statement and terminated by a matching END statement. A begin block is activated when encountered in sequential program flow or when control is transferred to the BEGIN statement by a GOTO statement.

BIT CONSTANT – Synonymous with bit string constant.

BIT STRING CONSTANT – A sequence of binary digits enclosed in apostrophes and followed by the radix factor B. A bit string constant does not represent an arithmetic value.

BIT STRING OPERATOR – An operator used in bit string operations. Bit string operators are ¬(NOT), & (AND), and ! (OR). The concatenate operator || can also be used for bit strings.

BLOCK – (1) As applied to PL/I, an area of a source program that delimits the scope of names, determines allocation and freeing of some kinds of data storage, and influences program flow of control. (2) As applied to CYBER Record Manager, the term block has several meanings depending on context. On tape, a block is information between interrecord gaps on tape. Block type C can be used for PL/I. CRM defines several blocks depending on organization.

Organization	Blocks
Indexed sequential	Data block; index block
Sequential	Block type I, C, K, E

BLOCK ACTIVATION – A specific execution of a block; the process of activating a block.

BLOCK TERMINATION – The removal of a block activation from the dynamic stack; the dynamic predecessor becomes the current block activation. Block termination is classified as normal or abnormal.

BOI (BEGINNING-OF-INFORMATION) – CYBER Record Manager defines beginning-of-information as the start of the first user record in a file. System-supplied information, such as an index block or control word, does not affect beginning-of-information. Any label on a tape exists prior to beginning-of-information.

BOOLEAN OPERATION – An algebraic operation on entities with two possible values, such as the individual bits found in a bit string.

BOUNDS – The lower bound and upper bound of a single dimension of an array.

BUFFER – See Allocated buffer.

BUILTIN FUNCTION – A function supplied by the language; performs a specific operation, usually on one or more arguments, and returns a single value as a result.

CHARACTER CODE – A code that specifies the type of character that can appear in each position. Character codes are 9 (digit or blank), A (letter A through Z or blank), and X (any character).

CHARACTER CONSTANT – Synonymous with character string constant.

CHARACTER PAIR – Occurrence of two consecutive characters.

CHARACTER STRING CONSTANT – A sequence of characters, including blanks, enclosed in apostrophes.

CLOSURE NAME – An optional name on an END statement denoting the block or do group that is to be closed. The name must match an entry name on a PROCEDURE statement, a label on a BEGIN statement, or a label on a DO statement.

COMMENT – A sequence of characters preceded by the character pair /* and followed by the character pair */. Comments are used for documentation purposes only.

COMMON BASE – The base that is common to a set of operands. The common base is DECIMAL if the base of all operands is DECIMAL; otherwise, the common base is BINARY.

COMMON DATA TYPE – The data type that is common to a set of operands. The common data type for a set of arithmetic operands is formed by taking the common base and the common scale of the operands.

COMMON SCALE – The scale that is common to a set of operands. The common scale is FIXED if the scale of all operands is FIXED; otherwise, the common scale is FLOAT.

COMPARISON OPERATOR – An operator used in comparison operations. Comparison operators are the infix operators = (equal), ¬= (not equal), < (less than), <= (less than or equal), ¬< (not less than), > (greater than), >= (greater than or equal), ¬> (not greater than).

COMPOUND STATEMENT – A statement that can contain embedded statements. IF and ON are classified as compound statements.

COMPUTATIONAL TYPE – Arithmetic, bit, character, or pictured data type.

CONCATENATE OPERATOR – The operator || for character strings or bit strings.

CONCATENATION – An operation in which two strings are joined to form one string.

CONDITION – An unusual situation that can occur during execution and can result in a program interrupt.

CONDITION BUILTIN FUNCTION – A builtin function that returns information concerning interrupts. A referenced condition builtin function name returns such information as the location and cause of an interrupt. Condition builtin functions are ONCHAR, ONCODE, ONFILE, ONKEY, ONLOC, and ONSOURCE.

CONDITION NAME – A keyword that denotes a condition that might occur during program execution. Condition names appear in the ON, REVERT, and SIGNAL statements.

CONDITION PREFIX – A parenthesized computational condition name or list of computational condition names that precede a statement. The condition prefix enables or disables program interrupts that can result when computational conditions are raised.

CONNECTED GENERATION – A generation whose storage description part specifies data items that are contiguous in storage.

CONSTANT – A data element with a value that does not change during program execution.

CONTAINED TEXT – All text that appears in a block or on-unit, including text in nested (contained) blocks. Also see Immediately contained text.

CONTEXTUAL DECLARATION – The establishment of a set of attributes for an identifier because the identifier has been used in a certain context, but has not been explicitly declared.

CONTROLLED VARIABLE – A variable that has generations maintained in a stack with only the top entry accessible at any one time.

CONVERSION – The transformation of a value to another computational type or to another data type.

COPY FILE – The file to which characters read in during a GET statement execution with the COPY option are copied.

CRM – CYBER Record Manager.

CRM FILE – A local file known to CYBER Record Manager and the operating system. Contrast with file.

CURRENCY CODE – The picture code $ (dollar sign).

CURRENT BLOCK ACTIVATION – The block activation most recently activated and not yet terminated, that is, the top activation in the dynamic stack.

CURRENT ESTABLISHED ON-UNIT – The on-unit that will be activated when the applicable condition is raised and an interrupt occurs.

CURRENT-RECORD DESIGNATOR – An indicator that identifies the current record to be processed.

CYBER RECORD MANAGER (CRM) – A generic term relating to the common products AAM and BAM that run under the NOS and NOS/BE operating systems and that allow a variety of record types, blocking types, and file organizations to be created and accessed.

DATA ELEMENT – A specification or representation of a value or values in the program.

DATA TYPE – The type of value represented by a variable or named constant.

DECIMAL POINT CODE – The picture code V (assumed decimal point).

DECLARATION – The establishment of an identifier with a set of attributes. Declarations are explicit, contextual, or implicit.

DEFINED VARIABLE – A variable that shares storage with another variable referred to as the host variable.

DELIMITER – One of a class of characters that serve to delimit statement elements such as words and literal constants.

DESCRIPTOR – The establishment of a set of attributes for a particular position in a parameter list or for the value returned from a procedure that is invoked as a function. A descriptor differs from a declaration in that there is no identifier.

DIGIT CODE – A code that specifies a character position in which a decimal digit can appear. Digit codes are 9 (digit), Z (leading zero suppression), * (leading zero replacement), and Y (zero suppression).

DIMENSION – An array has a specified number of dimensions; each dimension is a repetition of the array element the number of times indicated by the span of the dimension.

DIMENSIONALITY – The number of dimensions in an array.

DISABLED CONDITION – A condition that is in the disabled state; such a condition will not cause a program interrupt if it is raised. Contrast with enabled condition.

DO GROUP – A collection of statements headed by a DO statement and terminated by a matching END statement. A do group is executed when encountered in sequential program flow or when control is transferred to the DO statement by a GOTO statement.

DO-SPECIFICATION – The portion of an indexed DO that determines the number of times the group is executed.

DO WHILE – An iterative do group. The WHILE expression controls the number of times the group is executed.

DRIFTING SEQUENCE – A sequence of identical picture codes. The specified character is placed to the left of the most significant digit. One of the sign codes (S + -) or the currency code ($) can be used in a drifting sequence in a picture specification.

DUMMY ARGUMENT – A temporary value created and stored when an argument is passed by value. When a reference is made to the corresponding parameter, the dummy argument rather than the original argument is accessed.

DYNAMIC PREDECESSOR – A preceding block activation in the dynamic stack.

DYNAMIC STACK – An ordered list of block and on-unit activations. The last activation in the list is termed the top of the stack. Additions to and deletions from the stack are made only at the top of the stack. Each activation in the stack holds a set of values describing the current state of the activation, including pointers to the program, generations of certain variables, and so forth.

DYNAMIC SUCCESSOR – A succeeding block activation in the dynamic stack.

EDIT-DIRECTED I/O – A form of stream I/O. The data is transmitted in accordance with a format specification supplied in the program. Contrast with list-directed I/O.

EMBEDDED-DO – An indexed DO that specifies iteration in the list of target items or source values during stream I/O; used in GET or PUT statements.

ENABLED CONDITION – A condition that is in the enabled state; such a condition will cause a program interrupt if it is raised, thus activating the current established on-unit. Contrast with disabled condition.

ENTRY CONSTANT – An entry name that appears as an entry prefix on a PROCEDURE or ENTRY statement.

ENTRY POINT – A point at which a procedure can be invoked, and a complete description of all parameters and the returned value, if any.

ENTRY PREFIX – A prefix used only on an ENTRY and PROCEDURE statement to establish an entry constant.

ENTRY VALUE – A value that identifies both an entry point and a block activation.

ENTRY VARIABLE – A variable that is declared with the ENTRY attribute; it can only be used as a parameter in an invoked procedure.

ENVIRONMENT – (1) As applied to block or on-unit activations, the list of successive immediate environments of the activation, where each immediate environment is an activation of the successively containing block or on-unit. (2) As applied to files, the information indicating the characteristics of the local CRM file, as supplied by default or overridden with ENVIRONMENT options.

EOI (END-OF-INFORMATION) – CYBER Record Manager defines end-of-information in terms of the file organization and file residence.

File Organization	File Residence	Physical Position
Sequential	Mass storage	After last user record.
	Labeled tape in SI, I, S, or L format	After last user record and before any file trailer labels.
	Unlabeled tape in SI or I format	After last user record and before any file trailer labels.
	Unlabeled tape in S or L format	Undefined.
Word addressable	Mass storage	After last word allocated to file, which might be beyond the last user record.
Indexed Sequential	Mass storage	After record with highest key value.

ESTABLISHED ON-UNIT – An on-unit made available for execution when a particular condition is subsequently raised and an interrupt occurs. An on-unit is established when the ON statement in which it appears is executed. An on-unit can be in an established state, but not be the current established on-unit if another subsequently established on-unit remains established.

EXPLICIT ALLOCATION – The process of allocating storage by execution of an ALLOCATE statement.

EXPLICIT DECLARATION – The establishment of a set of attributes for an identifier by DECLARE statement, or by its appearance in a parameter list, or as a statement prefix.

EXPLICIT FILE OPENING – The process of opening a file by execution of an OPEN statement.

EXPLICIT FREEING – The process of freeing storage by execution of a FREE statement.

EXPONENT CODE – The picture code for the exponent. The exponent codes are E (supply E) and K (suppress E).

EXPRESSION – A sequence of operators and operands arranged in a way defined as legal by the rules of PL/I so as to constitute a valid rule for computing a value.

EXTENT – The bounds of an array dimension, the length (or maximum length) of a string, or the size of an area.

EXTERNAL ENTRY CONSTANT – An entry name on the PROCEDURE or ENTRY statement of an external procedure.

EXTERNAL PROCEDURE – A procedure block that is not contained in any other block.

EXTRALINGUAL CHARACTER – A character that can only be used in comments and character strings.

FILE – Contrast with CRM file. A file constant that has been opened, together with the completed file description attributes and the associated CRM file.

FILE CONSTANT – A constant that serves as a linkage to enable a PL/I program to communicate with external data storage (CRM files).

FILE DESCRIPTION – A set of attributes defining the characteristics of a file.

FILE INFORMATION TABLE (FIT) – A table through which a user program communicates with CYBER Record Manager. All CRM file processing executes on the basis of information in this table.

FILE TITLE – A name that is either supplied in the TITLE option of the OPEN statement or formed by concatenating the first four and last three characters of the file constant.

FILE VARIABLE – A variable that is declared with the FILE attribute; it can only be used as a parameter in an invoked procedure.

FIXED BINARY CONSTANT – An unsigned binary number containing an optional binary point at any position and followed by the radix factor B.

FIXED DECIMAL CONSTANT – An unsigned decimal number containing an optional decimal point at any position.

FIXED POINT – A method of calculation in which the location of the decimal or binary point is determined by the programmer, and a specified absolute precision is maintained. Contrast with floating point.

FLOAT BINARY CONSTANT – An unsigned binary number containing an optional binary point at any position, followed by a decimal exponent and the radix factor B.

FLOAT DECIMAL CONSTANT – An unsigned decimal number containing an optional decimal point at any position and followed by a decimal exponent.

FLOATING POINT – A method of calculation in which the location of the decimal or binary point is determined automatically, and a specified relative position is maintained. Contrast with fixed point.

FLOW OF CONTROL – The execution sequence of a PL/I program.

FORMAT CONSTANT – A format name that appears in a format prefix on a FORMAT statement.

FORMAT ITEM – An item used in edit-directed I/O to control the transmission of values, to control the spacing and positioning of the input or output data stream, and to control use of remote formats. Format items are F (fixed point decimal arithmetic), E (floating point decimal arithmetic), A (character string), B (bit string), P (pictured), X (ignore or create blanks), COLUMN (new column position), SKIP (skip lines), LINE (new line on a page), PAGE (next page), R (remote format list).

FORMAT PREFIX – A prefix used only on the FORMAT statement to establish a format constant.

FORMAT VALUE – A value that identifies both a format constant and a block activation.

FREEING – The process of releasing storage for a variable.

FULLY QUALIFIED REFERENCE – A reference to a structure member in which a structure-qualifier is included for each containing structure.

FUNCTION – A procedure that returns a value. A function is only invoked by a function reference in an expression.

FUNCTION REFERENCE – The use, in an expression, of the entry point name of a function so as to invoke the function and utilize its returned value in evaluating the expression.

GENERATION OF STORAGE – An entity associated with a variable and created either when the variable is allocated or when the variable is referenced (for certain storage types). A generation contains a data description for the variable and a storage description specifying the associated storage.

HOST VARIABLE – The variable with which a defined variable shares storage.

IDENTIFIER – A name used to identify and refer to data, statements, blocks, builtin functions, and programmer-named conditions.

IMMEDIATE DYNAMIC PREDECESSOR – The immediately preceding block activation in the dynamic stack.

IMMEDIATE ENVIRONMENT – an activation of the immediately containing block or on-unit.

IMMEDIATELY CONTAINED TEXT – All text that appears in a block or on-unit, excluding text in nested (contained) blocks.

IMPLICIT ALLOCATION – The process of allocating storage for an allocated buffer by execution of a READ or LOCATE statement.

IMPLICIT DECLARATION – The establishment of a set of attributes for an identifier because the identifier has been used, but has not been explicitly or contextually declared.

IMPLICIT FILE OPENING – The process of opening a file as a result of an I/O operation rather than by execution of an OPEN statement.

IMPLICIT FREEING – The process of freeing an allocated buffer by execution of a record I/O statement.

INDEXED DO – An iterative do group; the list of do-specifications determines the number of times the group is executed.

INDEXED SEQUENTIAL (IS) – A file organization in which records are stored in ascending order by key.

INFIX EXPRESSION – Two operands separated by a single infix operator.

INFIX OPERATOR – An operator that appears between two operands. The infix operators are + - * / ** & | || = ¬= < <= ¬< > >= and ¬>.

INHERITED DIMENSIONS – Dimensions added to the dimensions of a structure member as determined by the dimensionality of all containing structures.

INITIAL VALUE – The value assigned to a variable when the variable is allocated.

INSERTION CODE – A picture code that causes the insertion of a character in a specified position. The insertion codes are / (slash) , (comma) . (period) and B (blank).

INTERMEDIATE CALCULATION – The evaluation of a subexpression to produce an intermediate value.

INTERMEDIATE VALUE – The result of an operation to convert an operand to a different computational data type. Also, the result of a subexpression evaluation used as an operand in a subsequent subexpression evaluation in the same expression.

INTERNAL ENTRY CONSTANT – An entry name on the PROCEDURE or ENTRY statement of an internal procedure.

INTERNAL PROCEDURE – A procedure block that is contained in another block.

INTERRUPT – The disruption of program flow that results when an enabled condition is raised.

INVOCATION – The activation of a procedure.

iSUB – An element used in array defining to indicate dummy positions for the substitution of subscript values; iSUB is used in connection with the DEFINED attribute specification.

ITERATION FACTOR – An expression that indicates the number of consecutive array elements to be initialized or the number of times a format item is to occur.

ITERATIVE DO GROUP – A do group that is executed repetitively according to controls specified in the DO statement.

KEY – (1) As applied to PL/I, a character value used to identify a record on a KEYED file. (2) As applied to CYBER Record Manager, a group of contiguous characters that identify a record in an indexed sequential file. NOTE: When a KEYED file is opened on a local CRM file with WA organization, the PL/I keys used are character values that contain only digits, the represented number being the relative record number on the CRM file; CRM keys are not involved here.

KEYWORD – An identifier that has special significance to the compiler and run-time routines when used in the context for which it was designed.

LABEL CONSTANT – A label that appears in a label prefix on any statement except ENTRY, FORMAT, and PROCEDURE.

LABEL PREFIX – The portion of a statement that identifies and provides reference to the statement. The prefix can be used on any statement except ENTRY, FORMAT, and PROCEDURE.

LABEL VALUE – A value that identifies both a label constant and a block activation.

LABEL VARIABLE – A variable that is declared with the LABEL attribute and that can be set to a label value.

LANGUAGE CHARACTER SET – The set of characters that are defined as characters used in the PL/I language.

LEVEL NUMBER – An integer that specifies the position of an identifier in the hierarchy of a structure.

LIST-DIRECTED I/O – A form of stream I/O. The data is transmitted according to the data type of the transmitted value. Contrast with edit-directed I/O.

LITERAL CONSTANT – An unnamed data item that has a scalar computational value. Literal constants are arithmetic, character string, or bit string.

LOCAL FILE NAME – The name of a local CRM file as known by CYBER Record Manager and the operating system. The name consists of letters A through Z and digits 0 through 9; the name must be one to seven characters in length and must begin with a letter.

LOCAL GOTO – A GOTO statement that transfers control to a label value such that the label constant part identifies a statement immediately contained in the same block as the GOTO statement and such that the block activation part identifies the current block activation.

LOCAL REFERENCE – A reference to an identifier that is declared in the same block as the reference.

LOCATOR-QUALIFIED REFERENCE – One form of reference to a based variable.

LOCATOR QUALIFIER – A term that refers to both the locator and the -> symbol in a locator-qualified reference.

LOCATOR VARIABLE – A variable that holds locator values, that is, pointer or offset values.

LOGICAL RECORD – A data grouping under NOS that consists of one or more PRUs terminated by a short PRU or zero-length PRU; equivalent to a system-logical-record under NOS/BE.

LOWER BOUND – The lower limit of subscript values for a particular dimension of an array.

MAIN PROCEDURE – The single external procedure that includes the OPTIONS(MAIN) clause on the PROCEDURE statement. Program execution is initiated at the primary entry point of the main procedure.

MANTISSA – The portion of a floating point number that is raised to the power specified by the exponent portion.

MEMBER – A variable that is contained in a structure.

MODE – A characteristic of arithmetic data; mode is specified by the REAL attribute, which indicates that the data has no imaginary part.

NAMED CONSTANT – An identifier representing a noncomputational value that does not change during program execution.

NESTING – The occurrence of a block within another block; a group within another group; an IF statement in a THEN or ELSE clause; a function reference as an argument of a function reference; a remote format item in the format list of a FORMAT statement; a parameter descriptor list in another parameter descriptor list.

NOISE RECORD – Number of characters the tape drivers discard as being extraneous noise rather than a valid record. Value depends on installation settings.

NONCOMPUTATIONAL VARIABLE – A variable used to represent area, locator, or named constant values.

NONITERATIVE DO GROUP – A do group that is executed once each time it is encountered in the program.

NONLOCAL GOTO – A GOTO statement that transfers control to a label value such that the label constant part identifies a statement not immediately contained in the same block as the GOTO statement, or such that the block activation part identifies a block activation other than the current block activation.

NONLOCAL REFERENCE – A reference to an identifier that is declared in a containing block.

NORMAL BLOCK TERMINATION – Termination of a block activation in a manner such that control returns in the normal manner to the predecessor block activation. Execution of the END statement of a currently active block causes normal termination of that block activation. Execution of a RETURN statement causes normal termination of the most recently activated procedure block activation.

NORMAL ON-UNIT TERMINATION – Termination of an on-unit activation in a manner such that control returns to the point of interrupt. Execution of the END statement of an on-unit causes normal termination of that on-unit activation. Completion of execution of a single statement on-unit, when that statement is not a GOTO, causes normal termination of that on-unit activation.

NULL ARGUMENT LIST – A set of empty parentheses ( ) used as an argument list.

NULL ON-UNIT – An on-unit represented by a ; (semicolon). A null on-unit indicates no action is to be taken for the raised condition.

NULL STRING – A data string that has a length of zero.

OFFSET – A value that identifies the position of a based variable in an area, relative to the beginning of the area.

ON-UNIT – Action to be taken when the condition specified in an ON statement is raised and results in an interrupt.

ON-UNIT ACTIVATION – A specific execution of an on-unit; the process of activating an on-unit.

ON-UNIT TERMINATION – The removal of an on-unit activation from the dynamic stack. On-unit termination is classified as normal or abnormal.

OPERAND – An expression used in conjunction with an operator so that the indicated operation utilizes the value of the expression.

OPERATOR – A symbol that specifies an operation to be performed. See Arithmetic operator, Bit string operator, Concatenate operator, and Comparison operator.

OVERFLOW – A situation in which the result of a floating point calculation yields an exponent that exceeds the maximum allowed by the hardware.

PAGE EJECTION – Line printer action that advances paper to a new page.

PARAMETER – See Parameter variable.

PARAMETER DESCRIPTOR – The establishment of a set of attributes for a particular position in a parameter list. A descriptor differs from a declaration in that there is no identifier.

PARAMETER VARIABLE – A variable name that appears in a parameter list of a PROCEDURE or ENTRY statement. The variable name is used to refer to an argument received by a procedure.

PARTIALLY QUALIFIED REFERENCE – A reference that includes zero or more, but not all, names in the hierarchical structure above a member.

PARTITION – CYBER Record Manager defines a partition as a division within a file with sequential organization. Generally, a partition contains several records or sections. Implementation of a partition boundary is affected by file structure and residence.

Device	RT	BT	Physical Boundary
PRU device	W	C	A short PRU of level 0 containing a control word with a flag indicating partition boundary.
	F,Z	C	A short PRU of level 0 followed by a zero-length PRU of level 17.
S or L tape format	W	C	Separate tape block containing as many deleted records or record length 0 as required to exceed noise record size, followed by a control word with a flag indicating a partition boundary.
	F,Z	C	Tapemark
	S	C	Zero-length PRU of level number 0.
Any other tape format			Undefined

In a file with W type records, a short PRU of level 0 terminates both a section and a partition.

PICTURE – A group of codes that describe the type and form of a specific pictured data item. The picture specification is enclosed in apostrophes.

PICTURED CHARACTER ITEM – A character value associated with a picture that contains only character codes.

PICTURED NUMERIC ITEM – A character value associated with a picture that contains a valid combination of the codes 9 Z * Y (digit); V (decimal point); S + - (sign position); T I R (signed digit); CR DB (sign suffix); $ (currency); / , . B (insertion); F (scaling factor); and K E (exponent).

PICTURED VARIABLE – A variable that holds values under control of a declared picture.

PL/I FILE – See File.

POINT OF INTERRUPT – The point at which a condition, which results in a program interrupt, is raised during program execution.

POINTER – A value that identifies the position of storage allocated for a variable.

PRECISION – The value that establishes the number of significant digits for an arithmetic value.

PREFIX – The portion of a statement that precedes the statement body; prefixes are classified as condition and statement.

PREFIX EXPRESSION – A single prefix operator and a single operand.

PREFIX OPERATOR – An operator that operates on a single operand. The prefix operators are + (positive), - (negative), ¬ (NOT).

PRIMARY ENTRY POINT – An entry point denoted by a prefix on the PROCEDURE statement.

PRIMITIVE EXPRESSION – A literal constant, variable reference, or function reference used for its value in an expression.

PRINT FILE – A stream file intended for printing on a line printer.

PROCEDURE BLOCK – A collection of statements headed by a PROCEDURE statement and terminated by a matching END statement. A procedure block is activated when invoked at any entry point.

PROCEDURE INVOCATION – The process of activating a procedure by execution of a CALL statement or evaluation of a function reference.

PROGRAM – A set of one or more external procedures, including one main procedure.

PROGRAM ABORT – Immediate program termination; occurs when the system ERROR on-unit is activated, when the STORAGE on-unit cannot be activated, and when a missing external procedure is invoked.

PROGRAM INTERRUPT – See Interrupt.

PROGRAM TERMINATION – Termination of the first or only activation of the main procedure, or execution of a STOP statement.

PROGRAMMED ON-UNIT – Programmer-generated code that specifies the action to be taken when a condition is raised and an interrupt occurs.

PROGRAMMER-NAMED CONDITION – A condition that is defined by the programmer through keyword CONDITION; can be raised only by SIGNAL statement.

PROLOGUE – Actions taken when a block is activated. Actions include evaluation of extents, processing of automatic and defined variables, and establishment of appropriate condition builtin functions.

PRU (PHYSICAL RECORD UNIT) – Under NOS and NOS/BE, the amount of information transmitted by a single physical operation of a specified device. A PRU that does not contain as much user data as the PRU can hold is called a short PRU; a PRU that has a level terminator but no user data is called a zero-length PRU. The size of a PRU depends on the device.

Device	Size in Number of 60-Bit Words
Mass storage	64
Tape in SI format with coded data†	128
Tape in SI format with binary data	512
Tape in I format	512
Tape in other format	Undefined

†Not supported under NOS.

PRU DEVICE – A mass storage device or a tape in SI or I format. Records on these devices are written in PRUs.

PSEUDOVARIABLE – A builtin function name that is used as the target of an assignment operation. Pseudovariables are ONCHAR, ONSOURCE, PAGENO, SUBSTR, and UNSPEC.

RADIX FACTOR – The suffix B that indicates a binary base; used for binary arithmetic constants and for bit string constants.

RECORD – (1) As applied to PL/I, a unit of information for transmission by record I/O; the contents of an allocation unit represented in internal form. (2) As applied to CYBER Record Manager, a group of related characters. A record or a portion thereof is the smallest collection of information passed between CRM and a user program. Eight different record types exist, as defined by the RT field of the File Information Table. Other parts of the operating systems and their products might have additional or different definitions of records.

RECORD I/O – The input or output of complete records transmitted in internal form. Transmission is by READ, WRITE, REWRITE, DELETE, and LOCATE statements.

RECORD TYPE – The term record type can have one of several meanings, depending on the context. CYBER Record Manager defines eight record types established by an RT field in the File Information Table. Tables output by the loader are classified as record types such as text, relocatable, or absolute, depending on the first few words of the tables. Record types F, Z, W, and S can be used for data transmission in PL/I.

RECURSION – The activation of a block that is already active.

REFERENCE – The appearance of an identifier in any context other than one that represents an explicit declaration of the identifier.

REMOTE FORMAT ITEM – The letter R and the parenthesized remote-format name that appears in a format-specification.

REPETITION FACTOR – A parenthesized value indicating the number of times a character or bit constant is repeated, or the number of times a picture code occurs.

RETURNED VALUE – The value returned from a procedure invoked as a function, or the value returned by a builtin function.

RETURNS DESCRIPTOR – The establishment of a set of attributes for the value returned from a procedure that is invoked as a function. A descriptor differs from a declaration in that there is no identifier.

SCALAR – A data item that represents a single value.

SCALE – A characteristic of arithmetic data; either the FIXED or FLOAT attribute. The scale controls whether the fixed point or floating point method of calculation is used.

SCALING FACTOR CODE – The picture code F (scaling factor for arithmetic value).

SCOPE OF CONDITION PREFIX – The portion of the program within which a condition prefix applies.

SCOPE OF DECLARATION – The portion of the program within which a declared identifier is known.

SECONDARY ENTRY POINT – An entry point denoted by a prefix on an ENTRY statement.

SECTION – CYBER Record Manager defines a section as a division within a file with sequential organization. Generally, a section contains more than one record and is a division within a partition of a file. A section terminates with a physical representation of a section boundary.

Device	RT	BT	Physical Representation
PRU device	W	C	Control word with flags indicating a section boundary. The control word is in a short PRU of level 0.
	F,Z	C	Short PRU with level less than 17 octal.
	S	C	Undefined.
S or L format tape	W	C	A separate tape block containing as many deleted records of record length 0 required to exceed noise record size followed by a control word with flags indicating a section boundary.
	F,Z	C	Undefined.
	S	C	Undefined.
Any other tape format			Undefined.

The NOS and NOS/BE operating systems equate a section with a system-logical-record of level 0 through 16 octal.

SEQUENTIAL (SQ) – (1) As applied to PL/I, a type of record file in which records can be accessed in a sequential manner. (2) As applied to CYBER Record Manager, a file organization in which records are stored in the order they were generated.

SHORT PRU – A PRU that does not contain as much user data as the PRU can hold and that is terminated by a system terminator with a level number. Under NOS, a short PRU defines EOR. Under NOS/BE, a short PRU defines the end of a system-logical-record. In the CYBER Record Manager context, a short PRU can have several interpretations depending on the record and blocking types.

SIGN POSITION CODE – A picture code that specifies placement of a sign or a blank. The sign position codes are S (sign), + (positive), and - (negative).

SIGN SUFFIX CODE – A picture code that specifies placement of a character pair as a suffix that indicates the sign. The sign suffix codes are CR (credit) and DB (debit).

SIGNED DIGIT CODE – One of the picture codes T (overpunch, positive or negative), I (overpunch positive), and R (overpunch negative).

SIMPLE REFERENCE – A reference to a variable or a named constant.

SINGLE STATEMENT – A statement that cannot contain embedded statements and can be executed. All statements except compound statements (IF and ON), group statements (BEGIN, DO, and PROCEDURE), and the FORMAT statement are classified as single statements.

SNAP OUTPUT – Diagnostic information that is written to file SYSPRINT when an interrupt occurs; specified by the SNAP option in the ON statement.

SPACE – A blank character.

SPAN – The number of possible subscript values for a particular dimension of an array, as established by the lower bound and upper bound.

STACK – See Dynamic stack.

STANDARD SYSTEM ACTION – The system-specified action to be taken when an interrupt occurs and a programmed on-unit for the appropriate condition does not exist.

STATEMENT – An instruction that consists of the language keywords in combination with programmer-supplied elements; establishes names to be used in the program, states the characteristics that are to be associated with the names, and describes the operations to be performed at run time.

STATEMENT BODY – The portion of a statement between the prefix and the semicolon.

STATEMENT PREFIX – The portion of a statement that identifies and provides a means of referencing the statement; classified as entry, format, and label.

STATIC VARIABLE – A variable that has storage allocated before execution begins.

STORAGE ALLOCATION – The reserving of storage for a variable or buffer.

STREAM I/O – Transmission of input or output data where the data is considered as a continuous stream of characters; transmission is by GET and PUT statements.

STRING – A sequence of contiguous characters or bits treated as a unit.

STRING VARIABLE – A variable that holds string values.

STRUCTURE – A collection of data elements where each element can have a different name and a different set of attributes. Contrast with array.

STRUCTURE-QUALIFIED REFERENCE – One form of reference to a member of a structure.

SUBEXPRESSION – An expression that is part of a larger expression.

SUBROUTINE – A procedure that is invoked by CALL statement execution.

SUBSCRIPT – A value used to select an element in a particular dimension of an array.

SUBSCRIPTED REFERENCE – A reference to one or more elements of an array.

SUBSTRING – A character string that is part of a larger string.

SUBSTRUCTURE – A structure member that is a structure with members.

SYNTAX – The rules governing the structure of PL/I language components.

SYSIN – System default input file for program execution; associated with CRM file INPUT.

SYSPRINT – System default output file for program execution; associated with CRM file OUTPUT. Also used at run time as the error file.

SYSTEM-LOGICAL-RECORD – Under NOS/BE, a data grouping that consists of one or more PRUs terminated by a short PRU or zero-length PRU. These records can be transferred between devices without loss of structure. Equivalent to a logical record under NOS. Equivalent to a CYBER Record Manager S type record.

SYSTEM ON-UNIT – An on-unit specifying the standard system action for a particular condition.

TARGET VARIABLE – A variable to which a value is assigned.

TERMINATION OF A BLOCK – See Block termination.

TERMINATION OF AN ON-UNIT – See On-unit termination.

TITLE – See File title

UNALIGNED VALUE – A value not necessarily aligned on a word boundary in storage.

UNCONDITIONAL TRANSFER – The execution of a GOTO statement.

UNDERFLOW – A situation that occurs when the result of a floating point calculation yields an exponent that is smaller than the minimum allowed by the hardware.

UPPER BOUND – The upper limit of subscript values for a particular dimension of an array.

VARIABLE – An established identifier representing a value or values that can change during program execution.

WORD – A sequence of characters formed from the language character set; classified as keyword or identifier.

WORD ADDRESSABLE (WA) – A file organization in which records are identified by a key. The key indicates the relative record number within the file.

ZERO-BYTE TERMINATOR – A series of 12 zero bits in the low order position of a word. The terminator marks the end of the line to be displayed at a terminal or printed on a line printer. The image of cards input through the card reader or terminal also has such a terminator.

ZERO-LENGTH PRU – A PRU that contains system information, but no user data. Under CYBER Record Manager, a zero-length PRU of level 17 is a partition boundary. Under NOS, a zero-length PRU defines EOF.

KEYWORDS AND BUILTIN FUNCTION NAMES

This appendix provides an alphabetic list of PL/I keywords and builtin function names. The keywords (and the keyword abbreviations) are not reserved words; the special recognition of a keyword occurs only in the appropriate context. The builtin function names (and the abbreviations) are not keywords and not reserved words; the builtin function names are special identifiers used to reference the supplied functions.

Shading indicates keywords and builtin function names that are not recognized.

NOTE

The identifiers SYSIN and SYSPRINT should be treated as reserved identifiers. SYSIN is the default input file for stream I/O; SYSPRINT is the default output file for stream I/O and run-time error messages.

Keyword or identifier	Recognized in appropriate context as:
A	Format item
ABS	Builtin function
ACOS	Builtin function
ADD	Builtin function
ADDR	Builtin function
AFTER	Builtin function
ALIGNED	Attribute
ALLOCATE ALLOC	Statement
ALLOCATION ALLOCN	Builtin function
AREA	Attribute
AREA	Condition
ASIN	Builtin function
ATAN	Builtin function
ATAND	Builtin function
ATANH	Builtin function
AUTOMATIC AUTO	Attribute
B	Format item
B1,B2,B3,B4	Format items
BASED	Attribute
BEFORE	Builtin function
BEGIN	Statement
BINARY BIN	Attribute
BINARY BIN	Builtin function
BIT	Attribute
BIT	Builtin function
BOOL	Builtin function
BUILTIN	Attribute
BY	Option in do-specification in LIST or EDIT option (GET, PUT)
BY	Statement option (DO)
BY NAME	Statement option (Assignment)
C	Format item
CALL	Statement
CEIL	Builtin function
CHARACTER CHAR	Attribute
CHARACTER CHAR	Builtin function
CLOSE	Statement
COLLATE	Builtin function
COLUMN COL	Format item
COMPLEX CPLX	Attribute
COMPLEX CPLX	Builtin function
CONDITION COND	Attribute
CONDITION COND	Condition
CONJG	Builtin function
CONSTANT	Attribute
CONTROLLED CTL	Attribute

Keyword or identifier	Recognized in appropriate context as:
CONVERSION CONV	Condition
CONVERSION CONV	Enabled-condition name
COPY	Builtin function
COPY	Statement option (GET)
COS	Builtin function
COSD	Builtin function
COSH	Builtin function
DATA	Statement option (GET, PUT)
DATE	Builtin function
DECAT	Builtin function
DECIMAL DEC	Attribute
DECIMAL DEC	Builtin function
DECLARE DCL	Statement
DEFAULT DFT	Statement
DEFINED DEF	Attribute
DELETE	Statement
DIMENSION DIM	Attribute
DIMENSION DIM	Builtin function
DIRECT	Attribute
DIRECT	Statement option (OPEN)
DIVIDE	Builtin function
DO	Option in LIST or EDIT option (GET, PUT)
DO	Statement
DOT	Builtin function
E	Format item
EDIT	Statement option (GET, PUT)
ELSE	Statement option (IF)
EMPTY	Builtin function
END	Statement
ENDFILE	Condition
ENDPAGE	Condition
ENTRY	Attribute
ENTRY	Statement
ENVIRONMENT ENV	Attribute
ENVIRONMENT ENV	Statement option (CLOSE, OPEN)
ERF	Builtin function
ERFC	Builtin function
ERROR	Condition
EVERY	Builtin function
EXP	Builtin function
EXTERNAL EXT	Attribute
F	Format item
FILE	Attribute
FILE	Statement option (I/O statements)
FINISH	Condition
FIXED	Attribute
FIXED	Builtin function
FIXEDOVERFLOW FOFL	Condition
FIXEDOVERFLOW FOFL	Enabled-condition name
FLOAT	Attribute
FLOAT	Builtin function
FLOOR	Builtin function
FORMAT	Attribute
FORMAT	Statement
FREE	Statement
FROM	Statement option (WRITE, REWRITE)
GENERIC	Attribute
GET	Statement
GOTO GO TO	Statement
HBOUND	Builtin function

Keyword or identifier	Recognized in appropriate context as:
HIGH	Builtin function
IF	Statement
IGNORE	Statement option (READ)
IMAG	Builtin function
IN	Statement option (ALLOCATE, FREE)
%INCLUDE	Pre-processor statement
INDEX	Builtin function
INITIAL INIT	Attribute
INPUT	Attribute
INPUT	Statement option (OPEN)
INTERNAL INT	Attribute
INTO	Statement option (READ)
KEY	Condition
KEY	Statement option (READ, DELETE, REWRITE)
KEYED	Attribute
KEYED	Statement option (OPEN)
KEYFROM	Statement option (WRITE, LOCATE)
KEYTO	Statement option (READ)
LABEL	Attribute
LBOUND	Builtin function
LENGTH	Builtin function
LIKE	Attribute
LINE	Format item
LINE	Statement option (PUT)
LINENO	Builtin function
LINESIZE	Statement option (OPEN)
LIST	Statement option (GET, PUT)
LOCAL	Attribute
LOCATE	Statement
LOG	Builtin function
LOG10	Builtin function
LOG2	Builtin function
LOW	Builtin function
MAIN	Option in OPTIONS option (PROCEDURE)
MAX	Builtin function
MEMBER	Attribute
MIN	Builtin function
MOD	Builtin function
MULTIPLY	Builtin function
NAME	Condition
NOCONVERSION NOCONV	Disabled-condition name
NOFIXEDOVERFLOW NOFOFL	Disabled-condition name
NONE	Statement option (DEFAULT)
NONVARYING NONVAR	Attribute
NOOVERFLOW NOOFL	Disabled-condition name
NOSIZE	Disabled-condition name
NOSTRINGRANGE NOSTRG	Disabled-condition name
NOSTRINGSIZE NOSTRZ	Disabled-condition name
NOSUBSCRIPTRANGE NOSUBRG	Disabled-condition name
NOUNDERFLOW NOUFL	Disabled-condition name
NOZERODIVIDE NOZDIV	Disabled-condition name
NULL	Builtin function
OFFSET	Attribute
OFFSET	Builtin function
ON	Statement
ONCHAR	Builtin function
ONCHAR	Pseudovariable
ONCODE	Builtin function
ONFIELD	Builtin function
ONFILE	Builtin function
ONKEY	Builtin function

Keyword or identifier	Recognized in appropriate context as:
ONLOC	Builtin function
ONSOURCE	Builtin function
ONSOURCE	Pseudovariable
OPEN	Statement
OPTIONS	Statement option (PROCEDURE)
OUTPUT	Attribute
OUTPUT	Statement option (OPEN)
OVERFLOW OFL	Condition
OVERFLOW OFL	Enabled-condition name
P	Format item
PAGE	Format item
PAGE	Statement option (PUT)
PAGENO	Builtin function
PAGENO	Pseudovariable
PAGESIZE	Statement option (OPEN)
PARAMETER PARM	Attribute
PICTURE PIC	Attribute
POINTER PTR	Attribute
POINTER PTR	Builtin function
POSITION POS	Attribute
PRECISION PREC	Attribute
PRECISION PREC	Builtin function
PRINT	Attribute
PRINT	Statement option (OPEN)
PROCEDURE PROC	Statement
PROD	Builtin function
PUT	Statement
R	Format item
RANGE	Statement option (DEFAULT)
READ	Statement
REAL	Attribute
REAL	Builtin function
RECORD	Attribute
RECORD	Condition
RECORD	Statement option (OPEN)
RECURSIVE	Statement option (PROCEDURE)
REFER	Option in extent (DECLARE)
REPEAT	Statement option (DO)
RETURN	Statement
RETURNS	Attribute
RETURNS	Statement option (PROCEDURE, ENTRY)
REVERSE	Builtin function
REVERT	Statement
REWRITE	Statement
ROUND	Builtin function
SEQUENTIAL SEQL	Attribute
SEQUENTIAL SEQL	Statement option (OPEN)
SET	Statement option (ALLOCATE, LOCATE, READ)
SIGN	Builtin function
SIGNAL	Statement
SIN	Builtin function
SIND	Builtin function
SINH	Builtin function
SIZE	Condition
SIZE	Enabled-condition name
SKIP	Format item
SKIP	Statement option (GET, PUT)
SNAP	Statement option (ON)
SOME	Builtin function
SQRT	Builtin function
STATIC	Attribute
STOP	Statement
STORAGE	Condition

Keyword or identifier	Recognized in appropriate context as:
STREAM	Attribute
STREAM	Statement option (OPEN)
STRING	Builtin function
STRING	Pseudovariable
STRING	Statement option (GET, PUT)
STRINGRANGE STRG	Condition
STRINGRANGE STRG	Enabled-condition name
STRINGSIZE STRZ	Condition
STRINGSIZE STRZ	Enabled-condition name
STRUCTURE	Attribute
SUBSCRIPTRANGE SUBRG	Condition
SUBSCRIPTRANGE SUBRG	Enabled-condition name
SUBSTR	Builtin function
SUBSTR	Pseudovariable
SUBTRACT	Builtin function
SUM	Builtin function
SYSTEM	Statement option (ON)
TAB	Format item
TAB	Statement option (OPEN)
TAN	Builtin function
TAND	Builtin function
TANH	Builtin function
THEN	Statement option (IF)
TIME	Builtin function
TITLE	Statement option (OPEN)
TO	Option in do-specification in LIST or EDIT option (GET, PUT)
TO	Statement option (DO)
TRANSLATE	Builtin function
TRANSMIT	Condition
TRUNC	Builtin function
UNALIGNED UNAL	Attribute
UNDEFINEDFILE UNDF	Condition
UNDERFLOW UFL	Condition
UNDERFLOW UFL	Enabled-condition name
UNSPEC	Builtin function
UNSPEC	Pseudovariable
UPDATE	Attribute
UPDATE	Statement option (OPEN)
VALID	Builtin function
VARIABLE	Attribute
VARYING VAR	Attribute
VERIFY	Builtin function
WHEN	Option in GENERIC attribute (DECLARE)
WHILE	Option in do-specification in LIST or EDIT option (GET, PUT)
WHILE	Statement option (DO)
WRITE	Statement
X	Format item
ZERODIVIDE ZDIV	Condition
ZERODIVIDE ZDIV	Enabled-condition name

SYNTAX SUMMARY

E

A summary of language syntax appears in this appendix. Detailed information for each syntax is referenced by page number. The summary includes entries listed in alphabetic order, where appropriate, in the following major categories:

Blocks and Do Groups
Statements
Prefixes
Attributes
References
Expressions
Picture Specification

BLOCKS AND DO GROUPS

Page

BEGIN BLOCK

1-1

```
BEGIN-statement
    [ block-unit ] . . .
END-statement
```

where block-unit is

```
procedure-block
begin-block
do-group
ALLOCATE-statement
assignment-statement
CALL-statement
CLOSE-statement
DECLARE-statement
DELETE-statement
FORMAT-statement
FREE-statement
GET-statement
GOTO-statement
IF-statement
LOCATE-statement
null-statement
ON-statement
OPEN-statement
PUT-statement
READ-statement
RETURN-statement
REVERT-statement
REWRITE-statement
SIGNAL-statement
STOP-statement
WRITE-statement
```

Page

DO GROUP

1-2

DO-statement
    [block-unit] . . .
END-statement

where block-unit is

(procedure-block
| begin-block
| do-group
| ALLOCATE-statement
| assignment-statement
| CALL-statement
| CLOSE-statement
| DECLARE-statement
| DELETE-statement
| ENTRY-statement
| FORMAT-statement
| FREE-statement
| GET-statement
| GOTO-statement
| IF-statement
| LOCATE-statement
| null-statement
| ON-statement
| OPEN-statement
| PUT-statement
| READ-statement
| RETURN-statement
| REVERT-statement
| REWRITE-statement
| SIGNAL-statement
| STOP-statement
| WRITE-statement)

PROCEDURE BLOCK

1-1

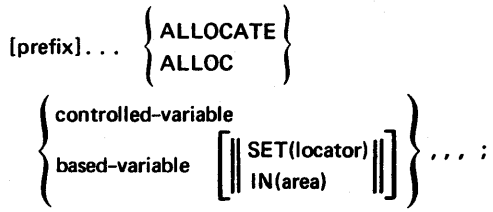
PROCEDURE-statement
    [block-unit] . . .
END-statement

where block-unit is

(procedure-block
| begin-block
| do-group
| ALLOCATE-statement
| assignment-statement
| CALL-statement
| CLOSE-statement
| DECLARE-statement
| DELETE-statement
| ENTRY-statement
| FORMAT-statement
| FREE-statement
| GET-statement
| GOTO-statement
| IF-statement
| LOCATE-statement
| null-statement
| ON-statement
| OPEN-statement
| PUT-statement
| READ-statement
| RETURN-statement
| REVERT-statement
| REWRITE-statement
| SIGNAL-statement
| STOP-statement
| WRITE-statement)

STATEMENTS

ALLOCATE STATEMENT



ASSIGNMENT STATEMENT

[prefix]... assignment-target, , , = expression ;

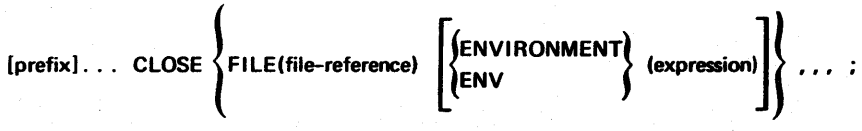
BEGIN STATEMENT

[prefix]... BEGIN ;

CALL STATEMENT

[prefix]... CALL subroutine-reference [([argument, , ,])] ;

CLOSE STATEMENT



[label:]. . . { DECLARE / DCL } declaration, , , ;

where declaration is

[level] { identifier / (declaration, , ,) } [dimension-suffix] [attribute] . . .

[prefix]. . . DELETE FILE(file-reference)

KEY(expression) ;

Noniterative DO

[prefix]. . . DO ;

DO WHILE

[prefix]. . . DO WHILE(expression) ;

Indexed DO

[prefix]. . . DO index = do-specification, , , ;

where do-specification is

start-expression [{ TO expression / BY expression }] [WHILE(expression)]

[label:]. . . END [closure-name] ;

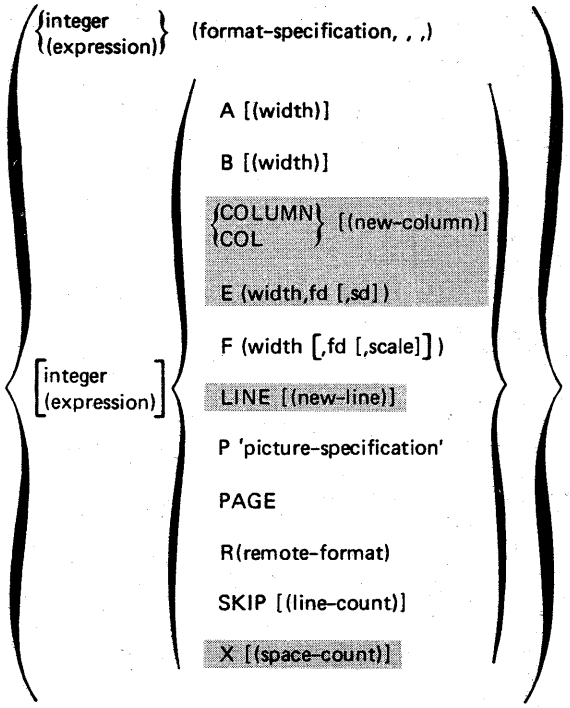
{entry-name:}. . . ENTRY [(parameter, , ,)]

[RETURNS [(returns-descriptor)]] ;

[condition-prefix]. . . {format-name:} . . .

FORMAT(format-specification, , ,) ;

where format-specification is



[prefix] . . . FREE

{controlled-variable
[locator-reference->]based-variable [IN(area)]}, , , ;

Page

GET STATEMENT

12-17

File Positioning

```
[prefix]. . . GET [COPY [(file-reference)]] [FILE(file-reference)] SKIP [(line-count)] ;
```

Stream Input

```
[prefix]. . . GET [ COPY [(file-reference)]] [ [FILE(file-reference)]   [SKIP [(line-count)]] ]
                                             [ STRING(input-source)                            ]

{ LIST(input-target , , ,)                                                                    }
{ EDIT {(input-target , , ,) (format-specification , , ,)}. . .                               } ;
```

where input-target is

```
{ target-variable                          }
{ pseudovariable                           }
{ (input-target , , , embedded-do)         }
```

where embedded-do is

```
DO index = { start-expression [ [TO expression] ]  [WHILE(expression)] } , , ,
                              [ [BY expression] ]
```

where format-specification is

```
{ {integer   }  (format-specification , , ,)                                                  }
{ {(expression)}                                                                              }
{                                                                                             }
{                                                                                             }
{ [integer   ]  { A (width)                                                                   }
{ [(expression)]{ B (width)                                                                   }
{               { {COLUMN} [(new-column)]                                                     }
{               { {COL   }                                                                    }
{               { E (width,fd [,sd])                                                          }
{               { F (width [,fd [,scale]])                                                    }
{               { P 'picture-specification'                                                   }
{               { R(remote-format)                                                            }
{               { SKIP [(line-count)]                                                         }
{               { X [(space-count)]                                                           }
```

Page

GOTO STATEMENT

12-20

[prefix]... { GOTO / GO TO } label-reference ;

IF STATEMENT

12-21

[prefix] ... IF expression

THEN executable-unit-1

[ELSE executable-unit-2]

where each executable-unit is

{ begin-block
do-group
ALLOCATE-statement
assignment-statement
CALL-statement
CLOSE-statement
DELETE-statement
FREE-statement
GET-statement
GOTO-statement
IF-statement
LOCATE-statement
null-statement
ON-statement
OPEN-statement
PUT-statement
READ-statement
RETURN-statement
REVERT-statement
REWRITE-statement
SIGNAL-statement
STOP-statement
WRITE-statement }

LOCATE STATEMENT

12-22

[prefix]... LOCATE based-variable

FILE(file-reference) [SET(pointer) / KEYFROM(expression)] ;

NULL STATEMENT

12-24

[prefix]... ;

Page

ON STATEMENT

12-24

[prefix]... ON condition [SNAP] { on-unit / SYSTEM ; }

where on-unit is

{ begin-block
ALLOCATE-statement
assignment-statement
CALL-statement
CLOSE-statement
DELETE-statement
FREE-statement
GET-statement
GOTO-statement
LOCATE-statement
null-statement
OPEN-statement
PUT-statement
READ-statement
RETURN-statement
REVERT-statement
REWRITE-statement
SIGNAL-statement
STOP-statement
WRITE-statement }

OPEN STATEMENT

12-24

[prefix]... OPEN

{ FILE(file-reference) [TITLE(expression) / { ENVIRONMENT / ENV } (expression)]
[[STREAM] [[INPUT] [LINESIZE(expression)] / [OUTPUT] [PRINT / LINESIZE(expression) / PAGESIZE(expression)]]
[RECORD] [INPUT / OUTPUT / UPDATE] [DIRECT / SEQUENTIAL / SEQL] [KEYED]] } ,...;

PROCEDURE STATEMENT

12-26

[condition-prefix]... {entry-name:}... { PROCEDURE / PROC }
[[(parameter, , ,)] [RETURNS [(returns-descriptor)] / RECURSIVE]
OPTIONS(MAIN) [RECURSIVE]] ;

Page

PUT STATEMENT

12-29

File Positioning

```
[prefix]. . . PUT [FILE(file-reference)] { SKIP [(line-count)]          }
                                         { LINE [(new-line)]            } ;
                                         { PAGE [LINE [(new-line)] ]    }
```

Stream Output

```
                                  [ [FILE(file-reference)] [ SKIP [(line-count)]          ] ]
[prefix]. . . PUT                 [                        [ [PAGE] [LINE [(new-line)] ] ] ]
                                  [ STRING(output-target)                                  ]

{ LIST(output-source , , ,)                                                  }
{ EDIT{(output-source , , ,) (format-specification , , ,)}. . .              } ;
```

where output-source is

```
{ expression                          }
{ (output-source , , , embedded-do)   }
```

where embedded-do is

```
DO index = { start-expression [ [ TO expression ] ] [WHILE(expression)] }, , ,
                              [ [ BY expression ] ]
```

where format-specification is

```
{ {integer    }                                                                  }
{ {(expression)} (format-specification , , ,)                                    }
{                                                                                }
{ [integer    ] { A [(width)]                                                    }
{ [(expression)] { B [(width)]                                                   }
{                { {COLUMN} [(new-column)]                                       }
{                { {COL   }                                                      }
{                { E (width,fd [,sd])                                            }
{                { F (width [,fd [,scale]] )                                     }
{                { LINE [(new-line)]                                             }
{                { P 'picture-specification'                                     }
{                { PAGE                                                          }
{                { R(remote-format)                                              }
{                { SKIP [(line-count)]                                           }
{                { X [(space-count)]                                             }
```

Page

READ STATEMENT 12-32

[prefix]. . . READ FILE(file-reference)

{INTO(read-target) / SET(pointer) / IGNORE(expression)} [KEY(expression) / KEYTO(key-target)] ;

RETURN STATEMENT 12-33

[prefix]. . . RETURN [(return-value)] ;

REVERT STATEMENT 12-34

[prefix]. . . REVERT condition ;

REWRITE STATEMENT 12-35

[prefix]. . . REWRITE FILE(file-reference)

[FROM(source) [KEY(expression)]] ;

SIGNAL STATEMENT 12-35

[prefix]. . . SIGNAL condition ;

STOP STATEMENT 12-36

[prefix]. . . STOP ;

WRITE STATEMENT 12-36

[prefix]. . . WRITE FILE(file-reference)

FROM(source) [KEYFROM(expression)] ;

PREFIXES

Page

CONDITION PREFIX

12-2

({ enabled-condition-name } , , ,) :
{ disabled-condition-name }

ENTRY PREFIX

12-2

entry-name :

FORMAT PREFIX

12-2

format-name :

LABEL PREFIX

12-2

label [(subscript , , ,)] :

ATTRIBUTES

ALIGNED ATTRIBUTE

4-6

ALIGNED

AREA ATTRIBUTE

4-7

AREA [(size)]

AUTOMATIC ATTRIBUTE

4-7

{ AUTOMATIC }
{ AUTO }

Page

DEFINED ATTRIBUTE 4-11

```
{DEFINED}  { host-reference  }  [{POSITION}  [(start-pos)]]
{DEF    }  {(host-reference)}  [{POS     }
```

DIMENSION ATTRIBUTE 4-12

(array-bounds , , ,)

where array-bounds is

```
{upper-bound          }
{lower-bound:upper-bound}
{*                    }
```

DIRECT ATTRIBUTE 4-12.1/4-12.2

DIRECT

ENTRY ATTRIBUTE 4-13

```
{ENTRY                              }
{ENTRY ( )                          }
{ENTRY (parameter-descriptor , , , )}
```

ENVIRONMENT ATTRIBUTE 4-14

```
{ENVIRONMENT} (options)
{ENV        }
```

EXTERNAL ATTRIBUTE 4-14

```
{EXTERNAL}
{EXT     }
```

For initialization of scalars

```
{INITIAL}  ( [±] arithmetic-constant
{INIT   }    simple-character-constant
             replicated-character-constant
             simple-bit-constant
             replicated-bit-constant
             reference
             (expression)
             * )
```

For initialization of arrays

```
{INITIAL}
{INIT   }   (initial-element , , , )
```

where initial-element is

```
[(iteration-factor)] { [±] arithmetic-constant
                     { replicated-character-constant
                     { replicated-bit-constant
                     { reference
                     { *

simple-character-constant
simple-bit-constant
(expression)
(iteration-factor) (initial-element , , , )
```

Page

PICTURE ATTRIBUTE 4-20

{PICTURE / PIC} 'picture-specification'

where picture-specification is

{'pictured-character' / 'pictured-numeric-fixed' / 'pictured-numeric-float'}

POINTER ATTRIBUTE 4-20

{POINTER / PTR}

POSITION ATTRIBUTE 4-11

See DEFINED

PRECISION ATTRIBUTE 4-20

{(p,q) / (p)}

PRINT ATTRIBUTE 4-21

PRINT

REAL ATTRIBUTE 4-21

REAL

RECORD ATTRIBUTE 4-21

RECORD

RETURNS ATTRIBUTE 4-22

{RETURNS / RETURNS (returns-descriptor)}

Page

SEQUENTIAL ATTRIBUTE 4-22

```
{SEQUENTIAL}
{SEQL      }
```

STATIC ATTRIBUTE 4-23

STATIC

STREAM ATTRIBUTE 4-23

STREAM

STRUCTURE ATTRIBUTE 4-23

For variables

```
1 structure-name { [attribute] . . . , { level member-name [attribute] . . .                                  }, , , }
                 {                     { level member-name LIKE structure-reference [attribute] . . .        }       }
                 { LIKE structure-reference                                                                        }
```

For parameter descriptors

```
1 [attribute] . . . , { level [attribute] . . . } , , ,
```

UNALIGNED ATTRIBUTE 4-6

```
{UNALIGNED}
{UNAL     }
```

UPDATE ATTRIBUTE 4-25

UPDATE

VARYING ATTRIBUTE 4-25

VARYING

REFERENCES

Page

BUILTIN FUNCTION AND PSEUDOVARIABLE REFERENCE 6-13

simple-reference [([argument , , ,])]

FUNCTION REFERENCE 6-4

simple-reference ([argument , , ,])

LOCATOR-QUALIFIED REFERENCE 6-3

locator-reference -> based-reference

SIMPLE REFERENCE 6-2

identifier

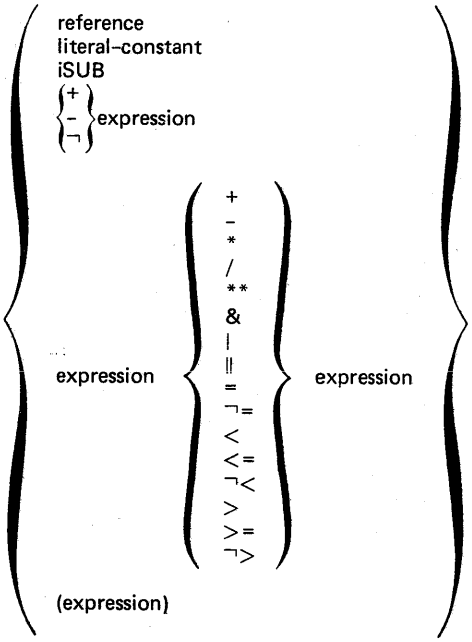
STRUCTURE-QUALIFIED REFERENCE 6-3

{structure-qualifier.}... member-reference

SUBSCRIPTED REFERENCE 6-2

identifier(subscript , , ,)

EXPRESSIONS 7-1



PICTURE SPECIFICATION

$$\left\{\begin{array}{l}\text{pictured-character}\\\text{pictured-numeric-fixed}\\\text{pictured-numeric-float}\end{array}\right\}$$

where pictured-character must include at least one A or X code and is

$$\left\{\begin{array}{l}9\\A\\X\end{array}\right\}\ldots$$

where pictured-numeric-fixed must include at least one of the codes 9 Z * Y T I or R, must include no more than one sign code S + - T I R CR or DB, can include insertion codes / , . or B at any position, and is

$$\left\{\begin{array}{l}\text{nondrifting-field}\\\text{drifting-field}\end{array}\right\}\ \left[F(\left[\begin{array}{l}+\\-\end{array}\right]\ \text{integer})\right]$$

where nondrifting-field is

$$\left\{\begin{array}{l}[\$]\bullet\left[\begin{array}{l}S\\+\\-\end{array}\right]\bullet\ \text{digits}\\[\$]\bullet\text{digits}\ \left\{\begin{array}{l}CR\\DB\end{array}\right\}\end{array}\right\}$$

where digits is

$$\left\{\begin{array}{l}\left[\begin{array}{l}Z\ldots\\*\ldots\end{array}\right][\text{digit-pos}]\ldots[V]\ [\text{digit-pos}]\ldots\\\left[\begin{array}{l}Z\ldots\\*\ldots\end{array}\right]V\,Z\ldots\\\qquad\quad V\,*\ldots\end{array}\right\}$$

where digit-pos is

$$\left\{\begin{array}{l}9\\Y\\T\\I\\R\end{array}\right\}$$

where drifting-field is

$$\left\{\begin{array}{l}[\$]\bullet\ \text{drifting-sign}\\\left[\begin{array}{l}S\\+\\-\end{array}\right]\bullet\left\{\begin{array}{l}\$\$\ldots[\text{digit-pos}]\ldots[V]\ [\text{digit-pos}]\ldots\\\$\,[\$\ldots]\,V\,\$\ldots\end{array}\right\}\\\left\{\begin{array}{l}\$\$\ldots\left[\begin{array}{l}9\\Y\end{array}\right]\ldots[V]\left[\begin{array}{l}9\\Y\end{array}\right]\ldots\\\$\,[\$\ldots]\,V\,\$\ldots\end{array}\right\}\left\{\begin{array}{l}CR\\DB\end{array}\right\}\end{array}\right\}$$

where drifting-sign is

$$\left\{\begin{array}{l}SS\ldots\\++\ldots\\--\ldots\end{array}\right\}\left[\begin{array}{l}9\\Y\end{array}\right]\ldots[V]\left[\begin{array}{l}9\\Y\end{array}\right]\ldots$$

where pictured-numeric-float is

$$\text{mantissa}\ \left\{\begin{array}{l}E\\K\end{array}\right\}\ \text{exponent}$$

where mantissa must include at least one of the codes 9 Z * Y T I or R, must include no more than one sign code S + - T I or R, can include insertion codes / , . or B at any position, and is

$$\left\{\begin{array}{l}\left[\begin{array}{l}S\\+\\-\end{array}\right]\ \text{digits}\\\text{drifting-sign}\end{array}\right\}$$

and where exponent must include at least one of the codes 9 Z * Y T I or R, must include no more than one sign code S + - T I or R, can include insertion codes / , . or B at any position, and is

$$\left[\begin{array}{l}S\\+\\-\end{array}\right]\ \left\{\begin{array}{l}\text{digit-pos}\ldots\\\left\{\begin{array}{l}Z\ldots\\*\ldots\end{array}\right\}[\text{digit-pos}]\ldots\end{array}\right\}$$

This appendix summarizes the relationship between this implementation of PL/I and the language described in American National Standard Programming Language PL/I, ANSI X3.53-1976.

In this appendix, each special notation within parentheses indicates the relevant section or line in the ANSI standard. The special notations take the following form:

(Sn.n. . .)	indicates a section number in the ANSI standard.
CHn	is used with the section number to indicate a specific line in the high-level concrete syntax.
CMn	is used with the section number to indicate a specific line in the middle-level concrete syntax.
CLn	is used with the section number to indicate a specific line in the low-level concrete syntax.

The relationship is described with respect to extensions, differences, and deviations. Information about implementation-defined values is also provided.

EXTENSIONS

An extension exists if program behavior or syntax is not standard but the behavior or syntax represents an additional feature. The following are extensions to the ANSI standard:

- Additional alphabetic characters in identifiers – The characters #, @, and $ are allowed as alphabetic characters in identifiers (S2.5.3).

- INRULE control statement option – The I through N rule is allowed as a control statement option to facilitate the conversion of existing programs. If the INRULE option is selected, default arithmetic attributes are added by taking the following rules in order:

  If the identifier begins with the letter I through N and if the base, scale, and mode are all absent, then FIXED and BINARY are added.

  If a precision of the form (p,q) is present, FIXED is added.

  If no base is present, DECIMAL is added.

  If no scale is present, FLOAT is added.

  If no mode is present, REAL is added.

- LINE option in PUT statement (S2.4.12.3-CM117) and COLUMN, LINE, and X format items (S2.4.12.3.3-CM151, CM148, CM149) – The syntax has been extended to allow omission of the parenthesized expression following the keyword. In each case, the default value of the expression is 1.

- Dependent declarations – The declaration of an AUTOMATIC or DEFINED variable can reference another AUTOMATIC or DEFINED variable declared in the same block.

- Precision greater than maximum – If a precision greater than the implementation-defined maximum, N, is specified, one of the following actions is taken:

  FIXED variable with p>N, q>=0, and q<=N: p is reduced to N, q is unchanged, and a warning diagnostic is produced.

  FIXED variable with p>N and q<0 or q>N: a fatal diagnostic is produced.

  FIXED constant with p>N and p-q<=N: the constant is truncated on the right, p is reduced to N, q is reduced so that p-q is unchanged, and a warning diagnostic is produced.

  FIXED constant with p>N and p-q>N: a fatal diagnostic is produced.

  FLOAT variable with p>N: p is reduced to N and a trivial diagnostic is produced.

  FLOAT constant with p>N: the mantissa is truncated on the right, p is reduced to N, and a trivial diagnostic is produced.

- LINESIZE for input files – LINESIZE is permitted for input files, and the default value is LINESIZE(80).

DIFFERENCES

A difference exists if the feature is supported but program behavior or syntax is different. The following are differences from the ANSI standard:

- Label constant arrays – A label constant array must be declared in a DECLARE statement. The standard prohibits such a declaration (S4.3.2.3).

- External names – External names and file titles that are longer than 7 characters are truncated to 7 characters. The first 4 and last 3 characters are used. A warning error message will be issued for external names that contain a special character in order to minimize the chance of conflict with system external names.

- ONCHAR and ONSOURCE pseudovariables – These pseudovariables modify the original string being converted rather than a copy of the string.

- Restrictions on CONVERSION on-unit – Repeated raising of CONVERSION without modifying the string is not allowed. Standard behavior would permit programs to loop on this condition.

- SYSPRINT – SYSPINT is an external name with the attributes EXTERNAL FILE STREAM PRINT OUTPUT. SYSPRINT and any other external name that maps to SYSPINT must have the same attributes.

DEVIATIONS

A deviation exists if the feature is not supported. The following are deviations from the ANSI standard:

- Context-only attributes – Certain attributes are supported by context, but the keywords are not supported (S2.4.4.2) (S2.4.4.3). The CONDITION keyword is supported in the ON, REVERT, and SIGNAL statements but not the DECLARE statement (S2.4.10.4-CM80). The FORMAT keyword is supported in the FORMAT statement only (S2.4.12.3.3-CM153). The following attribute keywords are not supported:

  CONSTANT
  DIMENSION
  MEMBER
  NONVARYING
  PARAMETER
  PRECISION
  STRUCTURE
  VARIABLE

- Unsupported attributes – The following attributes are not supported:

  GENERIC (S2.4.4.5)
  COMPLEX (S2.4.4.3-CM24)
  LOCAL (S2.4.4.2)

- Data types (S2.4.4.3-CM24) – The following data types are supported only for named constants that are scalars:

  ENTRY
  FILE
  FORMAT

  The following data types are supported only for variables that are parameters:

  ENTRY
  FILE

  The data type FORMAT is not supported for variables.

- INITIAL attribute – STATIC EXTERNAL variables can be initialized only in the outermost block of the main external procedure.

- Conditions and condition prefixes – The following conditions and prefixes are not supported:

  NAME condition (S2.4.10.4-CM79)
  STRINGSIZE prefix (S2.4.3.1-CM10)
  NOSTRINGSIZE prefix (S2.4.3.1-CM10)
  STRINGSIZE condition (S2.4.3.1-CM9)

- Builtin functions – The following builtin functions are not supported:

  COMPLEX (S9.4.4.18)
  CONJG (S9.4.4.19)
  DOT (S9.4.4.29)
  EVERY (S9.4.4.33)
  IMAG (S9.4.4.40)
  ONFIELD (S9.4.4.57)
  PROD (S9.4.4.65)
  REAL (S9.4.4.66)
  SOME (S9.4.4.73)
  STRING (S9.4.4.75)
  SUM (S9.4.4.78)

- Pseudovariables – The following pseudovariables are not supported:

  IMAG (S7.5.4.1)
  REAL (S7.5.4.5)
  STRING (S7.5.4.6)

- Constant specification – The following constant specifications are not supported:

  B1, B2, B3, and B4 radix factor (S2.5.5-CL23)

  F scale factor (S2.5.4-CL16)

  I imaginary constant indication (S2.5.4-CL21)

  P explicit precision indication (S2.5.4-CL13, CL18)

- Extents – Extents for STATIC and parameter variables, and in parameter and returns descriptors, are supported only as optionally signed decimal integers (S4.5.9).

- Aggregate operations (S9.1.1.6) – Aggregate operations are not supported. The following ramifications exist:

  Expressions that produce aggregate results are not supported.

  Promotion of operands is not supported.

  Aggregate assignment is not supported.

  Aggregate arguments which require the creation of aggregate dummy arguments are not supported.

  BY NAME assignment is not supported (S2.4.11-CM82).

  Functions with aggregate return values are not supported.

  Initialization of a member of an array of structures (unconnected storage) is not supported.

  Stream I/O of aggregates is not supported.

  AREA(*), BIT(*), and CHARACTER(*) are not permitted as return values.

- Data-directed I/O (S8.7.1.5) (S8.7.2.5) – Data-directed I/O operations are not supported, and the following features are not supported:

  DATA option on GET and PUT statements (S2.4.12.3.1-CM120)

  NAME condition (S2.4.10.4-CM79)

  ONFIELD builtin function (S9.4.4.57)

- DEFAULT statement – The DEFAULT statement is not supported (S2.4.4.7).

- DO statement and embedded-do in GET and PUT statements – The REPEAT option is not supported (S2.4.8-CM69).

- Formats – The following restrictions apply to formats:

  Nonlocal references to FORMAT statements are not supported.

Recursive remote references to a FORMAT statement are not supported.

Format variables are not supported.

C format item is not supported (S2.4.12.3.3-CM141).

E format item requires the d specification (S2.4.12.3.3-CM140).

TAB format item is not supported (S2.4.12.3.3-CM147).

- ON and REVERT statements – The condition-commalist is limited to a single condition (S2.3.3-CH8) (S2.4.10.4-CM75).
- OPEN statement – The TAB option is not supported (S2.4.12.1-CM92).
- LABEL values – Label comparison is not supported (S9.3.2.5.1). Functions that return a label value are not supported. (S2.4.4.7-CM59).
- %INCLUDE statement – The %INCLUDE statement is not supported (S2.5.7).
- Record I/O – The FROM and INTO options cannot contain subscripted references and cannot contain references to variables with the member, parameter, or DEFINED attributes.
- KEY option – The KEY option must be supplied for REWRITE and DELETE statements using KEYED SEQUENTIAL or KEYED DIRECT files.
- Duplicate keywords – Duplication of attribute keywords in a DECLARE statement is not supported.

IMPLEMENTATION-DEFINED VALUES

The ANSI standard describes a number of values that are to be defined by the implementation (S1.2). The implementation-defined values include:

- The ENVIRONMENT specification syntax (S2.4.4.4) corresponds in general to the parameter syntax on the FILE control statement.
- MAIN is the only supported option (S2.4.4.4) in the OPTIONS option of the PROCEDURE statement.
- Extralingual characters are ?, [,], ", \, and % (S2.5.5).
- The character sets supported (S2.6) are the CDC 63 and 64 character set and the ASCII 63 and 64 character set. A collating sequence (S9.4.4.17) is supported for each character set. The character set in use is selected by the installation.
- The default precisions for arithmetic data (S4.3.6.3) are the following:

  FIXED DECIMAL: (5,0)
  FIXED BINARY: (15,0)
  FLOAT DECIMAL: (14)
  FLOAT BINARY: (48)

- Expressions that are evaluated to integer are evaluated to FIXED BINARY (17,0) (S9.1.2).
- The maximum possible precision, N, for arithmetic data is 48 for BINARY and 14 for DECIMAL.
- The default AREA size is 150 words (S4.3.6.3). AREA size is expressed in 60-bit words.
- The SNAP option in the ON statement prints the appropriate values of ONCODE, ONCHAR, ONSOURCE, ONFILE, and ONLOC, as well as the statement number where the interrupt occurred, the names of the active procedures, and an informative message (S6.4.3).
- The standard system action for STORAGE condition is to raise the ERROR condition (S6.4.4).
- The standard system action for the ERROR condition is to produce a SNAP and to abort program execution (S6.4.4).
- The STORAGE condition is raised when the program attempts to exceed the maximum specified field length (S7.2.5).
- Concrete representation of a dataset is a CRM local file (S8.1).
- The default LINESIZE for a STREAM OUTPUT file is 80 characters and for a STREAM OUTPUT PRINT file is 136 characters (S8.5.1.3).
- The default PAGESIZE for a PRINT file is 60 lines (S8.5.1.3).
- The TIME builtin function returns six characters indicating hhmmss (S9.4.4.82).
- The results of floating point operations and numeric conversions are the hardware and software approximations to the actual values (S9.1.4) (S9.5.1.2).
- The currency symbol is $ (S9.5.2.2).
- The maximum length of an identifier is 40 characters.
- The result has a 3-digit exponent when a floating point value is converted to a character string (S9.5.1.5).
- The result of the ROUND builtin function with a floating point argument is not rounded (S9.4.4.68).
- Static nesting level for blocks cannot exceed 50.
- Evaluated extents have the following possible values:

  Area variable: 0<=size and size<=131071, but hardware always imposes a smaller maximum

  Bit variable: 0<=length and length<=131071

  Character variable: 0<=length and length<=131071

  Array bounds for an array: -131071<=(lower-bound) and (lower-bound)<(upper-bound) and (upper-bound)<=131071

- Character constant length is 0<=length and length<=16383.
- Bit constant length is 0<=length and length<=16383.
- Number of dimensions for an array cannot exceed 32.

- Length of the character value of a pictured item is 0<=length and length<=1000.
- Number of characters in the exponent of a pictured numeric floating point item cannot exceed 510.
- Number of blocks in an external procedure cannot exceed 255.
- Number of levels in a structure cannot exceed 16383, and no specified level number can exceed 16383.
- Number of arguments in an argument list and number of parameters in a parameter list cannot exceed 64.
- Number of unique external names in a program cannot exceed approximately 500, because of restrictions imposed by the loader.
- Precision q for a FIXED arithmetic variable is -255<=q and q<=255.
- Precision q for a FIXED BINARY arithmetic constant is 0<=q and q<=48. Precision q for a FIXED DECIMAL arithmetic constant is 0<=q and q<=14.

INDEX

COMMENT SHEET

CONTROL DATA CORPORATION

TITLE: PL/I Version 1 Reference Manual

PUBLICATION NO. 60388100 REVISION B

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

CUT ON THIS LINE

General comments:

FROM NAME: _____ POSITION: _____

COMPANY NAME: _____

ADDRESS: _____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Publications and Graphics Division
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ON THIS LINE

FOLD

FOLD