digital

VAX–11 PL/I

Encyclopedic Reference

Order No. AA-H952A-TE

VAX11

August 1980

Contains a definition of the VAX-11 PL/I programming language, including the keywords and the semantic and syntax rules of PL/I statements, attributes, and built-in functions.

VAX-11 PL/I
Encyclopedic Reference

Order No. AA-H952A-TE

SUPERSESSION/UPDATE INFORMATION:	This is a new document for this release.
OPERATING SYSTEM AND VERSION:	VAX/VMS V2.0
SOFTWARE VERSION:	VAX-11 PL/I V1.0

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, August 1980

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

ZKA27-80

Summary of Contents

Keywords in this manual are arranged in alphabetical order, as are numerous other topics of interest. For your convenience, the following summary includes page numbers for the general topics that are not associated with PL/I keywords.

Figures

Tables

Preface

■ Acknowledgment

The VAX-11 PL/I programming language is an implementation of the proposed PL/I G (General-Purpose) Subset, ANSI BSR X3.74.

■ How to Use This Manual

This manual provides language reference information for VAX-11 PL/I. All information in this manual is arranged in alphabetical order. For descriptions of individual attributes, built-in functions, or PL/I statements, look up the topic by its keyword.

■ Who Can Use This Manual

The manual is intended for use by all programmers who are designing or implementing applications using PL/I. Its readers should already understand the concepts of programming in PL/I and be familiar with the keywords and topics that will be searched for information. It is not, therefore, suitable for use as a strictly tutorial document.

■ Where to Find More Information

Introduction to VAX-11 PL/I contains an overview of the PL/I language and its implementation for the VAX-11 computer. The *Introduction* is recommended for all programmers who are not familiar with PL/I or who need information on the extensions made to PL/I for the VAX-11.

The companion document to this manual is the *VAX-11 PL/I User's Guide*. It contains information on program development with the VAX/VMS command language, on using the extensive I/O capabilities provided in VAX-11 PL/I, and on programming techniques available to PL/I programs executing under the exclusive control of the VAX/VMS operating system.

Figure 1 illustrates the relationship of documents available for VAX-11 PL/I, and lists the VAX/VMS operating system manuals that contain information of interest to PL/I programmers. For a complete list of all VAX/VMS documents and the order numbers of documents, see the *VAX-11 Information Directory and Index*.

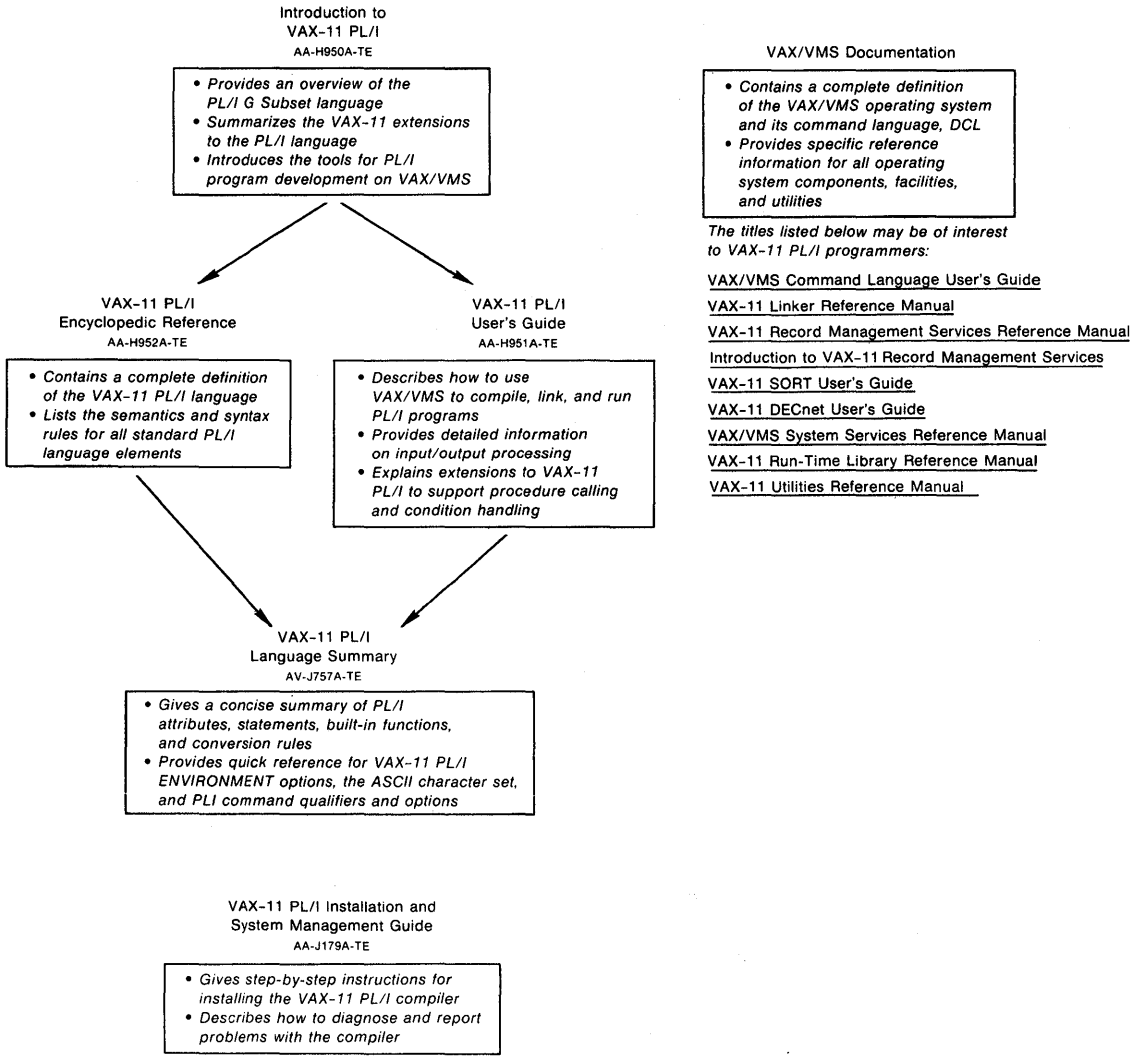


Figure 1: Documentation for VAX-11 PL/I Programmers

■ Conventions Used in This Document

ANY Attribute	All language items that are not included in the G Subset are printed in green ink.
Enter string>ABCD RET	In computer dialogs, the user's response to a prompt is printed in red ink.
DECLARE X FIXED; X = 5 ;	Vertical ellipses indicate that not all of the text of a program or program output is illustrated. Only relevant material is shown in the example.
option,...	Horizontal ellipses indicate that additional parameters, options, or values can be entered. When a comma precedes the ellipsis, it indicates that successive items must be separated by commas.
quotation mark apostrophes	The term quotation mark is used only to refer to the quotation mark symbol ("). The term apostrophe is used to refer to the single quotation mark symbol (′).
[OPTIONS (option,...)]	Except in VMS file specifications, square brackets indicate that a syntactic element is optional and you need not specify it.
[LIST EDIT]	Brackets surrounding two or more stacked items indicate conflicting options, one of which *may* be chosen.
{EXTERNAL INTERNAL}	Braces surrounding two or more stacked items indicate conflicting options, one of which *must* be chosen.
FILE (file-reference)	An uppercase word or phrase indicates a keyword that must be entered as shown; a lowercase word or phrase indicates an item for which a variable value must be supplied.
△	A delta symbol is used in some contexts to indicate a single ASCII space character.

■ Technical Assumptions

All descriptions of the effects of executing statements and evaluating expressions assume that the initial procedure activation of the program is through an entry point with OPTIONS(MAIN).

It is further assumed that any non-PL/I procedures called by the program follow all conventions of the PL/I run-time environment. Except as explicitly noted, descriptions of input/output statements do not cover the effects of VAX-specific options. For details on mixed-language programming and VAX-specific options, **see** the *VAX-11 PL/I User's Guide*.

A

A Format Item

The A format item describes the representation of a character string in the stream. The form of the A format item is:

A [(w)]

w

A nonnegative integer that specifies the width in characters of the field in the stream. If it is not included (PUT EDIT only), the field width equals the length of the converted output source.

The interpretation of the A format item on input and output is given below. For a general discussion of format items, **see** "Format Items and Their Uses."

■ Input with GET EDIT

The integer w must be included when the A format item is used with GET EDIT. If w is positive, a character-string value is acquired, comprising the next w characters in the input stream, and is assigned to the input variable. If w is zero, no operation is performed on the input stream, and a null character string is assigned to the input variable.

The acquired character string is converted, if necessary, to the data type of the input target, following the usual rules (for details, **see** "Conversion of Data"). Apostrophes should not enclose the stream data unless the apostrophes are intended to be acquired as part of the data.

■ Output with PUT EDIT

The output source associated with an A format item is converted, if necessary, to a string of characters. The result is assigned to a string of w characters, which are placed in the output stream. If w is omitted, the length of the output string is equal to the length of the converted output source. If w is zero, the A format item and the associated output source are skipped.

The output strings are not surrounded automatically by apostrophes. The converted output source is truncated or appended with trailing spaces, as required by the specification of w. The conversion is performed by the usual procedure for conversion of a computational data item to a character string (for details, **see** "Conversion of Data").

■ Examples

The tables below show the relationship between the internal and external representations of characters that are read or written with the A format item.

Input Examples

The "input stream" shown in the table is a field of characters beginning at the current position in the stream and continuing to the right. The target type is the type of the variable to which the input value is assigned.

Format Item	Input Stream	Target Type	Target Value
A(10)	ΔΔSHRUBBERYΔ...	CHAR(10)	ΔΔSHRUBBER
A(6)	ΔΔSHRUBBERYΔ...	CHAR(10)	ΔΔSHRUΔΔΔΔ
A(6)	ΔΔSHRUBBERYΔ...	CHAR(10)VAR	ΔΔSHRU
A(10)	ΔΔ1.2345ΔΔΔΔ...	DECIMAL(4,1)	001.2
A(5)	ΔΔ1.2345ΔΔΔΔ...	DECIMAL(4,2)	01.20
A(6)	ΔΔ1.2345ΔΔΔΔ...	DECIMAL(4,2)	01.23

Output Examples

The output source value shown in the table is either a constant or the value of a variable that is written out with the associated format item.

Output Source Value	Format Item	Output Value
'STRING'	A(10)	STRINGΔΔΔΔ
'STRING'	A	STRING
1.2345	A(2)	ΔΔ
1.2345	A	ΔΔ1.2345
-1.2345	A(4)	Δ-1.
-1.2345	A	Δ-1.2345
''	A(10)	ΔΔΔΔΔΔΔΔΔΔ
''	A	[no output]
0	A(3)	ΔΔΔ
0	A	ΔΔΔ0
-12345	A(6)	ΔΔ-123
-12345	A	ΔΔ-12345

ABS Built-In Function

The ABS built-in function returns the absolute value of an arithmetic expression x. Its format is:

ABS(x)

■ Examples

```
A = 3.567;
Y = ABS(A);    /*    Y = +3.567    */

A = -3.567;
Y = ABS(A);    /*    Y = +3.567    */

ROOT = SQRT (ABS(TEMP));
```

The last example shows a common use for the ABS built-in function, that is, to ensure that an expression has a positive value before requesting the square root (SQRT) built-in function.

ACOS Built-In Function

The ACOS built-in function returns a floating-point value that is the arc (inverse) cosine of an arithmetic expression x. The arc cosine is computed in floating point. The returned value is an angle w such that

$0 \leq w \leq \pi$

The absolute value of x, after its conversion to floating point, must be less than or equal to one. The format of the function is:

ACOS(x)

Addition

The plus sign character (+), when used as an infix operator, indicates an addition operation between two operands in an expression; the result is the sum of the operands. Both operands must be arithmetic or picture data.

The plus sign can also be used as a prefix operator. **See** "Operator."

■ Conversion of Operands

If both operands have the same base, precision, and scale, so has the result. The PL/I compiler converts operands of different data types as follows:

- If one operand has the FLOAT attribute and the other has the FIXED attribute, the fixed-point operand is converted to floating point before the operation.
- If one operand has the DECIMAL attribute and the other has BINARY, the decimal operand is converted to binary before the operation is performed. However, if a fixed-point decimal operand has fractional digits and the other operand is fixed-point binary, the binary operand is converted to fixed-point decimal, and a warning message is issued.

For an explanation of the precision of the value resulting from the conversion of an operand, **see** "Expression."

■ Precision of the Result

The precision of the resulting sum is based on the precision (or converted precision) of the two operands. For example, the title "Floating-Point Operands" below means that the operands were of floating-point types originally or that one was converted to floating point.

Floating-Point Operands

The result has the maximum of the precisions of the operands.

Fixed-Point Decimal Operands

If (p,q) and (r,s) represent the precisions and scale factors of the two operands, the resulting precision and scale factor are:

precision: min(31,max(p–q,r–s)+max(q,s)+1)

scale factor: max(q,s)

Fixed-Point Binary Operands

If (p) and (r) represent the precisions of the two operands, the resulting precision is:

MIN(31,MAX(p,r)+1)

ADDR Built-In Function

The ADDR built-in function returns a pointer to storage denoted by a specified variable. The only restriction on the variable reference is that it be addressable. The format of the function is:

ADDR(reference)

If the reference is to a parameter (or any element or member of a parameter), the pointer value obtained must not be used after return from the parameter's procedure invocation. (This could occur, for example, if the pointer were saved in a static variable or returned as a function value.)

See "Based Variable" for a general discussion of pointer values.

ALIGNED Attribute

The ALIGNED attribute controls the storage boundary of bit-string data in storage.

Specify the ALIGNED attribute in conjunction with the BIT attribute in a DECLARE statement to request alignment of a bit-string variable on a byte boundary. (**See** "Bit-String Data.") If you specify ALIGNED for an array of bit-string variables, each element of the array is aligned.

You can specify ALIGNED in the declaration of a nonvarying character-string variable. However, all character strings are byte-aligned on the VAX-11 machine; thus the specification of ALIGNED is superfluous and is not recommended. (**See** "Character-String Data.")

■ Restrictions

The ALIGNED attribute conflicts with the VARYING attribute and is invalid with all data type attributes other than BIT and CHARACTER. If you specify ALIGNED, you must specify either BIT or CHARACTER.

ALLOCATE Statement

The ALLOCATE statement obtains storage for a based variable and sets a pointer variable equal to the address of the allocated storage. The format of the ALLOCATE statement is:

{ALLOCATE / ALLOC} variable-reference [SET(pointer-reference)];

variable-reference

A based variable for which storage is to be allocated. The variable can be any scalar value, array, area, or major structure variable; it must be declared with the BASED attribute.

SET(pointer-reference)

The specification of the pointer variable that is assigned the value of the location of the allocated storage. If the SET option is omitted, the based variable must have been declared with BASED(pointer-reference), and the variable designated by that pointer reference is assigned the location of the allocated storage.

■ Examples

```
DECLARE STATE CHARACTER(100) BASED (STATE_POINTER),
        STATE_POINTER POINTER;

ALLOCATE STATE SET (STATE_POINTER);
```

This ALLOCATE statement allocates storage for the variable STATE and sets the pointer STATE_POINTER to the location of the allocated storage.

The ALLOCATE statement obtains as much storage as is necessary to accommodate the current extent of the specified variable. If, for example, a character-string variable is declared with an expression for its length, the ALLOCATE statement evalutes the current value of the expression to determine the amount of storage to allocate. For example:

```
DECLARE BUFFER CHARACTER (BUFLEN) BASED,
          BUF_PTR POINTER;
    .
    .
BUFLEN = 80;
ALLOCATE BUFFER SET (BUF_PTR);
```

Here, the value of BUFLEN is evaluted when the ALLOCATE statement is executed. The ALLOCATE statement allocates 80 bytes of storage for the variable BUFFER and sets the pointer variable BUF_PTR to its location.

For an additional example of the ALLOCATE statement and a description of based variables, **see** "Based Variable."

Storage within an area must be allocated by a user-written allocation procedure. For an example, **see** the *VAX-11 PL/I User's Guide*.

AND Operator

The & (ampersand) character is the logical AND operator in PL/I. In a logical AND operation, two bit-string operands are compared bit by bit. If two corresponding bits are 1, the corresponding bit in the result is 1; otherwise, the result is 0.

The result of a logical AND operation is a bit-string value. All relational expressions result in bit strings of length one, and they may therefore by used as operands in an AND operation. If the two operands have different lengths, the shorter operand is converted to the length of the longer operand, and that is the length of the result.

■ Examples

```
DECLARE (BITA, BITB, BITC) BIT (4);
BITA = '0011'B;
BITB = '1111'B;
BITC = BITA & BITB;
```

The resulting value of BITC is '0011'B.

The AND operator can test whether two or more expressions are both true in an IF statement. For example:

```
IF (LINENO(PRINT_FILE) < 60) &
   (MORE_DATA = YES) THEN ...
```

See also "Bit-String Data," "Logical Operator," and "Operator."

ANY Attribute

The ANY attribute specifies, for a parameter, that the corresponding argument can be of any data type. This attribute is applicable only to the declaration of entry names denoting non-PL/I procedures.

For complete details on using the ANY attribute, **see** the *VAX-11 PL/I User's Guide*.

■ Restrictions

- If you specify ANY for a parameter, you cannot specify any data type attributes for that parameter.
- The ANY attribute is valid only in a parameter descriptor.

■ Example

```
DECLARE SYS$SETEF ENTRY (ANY VALUE);
```

This statement identifies a system service procedure SYS$SETEF and indicates that the procedure accepts a single argument, which can be of any data type, to be passed by value.

ANYCONDITION Condition Name

The ANYCONDITION keyword can be specified in an ON or REVERT statement. It designates an ON-unit established for all signaled conditions that are not handled by specific ON-units.

The ANYCONDITION keyword is not defined in the PL/I language. It is provided specifically for use in the VAX-11/VMS operating system environment. For complete details on condition handling in VAX/VMS, **see** the *VAX-11 PL/I User's Guide*.

For information on defining ON-units for PL/I-specific conditions and PL/I default condition handling, **see** "ON Conditions and ON-Units" and "ON Statement."

Area

An area is a region of storage in which based variables may be allocated and freed. An area is defined by the declaration of a variable with the AREA attribute. An area variable can belong to any storage class. Areas provide the following programming capabilities:

- Based variables can be allocated within a specific area, and the entire area can be assigned or transmitted in a single operation. The variables can be referred to by offset values within the area; the offset values remain valid through assignment or transmission.
- The program can control the allocation of storage for related variables by placing them in the same area, thus improving the locality of reference. Also, the storage for all allocations within an area may be recovered in one operation by freeing the area itself.
- A structure containing an area can be used to represent a disk file that is mapped into a process's virtual memory space.

It is the responsibility of the user, in the program that declares and allocates an area, to control the allocation of variables within the area. Considerations for writing allocation procedures, as well as for using areas in conjunction with VAX/VMS memory allocation procedures, are discussed in the *VAX-11 PL/I User's Guide*.

■ Restriction

Do not write data in the first longword (32 bits) of an area. The first longword is reserved for future use by DIGITAL.

■ Area Assignment

You can specify an area variable as the target of an assignment statement only in the following case:

```
area-variable-1 = area-variable-2 ;
```

where both areas have the same extent. The complete contents of the source are copied to the target.

All other specifications of an area variable as the target of an assignment statement are invalid. An area variable cannot be used in an expression containing operators.

■ Reading and Writing Areas

An area can be the source or target of data transmission in either of the record I/O statements READ or WRITE. The complete contents of the area are transmitted.

AREA Attribute

The AREA attribute defines an area variable (**see** "Area"). Its format is:

AREA (extent)

extent

The size of the area in bytes. The extent must be a nonnegative integer value. The maximum size is 500 million bytes. The rules for specifying the extent are as follows:

- If AREA is specified for a static variable declaration, extent must be a decimal integer constant.
- If AREA is specified in the declaration of a parameter or in a parameter descriptor, extent may be specified as an integer constant or as an asterisk (*).
- If AREA is specified for an automatic or based variable, extent may be specified as an integer constant or as an expression. In the case of automatic variables, the extent expression must not contain any variables or functions declared in the same block, except for parameters.

■ Restrictions

- The AREA attribute is not allowed in a returns descriptor.
- The AREA attribute conflicts with all other data type attributes.

Argument

An argument is an expression or variable reference denoting a value to be used by a built-in function or a user-defined procedure or function. The maximum number of arguments that can be passed to a procedure is 253.

For full details, **see** "Parameters and Arguments."

■ Argument List

An argument list consists of zero or more arguments specified in the invocation of a procedure, built-in function, or built-in subroutine.

In the case of built-in functions, arguments are expressions that supply values to the built-in function, and the argument types must be those required by the specific function. In general, built-in functions can be considered as operators and their arguments, operands. For example, if two arithmetic arguments for a built-in function are of different arithmetic types, they are evaluated and converted to a common type as are the operands of an arithmetic expression. For further details, **see** "Built-In Function" and "Expression."

In the case of user-defined procedures, arguments correspond to parameters defined on the PROCEDURE or ENTRY statement of the invoked procedure.

■ Argument Passing

In PL/I, a parameter of a procedure is always associated with a variable passed to it by the calling procedure. This variable may be the original argument corresponding to the parameter or a dummy argument created by the compiler and assigned the original argument's value.

■ Dummy Argument

A dummy argument is a variable that is allocated by the compiler to pass an argument to an invoked procedure. The compiler creates a dummy argument when an argument specified in a procedure reference is a constant or an expression, is a variable with a different data type than that required by the corresponding parameter, or is enclosed in parentheses.

Arithmetic Data Types

Arithmetic data types are used for variables on which arithmetic calculations are to be performed. The arithmetic data types supported by VAX-11 PL/I are:

- Fixed-point binary — for integers (**see** "Fixed-Point Binary Data")
- Fixed-point decimal — for decimal data with a fixed number of fractional digits (**see** "Fixed-Point Decimal Data")
- Floating-point binary or decimal — for calculations on very large or very small numbers, with the decimal point (number of fractional digits) allowed to "float" (**see** "Floating-Point Data")
- Picture — for fixed-point decimal data that is stored internally in character form, with special formatting characters (**see** "Picture")

Arithmetic Operators

The arithmetic operators perform calculations. Programs that accept numeric input and produce numeric output use arithmetic operators to construct expressions to perform the required calculations. The arithmetic operators are:

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Included among the arithmetic operators are the two prefix operators:

Operator	Meaning
+	Unary plus
-	Unary minus

The unary plus is valid on any arithmetic operand, but it performs no actual operation.

The unary minus reverses the sign of any arithmetic operand.

For detailed descriptions of the other operands, **see** "Addition," "Division," "Exponentiation," "Multiplication," and "Subtraction."

For any arithmetic operator, operands must be arithmetic; that is, they must be constants, variables, or other expressions with one of the data type attributes BINARY, DECIMAL, or PICTURE. Operands of different arithmetic types are converted to a common type before the operation is performed (**see** "Expression").

Arithmetic operators have a predefined precedence that governs the order in which operations are performed. For further information, **see** "Operator." All expressions can also be enclosed in parentheses to override the rules of precedence.

Array

Arrays provide an orderly way to manipulate related variables of the same data type. An array variable is defined in terms of the number of variables, or elements, that it contains and the organization of those elements. These attributes of an array are called its dimensions.

The following subsections describe arrays in terms of scalar elements. For information on arrays whose elements are structures, **see** "Arrays of Structures."

■ Format of an Array Declaration

You specify the dimensions of an array in a DECLARE statement, as shown below:

DECLARE identifier (bound-pair,...) [attribute ...];

for declaring a single array; or

DECLARE (declaration,...) (bound-pair,...) [attribute ...];

for declaring two or more array variables with the same dimensions and bounds. Each declaration in this form can consist of a simple identifier, the declaration of another array, or the declaration of a structure. For further details on the syntax of declarations, **see** "DECLARE Statement." **See also** "Arrays of Structures."

identifier

A valid PL/I identifier to be used as the name of the array.

bound-pair

A specification of the number of elements in each dimension of the array. A bound pair can consist of:

- Two expressions separated by a colon, giving the lower and upper bounds for that dimension; or
- A single expression giving the upper bound only (the lower bound is then one by default); or
- An asterisk (*), used in the declaration of array parameters, and indicating that the parameter can be matched to array arguments with varying numbers of elements in that dimension.

The bound pairs must be separated by commas, and the list of bound pairs must be enclosed in parentheses. The list of bound pairs must immediately follow the identifier or the list of declarations.

Figure A–1 shows several forms of bound pairs as used in declarations.

attribute ...

One or more data type attributes of the elements of the array. All attributes you specify apply to each of the elements in the array.

Elements of an array can have any data type. If the array has the FILE or ENTRY attribute, it must also have the VARIABLE attribute.

ARRAY-NAME (Bound) A single value specifies: • That the array has a single dimension. • That the dimension has 'bound' number of elements; this is the extent of the dimension. • That the value specified is the high bound, that is, the largest numbered element. By default, the low bound is 1.	EXAMPLES `DECLARE VERBS (6) CHARACTER (12) ;`
ARRAY-NAME (Low-Bound:High-Bound) A single range of values specifies: • That the array has a single dimension. • That the number of elements in the dimension is (high-bound)-(low-bound)+1. • The index value assigned to the lowest-numbered element and the index value assigned to the highest-numbered element.	`DECLARE TEMPERATURS (-60:120) ;`
ARRAY-NAME (Bound1,Bound2,...) A list of values specifies: • That the array is multidimensional. Each bound value represents a dimension in the array. • The extent of each dimension. Each bound defines the number of elements in a dimension. • The high-bound value of each dimension. The low-bound value of each dimension defaults to 1.	`DECLARE TABLE (10,10) FIXED BINARY ;` `DECLARE SETS (5,5,5,5) CHARACTER (80) ;`
ARRAY-NAME (Low-Bound1:High-Bound1,Low-Bound2:High-Bound2,...) A set of ranges specifies: • That the array is multidimensional. Each range of values represents a dimension in the array (ranges can be intermixed with single-bound specifications). • The extent of each dimension. • The low-bound and high-bound values of each dimension.	`DECLARE WINDOWS (1:10,-2:32) FIXED ;` `DECLARE HISTORIES (10,30:102,50) ...`
ARRAY-NAME (*,...) Asterisk extents specify: • The number of dimensions in the array. Each asterisk indicates a dimension. • That the extent of each dimension will be defined by the actual argument passed to the procedure when it is invoked.	`ADDIT: PROCEDURE (ARR);` `  DECLARE ARR(*,*) FIXED ;`

Figure A–1: Specifying Array Dimensions

■ Rules for Specifying Dimensions

The following rules apply to specifying the dimensions of an array and the bounds of a dimension:

- The maximum number of dimensions that an array can have is eight.
- The values you can specify for bounds are restricted as follows:
  - If the array has the STATIC attribute, you must specify all bounds as restricted integer expressions (**see** "Integer Data").
  - If the array has the AUTOMATIC, BASED, or DEFINED attribute, you can specify the bounds as optionally signed integer constants or as expressions that yield integer values at run time. If the array has AUTOMATIC or DEFINED, the expressions must not contain any variables or functions that are declared in the same block, except for parameters.
  - If an array is a parameter, you can specify the bounds using optionally signed integer constants or asterisks (*). If you specify any bound as an asterisk, you must specify all bounds with asterisks. An array parameter declared this way inherits the dimensions of the corresponding argument. Passing array variables as arguments to a procedure is described below under "Passing Arrays as Arguments."
- The value of the lower bound you specify must be less than the value of the upper bound.

■ References to Individual Elements

You refer to an individual element in the array by means of subscripts. Since an array's attributes are common to all of its elements, a subscripted reference has the same properties as a reference to a scalar variable with these attributes.

Subscripts must be enclosed in parentheses in a reference to an array element. For example, in a one-dimensional array named ARRAY declared with the bounds (1:10), the elements are numbered 1 through 10 and are referred to as ARRAY(1), ARRAY(2), ARRAY(3), and so on.

The lower and upper bounds that you declare for a dimension determine the range of subscripts that you can specify for that dimension. If only an upper bound was specified for a dimension, the lower bound (minimum subscript) for that dimension is 1. The number of elements in any dimension of any array is:

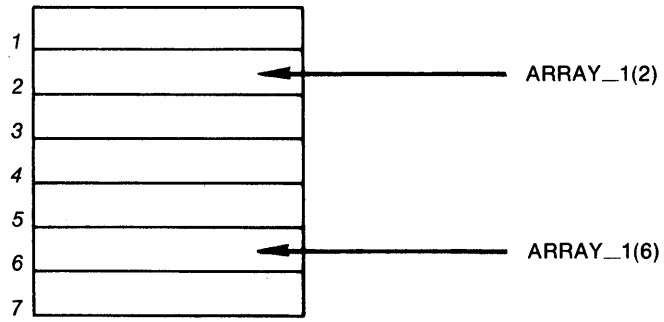
(upper-bound) - (lower-bound) + 1

The total number of elements in the array, called its "extent," is the product of the numbers of elements in all the dimensions of the array.

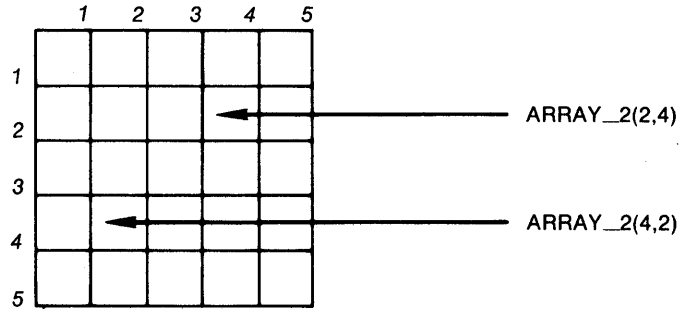
For multidimensional arrays, the subscript values represent an element's position with respect to each dimension in the array. Figure A-2 illustrates subscripts for elements of one-, two-, and three-dimensional arrays.

In subscripted references, the number of subscripts must match the number of dimensions of the array. This includes any dimensions that are inherited when an array results in the declaration of a dimensioned structure (**see** "Arrays of Structures").

DECLARE ARRAY_1 (7);



DECLARE ARRAY_2 (5,5);



DECLARE ARRAY_3 (3,4,4);

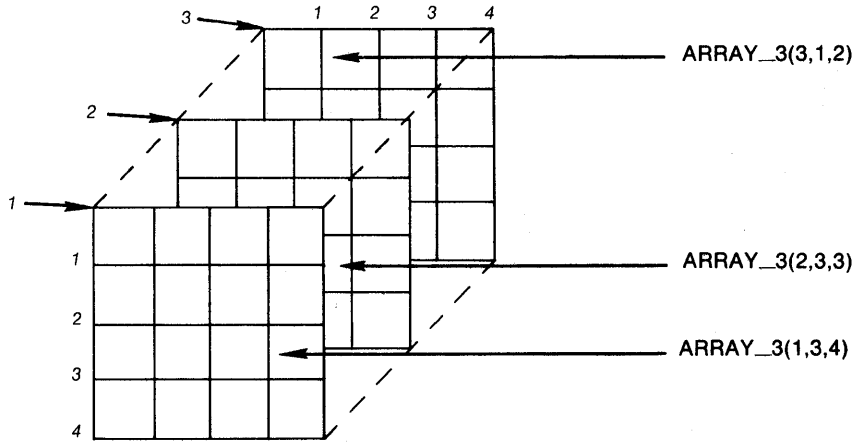


Figure A-2: Specifying Elements of an Array

■ Variable Subscripts

You can specify the subscript of an array element using any variables or expressions having integer values, that is, values that can be expressed as fixed binary or fixed decimal with a zero scale factor. For example:

```
DECLARE DAYS_IN_MONTH(12) FIXED BINARY;
DECLARE (COUNT, TOTAL) FIXED BINARY;
TOTAL = 0;
DO COUNT = 1 TO 12;
      TOTAL = TOTAL + DAYS_IN_MONTH(COUNT);
      END;
```

Here, the variable COUNT is used as a control variable in a DO loop. As the value of COUNT is incremented from 1 to 12, the value of the corresponding element of the array DAYS_IN_MONTH is added to the value of the variable TOTAL.

■ Initializing Arrays

The INITIAL attribute can be specified for arrays. For example:

```
DECLARE MONTHS (12) CHARACTER (9) VARYING
   INITIAL ('January', 'February', 'March', 'April',
      'May', 'June', 'July', 'August',
      'September', 'October', 'November', 'December');
```

In this example, each element of the array MONTHS is assigned a value according to the order of the character-string constants in the initial list: that is, MONTH(1) is assigned the value 'January'; MONTH(2) is assigned the value 'February'; and so on.

If the array being initialized is multidimensional, the initial values are assigned in row-major order.

For full details on the use of the INITIAL attribute, **see** "INITIAL Attribute."

■ Iteration Factors

In the INITIAL attribute, when more than one successive element of an array (or all elements of an array) are to be assigned the same value, you can specify an iteration factor. An iteration factor indicates the number of times that a specified value is to be used in the assignment of values to elements of an array. You can specify an iteration factor in one of the following formats:

(iteration-factor) arithmetic-constant

(iteration-factor) scalar-reference

(iteration-factor) (scalar-expression)

(iteration-factor) *

iteration-factor

An unsigned decimal constant indicating the number of times to use the specified constant in the assignment of an array element. The iteration factor can be zero.

arithmetic-constant

Any arithmetic constant whose data type is valid for conversion to the data type of the array.

scalar-reference

A reference to any scalar variable or to the NULL built-in function.

scalar-expression

Any arithmetic or string expression or string constant. The expression or constant must be enclosed in parentheses.

*

Symbol used to indicate that the corresponding array elements are not to be assigned initial values.

Any of the above forms may be used for arrays that have the AUTOMATIC attribute. For arrays with the STATIC attribute, only constants and the NULL built-in function may be used.

For example, this declaration of the array SCORES initializes all elements of the array to one:

```
DECLARE SCORES (100) FIXED STATIC INITIAL ((100)1);
```

This declaration initializes the first 50 elements to 1 and the last 50 elements to -1:

```
DECLARE SCORES(100) FIXED STATIC INITIAL((50)1,(50)-1);
```

The next example initializes all 10 elements of an array of character strings to the 26-character value in apostrophes. Note that the string constant is enclosed in parentheses; this is required syntax.

```
DECLARE ALPHABETS (10) CHARACTER(26) STATIC
   INITIAL((10)('ABCDEFGHIJKLMNOPQRSTUVWXYZ'));
```

■ Array Variables in Assignment Statements

You can specify an array variable as the target of an assignment statement in the following cases:

- array-variable = expression ;

where the expression yields a scalar value. Every element of the array is assigned the resulting value. The array variable must be a connected array whose elements are scalar. (**See** the subsection "Connected Arrays" in "Arrays of Structures.")

Note that the arithmetic operators, such as + and -, cannot have arrays as operands. An assignment of the form:

```
ARRAYC = ARRAYA + ARRAYB;
```

is invalid.

- array-variable-1 = array-variable-2 ;

where the specified array variables have identical data type attributes and dimensions. Each element in array-variable-1 is assigned the value of the corresponding element in array-variable-2.

In this type of assignment, both arrays must be connected. The actual storage occupied by the arrays must not overlap, unless the arrays are identical.

All other specifications of an array variable as the target of an assignment statement are invalid.

■ Using GET and PUT Statements with Array Variables

When you specify an array variable name in the input-target list of a GET LIST or GET EDIT statement, elements of the array are assigned values from the data items in the input stream. For example:

```
DECLARE VERBS (6) CHARACTER (15) VARYING; GET LIST (VERBS);
```

When this GET LIST statement executes, it accepts data from the default input stream. Each blank-, tab-, or comma-delimited input field is considered a separate string. The values of these strings are assigned to elements of the array VERBS in the order VERBS(1), VERBS(2), . . . VERBS(6). If a multidimensional array appears in an input-target list, input data items are assigned to the array elements in row-major order.

An array can also appear, with similar effects, in the output-source list of a PUT statement. For information on using the GET and PUT statements with arrays, **see** "GET Statement" and "PUT Statement."

■ Order of Assignment and Output for Multidimensional Arrays

When a multidimensional array is initialized without references to specific elements, PL/I assigns the values in row-major order. In row-major order, the rightmost subscript varies the most rapidly. For example, an array can be declared as follows:

```
DECLARE TESTS (2,2,3);
```

If TESTS is specified in a GET statement or in a declaration with the INITIAL attribute, values are assigned to the elements in the following order:

TESTS (1,1,1)
TESTS (1,1,2)
TESTS (1,1,3)
TESTS (1,2,1)
TESTS (1,2,2)
TESTS (1,2,3)
TESTS (2,1,1)
TESTS (2,1,2)
TESTS (2,1,3)
TESTS (2,2,1)
TESTS (2,2,2)
TESTS (2,2,3)

When an array is output with a PUT statement, PL/I uses the same order to output the array elements. For example:

```
PUT LIST (TESTS);
```

This PUT statement outputs the contents of TESTS in the order shown above.

■ Passing Arrays as Arguments

An array variable can be passed as an argument to another procedure. Within the invoked procedure, the corresponding parameter must be declared with the same number of dimensions. The rules for specifying the bounds in a parameter descriptor for an array parameter are:

- If the bounds are specified using integer constants, they must match exactly the bounds of the corresponding argument.
- All bounds can be specified as asterisks (*). In this case, the bounds of the array are determined from the bounds of the corresponding argument when the procedure is actually invoked. If any bound is specified as an asterisk, all bounds must be specified as asterisks.

For example:

```
DECLARE SCAN ENTRY (
                    (5,5,5) FIXED,
                    (*) FIXED),
        MATRIX (5,5,5) FIXED,
        OUTPUT (20) FIXED;
        CALL SCAN (MATRIX,OUTPUT);
```

The procedure SCAN receives two arrays as arguments. The first is a three-dimensional array whose bounds are known. The second is a one-dimensional array whose bounds are not known. The procedure SCAN may declare these parameters as follows:

```
SCAN: PROCEDURE (IN,OUT);
DECLARE IN (*,*,*) FIXED,
        OUT (*) FIXED;
```

An array whose storage is unconnected cannot be passed as an argument, nor can an array whose elements are label constants. Arrays are always passed by reference and cannot be passed by a dummy argument.

For full information on arguments and argument passing, **see** "Parameters and Arguments."

■ Array-Handling Functions

PL/I provides the following built-in functions that return information about the dimensions of an array:

- DIMENSION — returns the number of elements in a given dimension.
- HBOUND — returns the value of the upper bound of the array in a given dimension.
- LBOUND — returns the value of the lower bound of the array in a given dimension.

For the first dimension of an array X, the relationship of these functions can be expressed as follows:

DIMENSION (X,1) = HBOUND (X,1) - LBOUND (X,1) + 1

The simple procedure shown below uses the HBOUND and LBOUND built-in functions:

```
ADDIT: PROCEDURE (X);
DECLARE X (*) FIXED BINARY,
        (COUNT,I) FIXED BINARY;
        COUNT = 0;
        DO I = LBOUND (X,1) TO HBOUND(X,1);
            COUNT = COUNT + 1;
            X(I) = COUNT;
            END;
        RETURN;
        END;
```

This procedure receives a single-dimensioned array as a parameter and initializes the elements of the array with integral values beginning with one.

For more information, **see** the entries for these built-in functions, "Function," and "Procedure."

Arrays of Structures

An array of structures is an array whose elements are structures. Each structure has identical logical levels, minor structure names, and member names and attributes.

For example, a structure STATE can be declared an array, as shown below:

```
DECLARE 1 STATE (50),
          2 NAME CHARACTER (20) VARYING,
          2 POPULATION FIXED (31),
          2 CAPITAL,
            3 NAME CHARACTER (30) VARYING,
            3 POPULATION FIXED (31)
          2 SYMBOLS,
            3 FLOWER CHARACTER (20),
            3 BIRD CHARACTER (20);
```

A member of a structure that is an array inherits the dimensions of the structure. For example, the member CAPITAL.NAME of the structure STATE inherits the dimension 50. You must use a subscript whenever you refer to the variable CAPITAL.NAME, as in the example below:

```
PUT LIST (CAPITAL.NAME(I)) ;
```

A subscript for a member of a structure that is an array element can appear following any name within a qualified reference. For example, all of the following references are equivalent:

```
STATE(10).CAPITAL.NAME
STATE.CAPITAL(10).NAME
STATE.CAPITAL.NAME(10)
```

■ Arrays of Structures that Contain Arrays

A structure that is defined with a dimension can have members that are arrays. For example:

```
DECLARE 1 STATE (50),
          2 AVERAGE_TEMPS(12) FIXED DECIMAL (5,2),
            ,
            ,
```

In this example, the elements of the array STATE are structures. At the second level of the hierarchy of each structure is an array of 12 elements. Because this member of the structure inherits the dimension of the major structure, any of these elements must be referred to by two subscripts:

1. The first subscript references an element in the array STATE.
2. The second subscript references an element in the array AVERAGE_TEMPS.

These subscripts can appear following any name in the qualified reference. For example:

```
STATE(3).AVERAGE_TEMPS(4)
STATE.AVERAGE_TEMPS(3,4)
```

These references are equivalent.

Note the following rules for specifying subscripts for members of structures containing arrays:

- The number of subscripts specified for any member must include any dimensions inherited from a major or minor structure declaration, as well as those specified for the member itself.
- The subscripts that refer to a member of a structure in an array do not have to follow immediately the name to which they apply. However, the order of subscripts must be preserved.
- The total number of dimensions, including the inherited dimensions, must not exceed eight.

For information on structure declarations, **see** "Structure."

■ Connected Arrays

A connected array is an array whose elements occupy consecutive locations in storage. For example:

```
DECLARE NEWSPAPERS (10) CHARACTER (30) ;
```

In storage, the 10 elements of the array NEWSPAPERS occupy 10 consecutive 30-byte units. Thus, the array NEWSPAPERS is a connected array.

A connected array is valid as the target of an assignment statement, as long as the source expression is a similarly dimensioned array or is a single scalar value.

An unconnected array is an array whose elements do not occupy consecutive storage locations. A structure with the dimension attribute always results in unconnected arrays. When a structure is dimensioned, each member of the

structure inherits the dimensions of the structure and becomes, in effect, an array. For example:

```
DECLARE 1 STATE (50),
          2 NAME CHARACTER (20) VARYING,
          2 POPULATION FIXED (31);
```

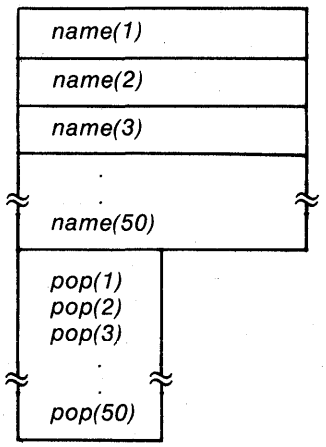
In the above example, the members NAME and POPULATION of the major structure STATE inherit the dimension 50 from the major structure. When PL/I allocates storage for a structure or a dimensioned structure, each member is allocated consecutive storage locations; thus the elements of the arrays NAME and POPULATION are not connected.

Figure A–3 illustrates the storage of connected and unconnected arrays.

CONNECTED:

```
DECLARE 1 STATE,
          2 NAME (50) CHAR(20),
          2  POP (50) FIXED(10);
```

The members NAME and POP of the structure STATE are dimensioned. The elements of each array occupy consecutive storage locations.



UNCONNECTED:

```
DECLARE 1 STATE (50),
          2 NAME CHAR(20),
          2 POP FIXED(10):
```

The array STATE is dimensioned. Its members NAME and POP inherit the dimension: each of these variables is an array of 50 elements, but the elements do not occupy consecutive storage locations.

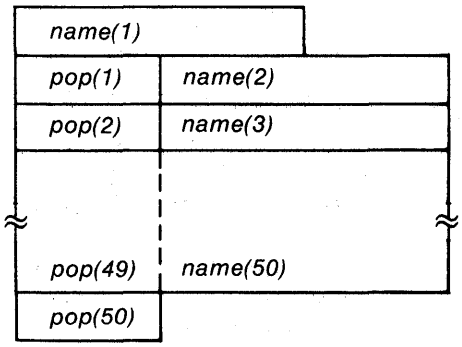


Figure A–3: Connected and Unconnected Arrays

ASCII Character Set

The American Standard Code for Information Interchange (ASCII) is a set of eight-bit numeric values that represent the alphabet, numerals, punctuation and symbols used in text and in communications protocol. This is the ASCII character set. Table A–1 lists the set and its numeric values.

Table A-1: ASCII Character Set

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
0	NUL	Null	64	@	At sign
1	SOH	Start of heading	65	A	Upper case A
2	STX	Start of text	66	B	Upper case B
3	ETX	End of text	67	C	Upper case C
4	EOT	End of transmission	68	D	Upper case D
5	ENQ	Enquiry	69	E	Upper case E
6	ACK	Acknowledgement	70	F	Upper case F
7	BEL	Bell	71	G	Upper case G
8	BS	Backspace	72	H	Upper case H
9	HT	Horizontal tab	73	I	Upper case I
10	LF	Line feed	74	J	Upper case J
11	VT	Vertical tab	75	K	Upper case K
12	FF	Form feed	76	L	Upper case L
13	CR	Carriage return	77	M	Upper case M
14	SO	Shift out	78	N	Upper case N
15	SI	Shift in	79	O	Upper case O
16	DLE	Data link escape	80	P	Upper case P
17	DC1	Device control 1	81	Q	Upper case Q
18	DC2	Device control 2	82	R	Upper case R
19	DC3	Device control 3	83	S	Upper case S
20	DC4	Device control 4	84	T	Upper case T
21	NAK	Negative acknowledgement	85	U	Upper case U
22	SYN	Synchronous idle	86	V	Upper case V
23	ETB	End of transmission block	87	W	Upper case W
24	CAN	Cancel	88	X	Upper case X
25	EM	End of medium	89	Y	Upper case Y
26	SUB	Substitute	90	Z	Upper case Z
27	ESC	Escape	91	[	Left square bracket
28	FS	File separator	92	\	Back slash
29	GS	Group separator	93	]	Right square bracket
30	RS	Record separator	94	^ or ↑	Circumflex or up arrow
31	US	Unit separator	95	← or _	Back arrow or underscore
32	SP	Space or blank	96	`	Grave accent
33	!	Exclamation mark	97	a	Lower case a
34	"	Quotation mark	98	b	Lower case b
35	#	Number sign	99	c	Lower case c
36	$	Dollar sign	100	d	Lower case d
37	%	Percent sign	101	e	Lower case e
38	&	Ampersand	102	f	Lower case f
39	'	Apostrophe	103	g	Lower case g
40	(	Left parenthesis	104	h	Lower case h
41	)	Right parenthesis	105	i	Lower case i
42	*	Asterisk	106	j	Lower case j
43	+	Plus sign	107	k	Lower case k
44	,	Comma	108	l	Lower case l
45	-	Minus sign or hyphen	109	m	Lower case m
46	.	Period or decimal point	110	n	Lower case n
47	/	Slash	111	o	Lower case o
48	0	Zero	112	p	Lower case p
49	1	One	113	q	Lower case q
50	2	Two	114	r	Lower case r
51	3	Three	115	s	Lower case s
52	4	Four	116	t	Lower case t
53	5	Five	117	u	Lower case u
54	6	Six	118	v	Lower case v
55	7	Seven	119	w	Lower case w
56	8	Eight	120	x	Lower case x
57	9	Nine	121	y	Lower case y
58	:	Colon	122	z	Lower case z
59	;	Semicolon	123	{	Left brace
60	<	Left angle bracket	124	|	Vertical line
61	=	Equal sign	125	}	Right brace
62	>	Right angle bracket	126	~	Tilde
63	?	Question mark	127	DEL	Delete

ASIN Built-In Function

The ASIN built-in function returns a floating-point value that is the arc (inverse) sine of an arithmetic expression x. The arc sine is computed in floating point. The returned value is an angle w such that

$-\pi/2 \leq w \leq \pi/2$

The absolute value of x, after its conversion to floating point, must be less than or equal to 1. The format of the function is:

ASIN(x)

Assignment Statement

The assignment statement gives a value to a specified variable. The format of the assignment statement is:

target = expression ;

target

The name of the variable to be assigned a value. It can be:

- Any reference to a scalar variable or scalar array element
- A pseudovariable (for example, SUBSTR)
- A reference to a major or minor structure name or any member of a structure
- A reference to an array variable

expression

Any valid expression.

PL/I evaluates an assignment statement and performs the assignment as follows:

1. The target is evaluated. If it contains a pseudovariable, any expressions in the argument list are evaluated.
2. The expression on the right-hand side of the assignment statement is evaluated, producing a result. An expression can consist of many subexpressions and operations, each of which must be evaluated. **See** "Expression" for a complete description.
3. If the data type of the result does not match the data type of the target variable, the resulting value is converted to the data type of the target.

Some general rules regarding the types of data you can specify in assignment statements are given below. For the complete rules for data conversion in assignments, **see** "Conversion of Data."

■ Arithmetic Data

PL/I converts an arithmetic expression to the type of the target, if their types are different. If the target is a character- or bit-string variable, PL/I converts the arithmetic expression to its character- or bit-string equivalent.

A character-string expression can be converted to the data type of an arithmetic target only if the string consists solely of characters that have numeric equivalents.

■ Arrays

You can specify an array variable as the target of an assignment statement in only the following ways:

- array-variable = expression ;

  where expression yields a scalar value. Every element of the array is assigned the resulting value.

- array-variable-1 = array-variable-2 ;

  where the specified array variables have identical data type attributes and dimensions. Each element in array-variable-1 is assigned the value of the corresponding element in array-variable-2.

  The storage occupied by the two arrays must not overlap.

Any array variable specified in an assignment statement must occupy connected storage.

All other specifications of an array variable as the target of an assignment statement are invalid.

■ Bit Data

When the target of an assignment is a bit-string variable, the resulting expression is truncated or padded with trailing zeros to match the length of the target.

■ Character Data

When the target of an assignment is a fixed-length character string, the resulting expression is truncated on the right or padded with trailing spaces to match the length of the target. If the target is a varying-length character string, the resulting expression is truncated on the right if it exceeds the maximum length of the target.

When one character-string variable is assigned to another, the storage occupied by the two variables cannot overlap.

■ Entry Data

If the specified expression is an entry constant, an entry variable, or a function reference that returns an entry value, the target variable must be an entry variable.

■ Label Data

If the specified expression is a label constant, a label variable, or a function reference that returns a label value, the target variable must be a label variable.

■ Pointer and Offset Data

If the specified expression is a pointer or offset, or a function reference that returns a pointer or offset, the target variable must be a pointer or offset variable.

■ Structures

You can specify the name of a major or minor structure as the target of an assignment statement only if the source expression is an identical structure with members in the same hierarchy and with identical sizes and data type attributes. The storage occupied by the two structures must not overlap.

Any structure variable specified in an assignment statement must occupy connected storage.

ATAN Built-In Function

The ATAN built-in function returns a floating-point value that is the arc tangent of an arithmetic expression y or an arc tangent computed from two arithmetic expressions y and x. The arc tangent is computed in floating point. If two arguments are supplied, they must not both be zero after their conversion to floating point.

The format of the function is:

ATAN(y[,x])

■ Returned Values

The returned value represents an angle in radians.

If x is omitted, the returned value v equals arctangent(s), such that

$-\pi/2 < v < \pi/2$

where s is the value of expression y after its conversion to floating point.

If x is present, the returned value v equals arctangent(s/r), such that

if $s \geq 0$ then $0 \leq v \leq \pi$, and
if $s < 0$ then $-\pi < v < 0$

where s and r are, respectively, the values of expressions y and x after their conversion to floating point.

ATAND Built-In Function

The ATAND built-in function returns a floating-point value that is the arc tangent of a single arithmetic expression y or an arc tangent computed from two arithmetic expressions y and x. The arc tangent is computed in floating point. If two arguments are supplied, they must not both be zero after their conversion to floating point.

The format of the function is:

ATAND(y[,x])

■ Returned Value

The floating-point value returned, representing an angle in degrees, equals ATAN(y,x)*180/π.

ATANH Built-In Function

The ATANH built-in function returns a floating-point value that is the inverse hyperbolic tangent of an arithmetic expression x. After its conversion to floating point, the absolute value of the argument x must be less than one.

The format of the function is:

```
ATANH(x)
```

Attribute

Attributes define and describe the characteristics of data used in a PL/I program. Each data item in a PL/I program has a set of attributes associated with it. Attributes can be specified in any of the following contexts:

- In a DECLARE statement for an identifier. These attributes are specified either by keyword or by syntax. In this text, keyword attributes are shown in uppercase letters. Attributes given by syntax are shown in lowercase letters. For example:

  ```
  DECLARE SIGNAL CHARACTER (20);
  ```

  In this declaration, the keyword attribute CHARACTER is associated with the identifier SIGNAL. The length attribute of the variable is specified in parentheses following the CHARACTER keyword.
- In an OPEN statement to describe a particular file. During the opening of a file, these attributes are merged with file description attributes specified in the declaration of the file.
- Within the ENTRY attribute to describe the parameters of an external procedure. These attributes must match the attributes given to corresponding parameters specified in the PROCEDURE or ENTRY statements of the invoked subroutine or function.
- Within the RETURNS attribute of a PROCEDURE or ENTRY statement to describe the value returned by a function.

Attributes can also be implied by the presence of other attributes. For example, if the RETURNS attribute is specified for an identifier, the compiler supplies the ENTRY attribute by default.

The entry for each attribute in this manual gives its syntax and abbreviation (if any) and describes related and conflicting attributes. See Table A–2 at the end of this entry for a concise alphabetical summary of PL/I attributes.

■ Computational Data Type Attributes

The attributes that define arithmetic and string data are:

```
CHARACTER   [ (length) ] [ VARYING ]

BIT   [ (length) ] [ ALIGNED ]

{ BINARY  } { FLOAT } [(precision)]
{ DECIMAL } { FIXED } [(precision[,scale-fractor])]

PICTURE 'picture'
```

These attributes can be specified for all elements of an array and for individual members of a structure.

■ Other Data Type Attributes

The following attributes apply to program data that is not used for computation:

```
AREA
ENTRY [VARIABLE]
FILE [VARIABLE]
LABEL
OFFSET
POINTER
```

■ Storage Class and Scope Attributes

The following attributes control the allocation and use of storage for a data variable and define the scope of the variable:

```
AUTOMATIC   [INITIAL(initial-element,...)]
BASED [(pointer-reference)]
DEFINED(variable-reference) [POSITION(expression)]
STATIC [READONLY] [INITIAL(initial-element,...)]
parameter

EXTERNAL [ GLOBALDEF [(psect-name)] [ VALUE    ] ]
         [ GLOBALREF                [ READONLY ] ]

INTERNAL
```

■ File Description Attributes

The following attributes can be applied to file constants and used in OPEN statements:

ENVIRONMENT(option,...)

{ RECORD [KEYED] } { INPUT }
{ STREAM } { OUTPUT [PRINT] }
{ UPDATE }

{ DIRECT }
{ SEQUENTIAL }

■ Entry Name Attributes

The following attributes can be applied to identifiers of entry points:

ENTRY [VARIABLE] [OPTIONS (VARIABLE)]
BUILTIN
RETURNS (returns-descriptor)

■ Argument-Passing Attributes

The following attributes describe parameters of external procedures that are not written in PL/I:

ANY
VALUE

Table A-2: Alphabetical Summary of PL/I Attributes

Attribute	Use
ALIGNED	Requests alignment of bit-string variables in storage
ANY	Indicates that a parameter may have any data type
AREA	Defines a unit of storage for the allocation of based variables
{AUTOMATIC | AUTO}	Requests dynamic allocation of storage for a variable
BASED [(pointer-reference)]	Indicates that a variable's storage is located by a pointer
{BINARY | BIN}	Defines a binary base for arithmetic data
BIT	Defines bit-string data
BUILTIN	Defines a built-in function name
{CHARACTER | CHAR} [(length)]	Defines character-string data
{DECIMAL | DEC}	Defines a decimal base for arithmetic data
{DEFINED | DEF} (variable-reference)	Indicates that a variable will share the storage allocated for another variable
dimension	Indicates that a variable is an array and defines the number and extent of its dimensions
DIRECT	Specifies that a file will be accessed only randomly
ENTRY (descriptor,...)	Describes an external procedure and its parameters
{ENVIRONMENT | ENV} (option,...)	Specifies system-dependent information about a file
extent	Gives the length or dimension of a variable
{EXTERNAL | EXT}	Identifies the name of a variable whose storage is referenced or defined in other procedures
FILE	Identifies a PL/I file constant or file variable
FIXED	Defines a fixed-point arithmetic variable
FLOAT	Defines a floating-point arithmetic variable
GLOBALDEF [(psect-name)]	Defines an external variable and specifies the program section in which the variable will reside
GLOBALREF	Defines an external variable whose value is defined in an external procedure
{INITIAL | INIT} (value,...)	Provides initial values for variables
INPUT	Specifies that a file will be used for input
{INTERNAL | INT}	Limits the scope of a variable to the block in which it is defined

(Continued on next page)

Table A-2 (Cont.): Alphabetical Summary of PL/I Attributes

Attribute	Use
KEYED	Specifies that a file may be accessed randomly by key
LABEL	Defines a label variable
length	Specifies a length for a string variable
OFFSET	Defines an offset variable
OPTIONS	Specifies attribute options
OUTPUT	Specifies that a file will be used for output
parameter	Indicates that a variable will be assigned a value when the procedure is invoked
{PICTURE / PIC} 'picture'	Specifies the format of numeric data stored in character form
{POINTER / PTR}	Defines a pointer variable
{POSITION / POS}	Specifies the position within a variable at which a defined variable begins
precision,[scale-factor]	Specifies the number of digits in an arithmetic variable and, with fixed-point decimal data, the number of fractional digits
PRINT	Specifies that a file is to be formatted for printing
READONLY	Specifies that a static variable's value does not change during program execution
RECORD	Specifies that a file will be accessed by record I/O statements
RETURNS(returns-descriptor)	Specifies that an external entry is a function and describes the value returned by it
{SEQUENTIAL / SEQL}	Specifies that a file may be accessed sequentially
STATIC	Requests static allocation of storage
STREAM	Specifies that a file will be accessed by stream I/O statements
UPDATE	Specifies that records in a file may be rewritten or deleted
VALUE	Requests (1) that a global symbol be accessed by value rather than by reference, or (2) that an argument be passed to a non-PL/I procedure by immediate value
VARIABLE	Defines variable entry and file data
{VARYING / VAR}	Defines a varying-length character string

AUTOMATIC Attribute

The AUTOMATIC attribute specifies, for one or more variables, that PL/I is to allocate storage only for the duration of a block. An automatic variable is not allocated storage until the block that declares it is activated. The storage is released when the block is deactivated. The format of the AUTOMATIC attribute is:

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

AUTOMATIC is the default for internal variables.

AUTOMATIC explicitly defines the storage class of a variable, array, or major structure in a DECLARE statement. Because AUTOMATIC is the default, you need not specify it.

■ Restrictions

- The AUTOMATIC attribute conflicts with the following attributes:

STATIC	parameter
BASED	EXTERNAL
DEFINED	READONLY
GLOBALDEF	GLOBALREF

- The AUTOMATIC attribute cannot be applied to minor structures, members of structures, parameters, or descriptions in an ENTRY or RETURNS attribute.

For a discussion of PL/I storage allocation, **see** "Storage Classes."

B

B Format Items

The B format items—B, B1, B2, B3, and B4—describe representations of bit strings in a stream. The form of the B format items is:

B[m] [(w)]

m

The integer 1, 2, 3, or 4, specifying the radix factor. B and B1 have the same meaning. When the radix factor is omitted or is 1, the bit string is represented by the characters 0 and 1 in the stream. When the radix factor is 2, the bit string is represented by the characters 0, 1, 2, and 3. When the radix factor is 3, the bit string is represented by the characters 0, 1, 2, 3, 4, 5, 6, and 7. When the radix factor is 4, the bit string is represented by the characters 0 through 9 and A through F.

w

A nonnegative integer that specifies the width in characters of the field in the stream.

The interpretation of the B format items on input and output is given below. For a general discussion of format items, **see** "Format Items and Their Uses."

■ Input with GET EDIT

The integer w must be included when the B format items are used with GET EDIT. If it is zero, no operation is performed on the input stream, and a null string is assigned to the input variable. The number of characters specified by w is acquired. The input characters are converted to an intermediate bit string of length w*m. If the input target is not a bit-string variable, then this intermediate bit string is converted to the type of the input target, following the usual rules (for details, **see** "Conversion of Data").

The string of characters in the stream can be preceded or followed by spaces, which are ignored. All characters in the input field (except the leading/trailing spaces) must be those implied by the radix factor; otherwise, the ERROR condition is signaled. Consequently, input strings should not be enclosed in apostrophes, nor should they include the suffix Bm.

■ Output with PUT EDIT

The output source is converted, if necessary, to a bit string, following the usual rules for conversion to bit strings (**see** "Conversion of Data"). If the length of the resulting bit string is not a multiple of the radix factor (m), the bit string is padded with zeros on the right to make its length the next higher multiple (**see** "Examples" below).

The bit string is then converted to a character representation appropriate to the radix factor and placed in the output stream. The character representation is left-justified in the field specified by w and is truncated or padded with spaces on the right if necessary. If w is not included, the output string is of the same length as the converted output source. If w is zero, the B format item and its associated output source are skipped.

■ Examples

```
BFORMAT_XM: PROCEDURE OPTIONS(MAIN);
/* This program prints incorrect values for an integer
*/
DECLARE I FIXED BINARY(31);
DECLARE BFORM STREAM OUTPUT PRINT FILE;
I = 5;
OPEN FILE(BFORM) TITLE('BFORMXM,OUT');
PUT SKIP FILE(BFORM) EDIT ('Decimal:',I) (A,X,F(2));
PUT SKIP FILE(BFORM) EDIT ('Binary:',I) (A,X,B);
PUT SKIP FILE(BFORM) EDIT ('Base-4:',I) (A,X,B2);
PUT SKIP FILE(BFORM) EDIT ('Octal:',I) (A,X,B3);
PUT SKIP FILE(BFORM) EDIT ('Hexadecimal:',I) (A,X,B4);
END BFORMAT_XM;
```

The above program produces the output:

```
Decimal: 5
Binary: 00000000000000000000000000000101
Base-4: 0000000000000022
Octal: 00000000024
Hexadecimal: 0000000A
```

The base-4, octal, and hexadecimal representations of I are incorrect, because the precision of I (31) is not a multiple of 2, 3, or 4. For the B2 and B4 format items, an extra zero bit was appended to the intermediate bit string, multiplying it by two. For B3, two extra bits were appended, to make the string 33 bits long and thus divisible into an exact number of three-bit segments. To avoid this problem, the precision of the output source must be a number that is evenly divisible by any radix factor with which it is to be written out, as in:

```
BFORMAT_XM: PROCEDURE OPTIONS(MAIN);
/* This program prints correct values for an integer */
DECLARE I FIXED BINARY(24); /* 24 is a multiple of
2*3*4 */
DECLARE BFORM STREAM OUTPUT PRINT FILE;
I = 5;
OPEN FILE(BFORM) TITLE('BFORMXM5,OUT');
PUT SKIP FILE(BFORM) EDIT ('Decimal:',I) (A,X,F(2));
PUT SKIP FILE(BFORM) EDIT ('Binary:',I) (A,X,B);
PUT SKIP FILE(BFORM) EDIT ('Base-4:',I) (A,X,B2);
PUT SKIP FILE(BFORM) EDIT ('Octal:',I) (A,X,B3);
PUT SKIP FILE(BFORM) EDIT ('Hexadecimal:',I) (A,X,B4);
END BFORMAT_XM;
```

This version of the program outputs:

```
Decimal: 5
Binary: 000000000000000000000101
Base-4: 000000000011
Octal: 00000005
Hexadecimal: 000005
```

The output values are correct representations of I because the precision (24) is evenly divisible by 2, 3, or 4.

The tables below show the relationship between the internal and external representations of characters that are read or written with the B format item.

Input Examples

The "input stream" shown in the table is a field of characters beginning at the current position in the stream and continuing to the right. The target type is the type of the variable to which the input value is assigned.

Format Item	Input Stream	Target Type	Target Value
B(12)	111000111110...	BIT(12)	'111000111110'B
B(12)	△△△△△△110011...	BIT(12)	'110011000000'B
B2(6)	123123...	BIT(12)	'011011011011'B
B3(4)	1775...	BIT(12)	'001111111101'B
B4(3)	1FA...	BIT(12)	'000111111010'B

Output Examples

The output source value shown in the table is either a constant or the value of a variable that is written out with the associated format item.

Output Source Value	Format Item	Output Value
4095	B	111111111111
4095	B(11)	11111111111
4095	B2	333333
4095	B3	7777
4095	B4	FFF

BASED Attribute

The BASED attribute defines a based variable, that is, a variable whose actual storage will be denoted by a pointer or offset reference. For general information, **see** "Based Variable." The format of the BASED attribute is:

BASED [(reference)]

reference

A reference to a pointer or offset variable or pointer-valued function. If the reference is to an offset variable, that variable must be declared with a base area. Each time a reference is evaluated that is to the based variable without an explicit pointer or offset qualifier, the reference is evaluated to obtain the pointer or offset value.

■ Restrictions

- The following attributes conflict with the BASED attribute:

AUTOMATIC	GLOBALDEF
DEFINED	GLOBALREF
EXTERNAL	STATIC
READONLY	INITIAL
VALUE	parameter

- The BASED attribute cannot be applied to minor structures, members of structures, parameters, or descriptions in an ENTRY or RETURNS attribute.

Based Variable

A based variable is a variable that describes storage that will be accessed through a pointer or offset value. PL/I does not automatically allocate any storage for a based variable. Instead, storage is allocated or specified explicitly by the user.

This entry gives the rules governing references to based variables and the use of pointer values. It also presents examples of dynamic storage allocation, the use of READ SET, and the use of the ADDR built-in function.

■ References to Based Variables

A reference to a based variable (except in an ALLOCATE statement) must specify a pointer or offset reference designating the storage to be accessed by the reference. This qualifying pointer or offset reference may be specified implicitly, by giving it in the BASED attribute, or explicitly, by prefixing the based variable reference with a locator qualifier. A complete based variable reference (with the locator qualifier) has the form:

qualifying-reference -> base-reference

Whether explicit or implicit, the qualifying reference must be a reference to a pointer variable, a pointer-valued function, or an offset variable declared with a base area. The qualifying reference is evaluated each time the complete reference is evaluated and must yield a valid pointer value (**see** "Pointer Values" below). If the qualifying reference is to an offset variable, the offset value is converted to a pointer using the base area specified in the offset variable's declaration. (For more details on offsets and areas, **see** the *VAX-11 PL/I User's Guide* and the entries "Offset" and "Area" in this manual.)

Both implicit and explicit qualification may be used with the same based variable; the explicit qualifier overrides the implicit one. For example:

```
DECLARE X FIXED BIN BASED(P);
P = ADDR(A);
X = ADDR(B)->X;
```

In the second assignment statement, the reference to X on the left-hand side of the assignment has the implicit qualifier P, which is the address of the variable A. The reference to X on the right-hand side is explicitly qualified with the address of another variable, B. This assigns the value of B to the variable A.

■ Pointer Values

In VAX-11 PL/I, a valid pointer value may be obtained in any of the following ways:

- Through the SET option of the ALLOCATE statement
- From a user-provided storage allocation routine

- Through the SET option of the READ statement
- From applying the ADDR built-in function to an addressable variable (**see** "Variable — Addressable Variable")
- By converting an offset value to a pointer value

A pointer value is valid only as long as the storage to which it applies remains allocated. Moreover, a pointer obtained by applying ADDR to a parameter is valid only as long as the parameter's procedure invocation exists, even though the storage to which the pointer points may exist longer.

The NULL built-in function returns a null pointer value that can be assigned to pointer and offset variables, but the null value is not valid as the pointer value qualifying a based variable reference.

It is possible, using the UNSPEC built-in function or based variables, to assign an arbitrary value to a pointer variable. Such a value is invalid even if it denotes allocated storage, and use of such values causes unpredictable program behavior and errors that are difficult to diagnose. For example, the following program attempts to use pointer arithmetic to "alias" two variables X and Y:

```
ALIAS: PROCEDURE OPTIONS(MAIN);

DECLARE INDEX FIXED BINARY(31),
        P POINTER BASED(ADDR(INDEX));
DECLARE (X,Y) FLOAT BINARY(24) STATIC, /* 4 bytes apart (?) */
        (A,B) FLOAT BINARY(24) BASED;

X = 1E0; Y = 2E0;
P = ADDR(X); /* INDEX holds the address of X */
P->A = Y + 1; /* Expect X = Y+1 */
INDEX = INDEX + 4; /* INDEX now holds address of Y (?) */
P->B = Y + 1; /* Expect Y = Y + 1 */
PUT SKIP LIST('P->A:',P->A,'P->B:',P->B);
END ALIAS;
```

The program may produce incorrect results in at least two ways:

1. It can be assumed that the programmer knows, perhaps from a storage map, that X and Y occupy adjacent storage and that Y can be accessed by incrementing INDEX. However, this is not necessarily true for any two variables, and the program does rely on the assumption.
2. If common subexpressions are eliminated during the compiler's optimization of this program, incorrect results occur. The optimization results in:

   ```
   T = Y + 1;
   P->A = T;
   P->B = T;
   ```

   The expected result of the program was to give B a value equal to the original value of Y plus 2. However, the assignment to B yields an incorrect result because the assignment to A modified Y, and the compiler had no way to discover that Y was an aliased variable.

■ Data Type Matching for Based Variables

In most applications, the data type of a based variable reference is identical to the data type under which the accessed storage is allocated. (For a discussion

of identical data types, **see** "Data and Data Types.") However, it is not required that the data types be identical. In standard PL/I, it is sufficient that the data types match as for overlay defining or that they are left-to-right equivalent. Moreover, in VAX-11 PL/I, the data types may be quite different, although the program will then depend on the VAX internal representation of data. The following subsections discuss these type-matching criteria in more detail.

Matching by Overlay Defining

This type of matching is in effect if the based variable reference and the variable for which the storage was originally allocated are both suitable for character (or bit) string overlay defining. (**See** "Defined Variable" for a discussion of string overlay defining.) The only further restriction is that the size n (in characters or bits) of the based variable must be less than or equal to the size in characters or bits of the original variable. The based variable reference accesses the first n characters or bits of the storage.

Matching by Left-to-Right Equivalence

This type of matching applies to structured variables that are identical up to a certain point. To see if this applies, examine the declaration of the based variable, and consider only the portion on the left that includes the referenced member and all of the level-2 substructure containing the referenced member (if the member is not itself at level 2). If the original variable's declaration has a similar left part with identical data type, then the based variable reference and the original reference match. For example:

```
DECLARE 1 S1 BASED (P),
 2 X,
         3 (A,B) FIXED BIN,
 2 Y,
         3 C CHAR(10),
         3 D(5) FLOAT;

DECLARE 1 S2 BASED(P);
 2 X,
         3 (A,B) FIXED BIN,
 2 Y,
         3 C CHAR(10),
         3 E BIT(32);

ALLOCATE S1;

S2.A = 3; /* valid left-to-right match */

S2.C = 'X'; /* INVALID */
```

In the first assignment, S2.A is a valid reference because S1 and S2 match through the level-2 structure X. In the second assignment, S2.C is invalid in standard PL/I because the level-2 structures S2.Y and S1.Y do not match. (However, the reference to S2.C does work in VAX-11 PL/I.)

This sort of matching is useful in connection with data structures and files, where the first part of a record contains a value indicating the precise structure of the remainder of the record.

Nonmatching Based Variable References

In VAX-11 PL/I, the base variable in a based variable reference need not match the variable for which the storage was originally allocated. The only requirement is that the size of the based variable in bits is less than or equal to the size of the original variable in bits. However, use of such nonmatching references requires knowledge of the VAX internal representation of data, and you should not expect the resulting code to be transportable to other PL/I implementations. For example:

```
DECLARE X FLOAT BINARY(24);
DECLARE 1 S BASED(ADDR(X)),
 2 FRAC_1 BIT(7),
 2 EXP BIT(8),
 2 SIGN BIT(1),
 2 FRAC_2 BIT(16);

EXP = '0'B; /* set exponent to 0 */
SIGN = '1'B; /* set sign negative */
X = X + 1;
```

The declaration of S describes the internal representation of a VAX-11 single-precision floating-point number. The first two assignments set the sign and exponent fields to the reserved operand combination. The assignment to X causes a reserved operand exception.

■ Based Variables and Dynamic Storage Allocation

These subsections discuss the dynamic allocation of storage by the ALLOCATE statement and the READ SET statement.

Using the ALLOCATE Statement

Each time it is executed, the ALLOCATE statement allocates storage for a based variable and, optionally, sets a pointer variable to the location of the storage in memory. For example:

```
DECLARE LIST (10) FIXED BINARY BASED,
          (LIST_PTR_A, LIST_PTR_B) POINTER;

ALLOCATE LIST SET (LIST_PTR_A);
ALLOCATE LIST SET (LIST_PTR_B);
```

In this example, the array LIST is declared with the BASED attribute; however, the declaration does not reserve storage for this variable. Instead, the ALLOCATE statements allocate storage for the variable and set the pointers LIST_PTR_A and LIST_PTR_B to the storage locations. LIST_PTR_A and LIST_PTR_B must both be declared with the POINTER attribute.

In references, the different allocations of LIST can then be distinguished (unless the pointers are assigned new values) by locator qualifiers that identify the specific allocation of LIST. For example:

```
LIST_PTR_A -> LIST(1) = 10;
LIST_PTR_B -> LIST(1) = 15;
```

The phrase LIST_PTR_A-> is a locator qualifier; it specifies the pointer that locates an allocation of storage for the variable. In this example, the first element of the storage pointed to by LIST_PTR_A is assigned the value 10. The first element of the storage pointed to by LIST_PTR_B is assigned the value 15.

Figure B-1 illustrates this example.

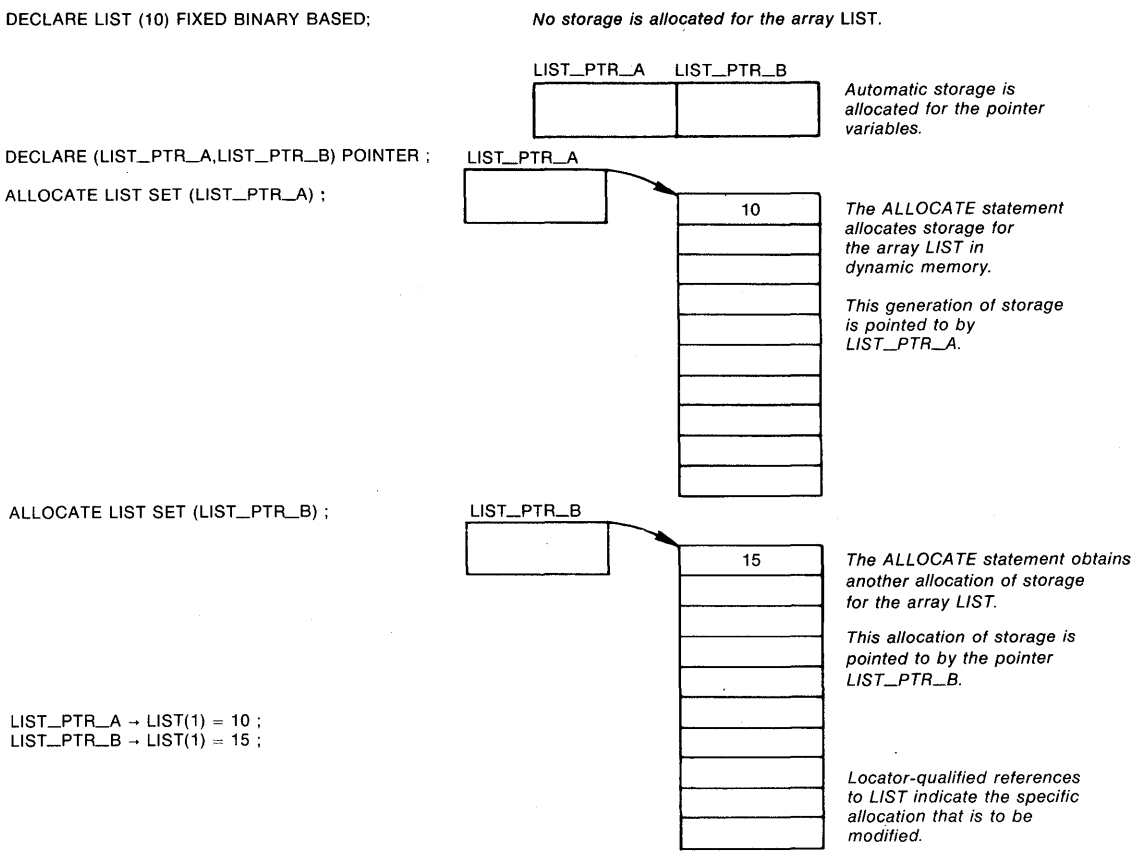


Figure B-1: Using the ALLOCATE Statement

Any extent expressions in the based variable declaration are evaluated each time the variable is allocated or referenced. Therefore, based variables can be used for data aggregates whose size depends on input data. Here is an example of dynamically allocating a matrix that will be accessed by several external procedures:

```
DECLARE 1 MATRIX_CONTROL_BLOCK STATIC EXTERNAL,
 2 MATRIX_POINTER POINTER,
 2 (ROW_SIZE,COL_SIZE) FIXED BINARY;

DECLARE 1 MATRIX(ROW_SIZE,COL_SIZE)
 BASED(MATRIX_POINTER);

GET LIST(ROW_SIZE,COL_SIZE);
ALLOCATE MATRIX;
```

The SET Option of the READ Statement

When you use the READ statement with a based variable, you do not have to define storage areas within your program to buffer records for input/output operations. If you specify the SET option on the READ statement, the READ statement places an input record in a system buffer and sets a pointer variable to the location of this buffer. For example:

```
DECLARE REC_PTR POINTER ,
        INFILE FILE RECORD INPUT SEQUENTIAL ;
DECLARE 1 RECORD_LAYOUT BASED (REC_PTR),
          2 NAME CHARACTER (15),
          2 AMOUNT PICTURE '999V99',
          2 BALANCE FIXED DECIMAL (6,2);
    .
    .
READ FILE (INFILE) SET (REC_PTR) ;
    .
    .
REWRITE FILE (INFILE) ;
```

In this example, the structure defined to describe the records in a file is declared with the BASED attribute; the declaration does not reserve storage for this structure. When the READ statement is executed, the record is actually read into a system buffer, and the pointer REC_PTR is set to its location.

When the SET option is used with the READ statement, a subsequently executed REWRITE statement need not specify the record to be rewritten. PL/I rewrites the record indicated by the pointer variable specified in the READ statement.

Figure B–2 illustrates this example.

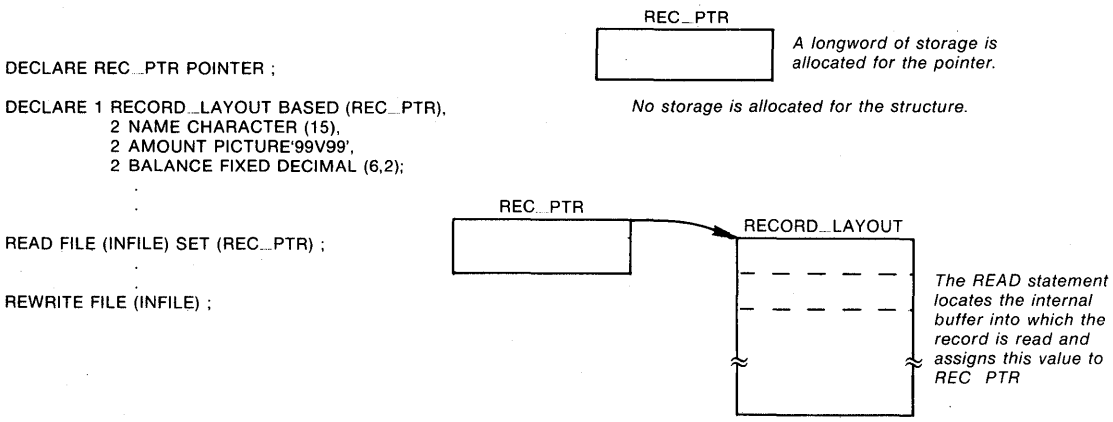


Figure B–2: Using the READ Statement with a Based Variable

■ Examples

The program DEFINED uses based variables and the READ SET statement to process a file of personnel data (PERSONNEL.DAT). The file has two types of valid records: a pay record and a health record. The different record types are identified by a one-character code in the first position. The two record types are declared as based structures (PAY_RECORD and HEALTH_RECORD), one of which is selected based on the record type character ('P' for pay, 'E' for health). Any record that does not begin with one of these characters is invalid and is written out as a reference to the based character variable INVALID_RECORD.

```
DEFINED: PROCEDURE OPTIONS(MAIN);

DECLARE P POINTER; /* pointer to structures */

DECLARE       1 PAY_RECORD BASED(P),
      2 RECORD_TYPE CHARACTER(1),
      2 NAME CHARACTER(20),
      /* the two structures differ in this member: */
      2 GROSS_PAY PICTURE '999999V.99';

DECLARE 1 HEALTH_RECORD BASED(P),
      2 RECORD_TYPE CHARACTER(1),
      2 NAME CHARACTER(20),
      2 EXAM_DATE CHARACTER(9);

DECLARE INVALID_RECORD CHARACTER(30) BASED(P);

DECLARE PERSONNEL RECORD FILE;
DECLARE PERSOUT STREAM OUTPUT PRINT FILE;

/* used to control DO group: */
%REPLACE NOTENDFILE BY '1'B;

ON ENDFILE(PERSONNEL) BEGIN;
      PUT FILE(PERSOUT) SKIP LIST
               ('All processing complete.');
      STOP;   /* program stops here */
      END;

OPEN FILE(PERSONNEL) INPUT TITLE('PERSONNEL.DAT');

DO WHILE(NOTENDFILE);
/* terminated by ENDFILE ON-unit */

READ FILE(PERSONNEL) SET(P);
/* P is the location of the
record acquired by the READ statement */

IF P->PAY_RECORD.RECORD_TYPE = 'P' THEN
      PUT FILE(PERSOUT) SKIP LIST
      ('Name=',P->PAY_RECORD.NAME,
      'Gross pay=',P->GROSS_PAY);
```

```
ELSE  /* either a health record or an invalid record */
      DO;
      IF P->HEALTH_RECORD.RECORD_TYPE = 'E' THEN
      PUT FILE(PERSOUT) SKIP LIST
      ('Name=',P->HEALTH_RECORD.NAME,
      'Exam date:',P->EXAM_DATE);
      ELSE /* invalid record type */
      PUT FILE(PERSOUT) SKIP LIST
            ('Invalid record:',P->INVALID_RECORD);
      END;

END; /* repeat DO group until ENDFILE is signaled */

END DEFINED;
```

For example, if the file PERSONNEL.DAT contains the following records:

```
PMary A. Ford         125000.55
EMary A. Ford         22July 80
t12345678901234567890ppppppp.pp
```

then the output file (PERSOUT.DAT) will contain the following output:

```
Name=    Mary A. Ford              Gross pay=     125000.55
Name=    Mary A. Ford              Exam date:     22July 80
Invalid record: t12345678901234567890ppppppp.pp
All processing complete.
```

Notice the following other features of the program:

- The references to based variables have a locator qualifier (P->) for clarity. However, since all were declared with P as their pointer reference, the locator qualifier could have been omitted.
- References to the structure members RECORD_TYPE and NAME must be fully qualified with the name of their containing structures (PAY_RECORD and HEALTH_RECORD) because both structures have members with these names. In contrast, GROSS_PAY and EXAM_DATE are unique to their structures and need not be fully qualified.

■ Using the ADDR Built-In Function

The ADDR built-in function returns the storage location of a variable. It can be used to associate the storage occupied by a variable with the description of a based variable. For example:

```
DECLARE A FIXED BINARY BASED (X),
        B FIXED BINARY,
        X POINTER;

        X = ADDR (B);
A = 15;
```

In this example, the variable A is declared as a based variable, with the pointer X designated as its pointer. The variable B is an automatic variable; PL/I allocates storage for B when the block is activated. When the ADDR built-in function is referenced, it returns the location in storage of the variable B and the assignment statement gives this value to the pointer, X. This

assignment associates the variable A with the storage occupied by B. Because A is based on X and X points to B, an assignment statement that gives a value to A actually modifies the storage occupied by the variable B.

Figure B–3 illustrates this example.

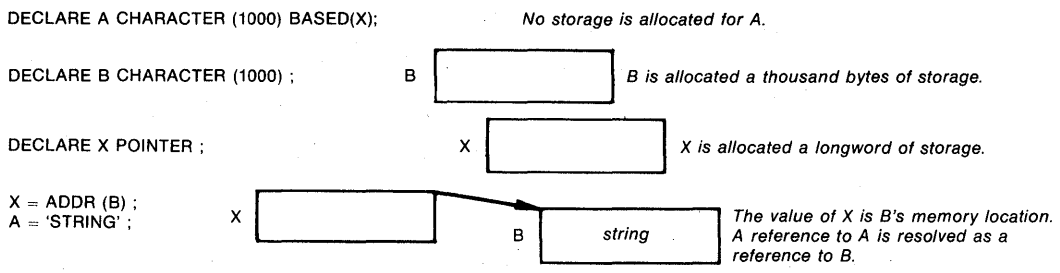


Figure B–3: Using the ADDR Built–In Function

■ Based Variables and List Processing

Data structures in which the elements have complex interactions or may be added or deleted are normally described with based variables. The simplest such structure is a linked list. For an example, **see** "List Processing."

Begin Block

A begin block is a sequence of statements headed with a BEGIN statement and terminated by an END statement. In general, a begin block can be used wherever a single executable statement is valid, for instance, in an ON-unit.

The statements in a begin block can be any PL/I statements, and begin blocks can contain DO-groups, DECLARE statements, procedures, and other (nested) begin blocks.

A begin block provides a convenient way to localize variables. The variables that are declared as internal variables within a begin block are not allocated storage until the begin block is activated. When the begin block terminates, storage for internal automatic variables is released. A begin block is terminated when:

- Its corresponding END statement is executed. Control continues with the next executable statement in the program.
- It executes a nonlocal GOTO to transfer control to a previous block.

A begin block differs from a DO-group chiefly in its ability to localize variables. Variables declared within DO-groups are not localized to the group (unless, of course, the group contains a begin block or procedure that declares internal variables). Begin blocks are preferable when you want to restrict the scope of variables, and there are some cases (such as ON-units) in which DO-groups cannot be used. Otherwise, DO-groups are often more efficient than begin blocks, because they do not have the overhead associated with block activation.

For more information, **see** "Block."

A begin block can designate a series of statements to execute depending on the success or failure of a test in an IF statement. For example:

```
IF A = B THEN BEGIN ;
     .
     .
END;
```

A begin block also provides the only way to denote a series of statements to be executed when an ON condition is signaled. For example:

```
ON ERROR BEGIN;  [statement ...] END;
```

For further information, **see** "ON Conditions and ON-Units."

BEGIN Statement

The BEGIN statement denotes the start of a begin block. The format of the BEGIN statement is:

BEGIN;

A begin block must be terminated with an END statement.

BINARY Attribute

The BINARY attribute specifies that an arithmetic variable has a binary base. The format of the BINARY attribute is:

{ BINARY }
{ BIN   }

When you specify the BINARY attribute for an identifer, you can also specify one of the following attributes to define the scale and precision of the data:

FIXED [(precision)]
FLOAT [(precision)]

where FIXED indicates a fixed-point binary value and FLOAT indicates a floating-point binary value. The precision of a binary value indicates the number of bits to be used to maintain its value. For a fixed-point binary value, the precision specifies the number of bits representing an integer and must be in the range 1–31. For a floating-point value, the precision specifies the number of bits representing the mantissa of a floating-point number and must be in the range 1–113. The maximum floating-point binary precision on a standard VAX–11/780 is 53. (Use of precisions in the range 54–113 requires optional hardware; **see also** "Floating-Point Data.") The default values applied to the BINARY attribute are:

Attributes Specified	**Defaults Supplied**
BINARY	FIXED (31)
BINARY FIXED	(31)
BINARY FLOAT	(24)

■ Restrictions

The BINARY attribute directly conflicts with the DECIMAL attribute and with any other data type attribute.

BINARY Built-In Function

The BINARY built-in function converts an arithmetic or string expression x to its binary representation with an optionally specified precision p. The returned value is either fixed- or floating-point binary, depending on whether x is a fixed- or floating-point expression.

The precision p, if specified, must be an integer constant greater than zero and less than or equal to the maximum precision of the result type (31 if fixed-point binary and 113 if floating-point binary). P must be specified if x is a fixed-point decimal value with fractional digits.

The scale factor q, if specified, must be the constant 0.

The format of the function is:

$$\left\{\begin{matrix}\text{BINARY}\\ \text{BIN}\end{matrix}\right\}\ (x[,p[,q]])$$

■ Returned Value

The result type is fixed- or floating-point binary, depending on whether the argument x is a fixed- or floating-point expression. (If the argument is a bit- or character-string expression, the result type is fixed-point binary.)

The argument x is converted to the result type, giving a value v, following the usual rules for conversion (**see** "Conversion of Data" for details).

The returned value is the value v, with precision p. If p is omitted (integer and floating-point arguments only), the precision of the returned value is the converted precision of x (**see** "Expression" for details). FIXEDOVERFLOW, OVERFLOW, or UNDERFLOW is signaled if appropriate.

BIT Attribute

The BIT attribute identifies a variable as a bit-string variable. The format of the BIT attribute is:

BIT [(length)]

length

The number of bits in the variable. If not specified, PL/I assumes a default length of one bit. The length must be in the range of 0 through 32767.

The rules for specifying the length are as follows:

- If the attribute is specified for a static variable declaration or in a returns descriptor, length must be an integer constant.

- If the attribute is specified in the declaration of a parameter or in a parameter descriptor, length may be specified as an integer constant or as an asterisk (*).
- If the attribute is specified for an automatic, based, or defined variable, length may be specified as an integer constant or as an expression. In the case of automatic or defined variables, the expression must not contain any variables or functions that are declared in the same block, except for parameters.

If specified, the length must immediately follow the keyword BIT.

If you give a variable the BIT attribute, you can also specify the ALIGNED attribute to request alignment of the variable on a byte boundary in storage.

■ Restrictions

The BIT attribute directly conflicts with the CHARACTER and VARYING attributes and with any other data type attribute.

BIT Built-In Function

The BIT built-in function converts an arithmetic or string expression x to a bit string of an optionally specified length. If x is a string expression, it must consist of 0s and 1s. If the length is specified, it must be a nonnegative integer. If the length is omitted, the returned value has a length determined by the usual rules for conversion to bit strings (**see** "Conversion of Data").

The format of the function is:

```
BIT(x[,length])
```

Bit-String Data

A bit string consists of a sequence of binary digits, or bits. A bit-string may be used as a Boolean value. A Boolean value has one of two states: true (if any bit is 1) or false (if all bits are 0).

Like a fixed-length character string, a bit string has a fixed length defined in the declaration or specified by the number of bits in a bit-string constant; bit-string variables cannot be declared with the VARYING attribute.

This discussion of bit-string data is divided into the following parts:

- Constants
- Variables
- Alignment
- Internal representation

■ Bit-String Constants

To specify a bit-string constant, enclose the string in apostrophes and follow the closing apostrophe with the letter B. Some examples of bit-string constants are:

```
'0101'B
'10101010'B
'1'B
```

The length of a bit-string constant is always the number of binary digits specified; the B does not count in the length of the string. The maximum length of any bit string is 32767 bits. A bit-string constant can be specified with a maximum of 1000 characters between the apostrophes.

You can also specify a bit-string constant using the syntax:

```
'character-string'Bn
```

where n specifies the number of bits to be represented by each character in the specified string. This format allows you to specify bit-string constants that have bases other than two. For example:

```
'EF8'B4
'117'B3
'223'B2
```

These constants specify the hexadecimal value EF8, the octal value 117, and the base 4 value 223.

All such constants are stored internally as bit strings. **See** "Internal Representation of Bit Data," below.

The characters that are valid for each type of bit-string constant are listed here:

- For B or B1, only the characters 0 and 1 are valid.
- For B2, only the characters 0, 1, 2, and 3 are valid.
- For B3, only the characters 0, 1, 2, 3, 4, 5, 6, and 7 are valid.
- For B4, the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are valid. (The letters A–F can be either upper or lower case.)

Using the B format items, you can also acquire or output bit-string data in binary, base 4, octal, or hexadecimal format. **See** "B Format Items."

■ Bit-String Variables

Use the keyword BIT to declare a bit-string variable. The format is:

```
DECLARE variable-name BIT [(n)];
```

where n is the length of the variable.

When a program assigns a value to a bit-string variable, the value can be larger or smaller than the defined length of the variable. In such cases, PL/I does the following:

- If the assigned string is shorter than the defined target length, PL/I pads the bit-string value in the direction of least significance with zeros. The "less significant" bits are those shown on the right, as the string is represented by PUT LIST.

- If the assigned string is longer than the target, PL/I truncates the least significant bits from the bit-string value.

If you do not specify a length for a bit-string variable, PL/I uses the default length of one bit.

NOTE

You should avoid using bit strings to represent integers. The truncation or padding that occurs in assignments between strings of different lengths results in an implicit division or multiplication of the numeric interpretation of the string; these implicit operations can introduce subtle errors in computations.

■ Alignment of Bit-String Data

PL/I distinguishes between aligned and unaligned bit-string variables. (Bit-string constants are always unaligned.) A bit-string variable is aligned only if it is declared with the ALIGNED attribute, as shown in the example below:

```
DECLARE FLAGS BIT (8) ALIGNED;
```

PL/I allocates storage for an aligned bit-string variable on a byte boundary and reserves an integral number of bytes to contain the variable.

Unaligned bit-string variables always occupy only as many bits as are needed to contain them. They need not be on byte boundaries.

In general, operations involving unaligned bit-string variables are less efficient than operations involving aligned bit-string variables. Unaligned bit-string variables are invalid as the targets of the FROM and INTO options of record I/O statements and as the argument of the ADDR built-in function. Moreover, most non-PL/I programs that accept bit-string arguments require that the strings be aligned.

This distinction affects argument passing. If a procedure declares a parameter as an aligned bit string, and if the corresponding argument that is passed to it is an unaligned bit-string variable, or vice versa, the actual argument will be a dummy variable. For example:

```
DECLARE GETSTRING ENTRY (BIT (*) ALIGNED);
DECLARE STRING BIT (8);
CALL GETSTRING (STRING);
```

In the above example, PL/I constructs a dummy variable to pass the argument STRING to the called procedure GETSTRING, rather than passing the actual argument by reference.

It is recommended that you declare bit-string variables using the ALIGNED attribute in most cases. Use unaligned bit-string variables when bit strings must be packed as tightly as possible, for example, in arrays and in structures.

■ Internal Representation of Bit Data

The way that PL/I allocates storage for a bit-string variable depends on whether the variable is declared with the ALIGNED attribute.

In this discussion, the term "most significant bit" means the leftmost bit in an external representation of the string, as, for example, when the string is output by the PUT LIST statement. The "least significant bit" is the rightmost bit in the external representation.

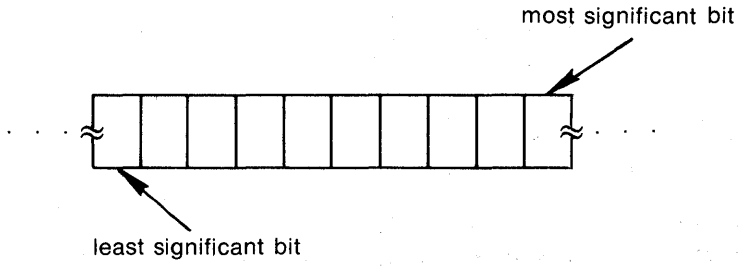
Of course, the notion of significance has no meaning for bit strings unless they are used to store integers. VAX-11 PL/I permits the use of bit strings for this purpose, and it has defined rules for conversions between bit strings and other data types (**see** "Conversion of Data"). Nevertheless, the use of PL/I bit-string data to store integers is not recommended, for two reasons:

1. In assignments involving two bit strings of different lengths, the source string is padded or truncated as required to make a string of the length of the target.
2. As shown in the following discussions, bit strings are stored, in regard to the "significance" of bits, in the reverse order from actual numeric data. In consequence, the conversion of bit strings to arithmetic data is an expensive one in terms of execution speed, except in the special case of a one-bit string.

It is recommended instead that you use the UNSPEC built-in function and UNSPEC pseudovariable when it is absolutely necessary to store integers in a compact form. Otherwise, use the data types FIXED BINARY and FIXED DECIMAL for integer arithmetic.

Unaligned Bit Strings

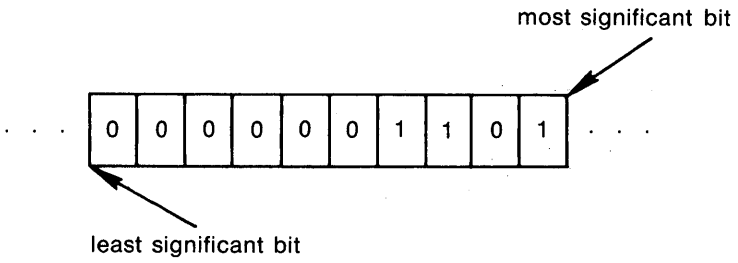
An unaligned bit string is stored beginning at an arbitrary bit location in storage; this location is the location of the most significant bit. The subsequent, less significant, bits are stored in progressively higher locations in memory, as shown here:



The following programming sequence illustrates how a value for an unaligned bit-string variable is stored:

```
DECLARE ABIT BIT (10);
ABIT = '1011'B;
```

After the assignment, the variable appears in storage like this:



Aligned Bit Strings

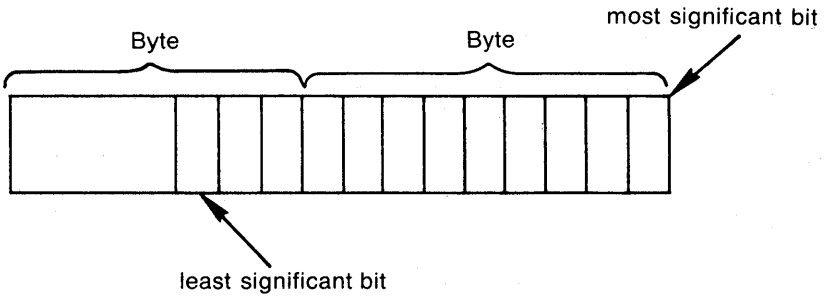
PL/I allocates storage for an aligned bit-string variable on a byte boundary and allocates an integral number of bytes. The number of bytes to allocate is calculated as follows:

ceil(n/8)

where n is the length specified for the bit string.

Beginning at bit 0 (the lowest memory location) of the lowest allocated byte, the bit string is stored like unaligned bit-string data; that is, the beginning bit is used to hold the most significant bit in the string. Less significant bits are stored in progressively higher memory locations. Unused bits are set to zeros each time the bit-string variable is assigned a value.

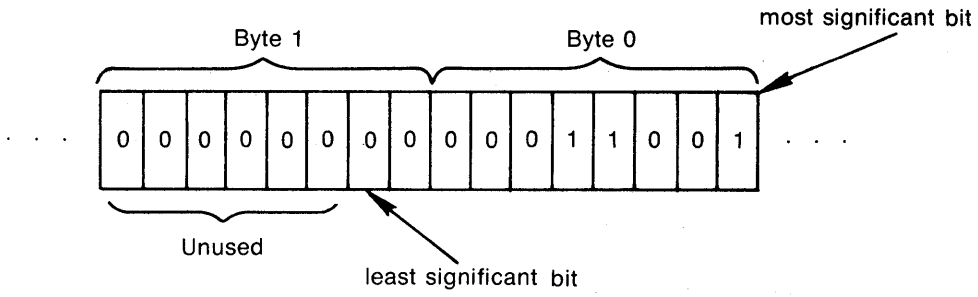
The representation is as follows:



The following programming sequence illustrates how values are stored for aligned bit strings:

```
DECLARE ABIT BIT (10) ALIGNED;
ABIT = '10011'B;
```

In this example, the variable ABIT is aligned. When it is assigned the value 10011, its storage appears as follows:



Block

A block is a sequence of PL/I statements. The types of block in a PL/I program are:

- Procedure blocks. A procedure block begins with a PROCEDURE statement and terminates with an END statement. A procedure is the basic program unit of PL/I; it also defines the scope of names declared within it.
- Begin blocks. A begin block begins with a BEGIN statement and terminates with an END statement. A begin block delimits a portion of a program and defines the scope of names declared within it.

Blocks control the scope of names, the allocation of storage for automatic variables, and the search for ON-units to respond to a condition.

■ Containment

A block A is said to be contained in another block B if all of A's source text, from label (if any) to END statement inclusive, is between B's BEGIN or PROCEDURE statement and B's END statement. If there is no block C contained in B and containing A, A is also said to be immediately contained in B. For example:

```
B: PROCEDURE OPTIONS(MAIN);
      A: PROCEDURE;
               CALL Q;
               END A;
      Q: PROCEDURE;
               .
               .
               .
               END Q;
      BEGIN;
               CALL A;
      END; /* of begin block */
END B;
```

The procedures A and Q, and the begin block, all are immediately contained in B.

If block A is contained in block B, they are also said to be nested. The maximum nesting level is 64.

■ Block Activation

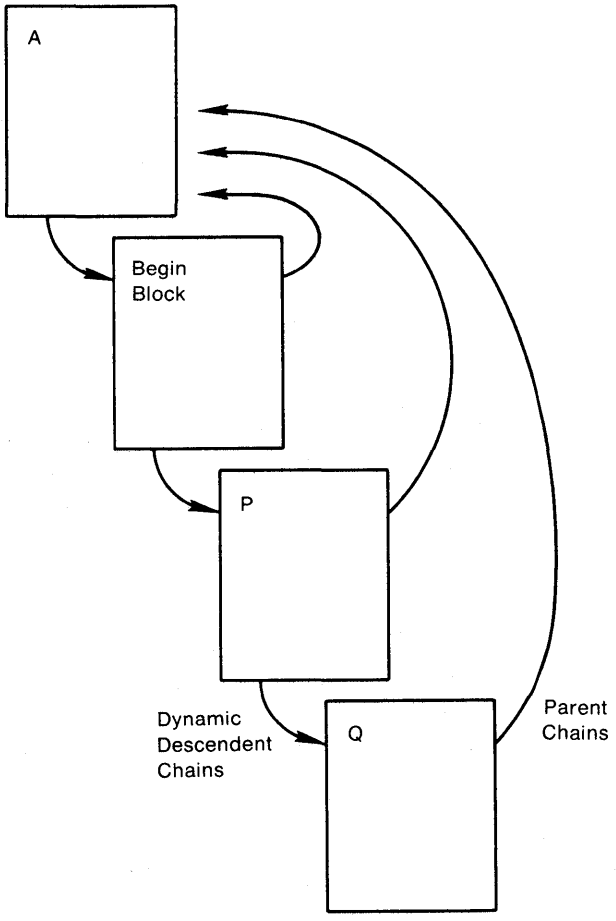
A block is activated when program execution flows into it. All automatic variables declared in the block become active. When control leaves the block, the variables become undefined and inaccessible.

A procedure block can be entered only by a CALL statement or a function reference. If an internal procedure is declared within a source program, control flows around the internal procedure during the normal sequence of execution.

A begin block, on the other hand, is entered when it is encountered during the normal flow of execution.

■ Relationship of Block Activations

During the execution of a program, many blocks can be simultaneously active. Two different relationships are defined among block activations and illustrated in Figure B–4.




```
A:  PROCEDURE OPTIONS (MAIN) ;

    P:  PROCEDURE;
        CALL Q;
          .
          .
        END P;
    Q:  PROCEDURE;
          .
          .
        END Q;

    BEGIN;
        CALL P;
        END; /* of begin block */

END A;
```

Figure B–4: Relationship of Block Activations

1. The first relationship is "immediate dynamic descendence." A block activation is the immediate dynamic descendent of the block that invoked it. At a given time, the chain of immediate dynamic descendents includes all existing block activations, starting with the activation of the main procedure and terminating in the current block activation. For example, in Figure B–4, the begin block is the immediate dynamic descendent of procedure A; the complete chain is: A, begin block, P, Q. This chain is used to find the applicable ON-unit when a condition is signaled. (**See also** "ON Conditions and ON-Units.")
2. The second relationship applies to activations of nested blocks. An activation of a block X that is a begin block or internal procedure has an "immediate parent activation." The immediate parent activation of X is an activation of the block that immediately contains X. The chain of immediate parent activations extends back to an activation of the external procedure containing X. In Figure B–4, the parent chain for each of the begin block, procedure P, and procedure Q leads directly back to the activation of A, because each of these blocks is immediately contained in A. This chain is used in interpreting references. (**See also** "Reference.")

When a block is activated, its immediate parent activation is determined as follows:

1. If the block is an external procedure, it has no parent activation.
2. If the block is a begin block, its immediate parent activation is the activation that invoked it. Therefore, the begin block is the immediate dynamic descendent of its immediate parent.
3. If the block is an internal procedure invoked in block activation A by a reference to an entry constant declared in block B, then the immediate parent of the new block activation is the activation of B in the parent chain starting at A.
4. If the block is an internal procedure invoked via an entry variable, the parent activation is taken from the entry value. It was originally set when the complete entry value was generated by assigning an entry constant to an entry variable. (**See** "Entry Data.")

■ Block Termination

When a block terminates normally, that is, when an END statement or a RETURN statement is executed, the current block is released and control goes to the preceding block activation. If a nonlocal GOTO statement is executed that transfers control out of the current block, the current block and any blocks between it and the block containing the label that is the target of the GOTO statement are released.

For more information, **see** "Begin Block," "Procedure," "Procedure Block," and "Scope of Names."

BOOL Built-In Function

The BOOL built-in function performs a Boolean operation on two bit-string arguments and returns the result as a bit string with the length of the longer argument. Its format is:

BOOL(string-1,string-2,operation-string)

string-1

A bit-string expression of any length.

string-2

A bit-string expression of any length.

operation-string

A bit-string expression that is converted to length 4. Each bit in the operation string specifies the result of comparing two corresponding bits in string-1 and string-2. Specify bit positions in the operation string from left to right to define the operation, as follows:

string-1-bit	string-2-bit	Result specified as
0	0	Bit 1 of operation string
0	1	Bit 2 of operation string
1	0	Bit 3 of operation string
1	1	Bit 4 of operation string

If string-1 and string-2 are of different lengths, the smaller is extended on the right with zeros to the length of the larger.

■ Example

```
X = '101010'B;
Y= '110011'B;
CHECK = BOOL (X,Y,'0110'B);
```

The operation is the exclusive OR. The result is '011001'B. Figure B–5 illustrates this example.

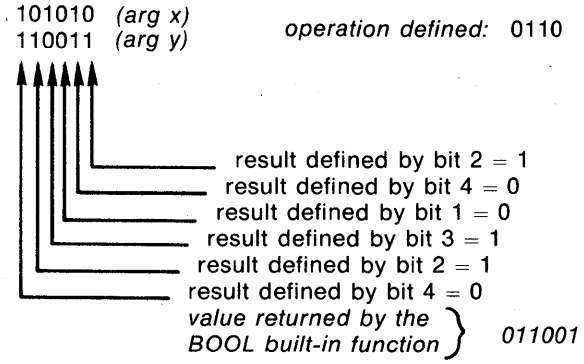


Figure B–5: Example of the BOOL Built-In Function

BUILTIN Attribute

The BUILTIN attribute indicates that the declared name is the name of a PL/I built-in function. Within the block in which the name is declared, all references to the name will be interpreted as references to the built-in function or pseudovariable of that name.

The BUILTIN attribute is used when you want to refer to a built-in function within a block in which the function's name has been used to declare a variable.

The BUILTIN attribute is also used when you want to invoke a built-in function that takes no arguments (such as the DATE function) and you do not want to include a null argument list.

■ Examples

```
OUTER: PROCEDURE;
DECLARE MAX FIXED BINARY STATIC INITIAL (10);
       .
       .
       INNER: PROCEDURE;
       DECLARE MAX BUILTIN;
              TEST = MAX(A,B);
                 .
                 .
       END INNER;
END OUTER;
```

The keyword MAX is used here as a variable name. In the internal procedure INNER, the MAX built-in function is invoked. Because the scope of the name MAX includes the internal procedure, the function must be redeclared with BUILTIN.

You can also use the BUILTIN attribute to declare PL/I built-in functions that have no arguments, if you want to invoke them without the empty argument list. For example:

```
DECLARE DATE BUILTIN;
PUT LIST(DATE);
```

Without the declaration, the PUT LIST statement would have to include an empty argument list for DATE:

```
PUT LIST(DATE());
```

■ Restrictions

When you specify the BUILTIN attribute, you cannot specify any other attributes.

Built-In Function

Built-in functions are procedures provided by the PL/I language. They can be used wherever an expression is valid.

■ Built-In Function Arguments

Built-in functions are similar to operators, and their arguments, to operands. Built-in function arguments, if arithmetic, are converted to their derived type before the function reference is evaluated. (**See also** "Expression — Conversion of Operands.") All evaluations of built-in functions are performed in the result type. The arguments are converted again from the derived type to the result type if necessary. The precision of the the result is the greater of the precisions of the two arguments.

For instance, all the mathematical functions listed in Table B–1 return floating-point values; their arguments are converted to floating point (binary or decimal, depending on the base of the argument) before the operation is performed.

■ Example

Like all mathematical functions, ATAN returns a floating-point result and is therefore computed in floating point. The base of the result is the same as the base of the converted arguments.

```
DCL J FIXED BINARY(8); FT = ATAN(J,2);
```

The derived type of J and 2 is fixed-point binary. The converted precision of 2 is min(ceil(1/3.32)+1,31), or 2. The result type is FLOAT BINARY(8). Both arguments are now converted to FLOAT BINARY(8), and the ATAN operation is performed.

Note also the following restrictions on built-in function arguments:

- If one argument of a function is fixed-point binary, no other argument should be fixed-point decimal with a nonzero scale factor or pictured with fractional digits.
- All arguments of all built-in functions except the array-handling, storage, file control, and STRING functions must be scalars of arithmetic, string, or pictured data types, as specified for the individual function.
- A reference to a built-in function that takes no arguments must still contain the pair of enclosing parentheses [example: NULL()] unless the function's name has been declared with the BUILTIN attribute.

■ Conditions Signaled

Built-in functions, like other operations, can signal conditions. The mathematical functions, which are computed in floating point, can signal OVERFLOW and UNDERFLOW under the appropriate conditions. Functions that are computed in fixed point can signal FIXEDOVERFLOW. In general, string and other functions signal ERROR if a result cannot be computed. **See also** the descriptions of individual conditions and built-in functions.

■ Summary

The built-in functions are summarized in Table B–1, according to the following functional categories:

- Arithmetic built-in functions — functions that provide information about the properties of arithmetic values, or that perform common arithmetic calculations
- Mathematical built-in functions — functions that perform standard mathematical calculations in floating point
- String-handling built-in functions — functions that process character-string and bit-string values
- Conversion built-in functions — functions that convert data from one data type to another
- Condition-handling built-in functions — functions that provide information about interrupts caused by signaled conditions
- Array-handling built-in functions — functions that provide information about arrays
- Storage control built-in functions — functions that return values concerning based variables
- Timekeeping built-in functions — functions that return the system date and time of day
- File control built-in functions — functions that return the current line number and page number of a file
- Miscellaneous — functions that check the validity of data, aid in argument passing, and perform other convenient operations

Table B–1: Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
Arithmetic	ABS(x)	Absolute value of x
	CEIL(x)	Smallest integer greater than or equal to x
	DIVIDE(x,y,p[,q])	Value of x divided by y, with precision p and scale factor q
	FLOOR(x)	Largest integer that is less than or equal to x
	MAX(x1,x2)	Larger of the values x1 and x2
	MIN(x1,x2)	Smaller of the values x1 and x2
	MOD(x,y)	Value of x mod y
	ROUND(x,k)	Value of x rounded to k digits
	SIGN(x)	-1, 0, or 1 to indicate the sign of x
	TRUNC(x)	Integer portion of x

(Continued on next page)

Table B-1 (Cont.): Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
Mathematical	ACOS(x)	Arc cosine of x (angle, in radians, whose cosine is x)
	ASIN(x)	Arc sine of x (angle, in radians, whose sine is x)
	ATAN(x)	Arc tangent of x (the angle, in radians, whose tangent is x)
	ATAN(x,y)	Arc tangent of x (the angle, in radians, whose sine is x and whose cosine is y)
	ATAND(x)	Arc tangent of x (the angle, in degrees, whose tangent is x)
	ATAND(x,y)	Arc tangent of x (the angle, in degrees, whose tangent is sine is x and whose cosine is y)
	ATANH(x)	Hyperbolic arc targent of x
	COS(x)	Cosine of radian angle x
	COSD(x)	Cosine of degree angle x
	COSH(x)	Hyperbolic cosine of x
	EXP(x)	Base of the natural logarithm, e, to the power x
	LOG(x)	Logarithm of x to the base e
	LOG10(x)	Logarithm of x to the base 10
	LOG2(x)	Logarithm of x to the base 2
	SIN(x)	Sine of the radian angle x
	SIND(x)	Sine of the degree angle x
	SINH(x)	Hyperbolic sine of x
	SQRT(x)	Square root of x
	TAN(x)	Tangent of the radian angle x
	TAND(x)	Tangent of the degree angle x
	TANH(x)	Hyperbolic tangent of x

(Continued on next page)

Table B–1 (Cont.): Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
String-Handling	BOOL(x,y,z)	Result of Boolean operation z performed on x and y
	COLLATE()	ASCII character set
	COPY(s,c)	c copies of specified string, s
	INDEX(s,c)	Position of the character string c within the string s
	LENGTH(s)	Number of characters or bits in the string s
	STRING(s)	Concatenation of values in array or structure s
	SUBSTR(s,i[,j])	Part of string s beginning at i for j characters
	TRANSLATE(s,c[,d])	String s with substitutions defined in c and d
	VERIFY(s,c)	Position of the first character in s which is not found in c
Conversion	BINARY(x[,p[,0]])	Binary value of x to the precision p and scale factor 0
	BIT(s[,l])	Value of s converted to a bit string of length l
	BYTE(x)	ASCII character represented by the integer x
	CHARACTER(s[,l])	Value of s converted to a character string of length l
	DECIMAL(x[,p[,q]])	Decimal value of x
	FIXED(x,p[,q])	Fixed arithmetic value of x
	FLOAT(x,p)	Floating arithmetic value of x
	RANK(c)	Integer representation of the ASCII character c
	UNSPEC(x)	Internal coded form of x
Condition-Handling	ONARGSLIST()	Pointer to argument lists of exception condition
	ONCODE()	Error code of the most recent run-time error
	ONFILE()	Name of file constant for which the most recent ENDFILE, ENDPAGE, KEY, or UNDEFINEDFILE condition was signaled
	ONKEY()	Value of key that caused KEY condition

(Continued on next page)

Table B-1 (Cont.): Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
Array-Handling	DIMENSION(x,n)	Extent of the nth dimension of x
	HBOUND(x,n)	Higher bound of the nth dimension of x
	LBOUND(x,n)	Lower bound of the nth dimension of x
Storage	ADDR(x)	Pointer identifying the storage referenced by x
	NULL()	A null pointer value
	OFFSET(p,a)	An offset into the location in area a pointed to by pointer p.
	POINTER(o,a)	A pointer to the location at offset o within area a.
Timekeeping	DATE()	System date in the form YYMMDD
	TIME()	System time of day in the form HHMMSSXX
File Control	LINENO(x)	Line number of the print file identified by x
	PAGENO(x)	Page number of the print file identified by x
Miscellaneous	DESCRIPTOR(x)	(The function forces its argument to be passed by descriptor to a non-PL/I procedure)
	VALID(p)	Boolean value, indicating whether the pictured variable p has a value consistent with its picture specification

BY Option

The BY option defines a value by which a control variable in a DO statement specification is modified. For example:

```
DO I = 10 BY 10 WHILE (X < Y);
```

The DO-group following this statement executes with values for I of 10, 20, and so on, until the specification in the WHILE option is no longer true. If no BY option is specified in a controlled DO statement, the default value of 1 is used. **See** "DO Statement."

BYTE Built-In Function

The BYTE built-in function returns the ASCII character whose ASCII code is the integer x; x must not be negative. The returned value is a character equivalent to BYTE(y), where y equals x mod 128. The format of the function is:

BYTE(x)

■ Example

```
DECLARE CHAR CHARACTER(1);
CHAR = BYTE(65);            /* CHAR = 'A' */
CHAR = BYTE(32);            /* CHAR = ' ' (space) */
```

CALL Statement

The CALL statement transfers control to an entry point of a procedure and optionally passes arguments to the procedure. The format of the CALL statement is:

CALL entry-name [(argument,...)] ;

entry-name

The name of an external or internal procedure that does not have the RETURNS attribute, or the name of an alternate entry point to a procedure. (The entry-name can also be an entry variable or a reference to a function that returns an entry value.)

argument,...

The argument list to be passed to the called procedure. If specified, the arguments must correspond to the parameters specified in the PROCEDURE or ENTRY statement that identifies the entry name of the called procedure.

Unless OPTIONS(VARIABLE) is specified in the declaration of an external entry name, the number of arguments must match the number of parameters in the parameter list of the invoked entry name. OPTIONS(VARIABLE) is valid only for use with non-PL/I procedures.

Arguments must be enclosed in parentheses. Multiple arguments must be separated by commas.

The CALL statement can be used to call an internal or external subroutine. The following example illustrates a main procedure, CALLER, and a call to an internal subroutine, PUT_OUTPUT. PUT_OUTPUT has two parameters, INSTRING and OUTFILE, that correspond to the arguments LINE and DEVICE specified in the CALL statement.

```
CALLER: PROCEDURE OPTIONS(MAIN);
  .
  .
        CALL PUT_OUTPUT(LINE,DEVICE);
  .
PUT_OUTPUT: PROCEDURE(INSTRING,OUTFILE);
  .
  .
END PUT_OUTPUT;
END CALLER;
```

For more information, **see** "Entry Data," "Parameters and Arguments," "Procedure," "Procedure Block," and "PROCEDURE Statement."

CEIL Built-In Function

The CEIL function returns the smallest integer that is greater than or equal to an arithmetic expression x. Its format is:

CEIL(x)

■ Returned Value

If x is a floating-point expression, a floating-point value is returned with the same precision as x. If x is a fixed-point expression, the returned value is a fixed-point value of the same base as x and with

precision = min(31,p–q+1)

scale factor = 0

where p and q are the precision and scale factor of x.

■ Examples

```
A = 4.3;
Y = CEIL(A);          /* Y = 5 */

A = -4.3;
Y = CEIL(A);          /* Y = -4 */
```

CHARACTER Attribute

The CHARACTER attribute identifies a variable as a character-string variable. The format of the CHARACTER attribute is:

{CHARACTER / CHAR} [(length)]

length

The number of characters in a fixed-length string or the maximum length of a varying-length string. If not specified, a length of 1 is assumed. The length must be in the range of 0 through 32767 characters.

The rules for specifying the length are as follows:

- If the attribute is specified for a static variable declaration or in a returns descriptor, length must be an integer constant.
- If the attribute is specified in the declaration of a parameter or in a parameter descriptor, length may be specified as an integer constant or as an asterisk (*).
- If the attribute is specified for an automatic, based, or defined variable, length may be specified as an integer constant or as an expression. In the case of automatic or defined variables, the expression must not contain any variables or functions that are declared in the same block, except for parameters.

If specified, the length must immediately follow the keyword CHARACTER, and it must be enclosed in parentheses.

If you give a variable the CHARACTER attribute, you can also specify the attribute VARYING.

■ Restrictions

The CHARACTER attribute directly conflicts with the BIT attribute and with any other data type attribute.

CHARACTER Built-In Function

The CHARACTER built-in function converts an arithmetic or string expression x to a character string of an optionally specified length. If the length is specified, it must be a nonnegative integer. If the length is omitted, the length of the returned value is determined by the usual rules for conversion to character strings (**see** "Conversion of Data"). The format of the function is:

CHARACTER(x[,length])

■ Examples

```
CHAR: PROCEDURE OPTIONS(MAIN);
DECLARE EXPRES FIXED DECIMAL(7,5);
DECLARE OUTPUT PRINT FILE;
EXPRES = 12.34567;
OPEN FILE(OUTPUT) TITLE('CHAR2.OUT');
PUT SKIP FILE(OUTPUT)
      LIST('No length argument: ',CHARACTER(EXPRES));
PUT SKIP FILE(OUTPUT)
      LIST('Length = 4: ',CHARACTER(EXPRES,4));
END CHAR;
```

The program CHAR produces the following output:

```
No length argument:        12.34567
Length = 4:        12
```

In the first PUT LIST statement, CHARACTER has only one argument, so the entire string is written out. The string ´12.34567´ is actually preceded by two spaces; that is the case with any nonnegative number converted to a character string (**see** "Conversion of Data"). In the second PUT LIST statement, CHARACTER has a length argument of 4, so the first four characters of the converted string are written out: ´△△12´.

Character-String Data

A character string is a sequence of ASCII characters (**see** "ASCII Character Set"). Every character-string variable has a length attribute that specifies either the length of all values of the variable (fixed-length strings) or the maximum length of a value of the variable (varying-length strings).

A character-string value can consist of any ASCII characters, to a maximum length of 32767 bytes.

This discussion of character-string data is divided into the following parts:

- Constants
- Variables
- Varying character strings
- Alignment of character strings
- Internal representation

■ Character-String Constants

When you use character-string constants in a program, you must enclose the character strings in apostrophes, as shown in the following examples:

```
'Total is:'
'Enter your name and age'
'Error - value is out of range'
```

To specify a string containing a literal apostrophe, use two apostrophes within the string, for example:

```
'Life isn''t fair'
```

When a character string that has embedded apostrophes is specified as shown above, the final result contains only a single apostrophe.

■ Character-String Variables

The CHARACTER keyword identifies a variable as a character-string variable in a declaration. The format for specifying a fixed-length character-string variable is:

DECLARE variable-name CHARACTER [(n)];

where n is the length of the variable, that is, the number of bytes needed to contain the value of the variable. If not specified, PL/I uses the default length of one character, or byte.

Fixed-Length Character-String Variables

When a program assigns a value to a fixed-length character-string variable, the value is not always exactly the same as the length defined for the variable. Depending on the size of the value, PL/I does the following:

- If the value is smaller than the length of the character string, PL/I pads the value with spaces on the right. For example:

```
DECLARE STRING CHARACTER (10);
STRING = 'ABCDEF';
```

  The final value of the variable STRING in the above example is 'ABCDEF△△△△', that is, the characters ABCDEF followed by four space characters.

- If the value is larger than the length of the variable, PL/I truncates the value on the right. For example:

```
DECLARE STRING CHARACTER (4);
STRING = 'ABCDEF';
```

  The final value of the variable STRING in this example is 'ABCD', that is, the first four characters of the value 'ABCDEF'.

Initializing Character-String Variables

You can use the INITIAL attribute to supply an initial value for the variable. For example:

```
DECLARE MESSAGE CHARACTER (20)
   INITIAL('Begin entering text');
```

If the initial value for a variable is longer than the length, the value is truncated. If the initial value is shorter than the specified length, it is padded with spaces on the right.

■ Varying Character Strings

When you define a character-string variable, you can also specify the VARYING attribute. A varying character-string variable is a variable whose length is not fixed. The length specified in the declaration of the variable defines the maximum length of any value that can be assigned to the variable. Each time a value is assigned to the variable, the current length changes. For example:

```
DECLARE NAME CHARACTER (20) VARYING;
NAME = 'COOPER';
NAME = 'RANDOM FACTOR';
```

The declaration of the variable NAME indicates that the maximum length of any character-string value it can have is 20. Although the maximum length of NAME is 20, the current length becomes 6 when NAME is assigned the value COOPER; the length becomes 13 when NAME is assigned the value RANDOM FACTOR; and so on.

When a varying character string is assigned a value with a length greater than the maximum defined, the value is truncated on the right.

The initial length of an automatic varying-length character-string variable is undefined. A static variable is initially a null string with a length of zero.

You can use the LENGTH built-in function to determine the current length of any string. **See** "LENGTH Built-In Function."

■ Alignment of Character Strings

The PL/I language makes a distinction between aligned and unaligned (fixed-length) character-string variables. (No such distinction is made for varying character strings or for character-string constants.) A character-string variable is aligned if it is declared with the ALIGNED attribute.

In VAX-11 PL/I, this distinction affects only argument passing. If a procedure declares a parameter as ALIGNED CHARACTER, and if the corresponding argument is an unaligned character-string variable or vice versa, the actual argument will be a dummy variable. For example:

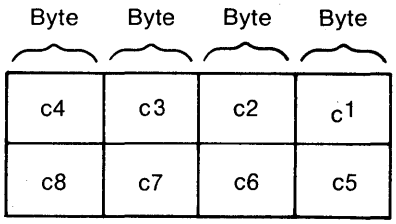
```
DECLARE GETSTRING ENTRY (CHARACTER (*) ALIGNED);
DECLARE STRING CHARACTER (8);
CALL GETSTRING (STRING);
```

PL/I constructs a dummy variable here to pass the unaligned string variable STRING to the called procedure GETSTRING, rather than passing the actual argument by reference. (**See** "Argument Passing.")

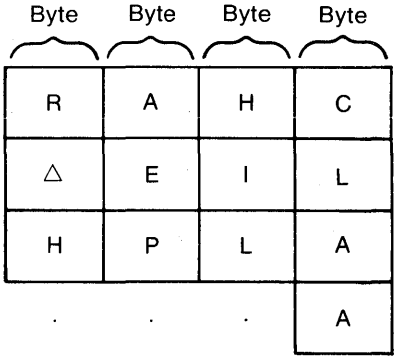
Note that all character strings on the VAX-11 hardware are aligned on byte boundaries. Thus it is recommended that you do not use the ALIGNED attribute to declare character-string variables.

■ Internal Representation of Character Data

PL/I stores fixed-length character-string data from right to left, with each character occupying a byte of storage, as shown here:



For example, a character string whose value is 'CHARLIE△ALPHA' appears as follows in storage:



Varying-length strings are stored in a number of bytes equal to n+2, where n is the declared maximum length. The two additional bytes contain, in the first two byte addresses, the current length of the value in bytes.

CLOSE Statement

The CLOSE statement dissociates a PL/I file from the physical file with which it was associated when it was opened. The format of the CLOSE statement is:

CLOSE FILE(file-reference) [ENVIRONMENT(option,...)] ;

file-reference

The file to be closed. If the file is already closed, the CLOSE statement has no effect.

ENVIRONMENT(option,...)

One or more of the ENVIRONMENT options listed below, separated by commas.

BATCH
DELETE
REWIND_ON_CLOSE
SPOOL
TRUNCATE

No other ENVIRONMENT options are valid. All ENVIRONMENT options are described in detail in the *VAX-11 PL/I User's Guide*.

■ Examples

```
CLOSE FILE(INFILE);
```

This CLOSE statement closes the file constant INFILE.

```
DECLARE STATE_FILE FILE KEYED;

OPEN FILE(STATE_FILE) DIRECT UPDATE;
   .
   .
CLOSE FILE(STATE_FILE);
OPEN FILE(STATE_FILE) INPUT SEQUENTIAL;
```

The file STATE_FILE is declared with the KEYED attribute. On the first OPEN statement that specifies this file, it is given the DIRECT and UPDATE attributes and opened for updating; that is, it can only be accessed by key.

The CLOSE statement closes the file and the second OPEN statement specifies the INPUT and SEQUENTIAL attributes; the file may now be accessed sequentially.

COLLATE Built-In Function

The COLLATE built-in function returns a 256-character string consisting of the ASCII character set in ascending order. Its format is:

COLLATE()

COLUMN Format Item

The COLUMN format item sets a stream file to a specific character position within a line. Effectively, COLUMN determines the position at which the next data will be output or from which the next data will be input. The COLUMN format item refers to an absolute character position in a line; to refer to a relative position, **see** "X Format Item."

The form of the COLUMN format item is:

$$\left\{\begin{matrix}\text{COLUMN}\\ \text{COL}\end{matrix}\right\}(w)$$

w

An integer that identifies the wth position from the beginning of the current line. The value of w must be zero or positive. If w is zero, a value of one is assumed.

If the file is already at the specified position, no operation is performed. If the file is already beyond the specified position, the format item is applied to the next line.

The interpretation of the COLUMN format item on input and output is given below. For a general discussion of format items, **see** "Format Items and Their Uses."

■ Input with GET EDIT

The file is positioned at the column specified by w. Characters between the beginning of the line and this column are ignored. If the file is already positioned beyond the specified column, the remainder of the line is skipped and the format item is applied to the next line.

■ Output with PUT EDIT

The file is positioned at the column specified by w. Within the current line, positions between the wth column and the position of the last output data are filled with spaces.

If the file is already positioned beyond the specified column, the format item is applied to the next line. If w exceeds the line size, a value of one is assumed. **See also** "LINESIZE Option."

■ Examples

```
COL: PROCEDURE OPTIONS(MAIN);

DECLARE IN STREAM INPUT FILE;
DECLARE OUT STREAM OUTPUT FILE;
DECLARE LETTER CHARACTER(1);

PUT FILE(OUT) SKIP
      EDIT('12345678901234567890123456789O') (A);
PUT FILE(OUT) SKIP
      EDIT('COL1','COL28') (A,COL(28),A);

GET FILE(IN) EDIT (LETTER) (A(1));
PUT FILE(OUT) SKIP(2)
      LIST('Letter in column 1:',LETTER);

GET FILE(IN)
      EDIT (LETTER) (COL(25),A(1));
PUT FILE(OUT) SKIP
      LIST ('Letter in column 25:',LETTER);

END COL;
```

If the stream input file IN.DAT contains the text:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

then the program COL writes the following output to the stream output file OUT.DAT:

```
12345678901234567890123456789O
COL1                          COL28

'Letter in column 1:' 'A'
'Letter in column 25:' 'Y'
```

Comment

A comment is an informational tool for documenting a PL/I program. To insert a comment in a program, enclose the comment within the character pairs /* and */. For example:

```
/* This is a comment.... */
```

Wherever the characters /* appear in a program, the compiler ignores all text until it encounters the characters */. Thus, a comment can span several lines.

The rules for entering comments are:

- A comment can appear anywhere that a space can appear, that is:
  - Between any identifiers, keywords, or constants. In this context, a comment separates tokens, or discrete text items, in a statement.
  - Preceding or following any other punctuation marks that normally delimit tokens, for example, spaces, tabs, or commas.
- A comment can contain any character except the pair */; comments cannot be nested.

Some examples of comments are:

```
A = B + C ;              /* Add B and C */

/* ********* START OF SECOND PHASE ********* */

DECLARE/*COUNTER*/A FIXED BINARY (7);

/* This module performs the following steps:
   1. Initializes all arrays and data structures.
   2. Establishes default condition handlers.
*/
```

Although complete comments cannot be nested, you can "comment out" a statement such as

```
DECLARE EOF BIT(1); /* end-of-file */
```

This statement can be commented out by preceding the DECLARE with another /* pair. The compiler will then ignore all text, including the DECLARE statement, until it reaches the */ pair.

Comparison Operator

See "Relational Operator."

Concatenation Operator

The concatenation operator produces a single string from two strings specified as operands. The concatenation operator is:

|| or !!

Both operands must be character strings, or else both must be bit strings. The result of the operation is a string of the same type as the operands.

■ Examples

```
CONCAT: PROCEDURE OPTIONS(MAIN);

DECLARE OUTFILE STREAM OUTPUT PRINT FILE;

PUT FILE(OUTFILE) SKIP LIST('ABC'||'DEF');

PUT FILE(OUTFILE) SKIP LIST('001'B||'110'B);

PUT FILE(OUTFILE) SKIP LIST('001'B||'07'B3);

END CONCAT;
```

The program CONCAT writes the following output to the file OUTFILE.DAT:

```
ABCDEF
'001110'B
'001000111'B
```

Condition Handling

A PL/I condition is any occurrence that causes the interruption of a program and a signal. When a condition is signaled, PL/I initiates a search for a user-written program unit called an ON-unit, to handle the condition. **See** "ON Conditions and ON-Units."

Constant

A constant is a data item whose value cannot change during the execution of a PL/I program. The converse of a constant is a variable, that is, a data item to which various values can be assigned during the execution of a program.

VAX-11 PL/I allows the following kinds of constants:

- Literal constants, which are actual numbers and strings written in the source program. Literal constant types are restricted to character strings, bit strings, and fixed- or floating-point decimal numbers. Unscaled fixed decimal numbers can be written with or without a decimal point. Arithmetic constants can be signed.

- Label constants, which are established by using a label in the source program. (Label constants cannot be declared in a DECLARE statement.)
- Declared constants (file and entry constants), which generally are established by DECLARE statements. (The default file constants SYSIN and SYSPRINT need not be declared.)
- Constant identifiers, which are identifiers assigned literal constant values with the %REPLACE statement. Constant identifiers are restricted to the same types as literal constants. **See** "%REPLACE Statement."

PL/I also has the computational types FIXED BINARY, FLOAT BINARY, and PICTURE, but there are no literal constants nor constant identifiers associated with these types. Binary variables usually are assigned values by assigning decimal constants or other binary variables to them and allowing PL/I to convert the assigned value to binary. Pictured variables are usually assigned values by assigning fixed-point decimal constants to them. For further details, **see** "Conversion of Data."

■ Examples

```
445              /* a fixed-point decimal constant */
-445.            /* a fixed-point decimal constant */
16.2             /* a fixed-point decimal constant */
129E-3           /* floating-point decimal constant */

'00101111'B      /* a bit-string constant */
'This is a string'  /* a character-string constant */

DECLARE E ENTRY;      /* an entry constant */
DECLARE F FILE;       /* a file constant */

STARTUP:             /* a label constant */

%REPLACE PI BY 3.14159       /* a fixed-point decimal
                                constant identifier */

STATUS = 25;           /* assignment of a fixed
                          decimal constant to a
                          variable */

C = 3E10;              /* assignment of a floating
                          decimal constant to a
                          variable */
```

Conversion of Data

Conversion is the changing of a data item from one data type to another. This entry describes the conversions performed in assignments. Conversions are also performed on operands in arithmetic expressions; **see** "Expression" for details of operand conversions.

In assignments, conversions are defined between the noncomputational types POINTER and OFFSET, and between any two computational types. The rules for assignments apply to:

- Assignment statements
- Arguments passed to a procedure
- Values specified in a RETURN statement
- An argument converted by the built-in function FIXED, FLOAT, BINARY, DECIMAL, BIT, or CHARACTER
- Conversions to and from character strings performed by the PUT and GET statements, respectively

If an attempt is made to assign a value to a target for which there is no defined conversion, the compiler generates a diagnostic message. For example, if F is a variable with the attributes FIXED DECIMAL (5,2), then the statement

```
F = '133.45';
```

assigns the numeric value 133.45 to F, as expected. However,

```
F = 'ABCD';
```

signals the ERROR condition.

Table C–1 illustrates the contexts in which PL/I performs conversions. Table C–1 also lists the built–in conversion functions, such as BINARY and CHARACTER. You can use these functions when you want to explicitly indicate a conversion and to specify such characteristics as the precision or string length of the converted result.

The rest of this section defines the rules and results of the following types of conversion:

- Assignments to arithmetic variables
  - from any arithmetic data type to any other arithmetic data type
  - from pictured to any arithmetic type
  - from a bit string to any arithmetic data type
  - from a character string to any arithmetic data type
- Assignments to bit-string variables
  - from any arithmetic data type to a bit string
  - from pictured to bit string
  - from a character string to a bit string
- Assignments to character-string variables
  - from any arithmetic data type to a character string
  - from pictured to character string
  - from a bit string to a character string
- Assignments to pictured variables
  - from any computational type to pictured
- Conversions between offsets and pointers

Table C–1: Contexts in Which PL/I Converts Data

Context	Conversion Performed
target = expression ;	In an assignment statement, the given expression is converted to the data type of the target.
entry-name RETURNS (attribute...) ; . . RETURN (value) ;	In a RETURN statement, the specified value is converted to the data type specified by the RETURNS option on the PROCEDURE or ENTRY statement.
x + y x - y x * y x / y x**y	In any arithmetic expression, if operands do not have the same data type, they are converted to a common data type before the operation. (See "Expression.")
BINARY (expression) BIT (expression) CHARACTER (expression) FIXED (expression) FLOAT (expression) OFFSET (variable) POINTER (variable)	PL/I provides built-in functions that perform specific conversions.
PUT LIST (item,...) ;	Items in a PUT LIST statement are converted to character-string data.
GET LIST (item,...) ;	Character-string input data is converted to the data type of the target item.
PAGESIZE (expression) LINESIZE (expression) DO control-variable ... SKIP (expression) LINE (expression)	Values specified for various options to PL/I statements must be converted to integer values.

Assignments to Arithmetic Variables

Expressions of any computational type can be assigned to arithmetic variables. The conversion rules are described below for each source type.

■ Arithmetic to Arithmetic Conversions

A source expression of any arithmetic type can be assigned to a target variable of any arithmetic type. Note the following qualifications:

- If the target is a variable of type FIXED BINARY or FIXED DECIMAL, then the FIXEDOVERFLOW condition is signaled when the source value has a larger number of integral digits than are specified in the precision of the target. If the target is a fixed-point binary variable, FIXEDOVERFLOW is signaled if the source value exceeds the storage allocated for the target, which may be larger than the target's declared precision (**see** "Fixed-Point Binary Data").

- If the target is a variable of type FIXED DECIMAL(p,q) and the source value has more than q fractional digits, then the excess fractional digits of the source are truncated, and no condition is signaled. If the source has fewer than q fractional digits, the source value is padded on the right with zeros.
- If the target value is floating point and the absolute source value is too large to be represented by a VAX floating-point type (**see** "Floating-Point Data"), then the OVERFLOW condition is signaled, and the value of the target is undefined. If the absolute source value is too small to be represented, the value zero is assigned to the target, and, if enabled, the UNDERFLOW condition is signaled.

Conversions to Fixed Point

In the following examples, the specified source values are converted to FIXED DECIMAL(4,1):

Source Value	Converted Value
25.505	25.5
-2.562	-2.5
101	101.0
5365	FIXEDOVERFLOW - value undefined

Conversions to Floating Point

Let p be the precision of the floating-point target. If the source value is an integer that can be represented exactly in p digits, then the source value is converted to floating-point binary with no loss of accuracy.

Otherwise, the source value is converted to floating-point binary with rounding to precision p. For example, the constant 479 will be converted to FLOAT BINARY(24) without loss of accuracy, while the constant 16777217, which cannot be represented exactly in 24 bits, will be rounded during conversion.

■ Pictured to Arithmetic Conversions

In VAX-11 PL/I all pictured values have the associated attributes FIXED DECIMAL(p,q), where p is the total number of characters in the picture specification that specify decimal digits, and q is the total number of these digits that occur to the right of the V character. If the picture specification does not include a V character, then q is zero. This associated fixed-point decimal value is assigned to the target, following the usual rules for arithmetic to arithmetic conversion.

■ Bit String to Arithmetic Conversions

When a bit-string value is assigned to an arithmetic variable, PL/I treats the bit string as a fixed-point binary value. A string of type BIT(n) is converted to FIXED BINARY(m), where

$$m = \min(n,31)$$

If the converted value is greater than or equal to 2^{31}, then FIXEDOVERFLOW is signaled. The leftmost bit in the bit string (as output by PUT LIST) is the most significant bit in the fixed-point binary value, not its sign. If the bit string is null, the fixed-point binary value is zero.

The intermediate fixed-point binary value is then converted to the target arithmetic type.

Note that bit strings are stored internally with the leftmost bit in the lowest address. The conversion to an arithmetic type must reverse the bits from this representation and should therefore be avoided when performance is a consideration.

■ Examples

```
CONVTB: PROCEDURE OPTIONS(MAIN);

DECLARE STATUS FIXED BINARY(8);
DECLARE STATUS_D FIXED DECIMAL(10);
DECLARE OUT PRINT FILE;

OPEN FILE(OUT) TITLE('CONVTB.OUT');
ON FIXEDOVERFLOW PUT SKIP FILE(OUT)
      LIST('Fixedoverflow:');

STATUS = '1001101'B;
PUT SKIP FILE(OUT) LIST(STATUS);

STATUS_D = '001101'B;
PUT SKIP FILE(OUT) LIST(STATUS_D);

STATUS = '1232'B2;
PUT SKIP FILE(OUT) LIST(STATUS);

STATUS = 'FF'B4;
PUT SKIP FILE(OUT) LIST(STATUS);

STATUS_D = '10111111111111111111111111111111'B;
PUT SKIP FILE(OUT) LIST(STATUS_D);

END CONVTB;
```

The program CONVTB produces the output:

```
      77
              13
     110
     255
Fixedoverflow:
              13
```

Notice that the leftmost bit of all the bit-string constants is treated as the most significant numeric bit, not as a sign. For instance, the hexadecimal constant 'FF'B4 is converted to 255 instead of -127. The last assignment to STATUS_D signals the FIXEDOVERFLOW condition, because the bit-string constant, when represented as a binary integer, is greater than 2^{31}. The resulting value of STATUS_D is undefined.

■ Character String to Arithmetic Conversions

When a character string is assigned to an arithmetic value, PL/I creates an intermediate numeric value based on the characters in the string. The type of

this intermediate value is the same as that of an ordinary arithmetic constant comprising the same characters; that is, 342.122E-12 and '342.122E-12' are both floating-point decimal.

The character string can contain any series of characters that describes a valid arithmetic constant. That is, the character string can contain any of the numeric digits 0 through 9, optionally preceded by a plus (+) or minus (-) sign and optionally containing a decimal point. It can also contain the letter E followed by a signed exponent, and it can contain a single decimal point (.). If the character string contains any invalid characters, the ERROR condition is signaled. **See** "Examples," below.

If the implied data type of the character string does not match the data type of the arithmetic target, PL/I converts the intermediate value to the data type of the target, following the rules for arithmetic to arithmetic conversions. In conversions to fixed point, FIXEDOVERFLOW is signaled if the character string specifies too many integral digits. Excess fractional digits are truncated without signaling a condition.

If the source character string is null or contains all spaces, the resulting arithmetic value is zero.

■ Examples

```
DECLARE SPEED FIXED DECIMAL (9,4);

SPEED = '23344.3882';
      /* string converted to 23344.3882 */

SPEED = '32423.23SD';
      /* ERROR condition */

SPEED = '4324324.3933';
      /* FIXEDOVERFLOW condition */

SPEED = '1.33336';
      /* string converted to 1.3333 */
```

Assignments to Bit-String Variables

In the conversion of any data type to a bit string, PL/I first converts the source data item to an intermediate bit-string value. Then, based on the length of the target string, it performs the following:

- If the length of the target bit-string value is greater than the length of the intermediate string, the target bit string (as represented by PUT LIST) is padded with zeros on the right.
- If the length of the target bit-string value is less than the length of the intermediate string, the intermediate bit string (as represented by PUT LIST) is truncated on the right.

The next sections describe how PL/I arrives at the intermediate bit-string value for each data type.

■ Arithmetic to Bit String Assignments

In converting an arithmetic value sv to a bit-string value, PL/I performs the following steps:

1. Let v = abs(sv)
2. Determine a precision p as follows:

Source	Precision p
FIXED BINARY(r)	min(31,r)
FLOAT BINARY(r)	min(31,r)
FIXED DECIMAL(r,s)	min(31,ceil((r–s)*3.32))
FLOAT DECIMAL(r)	min(31,ceil(r*3.32))

3. If p=0 (for example, when r=s), the intermediate string is a null bit string. Otherwise, the value v is converted to an integer n of type FIXED BINARY(p). If $n \geq 2^p$, the FIXEDOVERFLOW condition is signaled; otherwise, the intermediate bit string is of length p, and each of its bits represents a binary digit of n.

Note that bit strings are stored internally with the leftmost bit in the lowest address. The conversion must reverse the bits from this representation and should therefore be avoided when performance is a consideration. Note also that during the conversion, the sign of the arithmetic value and any fractional digits are lost.

■ Examples

```
CONVB: PROCEDURE OPTIONS(MAIN);

DECLARE NEW_STRING BIT(10);
DECLARE LONGSTRING BIT(16);
DECLARE OUT PRINT FILE;

OPEN FILE(OUT) TITLE('CONVB1.OUT');

NEW_STRING = 35;
PUT FILE(OUT) SKIP
      LIST('35 converted to BIT(10):',NEW_STRING);

NEW_STRING = -35;
PUT FILE(OUT) SKIP
      LIST('-35 converted to BIT(10):',NEW_STRING);

NEW_STRING = 23.12;
PUT FILE(OUT) SKIP
      LIST('23.12 converted to BIT(10):',NEW_STRING);

NEW_STRING = .2312;
PUT FILE(OUT) SKIP
      LIST('.2312 converted to BIT(10):',NEW_STRING);
```

```
NEW_STRING = 8001;
PUT FILE(OUT) SKIP
      LIST('8001 converted to BIT(10):',NEW_STRING);

LONGSTRING = 8001;
PUT FILE(OUT) SKIP
      LIST('8001 converted to BIT(16):',LONGSTRING);

END CONVB;
```

The program CONVB produces the output:

```
35 converted to BIT(10):          '0100011000'B
-35 converted to BIT(10):         '0100011000'B
23.12 converted to BIT(10):       '0010111000'B
.2312 converted to BIT(10):       '0000000000'B
8001 converted to BIT(10):        '0111110100'B
8001 converted to BIT(16):        '0111110100000100'B
```

Note that 35 and -35 produce the same bit string, since the sign is lost in the conversion. In the first assignment, the constant 35 [type: FIXED DECIMAL(2,0)] is converted to FIXED BINARY(7) and then to a seven-bit string ('0100011'B). Three additional bits are appended to this intermediate bit string when it is assigned to NEW_STRING. Notice also that the low-order bit of 8001 is lost when the constant is assigned to a BIT(10) variable.

■ Pictured to Bit-String Conversions

If the source value is pictured, its associated fixed-point decimal value is extracted. The fixed-point decimal value is then converted to a bit string, following the previous rules for arithmetic to bit-string conversion.

■ Character-String to Bit-String Conversions

PL/I can convert a character string of 0s and 1s to a bit string. Any character in the character string other than 0 or 1, including spaces, will signal the ERROR condition.

PL/I converts each 0 or 1 character in the character string to a 0 or a 1 bit in the corresponding position (as represented by PUT LIST) in the intermediate bit string.

If the source is a null character string, the intermediate string is a null bit string.

■ Examples

```
DECLARE NEW_STRING BIT(4);

NEW_STRING = '0010';
      /* NEW_STRING = '0010'B */

NEW_STRING = '11';
      /* NEW_STRING = '1100'B */

NEW_STRING = 'AS110';
      /* ERROR condition */
```

Assignments to Character-String Variables

In the conversion of any data type to a character string, PL/I first converts the source value to an intermediate character-string value. Then it performs one of the following:

- If the length of the intermediate string is zero, a null string is assigned to the target.
- If the target is a parameter or return value with an asterisk extent (as in RETURNS CHAR(*)), the intermediate string is assigned to the target.
- If the target is of type CHARACTER, and the intermediate string is shorter than the maximum length of the target, the target is assigned the value of the intermediate string without trailing spaces if the target has the VARYING attribute. If the target does not have the VARYING attribute, the string is padded with trailing spaces.
- If the maximum length of the target character string is less than the length of the intermediate string, the intermediate string is truncated.

The next sections describe how PL/I arrives at the intermediate string for conversion of each data type. Examples at the end of each section illustrate the intermediate value, as well as the resulting value.

■ Arithmetic to Character-String Conversions

The manner in which PL/I converts the arithmetic item depends on the data type of the source, as described below.

Conversion from Fixed-Point Binary or Decimal

If the source value is of type FIXED BINARY(p1), PL/I first converts it to type FIXED DECIMAL(p2,0), where p2 is given by

$$p2 = \min(\text{ceil}(p1/3.32)+1,31)$$

PL/I converts a value with attributes FIXED DECIMAL(p,q) to an intermediate string of length p+3. The numeric value is right justified in the string. If the value is negative, a minus sign immediately precedes the value. If q is greater than zero, the value contains a decimal point followed by q digits. When p equals q, a 0 character precedes the decimal point. When q equals zero, a value of zero is represented by the zero character.

Alternately, the format of the intermediate string can be described by picture specifications, as follows:

1. If q=0, the intermediate string is the string created by the picture specification:

   'BB(p)-9'

   That is, the first two characters of the string are spaces. The last p characters in the string are the digit characters representing the integer; leading zeros are replaced by spaces except in the last position. If the integer is negative, a minus sign immediately precedes the first digit; if the number is not negative, this position contains a space. At least one digit always appears, in the last position in the string.

2. If p=q, the intermediate string is the string created by the picture specification:

   '-9V.(q)9'

   That is, the first three characters are (in order) an optional minus sign if the fraction is negative, the digit 0, and a decimal point. If the number is not negative, the first character is a space. The last q characters in the string are the fractional digits of the number.

3. If p>q, the intermediate string is the string created by the picture specification:

   'B(p-q)-9V.(q)9'

   That is, the first character is always a space; the last q characters are the fractional digits of the number and are preceded by a decimal point; the decimal point is always preceded by at least one digit, which may be 0; all integral digits appear before the decimal point, and leading zeros are replaced by spaces; a minus sign precedes the first integral digit if the number is negative; if the number is not negative then the minus sign is replaced by a space.

■ Examples

```
DECLARE STRING_1 CHARACTER (8),
        STRING_2 CHARACTER (4);

STRING_1 = 283472.;
        /* intermediate string = '△△△283472',
STRING_1 = '△△△28347' */

STRING_2 = 283472.;
        /* intermediate string = '△△△283472',
        STRING_2 = '△△△2' */

STRING_2 = -283472.;
        /* intermediate string = '△△-283472',
        STRING_2 = '△△-2' */

STRING_2 = -.003344;
        /* intermediate string = '-0.003344',
        STRING_2 = '-0.0' */

STRING_2 = -283.472;
        /* intermediate string = '△-283.472',
        STRING_2 = '△-28' */

STRING_2 = 283.472;
        /* intermediate string = '△△283.472',
        STRING_2 = '△△28' */
```

Conversion from Floating-Point Binary or Decimal

If the source value is of type FLOAT BINARY(p1), it is converted to FLOAT DECIMAL(p2), where p2 is given by

p2 = min(ceil(p1/3.32),34)

For a value of type FLOAT DECIMAL(p), where p is less than or equal to 34, the intermediate string is of length p+6; this allows extra characters for the sign of the number, the decimal point, the letter E, the sign of the exponent, and the two-digit exponent.

NOTE

If the value is a floating-point number of the VAX type G-float, three characters are allocated to the exponent, and the length of the string is p+7. If the value is of type H-float, four characters are allocated to the exponent, and the length of the string is p+8. (**See** "Floating-Point Data.")

If the number is negative, the first character is a minus sign; otherwise, the first character is a space. The subsequent characters are a single digit (which may be 0), a decimal point, p–1 fractional digits, the letter E, the sign of the exponent (always + or –), and the exponent digits. The exponent field is of fixed length, and the zero exponent is shown as all zeros in the exponent field.

■ Examples

```
CONCH: PROCEDURE OPTIONS(MAIN);

DECLARE OUT PRINT FILE;

OPEN FILE(OUT) TITLE('CONCH.OUT');

PUT SKIP FILE(OUT) EDIT('''',25E25,'''') (A);
PUT SKIP FILE(OUT) EDIT('''',-25E25,'''') (A);
PUT SKIP FILE(OUT) EDIT('''',1.233325E-5,'''') (A);
PUT SKIP FILE(OUT) EDIT('''',-1.233325E-5,'''') (A);

END CONCH;
```

The program CONCH produces the output:

```
' 2.5E+26'
'-2.5E+26'
' 1.233325E-05'
'-1.233325E-05'
```

The PUT statement converts its output sources to character strings, following the rules described in this section. (The output strings were surrounded with apostrophes to make the spaces distinguishable.) Note that, in each case, the width of the quoted output field (that is, the length of the converted character string) is the precision of the floating-point constant plus 6.

■ Pictured to Character-String Conversion

If the source value is pictured, its internal, character-string representation is used without conversion as the intermediate character string.

■ Bit String to Character String Conversion

When PL/I converts a bit string to a character string, it converts each bit in the bit string (as represented by PUT LIST) to a 0 or 1 character in the corresponding position of the intermediate character string.

If the bit string is a null string, the intermediate character string is also a null string.

■ Examples

```
DECLARE STRING_1 CHARACTER (4),
        STRING_2 CHARACTER (8);

STRING_1 = '1010'B;
        /* STRING_1 = '1010' */

STRING_2 = '1010'B;
        /* STRING_2 = '1010△△△△' */

STRING_1 = '010011'B;
        /* STRING_1 = '0100' */
```

Assignments to Pictured Variables

A source expression of any computational type can be assigned to a pictured variable. The target pictured variable has a precision (p), which is defined as the number of characters in its picture specification that specify decimal digits. It also has a scale factor (q), which is defined as the number of picture characters that specify digits and occur to the right of the V character in the picture specification. If the picture specification contains no V character, or if all digit-specification characters are to the left of V, then q is zero.

The source expression is converted to a fixed-point decimal value v of precision (p,q), following the usual rules for the source data type. This value is then edited to a character string s, as specified by the picture specification (**see also** "Picture"), and the value s is assigned to the pictured target.

When the value v is being edited to the string s, the ERROR condition is signaled if the value of v is less than zero and the picture specification does not contain one of the characters S, +, –, T, I, R, CR, or DB. The value of s is then undefined. FIXEDOVERFLOW is also signaled if the value v has more integral digits than are specified by the picture specification of the target.

Conversions Between Offsets and Pointers

Offset variables are given values by assignment from existing offset values or from conversion of pointer values. Pointer variables are given values by assignment from existing pointer values or from conversion of offset values.

The OFFSET built-in function converts a pointer value to an offset value. The POINTER built-in function converts an offset value to a pointer.

PL/I also automatically converts a pointer value to an offset value, and vice versa, in an assignment statement. The following assignments are valid:

1. pointer-variable = pointer-value ;
2. offset-variable = offset-value ;
3. pointer-variable = offset-variable ;
4. offset-variable = pointer-value ;

In (3) and (4), above, the offset variable must have been declared with an area reference. **See also** "Offset," "OFFSET Built-In Function," "Pointer," and "POINTER Built-In Function."

COPY Built-In Function

The COPY built-in function copies a given string a specified number of times and concatenates the result into a single string. Its format is:

COPY(string,count)

string

Any bit- or character-string expression. If the expression is a bit string, the result is a bit string. Otherwise, the result is a character string.

count

Any expression that yields a nonnegative integer. The specified count controls the number of copies of the string that are concatenated, as follows:

Value of Count	String Returned
0	a null string
1	the string argument
n	concatenated copies of the string argument

■ Example

The function reference

```
COPY('12',3)
```

returns the character-string value '121212'.

COS Built-In Function

The COS function returns a floating-point value that is the cosine of an arithmetic expression x, where x represents an angle in radians. The cosine is computed in floating point. The format of the function is:

COS(x)

COSD Built-In Function

The COSD built-in function returns a floating-point value that is the the cosine of an arithmetic expression x, where x is an angle in degrees. The cosine is computed in floating point. The format of the function is:

COSD(x)

COSH Built-In Function

The COSH built-in function returns a floating-point value that is the hyperbolic cosine of an arithmetic expression x. The hyperbolic cosine is computed in floating point. The format of the function is:

COSH(x)

D

Data and Data Types

All programs process information, or data. The way you choose to represent different items of data in a program depends on how the program will use or manipulate the data.

The data type of a variable or a constant reflects the kind of information that is being processed. For example, names and addresses within a personnel record are character-string data; weekly salaries and taxes and cumulative totals of salaries and taxes are arithmetic data.

Variables that represent single elements or items of data are called scalar variables. Variables can also be grouped into aggregates. There are two types of aggregate:

- An array is an aggregate in which all items, called elements, have the same data type. Individual elements of an array are referred to by position, or order, in the array by using subscripts. Elements can be scalar data items or structures. (**See** "Array.")
- A structure is an aggregate in which individual items, called members, can have different data types. Individual members are referred to with qualified references that give, in general, the names of the structure itself and of the individual member. (**See** "Structure.")

Aggregates can also be formed from arrays whose elements are structures, or from structures whose individual members are arrays.

■ Summary of Data Types

VAX-11 PL/I supports the following data types that are used in computations and, therefore, called "computational" data types.

- The arithmetic data types define values that can be used in arithmetic computation. These data types are:

  fixed-point binary (for integers)
  fixed-point decimal (for decimal integers and fractions)
  floating-point (binary and decimal)

  See "Fixed-Point Binary Data," "Fixed-Point Decimal Data," and "Floating-Point Data."

- Picture data represents fixed-point decimal values that are stored as character strings; the strings contain the characters representing the numeric value, formatted with special symbols. In computations and other expressions, a data item of this type (that is, a "pictured value") can be used wherever an arithmetic value is valid.

  See "Picture."

- Character-string data consists of a sequence of ASCII characters. VAX-11 PL/I supports:

  fixed-length character strings
  variable-length character strings

  See "Character-String Data."

- Bit-string data consists of sequences of binary digits. VAX-11 PL/I supports:

  aligned bit strings
  unaligned bit strings

  See "Bit-String Data."

The following data types represent noncomputational program values that are used within a PL/I program for control:

areas
entry data
file data
label data
offsets
pointers

The rest of this section discusses declarations and default attributes, including the default attributes of constants, for computational data types. For similar information on the noncomputational types, **see** "Area," "Entry," "File," "Label," "Offset," and "Pointer."

■ Declarations

All variables in a PL/I program must be declared. With the exception of entry point names, statement labels, built-in functions, and the default file constants SYSIN and SYSPRINT, all names referenced must be declared explicitly. You declare a name and its data type attributes in a DECLARE statement. For example:

```
DECLARE AVERAGE FIXED DECIMAL;
DECLARE NAME CHARACTER (20);
```

The keywords DECIMAL, FIXED, and CHARACTER describe characteristics, or attributes, of the variables AVERAGE and NAME. (**See** "DECLARE Statement.")

■ Default Attributes

It is not always necessary to define all the characteristics, or attributes, of a variable; the PL/I compiler makes assumptions about attributes that are not explicitly defined. For example:

```
DECLARE NUMBER FIXED;
```

The FIXED attribute implies the attributes BINARY(31). Thus, the variable NUMBER has the attributes FIXED BINARY(31).

Table D–1 shows the default attributes implied by each computational data attribute.

Table D–1: Implied Attributes for Computational Data

Specified	Implied
FIXED	BINARY(31)
FLOAT	BINARY(24)
BINARY	FIXED(31)
DECIMAL	FIXED(10,0)
FIXED BINARY	(31)
FLOAT BINARY	(24)
FIXED DECIMAL	(10,0)
FIXED DECIMAL(p)	(p,0)
FLOAT DECIMAL	(7)
BIT [ALIGNED]	(1)
CHARACTER [VARYING]	(1)
PICTURE 'picture'	see "Picture"

Attributes of Constants

Constants have attributes implied by the characters used to specify them:

- A series of characters enclosed in apostrophes is assumed to be a string constant:
  - If the letter b or B is appended after the closing apostrophe, the constant is a bit-string constant, for example, '00010101'B. If the integer 2, 3, or 4 is appended to the letter B, the constant is a bit-string constant with the base 4, 8, or 16, respectively. For example, '17777'B3 is an octal constant that is represented internally as a string of 13 bits.
  - If the constant does not have the letter B or b appended, it is a character-string constant even when it contains only the characters 0 and 1. (However, a character string of 0s and 1s can be converted by a simple assignment to a bit string.)
- If the constant is an integer, it has the attributes FIXED DECIMAL(n,0), where n is the number of digits in the integer. For example, the constant 1777 is a constant of type FIXED DECIMAL(4,0).

- Constants with an appended or embedded decimal point, but with no following exponent, are of type FIXED DECIMAL(p,q), where p is the total number of digits and q is the number of digits to the right of the decimal point.
- If a fixed-point decimal constant has the appended characters

  E [+ / -] digit...

  then it is of type FLOAT DECIMAL(p), where p is the total number of digits in the fixed-point constant (that is, the total number to the left of the letter E).

Note that PL/I has no constants with the attributes FIXED BINARY, FLOAT BINARY, or PICTURE. However, this presents no problems in programming, since constants of any computational type can be assigned to variables of any computational type and are converted automatically to the target type (**see** "Conversion of Data" for details).

Binary variables are usually given values by assigning decimal constants to them. For example,

```
I = 1;
```

converts the decimal integer 1 and assigns the converted value to a fixed-point binary variable I; also,

```
F = 1.333E-12;
```

converts the floating-point decimal constant 1.333E-12 and assigns the converted value to a floating-point binary variable F.

Picture variables are usually given values by assigning fixed-point decimal constants. For example,

```
PAY_PIC = 123.44;
```

assigns the fixed-point decimal value 123.44 to a picture variable PAY_PIC. The value of PAY_PIC is a "pictured value," stored internally as a character string containing the characters 1, 2, 3, 4, and 4, along with any special formatting symbols defined for PAY_PIC (**see** "Picture" for details).

Arithmetic Operands

The implied data types of constants are important primarily because of PL/I's rules for converting operands in an arithmetic operation. (Bit-string and character-string operations must have bit- and character-string operands, respectively.) All operations, including arithmetic operations, must be performed in a single data type, and automatic conversions are performed on arithmetic operands to make this possible. For example:

```
DECLARE X FLOAT DECIMAL (9) ;
X = X + 1.3 ;
```

In this example, the fixed-point decimal constant 1.3 is converted to floating-point decimal before the addition is performed. For the de-

tailed definition of operand conversion, **see** "Expression." Stated briefly, the rules for operand conversion are as follows:

- If either operand is binary, the operation is performed in binary.
- If either operand is floating point, the operation is performed in floating point.

These rules apply both to the declared attributes of variable operands and to the implied attributes of constant operands. Operands are converted as required to follow these rules; each converted operand then has the type (for instance, floating-point decimal) in which the operation will be performed, but it also has an individual precision based on its own attributes. These "converted precisions" (which include scale factors in fixed-point decimal operations) are used to determine the precision of the result of the operation.

■ Identical Data Types

In PL/I, the notion of identical data types is used in the rules for passing arguments by reference, for defined variables, for based variables, and for external variables. For two nonstructure variables to have identical data types, the following attributes must agree. That is, if one variable has the attribute, the other must also have it after the application of default rules:

ALIGNED	FILE	picture
BINARY	FIXED	PICTURE
BIT	FLOAT	POINTER
CHARACTER	LABEL	precision
DECIMAL	length	VARYING
ENTRY	OFFSET	array bounds
AREA		

The lowercase word "picture" in the list means that two pictured variables must have identical pictures after the expansion of iteration factors.

In addition, the following values must be equal:

- Precisions and scale factors for arithmetic data
- String lengths and area sizes
- Number of dimensions for arrays and bounds in each dimension

Two structure variables have identical data types if they have the same number of immediate members and if corresponding members have identical data types.

In general, string lengths, area sizes, and array bounds may be specified by expressions or by asterisks for parameters. The values used to determine whether two variables have identical data types are obtained as follows:

- For STATIC variables, the values must be constants.
- For AUTOMATIC and DEFINED variables, the expressions are evaluated when the block is activated that contains such a variable's declaration. The resulting values are used for all references to the variable within that block activation.

- For parameters, the declaration specifies any extents either with constants or with asterisks. In the case of asterisks, the extent in a particular procedure invocation is determined by the argument passed to the parameter. The extent remains the same throughout the procedure invocation.
- For based variables, extent expressions are evaluated each time the based variable is referenced.

■ Example

```
/* Example of extent determination */

DATAT: PROCEDURE (PTR1);

DECLARE N FIXED, S CHARACTER(N) BASED(PTR1);
DECLARE PTR1 POINTER;

N = 10;

CALL P(S);

P: PROCEDURE(A);

      DECLARE A CHARACTER(*), B CHARACTER(N);
      N = 20;
      PUT LIST(LENGTH(A),LENGTH(B),LENGTH(S));
      END P;

END DATAT;
```

The PUT statement writes out:

```
10     10       20
```

The assignment to N inside the procedure P affects the extent of S, but not the extents of A or B, which were "frozen" when P was invoked.

DATE Built-In Function

The DATE built-in function returns a six-character string in the form yymmdd, where:

yy is the current year (00–99)
mm is the current month (01–12)
dd is the current day of the month (01–31)

Its format is:

DATE()

DECIMAL Attribute

The DECIMAL attribute specifies that an arithmetic variable has a decimal base. The format of the DECIMAL attribute is:

{DECIMAL / DEC}

When you specify the DECIMAL attribute for a variable, you can also specify the following attributes to define the scale factor and precision of the data:

```
FIXED (precision[,scale-factor])
FLOAT (precision)
```

where FIXED indicates a fixed-point value and FLOAT indicates a floating-point decimal value. The precision specifies the number of decimal digits that represent values of the variable. The precision of a fixed-point decimal value is the total number of integral and fractional digits. The precision of a floating-point decimal value is the total number of digits in the mantissa. The precision for a fixed-point decimal value must be in the range 1–31; the scale factor, if specified, must be greater than or equal to zero and less than or equal to the specified precision. The precision for a floating-point decimal value must be in the range 1–34.

The default values applied to the DECIMAL attribute are:

Attributes Specified	**Defaults Supplied**
DECIMAL	FIXED (10,0)
DECIMAL FIXED	(10,0)
DECIMAL FIXED (n)	(n,0)
DECIMAL FLOAT	(7)

(**See** "Fixed-Point Decimal Data" and "Floating-Point Data.")

■ Restrictions

The DECIMAL attribute conflicts with the BINARY attribute and with any other data type attribute.

DECIMAL Built-In Function

The DECIMAL built-in function converts an arithmetic or string expression x to a decimal value of an optionally specified precision p and scale factor q.

P and q, if specified, must be integer constants. P must be greater than zero and less than or equal to the maximum precision for the result type (31 for fixed-point decimal, 34 for floating-point decimal). If q is specified, x must be a fixed-point expression and p must also be specified; if q is omitted, the scale factor of the result is zero.

The format of the function is:

```
{DECIMAL} (x[,p[,q]])
{DEC    }
```

■ Returned Value

The result type is fixed-point or floating-point decimal, depending on whether x is a fixed- or floating-point expression. (If x is a bit- or character-string expression, the result type is fixed-point decimal.)

The expression x is converted to a value v of the result type, following the usual rules (**see** "Conversion of Data" for details). The returned value is v with precision p and scale factor q. If p and q are omitted, they are the converted precision and scale factor of x (**see** "Expression" for details). FIXEDOVERFLOW, UNDERFLOW, or OVERFLOW is signaled if appropriate.

Declarations

The declaration of a name in a PL/I program consists of a user-specified identifier and the attributes of the name. The attributes describe:

- The data type of the name, that is, whether it is a computational data item such as a number or a string, or whether it is noncomputational program data
- The storage class to which the name belongs, that is, whether the compiler allocates storage for it, and how the storage is allocated
- The scope of the name, that is, whether the name is known only within the block in which it is declared and its contained blocks, or is known in external blocks

A name is declared either explicitly in a DECLARE statement or implicitly by its appearance in a particular context. Only two types of name can be declared implicitly. These are entry constants and label constants. For example:

```
CALC:  PROCEDURE;
```

This statement is an implicit declaration of the name CALC as an entry constant. All other names must be declared explicitly.

In a PL/I source program, the DECLARE statements that provide the declarations of names to be used in a given block may appear anywhere in that block. However, for clarity and readability of programs, most programmers place all the declarations for a block at the beginning of the block, and follow the declarations with the executable statements of the program. For example:

```
CALC:  PROCEDURE (X,Y);
DECLARE (X,Y) FLOAT,
        COPYSTRING ENTRY (CHARACTER(*)),
        MESSAGE_TEXT CHARACTER(40);
         .
         .
```

See "Attribute," "Data and Data Types," and "DECLARE Statement."

DECLARE Statement

The DECLARE statement specifies the attributes associated with names. The general format of the DECLARE statement is:

$$\left\{\begin{matrix}\text{DECLARE}\\ \text{DCL}\end{matrix}\right\}\ \text{declaration[,...];}$$

declaration

One or more declarations consisting of an identifier and attributes.

Formally, each declaration has the format:

[level] identifier [(bound-pair,...)] [attribute ...]

or

[level] (declaration,...) [(bound-pair,...)] [attribute ...]

The format of the DECLARE statement varies according to the number and nature of the items being declared. The DECLARE statement can list a single identifier, optionally specifying a level, bound-pair list, and other attributes for that identifier. Alternately, the statement can include, in parentheses, a list of declarations to which the level and all subsequent attributes apply. The declarations in the second case can be simple identifiers or can include attributes that are specific to individual identifiers (**see** "Factored Declarations" below).

Bound pairs are used to specify the dimensions of arrays. If bound pairs are present, they must be in parentheses and must immediately follow the identifier or the parenthetical list of declarations.

Levels are used to specify the relationship of members of structures; if a level is present in the declaration, it must be written first.

The various formats are described individually, below. **See also** "Array" and "Structure."

■ Simple Declarations

A simple declaration defines a single name and describes its attributes. The format of a simple declaration is:

DECLARE identifier [attribute ...] ;

identifier

A 1- to 31-character user-supplied name. The name must be unique within the current block.

An identifier can consist of any of the alphanumeric characters A through Z, a through z, 0 through 9, $ and _, but must begin with an alphabetic letter, dollar sign, or underline. **See also** "Identifier."

attribute ...

One or more attributes of the name. Attributes, if specified, must be separated by spaces. They can appear in any order.

The valid attribute keywords and their meanings are summarized under "Attribute."

Some examples of simple declarations are:

```
DECLARE COUNTER FIXED BINARY (7);
DECLARE TEXT_STRING CHARACTER (80) VARYING;
DECLARE INFILE FILE;
```

Names that are not given specific attributes in a DECLARE statement or that are referenced without being declared are given the default attributes:

BINARY FIXED (31,0) AUTOMATIC

Note that the compiler issues a warning message whenever it gives a name these default attributes.

■ Multiple Declarations

Multiple declarations define two or more names and their individual attributes. This format of the DECLARE statement is:

```
DECLARE identifier [attribute ...]
        [,identifier [attribute ...]] ...;
```

When you specify more than one set of names and their attributes, separate each name and attribute set from the preceding set with a comma. A semicolon must follow the last name.

Some examples of multiple declarations are:

```
DECLARE COUNTER FIXED BINARY (7),
        TEXT_STRING CHARACTER (80) VARYING,
        Y FILE;
```

This DECLARE statement defines the variables COUNTER, TEXT_STRING, and Y. The attributes for each variable follow the name of the variable.

■ Factored Declarations

When two or more names have the same attribute, you can combine the declarations into a single, factored declaration. This format of the DECLARE statement is:

```
DECLARE (identifier[,identifier,...])
        [attribute ...] ;
```

When you use this format, you must place names that share common attributes within parentheses, separated by commas. The attributes that follow the parenthetical list of names are applied to all the named identifiers.

Some examples of factored declarations are:

```
DECLARE (COUNTER, RATE, INDEX) FIXED BINARY (7);
DECLARE (INPUT_MESSAGE, OUTPUT_MESSAGE, PROMPT)
        CHARACTER (80) VARYING;
```

In the preceding declarations, the variables COUNTER, RATE, and INDEX share the attributes FIXED BINARY (7). The variables INPUT_MESSAGE, OUTPUT_MESSAGE, and PROMPT share the attributes CHARACTER (80) VARYING.

You can also specify, within the parentheses, attributes that are unique to specific variable names, using this format:

```
DECLARE (identifier attribute ...,
         identifier [attribute ...],...)
         attribute ...
```

For example:

```
DECLARE (INFILE INPUT RECORD,
         OUTFILE OUTPUT STREAM) FILE;
```

The DECLARE statement declares INFILE as a RECORD INPUT file and OUTFILE as an OUTPUT STREAM file.

The parentheses can be nested. For example:

```
DECLARE ( (INFILE INPUT, OUTFILE OUTPUT) RECORD,
          SYSFILE STREAM ) FILE;
```

The DECLARE statement declares INFILE as a RECORD INPUT file, OUTFILE as a RECORD OUTPUT file, and SYSFILE as a STREAM INPUT file (STREAM implies INPUT).

■ Array Declarations

The declaration of an array specifies the dimensions of the array and the bounds of each dimension. This format of a DECLARE statement is:

DECLARE identifier (bound-pair,...) [attribute ...];

or

DECLARE (declaration,...) (bound-pair,...) [attribute ...];

where each bound pair has the format:

[lower-bound:]upper-bound

or

*

One bound pair is specified for each dimension of the array. The number of elements per dimension is defined by the bound pair. The extent of an array is the product of the numbers of elements in its dimensions. If the lower bound is omitted, the lower bound for that dimension is 1 by default.

The asterisk (*) can be used as the bound pair when arrays are declared as parameters of a procedure. The asterisk indicates that the parameter can accept array arguments with any number of elements. (If one dimension is specified with an asterisk, all must be specified with asterisks.)

As an example, the statement:

```
DECLARE SALARIES(100) FIXED DECIMAL(7,2);
```

declares a 100-element array with the identifier SALARIES. Each element is a fixed-point decimal number with a total of seven digits, two of which are fractional. The identifier in the statement can be replaced with a list of declarations, to declare several objects with the same attributes. For instance,

```
DECLARE (SALARIES,PAYMENTS) (100) FIXED DECIMAL(7,2);
```

declares SALARIES and another array, PAYMENTS, with the same dimensions and other attributes.

For further details on how to specify the bounds of an array, and for examples of array declarations, **see** "Array."

■ Structure Declarations

The declaration of a structure defines the organization of the structure and the names of members at each level in the structure. This format of a DECLARE statement is:

DECLARE declaration[,...]

where each declaration is:

level identifier [(bound-pair,...)] [attribute ...]

or

level (declaration,...) [(bound-pair,...)] [attribute ...]

Each declaration specifies a member of the structure and must be preceded by a level number. As shown, a single variable can be declared at a particular level; or the level can contain one or more complete declarations, including declarations of arrays or of other structures. The major structure name is declared as structure level 1; minor members must be declared with level numbers greater than 1. For example, the statement:

```
DECLARE 1 PAYROLL,
        2 NAME,
                3 LAST CHARACTER(80) VARYING,
                3 FIRST CHARACTER(80) VARYING,
        2 SALARY FIXED DECIMAL(7,2);
```

declares a structure named PAYROLL. The last name can be accessed with a qualified reference:

```
PAYROLL.NAME.LAST = 'ROOSEVELT';
```

Alternately, since the last and first names have the same attributes, the same structure can be declared as:

```
DECLARE 1 PAYROLL,
        2 NAME,
                3 (LAST,FIRST) CHARACTER(80) VARYING,
        2 SALARY FIXED DECIMAL(7,2);
```

For details and examples of structure declarations, **see** "Structure."

DEFINED Attribute

The DEFINED attribute indicates that PL/I is not to allocate storage for the variable, but is to map the description of the variable onto the storage of a base variable. The DEFINED attribute provides a way to access the same data using different names. Its format is:

{DEFINED / DEF} (variable-reference)

variable-reference

A reference to a base variable that has storage associated with it. The base variable must not have the BASED or DEFINED attribute. The base variable and the declared variable must satisfy the rules given under "Defined Variable."

The DEFINED attribute may optionally specify a position within the base variable at which the definition begins. For example:

```
DECLARE ZONE CHARACTER(10)
        DEFINED(ZIP) POSITION(4);
```

For more information, **see** "POSITION Attribute" and "Defined Variable."

■ Restrictions

- The following attributes conflict with the DEFINED attribute:

AUTOMATIC	GLOBALDEF
BASED	GLOBALREF
EXTERNAL	READONLY
INITIAL	STATIC
VALUE	parameter

- The DEFINED attribute cannot be applied to minor structures, members of structures, parameters, or descriptions in an ENTRY or RETURNS attribute.

For additional information on defined variables, **see** "Defined Variable."

Defined Variable

A defined variable is a variable for which no storage is allocated. Instead, the variable shares the storage of a specified base variable. A defined variable is declared with the DEFINED attribute, which also specifies the base variable. Any reference to the defined variable is a reference to part or all of the storage of the base variable. For example:

```
DECLARE A(10) FIXED, B FIXED DEFINED(A(I));
```

The variable B is a defined variable, with A as its base reference. A reference to B is a reference to the element of A denoted by the current value of I.

The extents of a defined variable are determined at the time of block activation, but the base reference (and the position, if the POSITION attribute is also specified) are interpreted each time the defined variable is referenced.

For example:

```
DECLARE A(10) FIXED, B FIXED DEFINED(A(I));
DO I = 1 TO 10;
B = I;
END;
```

The DO group assigns I to A(I) for I = 1,2,..10.

The base reference of a defined variable may not be a reference to a based variable or to another defined variable.

A defined variable and its base reference must satisfy one of the following criteria:

- They must have identical data types (**see** "Data and Data Types").
- They must both be suitable for character-string overlay defining.
- They must both be suitable for bit-string overlay defining.

If the defined variable is specified with the POSITION attribute, then both the defined variable and the base reference must be suitable for bit- or character-string overlay defining.

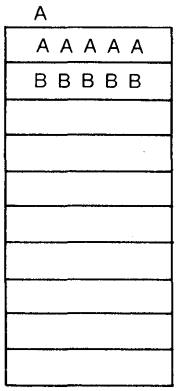
Roughly, a variable is suitable for overlay defining if it consists entirely of characters or bits, and these characters or bits are packed into adjacent storage without gaps. (Precise rules are given below.) Such a variable can be treated as a string (**see also** "STRING Built-In Function" and "STRING Pseudovariable") or can be interpreted as different types of aggregates. For example, Figure D–1 shows a 50-byte region of storage treated either as a 10-element array (A) of five-character strings or as a five-element array (B) of 10-character strings.

```
DECLARE A (10) CHARACTER (5) ;

DECLARE B (5) CHARACTER (10) DEFINED(A);

A(1)   'AAAAA';
A(2)   'BBBBB';
PUT LIST (B(1)) ;

Result: AAAAABBBBB
```



The declaration of A reserves storage for a 10-element array of 5-byte character strings.

The declaration of B defines B's storage as equivalent to A's. Any reference to B will access the same storage as that allocated for A.

Figure D–1: An Overlay Defined Variable

If the defined variable and its base reference have identical data types, a reference to the defined variable is equivalent to the base reference. In the case of overlay defining, the defined variable maps onto part of the base reference's storage as follows:

1. If the POSITION attribute was specified, let pos be its value at the moment of reference; otherwise, let pos equal one.
2. Let m be the total number of characters (or bits) specified by the data type of the defined variable. (Note that, for pictured data, m is the total number of characters in the picture specification, exclusive of the V character.)
3. A reference to the defined variable accesses m characters (or bits) of the base reference, beginning with the character or bit specified by pos. The reference must lie entirely within the base reference; that is, pos and m must satisify

   $$1 \leq pos \leq pos+m \leq n+1$$

   where n is the total number of characters or bits in the base reference.

■ Precise Rules for Overlay Defining

A variable V is suitable for character-string overlay defining if V is not an unconnected array and if one of the following criteria is satisfied:

1. V has the attribute CHARACTER but not ALIGNED or VARYING.
2. V has the attribute PICTURE.
3. V is a structure, and each of V's members and submembers that is not a structure satisfies criterion 1 or 2.

A variable V is suitable for bit-string overlay defining if V is not an unconnected array and if one of the following criteria is satisfied:

1. V has the attribute BIT but not ALIGNED.
2. V is a structure, and each of V's members or submembers that is not a structure satisfies criterion 1.

DELETE Statement

The DELETE statement deletes a record from a file, either the current record (**see** "Record Input/Output") or the record specified by the KEY option. The file must have the UPDATE attribute.

The format of the DELETE statement is:

```
DELETE FILE(file-reference) [ KEY (expression) ]
                            [ OPTIONS(option,...) ]
```

file-reference

A reference to the file from which the specified record is to be deleted. If the file is not currently opened, PL/I opens the file with the implied attributes RECORD and UPDATE; these attributes are merged with the attributes specified in the file's declaration.

KEY (expression)

An option specifying that the record to be deleted is to be located using the key specified by expression. The file must have the KEYED attribute.

The nature of the key depends on the file's organization, as follows:

- If the file is a relative file, the key is a fixed binary value indicating the relative record number of the record to be deleted.
- If the file is an indexed sequential file, the key is contained in the record; its position in the record and its data type are as determined when the file was created.

The value of the specified expression is converted to the data type of the key. If no record with the specified key exists in the file, or if the value specified is not valid for conversion to the data type of the key, the KEY condition is signaled.

OPTIONS(option,...)

An option giving one or more of the DELETE statement options listed below, separated by commas:

FAST_DELETE
INDEX_NUMBER (expression)
MATCH_GREATER
MATCH_GREATER_EQUAL
RECORD_ID (expression)

These options are described fully in the *VAX-11 PL/I User's Guide*.

■ File Positioning

The next record is set to denote the record following the deleted record. The current record is undefined.

■ Examples

The program BAD_RECORD, below, deletes an erroneous record in an indexed sequential file containing data about states. The primary key in the file is the name of a state.

```
BAD_RECORD:  PROCEDURE OPTIONS(MAIN);

DECLARE STATE_FILE FILE KEYED UPDATE;
        OPEN FILE(STATE_FILE) TITLE('STATEDATA.DAT');
        DELETE FILE(STATE_FILE) KEY('Arklansas');
        CLOSE FILE(STATE_FILE);
RETURN;
END;
```

The file is opened with the UPDATE attribute, and the OPEN statement gives the file specification of the file from which the record is to be deleted.

DESCRIPTOR Built-In Function

The DESCRIPTOR built-in function forces its argument to be passed by descriptor to a non-PLI procedure. The corresponding parameter descriptor must specify the ANY attribute without the VALUE attribute. A reference to the built-in function may occur only as an argument in such a context and has no other use. The format of the function is:

DESCRIPTOR(expression)

expression

The argument to be passed by descriptor. Its data type must be computational but may not be pictured. (It may be an array variable.)

For a full discussion of argument passing to non-PL/I procedures, **see** the *VAX-11 PL/I User's Guide* and the entry "ANY Attribute" in this manual.

Diagnostic Messages

Diagnostic messages are produced by the PL/I compiler to inform you of programming errors detected by the compiler, and to warn you of certain exceptional conditions, such as the compiler's assignment of type and precision to an undeclared variable.

For full details on diagnostic messages, **see** the *VAX-11 PL/I User's Guide.*

Dimension Attribute

The dimension attribute defines a variable as an array. It specifies the number of dimensions of the array and the bounds of each dimension. The format of the dimension attribute is:

(bound-pair[,bound-pair]...)

bound-pair

One or two expressions that indicate the number of elements in a single dimension of the array. The list of bound pairs must be specified immediately following the name of the identifier in the array declaration.

The maximum number of dimensions allowed is eight.

A bound pair can be specified as follows:

- [lower-bound:]upper-bound

  This format of a bound pair specifies the minimum and maximum subscripts that can be used for the dimension. The number of elements is therefore:

  (upper-bound - lower-bound) + 1

  If the lower bound is omitted, it is assumed to be one.

- *

  This format of a bound pair, when used to define a parameter for a procedure or function, indicates that the bounds are to be determined from the associated argument. If one bound pair is specified as an asterisk, all bound pairs must be specified as asterisks.

For the complete rules for specifying dimensions and bounds, **see** "Array — Rules for Specifying Dimensions."

DIMENSION Built-In Function

The DIMENSION built-in function returns a fixed-point binary integer that is the number of elements in a specified dimension of an array. Its format is:

{DIMENSION / DIM} (reference,dimension)

reference

The name of an array variable.

dimension

An integer constant specifying the dimension of the array for which the extent is to be determined.

■ Example

```
INIT:  PROCEDURE (ARRAY);
DECLARE ARRAY(*) FIXED,
        I FIXED;

        DO I = 1 TO DIM(ARRAY,1);
            ARRAY(I) = I;
            END;
```

This procedure is passed a one-dimensional array of an unknown extent. The DIMENSION built-in function is used as the end value in a controlled DO statement. This DO-group assigns integral values to each element of the array ARRAY so that the first element has the value 1, the second element has the value 2, and so on to the last element of the array.

DIRECT Attribute

The DIRECT file description attribute indicates that a file will be accessed only in a nonsequential manner, that is, by key or by relative record number.

The DIRECT attribute implies the RECORD and KEYED attributes.

Specify the DIRECT attribute on a DECLARE statement for a file constant or on an OPEN statement to access the file. A file declared with the DIRECT attribute must be one of the following:

- A relative file
- An indexed sequential file
- A sequential disk file with fixed-length records
- A sequential file opened with ENVIRONMENT(BLOCK_IO)

See "File" and "Record Input/Output."

To access a file both randomly and sequentially, use the SEQUENTIAL attribute instead of DIRECT (**see** "SEQUENTIAL Attribute").

■ Restrictions

The DIRECT attribute conflicts with the SEQUENTIAL, STREAM, and PRINT attributes.

DIVIDE Built-In Function

The DIVIDE built-in function divides an arithmetic expression x by an arithmetic expression y and returns the quotient with a specified precision p and optionally specified scale factor q. The scale factor q must be a nonnegative integer following these rules:

- If either x or y is binary, q must be zero.
- If q is not zero, both expressions must be fixed-point decimal.
- If q is omitted, it is assumed to be zero.

The expressions x and y are converted to their derived type before the division is performed (**see** "Expression"). If y is zero after this conversion, the ZERODIVIDE condition is signaled. The quotient has the derived type of the two arguments.

Division of fixed-point binary data must be performed with the DIVIDE built-in function, to avoid a result with a nonzero scale factor.

The format of the function is:

```
DIVIDE(x,y,p[,q])
```

Division

The slash sign character (/) indicates a division operation in an expression; the result is the quotient of the first operand divided by the second operand. Both must be arithmetic or picture data. The division operator should not be used to divide two fixed-point binary operands; use the DIVIDE built-in function instead.

■ Precision of the Result

Before the division is performed, the two operands are converted to their derived type (**see** "Expression — Conversion of Operands"). Each converted operand has an individual converted precision, and the two precisions are used to determine the precision of the result.

Floating-Point Operands

The floating-point result has the maximum of the converted precisions of the operands.

Fixed-Point Operands

If (p,q) and (r,s) represent the converted precisions and scale factors of the two operands, the resulting precision and scale are:

```
precision:    31
scale factor: 31-p+q-s
```

If the quotient exceeds the precision of the result, the least significant digits of the quotient are truncated.

■ Restrictions

- The divisor (that is, the second operand) must not be zero. If the divisor equals zero, the ZERODIVIDE condition is signaled; if no ON-unit exists to handle this condition, the program terminates.
- Both operands cannot be fixed binary. To divide fixed-point binary operands, use the DIVIDE built-in function.
- For division of fixed-point decimal operands, the precisions of the operands must be such that the result does not have a negative scale factor.

The DIVIDE built-in function also performs division on arithmetic data; it allows you to control precisely the precision of the result.

DO-Group

A DO-group is a sequence of PL/I statements delimited by a DO statement and its corresponding END statement. The statements in a DO-group are executed as the result of an unconditional DO statement or as the result of the successful test of a conditional DO.

For example:

```
IF A > B THEN DO ;
      .
      .
   END ;
```

The statements that occur between the DO and the END are a DO-group. After all statements are executed in this unconditional DO-group, execution continues with the next executable statement following the END statement.

Normally, all the statements in the group are executed. However, control can be transferred out of a DO-group in the following ways:

- Execution of a GOTO statement that transfers control outside of the DO-group. The GOTO statement can be present in the DO-group itself, in a procedure invoked from within the DO-group, or in an ON-unit executed while the DO-group is active.
- Execution of a RETURN or STOP statement that terminates the current procedure or program.

DO-groups can be nested to a maximum level of 64.

DO Statement

The DO statement defines the beginning of a sequence of statements to be executed in a group. The group begins with the DO statement and ends with the nonexecutable statement END. DO-groups have several formats. These formats are summarized in Figure D–2 and described individually, below, under the following subheadings:

- Simple DO
- DO WHILE
- Controlled DO
- DO REPEAT

DO ; . . END ;	*The statements in a simple, noniterative DO-group are executed a single time.*
DO WHILE (test-expression) ; . . END ;	*The statements in the DO-group following a DO WHILE are executed in a loop as long as the condition specified in the test expression is satisfied.*
DO control-variable start-value TO end-value [WHILE (test-expression)]; . . END ;	*Each time the statements in the DO-group are executed, the specified control variable has a different value. When the DO statement is evaluated at the start of each execution, the control variable is incremented by 1. When its value exceeds the specified end value, control passes out of the DO-group.* *Optionally, a WHILE clause can provide a condition that must also be satisfied in order for the DO-group to be executed.*
DO control-variable start-value BY modify-value [WHILE (test-expression)]; . . END ;	*The value of the control variable is modified by a specified positive or negative value; for each iteration of the DO-group, it has a different value. The DO-loop is terminated by a statement within the loop or, if the optional WHILE clause is specified, when the test expression yields a false value.*
DO control-variable start-value TO end-value BY modify-value [WHILE (test-expression)] ; . . END ;	*The DO statement can specify a range of values to use for the control variable as well as a value by which it is to be modified.* *Optionally, a WHILE clause can provide a condition that must be met in order for the DO-group to be executed.*
DO control-variable start-value REPEAT expression WHILE (test-expression);	*The repetition of the statements in the DO-group is controlled by the expression in the REPEAT option. This expression defines how the control variable is to be modified.* *The WHILE clause provides the condition that must be met in order for execution to continue.*

Figure D–2: Forms of the DO Statement

■ Simple DO

A simple DO statement is a noniterative DO. The format of a simple DO statement is:

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

The statements that appear between the DO statement and its corresponding END statement are executed once. After all statements in the group are executed, control passes to the next executable statement in the program.

■ Examples

```
IF A < B THEN DO;
   PUT LIST ('More data needed');
   GET LIST (VALUE);
   A = A + VALUE;
   END;
```

A common use of the simple DO statement is as the action of the THEN clause of an IF statement, as shown above, or of an ELSE option.

■ DO WHILE

A DO WHILE statement executes a group of statements as long as a particular condition is satisfied. When the condition is not true, the group is not executed. This format of the DO statement is:

```
DO WHILE (test-expression);
  .
  .
END;
```

test-expression

Any expression that yields a scalar bit-string value. If any bit of the value is a 1, then the test expression is true; otherwise, the test expression is false. The test expression must be enclosed in parentheses.

This expression is evaluated before each execution of the DO-group. It must have a true value in order for the DO-group to be executed. Otherwise, control passes outside of the DO-group, to the next executable statement following the END statement that terminates the group.

■ Examples

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

The DO-group executes as long as the value of the variable A is less than the value of the variable B.

```
DO WHILE (LIST->NEXT ^= NULL());
```

The DO-group executes until a forward pointer in a linked list has a null value. (**See** "List Processing.")

```
DECLARE EOF BIT(1) INITIAL('0'B);
        .
        .
ON ENDFILE(INFILE) EOF = '1'B;
DO WHILE (^EOF);
READ FILE(INFILE) INTO(INREC);

        .
        .
END;
```

This DO-group reads records from the file INFILE until the end of the file is reached. At the beginning of each iteration of the DO-group, the expression

^EOF is evaluated; the expression is '1'B until the ENDFILE ON-unit sets the value of EOF to '1'B.

■ Controlled DO

A controlled DO statement identifies a variable whose value controls the execution of the DO-group and defines the conditions under which the control variable is to be modified and tested. The format of the controlled DO statement is:

```
DO control-variable = start-value
{ TO end-value [BY modify-value] }
{ BY modify-value                }
      [ WHILE (test-expression) ] ;
  .
  .
END;
```

control-variable
A reference to a variable whose current value determines whether the DO-group is executed. The control variable must be of an arithmetic data type.

start-value
An expression specifying the initial value to be given to the control variable. Evaluation of this expression must yield an arithmetic value.

end-value
An expression giving the value to be compared with the control variable during successive iterations. Evaluation of this expression must yield an arithmetic value.

modify-value
An expression giving a value by which the control value is to be modified. Evaluation of this expression must yield an arithmetic value. If the BY option is not specified, the modify value is 1 by default.

WHILE (test-expression)
An option specifying a condition that further controls the execution of the DO-group. The specified test expression must yield a scalar bit-string value. If any bit in the value is a 1, then the test expression is true; otherwise, the test expression is false. The test expression must be enclosed in parentheses.

The controlled DO-group is executed by the following steps:

1. The following steps are taken to prevent the allocation of a new control variable during the execution of the DO-group:
   - If the control variable is based, its pointer qualifier is evaluated and a temporary reference of the control variable type is created. The temporary reference is used as the control variable in subsequent steps.

- If the control variable is subscripted, its subscripts are evaluated and a temporary reference of the control variable type is created. The temporary reference is used as the control variable in subsequent steps.
- If the control variable is neither based nor subscripted, its reference is used in subsequent steps.

2. The start value expression is evaluated and assigned to the control variable. The expressions specified in the TO and (if specified) BY options are evaluated and their values are stored. These expressions may contain references to the object referenced by the control variable. If so, the original reference, not the temporary reference, is used in evaluation of the expressions.
3. If the TO option is present, the value of the control variable is compared with the end value specified in the TO option. Otherwise, this step is skipped. Execution of the DO-group terminates if either of the following is true:
   - The modify value is greater than zero and the control variable is greater than the end value.
   - The modify value is less than zero and the control variable is less then the end value.

   Note that if this step terminates the DO-group on the first iteration, the control variable has a final value assigned by the start value. If the group is terminated on a subsequent iteration, the control variable has a final value assigned by step 6.
4. If a test expression is present, it is evaluated. If it does not produce a true value, the execution of the DO-group terminates.
5. The body of the DO-group is executed. The execution of the DO-group may be terminated during this step by the execution of a STOP or RETURN statement or by the execution of a GOTO statement that transfers control out of the DO-group.

   The body of the DO-group may also contain statements that change the values of the control variable, modify value, end value, or test expression.
6. The value of the control variable is modified as follows:

   control variable = control variable + modify value ;
7. Execution continues at step 3.

■ Examples

```
DO I = 2 TO 100 BY 2;
```

The DO-group executes 50 times, with values for I of 2, 4, 6, and so on.

```
DO I = LBOUND(ARRAY,1) TO HBOUND(ARRAY,1);
```

The DO-group executes as many times as there are elements in the array variable ARRAY, using the subscript values of the array's elements.

```
DO I = 1 BY 1 WHILE (X < Y);
```

The DO-group continues executing with successively higher values for I until the value of the variable X equals or is greater than the value of the variable Y.

■ DO REPEAT

The DO REPEAT statement executes a DO-group repetitively for different values of a control variable. The control variable is assigned a start value that is used on the first iteration of the group. The REPEAT expression is evaluated before each subsequent iteration, and its result is assigned to the control variable. A WHILE clause may also be included; if it is, the WHILE expression is evaluated before each iteration, including the first. The format of the DO REPEAT statement is:

DO control-variable = start-value REPEAT expression

[WHILE (test-expression)] ;

control-variable

A reference to a variable whose current value determines whether the DO-group is executed. The control variable can be a scalar variable of any computational type.

start-value

An expression specifying the initial value to be given to the control variable. The evaluation of this expression must yield a value that is valid for assignment to the control variable.

expression

An expression giving the value to be assigned to the control variable on reiterations of the DO REPEAT group. The expression is evaluated before each reiteration. Evaluation of this expression must yield a result that is valid for assignment to the control variable.

WHILE (test-expression)

An option specifying a condition that controls the termination of the DO REPEAT group. The specified test expression must yield a scalar bit-string value. If any bit of the value is 1, then the test expression is true; otherwise, the test expression is false. The test expression must be enclosed in parentheses.

This expression is evaluated each time control reaches the DO statement; the test expression must have a true value in order for the DO-group to be executed. Otherwise, control passes outside of the DO-group, to the next executable statement following the END statement that terminates the group.

NOTE

If the WHILE option is omitted, the DO REPEAT statement specifies no means for terminating the group; the execution of the group must be terminated by a statement or condition occurring within the group.

A DO REPEAT group is executed in the following manner:

1. The following steps are taken to prevent the allocation of a new control variable during the execution of the DO-group:
   - If the control variable is based, its pointer qualifier is evaluated and a temporary reference of the control variable type is created. The temporary reference is used as the control variable in subsequent steps.
   - If the control variable is subscripted, its subscripts are evaluated and a temporary reference of the control variable type is created. The temporary reference is used as the control variable in subsequent steps.
   - If the control variable is neither based nor subscripted, its reference is used in subsequent steps.
2. The start value expression is evaluated and assigned to the control variable.
3. If the test expression is present, it is evaluated. If it does not produce a true value, the execution of the DO-group terminates. If the test expression is not present, execution continues.
4. The body of the DO-group is executed. The execution of the DO-group may be terminated during this step by the execution of a STOP or RETURN statement or by the execution of a GOTO statement that transfers control outside the DO-group. Statements in the group can also modify the values of the control variable, REPEAT expression, and test expression.
5. The REPEAT expression is evaluated and its value is assigned to the control variable.
6. Execution continues at step 3, above.

■ Examples

```
DO LETTER='A' REPEAT BYTE(I);
```

This example will repeat the group with an initial LETTER value of 'A' and with subsequent values assigned by the built-in function BYTE(I). The variable I may be assigned new values within the group. The group will iterate endlessly unless terminated by a statement or condition within the group.

```
DO I = 1 REPEAT I + 2 WHILE ( I <= 100 );
```

This example has the same effect as the controlled DO statement:

```
DO I = 1 TO 100 BY 2;
```

The most common use of the DO REPEAT statement is in the manipulation of lists. For example:

```
DO P = LIST_HEAD REPEAT P->LIST.NEXT
    WHILE ( P ^= NULL() );
```

In this example, the pointer P is initialized with the value of the pointer variable LIST_HEAD. The DO-group is executed with this value of P. The REPEAT option specifies that, each time control reaches the DO statement after the first execution of the DO-group, P is to be set to the value of LIST.NEXT in the structure currently pointed to by P. For an expanded example of this technique, **see** "List Processing."

E

E Format Item

The E format item describes the representation of a fixed- or floating-point value as a decimal floating-point number in a stream.

The form of the item is:

E(w[,d])

w

A nonnegative integer that specifies the total width in characters of the field in the stream.

d

An optional nonnegative integer that specifies the number of fractional digits in the stream representation. If d is omitted on output, all fractional digits are written out. If d is omitted on input, it is assumed to be zero (no fractional digits). If the input value contains a decimal point, the value of d is ignored.

The interpretation of the E format item on input and output is given below. For a general discussion of format items, **see** "Format Items and Their Uses."

■ Input with GET EDIT

Used with GET EDIT, the E format item acquires a character-string value representing a floating-point decimal value and assigns it, with necessary conversions, to an input target of any computational type. If w is zero, no operation is performed on the input stream, and a null character string is converted and assigned to the input target.

For input, floating-point values can be represented in the stream in the following forms:

Form	Example
mantissa	124333
sign mantissa	-123.333
sign mantissa sign exponent	-123.333-12
sign mantissa E exponent	-123.333E12
sign mantissa E sign exponent	-123.343E-12

The mantissa is a fixed-point decimal constant, the sign is a + or - symbol, and the exponent is a decimal integer. A zero exponent is assumed if both the letter E and the exponent are omitted.

If, on input, the mantissa includes a decimal point, it overrides the specification of d. If no decimal point is included, then d specifies the number of fractional digits.

The value of w should be only large enough to include the mantissa, the optional decimal point in the mantissa, the signs on the exponent and mantissa, the optional letter E, and the exponent. If the field width is too narrow, the stream representation may be truncated on the right; if the field width is too wide, excess characters are acquired on the right and may contain invalid input.

Spaces can precede or follow the value in the stream and are ignored. If the entire field contains spaces, zero is assigned to the input target. If the stream representation is not one of the forms shown previously, the ERROR condition is signaled.

■ Output with PUT EDIT

Used in a PUT EDIT statement, the E format item converts an output source of any computational type to the following form for representation in the stream:

[-] digit . [fractional-digits] E sign exponent

Typical representations are:

```
1.E+07
3.33E-10
-2.7186E+00
```

If d is omitted from the format item, then d = s–1, where s is the precision of the output source expressed in decimal. The decimal value is rounded before being written out.

The exponent ordinarily is a two-digit decimal integer and is always signed. The exponent is adjusted so that the first digit of the mantissa is not zero, except that the value 0 is represented as

0.0000...E+00

with a number of zeros to the right of the decimal point equal to the specified number of fractional digits.

To account for negative values with fractional digits, the specified width integer should be 6 greater than the number of digits to be represented in the mantissa: one character for the preceding minus sign, one for the decimal point in the mantissa, one for the letter E, one for the sign of the exponent, and two for the exponent itself. (For values of type G-float or H-float, the value of w should be 7 or 8 greater than the number of digits, respectively.)

If the number's representation is shorter than the specified field, the representation is right-justified in the field and the number is extended on the left with spaces.

If the field specified by w is too narrow, the ERROR condition is signaled.

■ Examples

The tables below show the relationship between the internal and external representations of numbers that are read or written with the E format item.

Input Examples

The "input stream" shown in the table is a field of characters beginning at the current position in the stream and continuing to the right. The target type is the type of the variable to which the input value is assigned.

Format Item	Input Stream	Target Type	Target Value
E(6,0)	124333...	DECIMAL(10,2)	124333.00
E(6,0)	-123333...	DECIMAL(10,2)	-12333.00
E(8)	-123.333...	DECIMAL(8,5)	-123.33300
E(11)	-123.333-12...	FLOAT DEC(7)	-1.233330E-10
E(11,3)	-123343E-12...	FLOAT DEC(15)	-1.233429998137 58E-07

Output Examples

The output source value shown in the table is either a constant or the value of a variable that is written out with the associated format item.

Output Source Value	Format Item	Output Value
-12234	E(11)	-1.2234E+04
-12234	E(11,2)	ΔΔ-1.22E+04
-12.234	E(11,1)	ΔΔΔ-1.2E+01
-1.23456E3	E(12)	-1.23456E+03
-1.23456E3	E(12,2)	ΔΔΔ-1.23E+03

EDIT Option

The EDIT option is used with the GET and PUT statements to perform edit-directed stream input or output.

The EDIT option allows you to include a format-specification list that matches the list of input targets (GET statement) or output sources (PUT statement). When used in the GET statement, the EDIT option and format-specification list control the interpretation of ASCII characters being input from a stream file. When used with the PUT statement, the two items control the representation of program data as ASCII characters in a stream output file.

For further details, **see** "GET Statement" and "PUT Statement."

ELSE Option

The ELSE option may be specified in an IF statement to define the action to be taken if a given expression is false. For example:

```
IF ^SUCCESS THEN
        CALL PRINT_ERROR;
   ELSE
      CALL PRINT_SUCCESS;
```

The action following the ELSE option may be null. For more information, **see** "IF Statement."

END Statement

The END statement terminates a block or a group that is headed by the most recent BEGIN, DO, or PROCEDURE statement. The format of the END statement is:

END [label-reference] ;

label-reference

A reference to the label on the PROCEDURE, BEGIN, or DO statement for which the specified END statement is the termination. A label is not required. If specified, the label reference must match only one label, which is the label of the most recent BEGIN, DO, or PROCEDURE statement that is not already matched with an END statement. If the label reference is omitted, the most recent statement is matched by default.

The END statement performs one of the following actions, depending on the type of block or group that it terminates:

- When an END statement denotes the end of a procedure, the current procedure is terminated. The storage allocated for the block is released, and all automatic variables are made inaccessible. If the current procedure is the main, or only, procedure, the program terminates. Otherwise, control is returned to the point following the CALL statement or function reference that invoked the procedure.
- When an END statement denotes the end of a BEGIN block, the storage allocated for the block is released, and all automatic variables are made inaccessible. Control passes to the next executable statement following the END statement.
- When an END statement denotes the end of a DO-group, control returns either to the DO statement that heads the group or to the next outer statement. If the DO-group is headed by a noniterative DO, that is, a DO-group that is executed only once, control passes to the next executable statement. Otherwise, control returns to the head of the DO-group, where the control variable or expression is tested.

ENDFILE Condition Name

The ENDFILE condition name can be specified in an ON, SIGNAL, or REVERT statement to designate an end-of-file condition or ON-unit for a specific file.

PL/I signals the ENDFILE condition when a GET or READ statement attempts an input operation on a file or device after the last data item has been input. The format of the ENDFILE condition name is:

ENDFILE (file-reference)

file-reference

The name of a file constant or file variable for which the ENDFILE ON-unit is established. If the name of a file variable is specified, the variable must be resolved to the name of a file constant when the condition is signaled.

An ENDFILE ON-unit can be established for any input file. For any particular file, the meaning of the end-of-file condition depends on the type of device. For example, end-of-file is signaled for a terminal device when the CTRL/Z character is read.

For a stream file, an end-of-file condition is signaled whenever a GET statement attempts to access an empty file or attempts to access a file after its last input field has been read.

For a record file, an end-of-file condition is signaled when a READ statement is executed with the file at the end-of-file position or when a read is attempted beyond the last record in the file. For example:

```
        ON ENDFILE (RECEIPTS) GOTO PRINT_REPORT;
        OPEN FILE (RECEIPTS) RECORD SEQUENTIAL;
LOOP:   READ FILE (RECEIPTS) INTO (RECORD);
           .
           .
        GOTO LOOP;
```

In this example, the ON statement establishes the default action to take when the last record in the input file has been processed: control is transferred to the label PRINT_REPORT.

An ON-unit established to handle end-of-file conditions can reference the ONFILE built-in function to determine the name of the file constant for which the condition was signaled.

■ ON-Unit Completion

If the ON-unit for the ENDFILE condition does not transfer control elsewhere in the program, control returns to the statement following the GET or READ statement that caused the condition to be signaled.

When the ENDFILE condition is signaled, it remains in effect until the file is closed. Subsequent GET or READ statements for the file cause the ENDFILE condition to be signaled repeatedly.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement."

ENDPAGE Condition Name

The ENDPAGE condition name can be specified in an ON, SIGNAL, or REVERT statement to designate an end-of-page condition or ON-unit for a specific print file. The format of the ENDPAGE condition name is:

ENDPAGE (file-reference)

file-reference

The name of the file constant or file variable for which the ENDPAGE ON-unit is to be established. If the name of a file variable is specified, the variable must be resolved to the name of a file constant when the condition is signaled. The file must have the PRINT attribute.

The maximum number of lines that can be output on a single page is set by the PAGESIZE option of the OPEN statement. If not specified, PL/I uses the default page size (**see** "PAGESIZE Option").

PL/I signals the ENDPAGE condition when a PUT statement attempts to output a line beyond the last line specified for an output page. When the ENDPAGE condition is signaled, the current line number associated with the file is (pagesize+1). An ENDPAGE ON-unit allows you to provide special processing before output continues on a new page. For example:

```
ON ENDPAGE (PRINTFILE) BEGIN;
   PUT FILE (PRINTFILE) PAGE;
   PUT FILE (PRINTFILE) LIST(HEADER_LINE);
   PUT FILE (PRINTFILE) SKIP(2);
   END;
```

The ON-unit for the ENDPAGE condition for the file PRINTFILE outputs a page eject and a header line for the new output page.

To cause PL/I to ignore the ENDPAGE condition when a lot of output is written to a terminal, you can use the following ON-unit, which contains only the null statement:

```
ON ENDFILE(SYSPRINT);
```

An ON-unit established to handle end-of-page conditions can reference the ONFILE built-in function to determine the name of the file constant for which the condition was signaled.

■ ON-Unit Completion

If the ON-unit does not transfer control elsewhere in the program, the line number is set to one and the program continues execution of the PUT statement. Note that:

- If the ENDPAGE condition was signaled during data transmission, the data is written on the new current line.
- If the ENDPAGE condition was caused by a LINE or a SKIP option on the PUT statement, then the action specified by these options is ignored on return.

An ENDPAGE condition can occur only once per page of output. If the ON-unit specified does not specify a new page, then execution and output continue. The current line number can increase indefinitely; PL/I does not signal the ENDPAGE condition again. If, however, a LINE option on a PUT statement specifies a line number that is less than the current line, a new page is output and the current line is set to one.

■ Default PL/I Action

If the ENDPAGE condition is signaled during file processing, PL/I starts output on a new page and continues processing. If the ENDPAGE condition is signaled as a result of a SIGNAL statement, the statement following the SIGNAL statement is executed and no page is output by default.

ENTRY Attribute

The ENTRY attribute declares a constant or variable whose value is an entry point, and it describes the attributes of the parameters (if any) that are declared for the entry point. The format of the ENTRY attribute is:

```
ENTRY [ (parameter-descriptor, ...) ]
      [ OPTIONS (VARIABLE) ]
      [ RETURNS (returns-descriptor) ]
```

parameter-descriptor

A set of attributes describing a parameter of the specified entry. (**See also** "Procedure — Entry Points.") Attributes describing a single parameter must be separated by spaces; sets of attributes (each set describing a different parameter) must be separated by commas. Parameter descriptors are not allowed if the ENTRY attribute is within a RETURNS descriptor (**see** "RETURNS Attribute and Option" for more information on RETURNS descriptors).

The following rules apply to the specification of a parameter descriptor for an array or structure:

- If the parameter is an array, the dimensions must be specified first; otherwise, the attributes can be specified in any order.
- If the parameter is a structure, the level number must precede the attributes for each member.
- Extents for any parameter may be specified using only integer constants, restricted integer expressions, or asterisks (*).
- No storage class attributes may be specified.

OPTIONS (VARIABLE)

An option indicating that the specified external procedure can be invoked with a variable number of arguments. At least one parameter descriptor must be specified following the ENTRY keyword if OPTIONS(VARIABLE) is specified.

This option is provided for use in calling non-PL/I procedures. For complete details on using OPTIONS (VARIABLE), **see** the *VAX-11 PL/I User's Guide*.

RETURNS (returns-descriptor)

An option giving, for an entry that is invoked as a function reference, the data type attributes of the function value returned. (**See also** "RETURNS Attribute and Option.") For entry points that are invoked by function references, the RETURNS attribute is required; for procedures that are invoked by CALL statements, the RETURNS attribute is invalid.

The ENTRY attribute without the VARIABLE attribute implies the EXTERNAL attribute (and implies that the declared item is a constant), unless the ENTRY attribute is used to declare a parameter.

You must declare all external entry constants with the ENTRY attribute. When you declare an external entry constant, you must also specify the RETURNS attribute if the constant will be used to invoke a function. The RETURNS attribute indicates that the entry point is invoked via a function reference and defines the data type of the value it returns. **See** "RETURNS Attribute and Option."

■ Restrictions

- Internal entry constants must not be declared with the ENTRY attribute in the procedure to which they are internal. Internal entry constants are declared implicitly by the labels on the PROCEDURE or ENTRY statements of an internal procedure.
- The ENTRY attribute conflicts with all other data type attributes.

■ Example

```
DECLARE COPYSTRING ENTRY (CHARACTER (40) VARYING,
                          FIXED BINARY(7))
        RETURNS (CHARACTER(*));
```

This declaration describes the external entry COPYSTRING. This entry has two parameters: (1) a varying-length character string with a maximum length of 40 and (2) a fixed-point binary value. The RETURNS attribute indicates that COPYSTRING is invoked as a function and that it returns a character string with any length.

Entry Data

Entry constants and variables are used to invoke procedures through specified entry points. An entry value specifies an entry point and a block activation of a procedure.

■ Entry Constants

Entry constants are declared by writing labels on PROCEDURE or ENTRY statements.

Internal entry constants are declared by writing labels on PROCEDURE or ENTRY statements whose procedure blocks are nested in another block. An internal entry constant can be used anywhere within its scope to invoke its procedure block.

External entry constants are declared either by writing labels on PROCEDURE or ENTRY statements that belong to external procedures, or by explicitly declaring the name with the ENTRY attribute. An external entry constant can be used to invoke its procedure block from any program location that is within its scope. Its scope is either the scope of its declaration (as a label) or the scope of a DECLARE statement for the constant.

In DECLARE statements, external entry constants are declared with the ENTRY attribute. The declaration must agree with the actual entry point. That is, the declaration of the external entry constant must contain parameter descriptors for any parameters specified at the entry point, and, if the entry constant is to be used in a function reference, the declaration must have a returns descriptor describing the returned value. For the syntax and rules governing parameter descriptors, **see** "ENTRY Attribute." For the syntax and rules governing returns descriptors, **see** "RETURNS Attribute and Option."

■ Entry Values

Whenever a reference to an entry constant is interpreted, the result is an entry value. An entry value has two components:

1. The first component designates an entry point of a procedure.
2. The second component designates an activation of the block in which the entry point is declared (that is, the block in which the entry point's name appears as the label of a PROCEDURE or ENTRY statement). This block activation is the current block activation if the entry point belongs to the current block. If the entry point belongs to a containing block, the activation is on the chain of parent activations that ends at the current block activation. (For additional details on block activations, **see** "Block.")

No conversions are defined between entry data and other data types. An entry variable can be assigned only the value of an entry constant or the value of another entry variable. The only operations that are valid for entry data are comparisons for equality (=) and inequality (^=). Two entry values are equal if they refer to the same entry point in the same block activation.

■ Entry Variables

Entry variables are variables (including parameters) that take entry values. If the VARIABLE attribute is specified with the ENTRY attribute in a DECLARE statement, the declared identifier is an entry variable. You can assign an entry constant to an entry variable, or you can assign to it the value of another entry variable.

When an entry variable is used to invoke a procedure, its declaration must agree with the definition of the entry point. If the value you assign to an entry variable specifies an entry point with parameters, the parameters must be described with parameter descriptors in the declaration of the variable. If the assigned value specifies an entry point that is invoked as a function, then the declaration of the entry variable must have a RETURNS attribute that describes the data type of the returned value.

The scope of an entry variable name can be either INTERNAL or EXTERNAL. If neither EXTERNAL nor INTERNAL is specified with ENTRY VARIABLE, the default is INTERNAL. (**See also** "Scope of Names.")

The entry variable can be used to represent different entry points during the execution of the PL/I program. For example:

```
DECLARE E ENTRY VARIABLE,
        (A,B) ENTRY;

        E = A;
        CALL E;
```

In this example, the entry constant A is assigned to the entry variable E. The CALL statement results in the invocation of the external entry point A.

You can also declare arrays of entry variables. The following example shows an array of external functions:

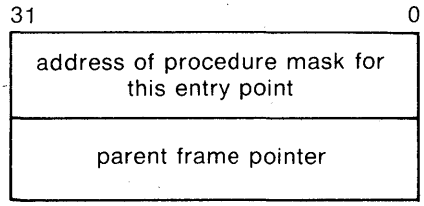
```
DECLARE EXTRACT(10) ENTRY (FIXED,FIXED)
                    VARIABLE RETURNS (FLOAT),
        GETVAL FLOAT;

        GETVAL = EXTRACT(3)(1,3);
```

This assignment statement references the third element of the array EXTRACT. When the statement is executed, this array element must contain a valid entry value.

Exercise caution using static entry variables. The value of a static entry variable is valid only as long as the block in which that value was declared is active.

■ Internal Representation of Variable Entry Data



ENTRY Statement

The ENTRY statement defines an alternate entry point to a procedure. Its format is:

entry-name: ENTRY [(parameter,...)]
[RETURNS (returns-descriptor)] ;

entry-name

A 1- to 31-character identifier for the entry point. Specifying the entry name declares the name as an entry constant. The scope of the name is external if the ENTRY statement is contained in an external procedure and internal if it is contained in an internal procedure.

parameter,...

One or more parameters that the procedure requires at this entry point. Each parameter specifies the name of a variable declared in the block to which this ENTRY statement belongs. The parameters must correspond, one by one, with arguments specified for the procedure when it is invoked via this ENTRY statement.

For more information, **see** "Parameters and Arguments."

RETURNS (returns-descriptor)

An option giving, for an entry that is invoked as a function reference, the data type attributes of the function value returned. (**See also** "RETURNS Attribute and Option.") For entry points that are invoked by function references, the RETURNS option is required; for procedures that are invoked by CALL statements, the RETURNS option is invalid.

■ Restrictions

An ENTRY statement is not allowed in a begin block, in an ON-unit, or in a DO-group except for a simple DO.

For more information on entry data, **see** "Entry Data." For more information on entry points, **see** "Procedure."

ENVIRONMENT Attribute

The ENVIRONMENT file description attribute specifies options that:

- Define file characteristics that are specific to the VAX-11 file system
- Request special processing not available in the standard PL/I language

The format of the ENVIRONMENT attribute is:

ENVIRONMENT(option,...)

option,...

One or more keyword options, separated by commas. Options are summarized below. The options are described in detail in the *VAX-11 PL/I User's Guide*.

■ Specifying Values for ENVIRONMENT Options

All ENVIRONMENT options may be specified in DECLARE and OPEN statements. Certain disposition options may also be specified in a CLOSE statement.

Options that require values may be specified using only literal constants in DECLARE statements; in an OPEN or CLOSE statement, they may be specified using expressions or literal constants.

Any option that does not require a value may optionally be specified with a Boolean constant or expression that indicates whether the option is to be enabled (if true) or disabled (if false). For example:

```
DECLARE IFDELETE BIT(1);
   .
   .
OPEN FILE (XYZ) ENVIRONMENT(DELETE(IFDELETE));
```

This DELETE option specifies a Boolean variable whose value may be true or false at run-time. Boolean values must be specified using only constants in a DECLARE statement; in an OPEN statement or CLOSE statement, Boolean values may be specified using constants or expressions.

Options that require variable references may be specified only on OPEN statements.

■ Alphabetic List of Options

An item with an asterisk (*) indicates an option that may be specified in a CLOSE statement.

APPEND
*BATCH
BLOCK_BOUNDARY_FORMAT
BLOCK_IO
BLOCK_SIZE(expression)
BUCKET_SIZE(expression)
CARRIAGE_RETURN_FORMAT
CONTIGUOUS
CONTIGUOUS_BEST_TRY
CREATION_DATE(variable-reference)
CURRENT_POSITION
DEFAULT_FILE_NAME(character-expression)
DEFERRED_WRITE
*DELETE
EXPIRATION_DATE(variable-reference)
EXTENSION_SIZE(expression)
FILE_ID(variable-reference)
FILE_ID_TO(variable-reference)
FILE_SIZE(expression)
FIXED_CONTROL_SIZE(expression)
FIXED_CONTROL_SIZE_TO(variable-reference)
FIXED_LENGTH_RECORDS
GROUP_PROTECTION(character-expression)
IGNORE_LINE_MARKS
INDEX_NUMBER(expression)
INDEXED
INITIAL_FILL
MAXIMUM_RECORD_NUMBER(expression)
MAXIMUM_RECORD_SIZE(expression)
MULTIBLOCK_COUNT(expression)
MULTIBUFFER_COUNT(expression)

NO_SHARE
OWNER_GROUP(expression)
OWNER_MEMBER(expression)
OWNER_PROTECTION(character-expression)
PRINTER_FORMAT
READ_AHEAD
READ_CHECK
RECORD_ID_ACCESS
RETRIEVAL_POINTERS(expression)
*REWIND_ON_CLOSE
REWIND_ON_OPEN
SCALARVARYING
SHARED_READ
SHARED_WRITE
*SPOOL
SUPERSEDE
SYSTEM_PROTECTION(character-expression)
TEMPORARY
TRUNCATE
WORLD_PROTECTION(character-expression)
WRITE_BEHIND
WRITE_CHECK

Error and Condition Handling

All error conditions that occur during the execution of PL/I run-time procedures result in the interruption of the program and a signal that indicates the type of error, or condition, that occurred.

When an error is signaled, PL/I attempts to locate a user-written program unit, called an ON-unit, to handle the condition. An ON-unit is established for a specific condition by means of an ON statement. If no ON-unit exists for a specific condition, PL/I performs a default action, which in most cases results in the termination of the program.

PL/I conditions have language keywords, or ON condition names. For example, the keyword ENDFILE is the name of the condition that is signaled when an end-of-file is encountered during an input operation. Thus, a program could handle an end-of-file condition for a given file as follows:

```
DECLARE INFILE FILE RECORD INPUT;

ON ENDFILE (INFILE) GOTO LAST;

OPEN FILE (INFILE);
```

For details on condition handling, **see** "ON Conditions and ON-Units." For additional information on end-of-file handling, **see** "ENDFILE Condition Name."

ERROR Condition Name

The ERROR condition name can be specified in an ON, SIGNAL, or REVERT statement to designate an error condition or ON-unit.

PL/I signals the ERROR condition in the following contexts:

- When a condition occurs for which the default PL/I action is to signal ERROR
- When the SIGNAL ERROR statement signals the condition
- When there is a default PL/I ON-unit and a condition is signaled for which there is no corresponding ON-unit

When any condition is signaled for which no specific ON-unit is established, the default PL/I action for all conditions except ENDPAGE is to signal the ERROR condition.

When any ON-unit is executed, the ON-unit can reference the built-in function ONCODE. This function returns the numeric condition value associated with the specific error that signaled the condition.

■ ON-Unit Completion

If an ERROR ON-unit does not handle the condition, the program is terminated at the completion of the ON-unit.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement." For more details on condition handling in the VAX/VMS environment, **see** the *VAX-11 PL/I User's Guide*.

EXP Built-In Function

The EXP built-in function returns a floating-point value that is the base *e* to the power of an arithmetic expression x. The computation is performed in floating point. The format of the function is:

```
EXP(x)
```

Exponentiation

Double asterisks (**) indicate exponentiation in an expression; the result is the value of the first operand raised to the power of the second operand. Both operands must have arithmetic data types.

■ Conversion of Operands

If the second operand is not a decimal integer constant, both operands are converted to FLOAT BINARY.

■ Precision of the Result

If the operation is expressed as x**y and if y is a positive decimal integer constant, the following rules apply to the result precision based on the data type of x and the value of y:

- where x is FIXED BINARY(p) and ((p+1)*y-1) ≤ 31, the result has the fixed binary precision:

  ((p+1)*y-1)

- where x is FIXED DECIMAL(p,q) and ((p+1)*y-1) ≤ 31, the result has the fixed decimal precision:

  ((p+1)*y-1)

  and scale factor:

  q*y

In all other cases, the operands are converted to floating point as described above. The result is a floating-point binary value whose precision is the maximum precision of the converted operands.

Expression

An expression is a representation of a value or of the computation of a value. In a PL/I program, you can use expressions to:

- Indicate constant values or scalar variables. For example:

```
B = 55;
NAME = 'HECTOR';
B = A;
```

- Perform algebraic or logical calculations on variables or constants. For example:

```
B = A + 10;
C = A + B * 40;
B = ^A;
COMMON = A & B;
```

- Compare the values of two or more expressions and obtain a Boolean result. For example:

```
IF A < B THEN C = 10;
IF NAME = SAVED_NAME THEN GOTO REPEAT;
```

- Concatenate character- or bit-string values. For example:

```
NAME = FIRST_NAME||LAST_NAME;
```

All expressions except simple constants and references consist of an operator and one or more operands. Each operator requires operands of specific types (either arithmetic, character string, or bit string) and produces a result of a specific type. The operands may be constants, variable references, function references, or other expressions, so long as they are objects of the type required by the operator.

Built-in functions may also be considered operators in this sense, and their arguments, operands.

All PL/I expressions and functions have scalar results.

Arithmetic expressions must have arithmetic operands. **See** "Arithmetic Operator," "Addition," "Subtraction," "Multiplication," "Division," and "Exponentiation."

Logical expressions must have bit-string operands, and all logical expressions have bit-string results. **See** "Logical Operator."

Relational, or comparison, expressions must have two operands of the same type. All relational expressions have Boolean results of type BIT(1), where '0'B signifies "false" and '1'B signifies "true." **See** "Relational Operator."

Concatenation expressions must have two string operands of the same type (bit or character). The result is a string of the operands' type. **See** "Concatenation Operator."

■ Expression Evaluation and Precedence of Operations

Expressions may be evaluated in any order, with the following qualifications:

- Some PL/I operators take precedence over others used in the same expression. Operations with higher priority are evaluated first, and their results are used as single operands. The rules of precedence usually guarantee an algebraically correct result without the use of parentheses. All built-in functions are of equal priority. **See** "Operator" for a table listing the priorities of PL/I operators.
- Any expression can be enclosed in parentheses, to override the usual rules of precedence. Expressions at the deepest level of nested parentheses are always evaluated first, and their results are used as single operands.
- Exponential operations of the form A**B**C are evaluated from right to left.
- The run-time evaluation of a logical expression may be terminated as soon as its result is known. For instance, evaluation of the expression

```
A & USER_FUNCTION(ALPHA,BETA)
```

  may be terminated without evaluating the USER_FUNCTION reference if the evaluation of A results in a "false" Boolean value.
- If a function referenced in an expression executes a nonlocal GOTO statement, the expression is not evaluated further.

■ Conversion of Operands

This section applies only to arithmetic operations, which must always have arithmetic operands. (However, **see also** "Built-In Conversion Functions," below.)

Even though arithmetic operands can be of different arithmetic types, all operations must be actually performed on objects of the same type. Any set of operands of different arithmetic types has an associated derived type, as follows:

- If any operand has the attribute BINARY, the derived base is BINARY. Otherwise, the derived base is DECIMAL.
- If any operand has the attribute FLOAT, the derived scale is FLOAT. Otherwise, the derived scale is FIXED.

Table E–1 gives the derived data type for two arithmetic operands of different types. (Note that the types derived from FIXED DECIMAL in Table E–1 also are derived when one operand is pictured.)

Table E–1: Derived Types

Operand-1 Type	Operand-2 Type	Derived Type
FIXED BINARY	FLOAT BINARY	FLOAT BINARY
FIXED BINARY	FLOAT DECIMAL	FLOAT BINARY
FIXED DECIMAL	FLOAT DECIMAL	FLOAT DECIMAL
FIXED DECIMAL	FLOAT BINARY	FLOAT BINARY
FIXED BINARY	FIXED DECIMAL	FIXED BINARY

NOTE

If one operand is fixed-point binary, the other should not be fixed-point decimal with a nonzero scale factor (or pictured with fractional digits). Because VAX–11 PL/I does not support fixed-point binary data with fractional digits, the conversion of the decimal operand would result in the loss of its fractional digits.

Table E–2 gives the precision resulting from the conversion of an operand to its derived type. The values p and q are known as the converted precision of an operand and are based on the values p and q of the source operand.

Table E–2: Converted Precision as a Function of Target and Source Attributes

Target Data Type	Source Data Type[1]			
	Binary Fixed	Decimal Fixed	Binary Float[2]	Decimal Float[2]
Binary Fixed	p	min(ceil(p*3.32)+1,31)		
Decimal Fixed	min(ceil(p/3.32)+1,31) scale factor: 0	p scale factor: q		
Binary Float	min(p,113)	min(ceil(p*3.32),113)	p	min(ceil(p*3.32),113)
Decimal Float	min(ceil(p/3.32),34)	min(p,34)	min(ceil(p/3.32),34)	p

1. The constant 3.32 is an approximation of log2(10), the number of bits required to represent a decimal digit.
2. The blank entries are cases that never occur in the language.

All arithmetic operations except exponentiation are performed in the derived type of the two operands. Note that the two converted operands, although they have the same derived base and scale, may have different values for p and q, as shown by Table E-2. Exponential operations are performed in a data type that is based on the derived type of the operands; for details, **see** "Exponentiation."

All operations, including exponentiation, have results of the same type as the type in which they are performed. The precision and scale factor of the result differ depending on the operation being performed. For details, **see** "Addition," "Subtraction," "Multiplication," "Division," "Exponentiation," "Built-In Function," or the section on an individual built-in function.

When the result of an arithmetic operation is assigned to a target variable, the target variable can be of any computational type. The result is converted to the target type, following the rules in "Conversion of Data."

■ Built-In Conversion Functions

The built-in conversion functions FLOAT, FIXED, BINARY, and DECIMAL can take arguments that are either arithmetic or string expressions. They are, in fact, often used to convert an operand to the type required in a certain context, for instance, to convert a bit string to an arithmetic value for use as an arithmetic operand.

For the purpose of these functions, and a few other contexts, derived arithmetic attributes are also defined for bit- and character-string expressions:

- The derived type of a bit string is fixed-point binary; its converted precision is 31.
- The derived type of a character string is fixed-point decimal; its converted precision is also 31.

These derived attributes are used to determine the precision of values returned by the conversion functions if no precision is specified in the functions' argument lists. Of course, the value of a string argument must also be convertible to the result type; for instance, '1.333' is convertible to arithmetic, but 'ABCD' is not. For more information, **see** "Conversion of Data" and the sections on the FLOAT, FIXED, BINARY, and DECIMAL built-in functions.

Extent

An extent gives a length or dimension of a variable. The rules for specifying extents apply to the length of a character-string or bit-string variable and to the dimensions of an array. The length of a character string or a bit string is the number of characters or bits of its value. The dimensions of an array are expressed in terms of bounds, and the rules for specifying extents apply to those bounds. These rules are:

- If an extent is specified in a static variable declaration, the extent must be specified as an integer constant or as a restricted integer expression. (A restricted integer expression is an expression consisting solely of integer constants, identifiers given values by %REPLACE statements, and any of the operators +, -, *, or the DIVIDE built-in function.)

- If an extent is specified in the declaration of a parameter, in a parameter descriptor, or in a returns descriptor, the extent may be specified as an integer constant, as a restricted integer expression, or as an asterisk (*). If one dimension of an array is specified with an asterisk, all must be specified with asterisks.
- If the extent is specified for an automatic, based, or defined variable it may be specified as an integer constant or as an expression.
- The maximum value that can be specified for an extent is 500 million bytes.

EXTERNAL Attribute

The EXTERNAL attribute declares an external name, that is, a name whose value can be known to blocks outside the block in which it is declared.

The format of the EXTERNAL attribute is:

```
{ EXTERNAL }
{ EXT      }
```

The EXTERNAL attribute is implied by the FILE, GLOBALDEF, and GLOBALREF attributes. EXTERNAL is also implied by declarations of entry constants (that is, declarations that contain the ENTRY attribute but not the VARIABLE attribute). For variables, the EXTERNAL attribute implies the STATIC attribute.

■ Restrictions

The following rules apply to the use of external names:

- The EXTERNAL attribute directly conflicts with the AUTOMATIC, BASED, and DEFINED attributes.
- The EXTERNAL attribute cannot be applied to minor structures, members of structures, parameters, or descriptions in an ENTRY or RETURNS attribute.
- The EXTERNAL attribute is invalid for variables that are the parameters of a procedure.
- If a variable is declared as EXTERNAL STATIC INITIAL, all blocks that declare the variable must initialize the variable with the same value.
- The maximum number of external variables that can be declared in a single compilation is 254. This number includes file constants.

External Procedure

An external procedure is one whose text is not contained within another procedure. An external procedure must be explicitly declared with the ENTRY attribute before it can be invoked or referenced.

See "Procedure."

External Variable

An external variable provides a way for external procedures to share common data. All declarations that refer to an external variable must also declare the variable with the attribute EXTERNAL and with identical data type attributes. Figure E–1 illustrates how procedures can use external variables.

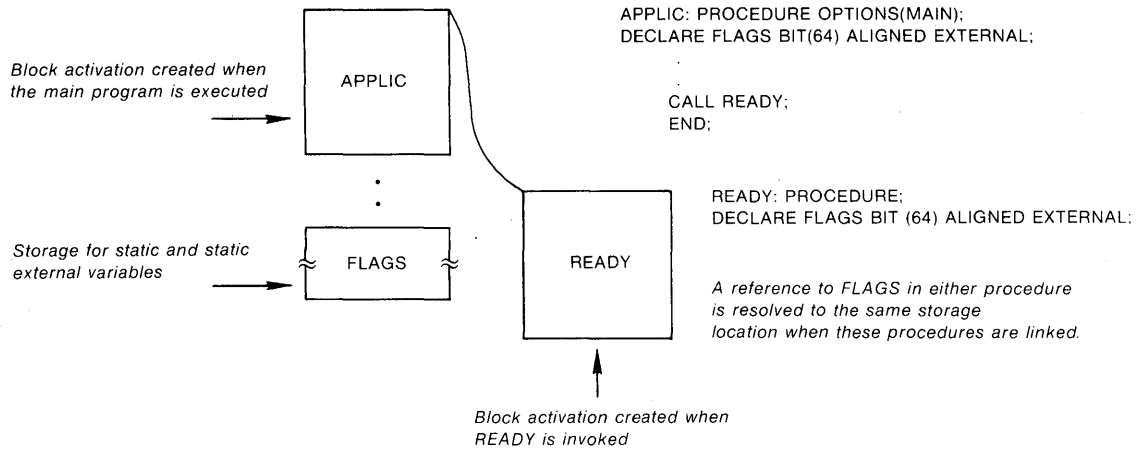


Figure E–1: External Variables

The VAX/VMS linker allows more control than does PL/I over the definition and allocation of external variables. With the GLOBALDEF attribute, you can define the allocation and initialization of an external variable in a single module. Other PL/I modules can then declare the variable with the GLOBALREF attribute and with no INITIAL attribute.

Further control is provided by the VALUE attribute, which can be used in conjunction with GLOBALDEF and GLOBALREF. A variable declared in this way is actually a constant whose value is used immediately in instructions generated by the compiler.

Declarations of names with the attributes FILE, GLOBALREF, and GLOBALDEF are not subject to the limit of 254 per compilation.

For more information, **see** "GLOBALDEF Attribute," "GLOBALREF Attribute," and "VALUE Attribute." For more information on the use of the linker, **see** the *VAX-11 PL/I User's Guide*.

F

F Format Item

The F format item describes the representation of a fixed- or floating-point value as a decimal fixed-point number in a stream.

The form of the item is:

F(w[,d])

w

A nonnegative integer that specifies the total width in characters of the field in the stream.

d

An optional nonnegative integer that specifies the number of fractional digits in the stream representation.

The interpretation of the F format item on input and output is given below. For a general discussion of format items, **see** "Format Items and Their Uses."

■ Input with GET EDIT

Used with GET EDIT, the F format item acquires a fixed-point decimal value from the next w characters in the stream and assigns it to an input target of any computational type. For input, fixed-point decimal values can be represented in the stream in the following forms:

number
sign number

The number is a fixed-point decimal constant and the sign is a + symbol or – symbol.

The following are valid representations:

124333
-123333
-123.333

The ERROR condition is signaled if the field is not blank and does not contain one of the valid representations shown above; otherwise, the fixed-point decimal number is extracted from the field and is assigned to the input target, with any necessary conversions. If the number includes a decimal point, it overrides the specification of d. If no decimal point is included, d specifies the number of fractional digits. If d is omitted, it is assumed to be zero.

The integer w should be only large enough to include the number, the optional decimal point in the number, and the optional sign. If w is too small, the stream representation is truncated on the right. If w is too large, extra characters are acquired, which may include invalid syntax.

If w is zero, a null character string is converted and assigned to the input target, and no operation is performed on the stream.

Spaces can precede or follow the number in the stream and are ignored. If the entire string contains spaces or is a null string, the fixed-point decimal constant 0 is converted and assigned to the input target.

■ Output with PUT EDIT

Used in a PUT EDIT statement, the F format item converts an output source of any computational type to one of the following forms for representation in the stream:

```
integer
integer.fractional-digits
-integer.fractional-digits
```

Typical representations are:

```
3234
0.23432
3.33
-3234.33
```

The decimal value is rounded before being written out. If d is omitted from the format item, the decimal point is not shown, and only the integral part of the number is shown.

If d is larger than the number of fractional digits to be output, trailing zeros are appended to the output number. All leading zeros to the left of the decimal point are suppressed unless the integral part of the number is zero, in which case one 0 appears to the left of the decimal point.

To account for negative values with fractional digits, the specified width integer should be two greater than the number of digits to be represented: one character for the preceding minus sign and one for the decimal point in the number.

If the number's representation is shorter than the specified field, the representation is right-justified in the field and the number is extended on the left with spaces.

If the field is too narrow to represent the integral portion of the output number, the ERROR condition is signaled.

■ Examples

The tables below show the relationship between the internal and external representations of numbers that are read or written with the F format item.

Input Examples

The "input stream" shown in the table is a field of characters beginning at the current position in the stream and continuing to the right. The target type is the type of the variable to which the input value is assigned.

Format Item	Input Stream	Target Type	Target Value
F(10,2)	-123456.78...	DECIMAL(10,2)	-123456.78
F(10,4)	-1234.56789...	DECIMAL(10,2)	-1234.56
F(8,5)	-.123456789...	DECIMAL(5,5)	-0.12345
F(10)	1234.56789...	FLOAT DEC(7)	1.234568E+03

Output Examples

The output source value shown in the table is either a constant or the value of a variable that is written out with the associated format item.

Output Source Value	Format Item	Output Value
-12.234	F(3,0)	-12
-12.234	F(6,2)	-12.23
-12.234	F(7,3)	-12.234
-1.23456E3	F(8)	△△△-1235
-1.23456E3	F(8,2)	-1234.56
'1000'B3	F(4)	△512
'1000000000000000'B	F(5)	32768
'100000'B3	F(5)	32768
'ABCEDF'B4	F(10)	△△11259615

File

A PL/I file is a source of input data or a target for output data. All I/O operations must specify the name of the PL/I file on which the operation is to be performed; the name of a PL/I file is declared in a DECLARE statement. When a file is opened, it must also be associated with an external, or physical, file or device.

PL/I provides two distinct types of I/O processing. Each type of processing handles input and output data in a different manner, and each has a unique set of input/output statements. These types of input/output are:

- Stream input/output, or simply stream I/O. The stream I/O statements are GET and PUT.
- Record input/output, or simply record I/O. The record I/O statements are READ, WRITE, DELETE, and REWRITE.

When a file is read or written using stream I/O, the data is treated as if it forms a continuous stream. Individual fields of data within the stream are delimited by commas, spaces, and record boundaries. A stream I/O statement specifies one or more fields to be processed in a single operation.

When a file is read or written using record I/O, however, a single record is processed upon the execution of an I/O statement.

The following subsections discuss input/output concepts that apply to both stream and record I/O. Additional details on each of these forms of I/O can be found under the entries "Stream Input/Output" and "Record Input/Output."

■ File Declarations

A file declaration specifies an identifier, the FILE attribute, and one or more file description attributes that describe the type of input/output operation that will be used to process the file.

A file is denoted in an input/output statement by a FILE option:

FILE(file-reference)

where file-reference is the name specified in the file's declaration. For example:

```
DECLARE INFILE FILE SEQUENTIAL INPUT;
OPEN FILE(INFILE);
```

Here, INFILE is the name of a file constant. A file constant is an identifier declared with the FILE attribute and without the VARIABLE attribute. Except for the default file constants SYSIN and SYSPRINT, all files must be declared before they can be opened and used.

By default, all file constants have the EXTERNAL attribute. Any external procedure that declares the identifier with the FILE attribute and without the INTERNAL attribute can access the same file constant and therefore the same physical file.

■ File Variables

In PL/I, you can also refer to files using file variables and file-valued functions. For example:

```
DECLARE ANYFILE FILE VARIABLE;

ANYFILE = INFILE;
OPEN FILE(ANYFILE);
```

If INFILE is declared as in the preceding example, the OPEN statement opens the file INFILE.

A file variable can also be given a value by passing a file constant as an argument or by return of a file constant as the value of a function. For example:

```
GETFILE: PROCEDURE (PRINTFILE);
DECLARE PRINTFILE FILE VARIABLE;
```

This file variable is given a value when the procedure GETFILE is invoked.

FILE Attribute

The FILE attribute declares a file constant or file variable.

The FILE attribute is implied by any of the following file description attributes:

DIRECT	OUTPUT	SEQUENTIAL
ENVIRONMENT	PRINT	STREAM
INPUT	RECORD	UPDATE
KEYED		

If the VARIABLE attribute is not specified, the FILE attribute declares a file constant. If the INTERNAL attribute is not specified, the file has the EXTERNAL attribute by default. All external declarations of a file constant are associated with the same file.

■ Restrictions

- The FILE attribute conflicts with all other data type attributes.
- If the VARIABLE attribute is not specified, no storage class attributes are allowed.
- If the FILE attribute is used to declare a variable or parameter, no file description attributes may be specified.

File Data

A PL/I file, or file constant, is represented by a file control block. A file control block is an internal data structure maintained by PL/I.

A file variable is represented internally as a longword that contains a pointer to a file control block. The value of the file variable, when evaluated, is the address of the file control block for the file with which the variable is currently associated.

File Description Attributes and Options

The operations that can be performed on an open file depend on both the attributes of the file and the physical organization of the file or device that is associated with the PL/I file constant.

Attributes can be specified for a file constant in its declaration or its opening. The file description attributes specified in the DECLARE statement for a file are permanent attributes. The file description attributes used in a particular opening of a file are obtained by merging the permanent attributes and attributes specified at the opening. For example:

```
DECLARE TAPEIO FILE RECORD;
OPEN FILE(TAPEIO) OUTPUT;
```

The DECLARE statement specifies that a permanent attribute of the file is RECORD, that is, it will be processed using record I/O statements. The OPEN statement adds the attribute OUTPUT to the file's description.

The rules governing the merging of attributes during file opening are given under the heading "Opening a File." Particular implications of using a specific file description attribute are given under the entry for the attribute.

The file description attributes are summarized in Table F–1. These attributes can be specified on either DECLARE or OPEN statements.

Table F-1: Summary of File Description Attributes

Attribute	Meaning
DIRECT	Records in the file will be accessed randomly only.
INPUT	The file is an input file and will only be read.
KEYED	Records in the file will be accessed by key.
OUTPUT	The file is an output file and will only be written.
PRINT	The file will be output on a printer or terminal.
RECORD	The file will be accessed using record I/O statements.
SEQUENTIAL	Records in the file will be accessed sequentially.
STREAM	The file will be accessed using stream I/O statements.
UPDATE	The file will be accessed for both reading and writing and records may be rewritten and deleted.

■ File Access Modes

Most file description attributes relate to the way in which a file will be used, for example, whether it will be an input or an output file, or whether it will be used for record I/O or stream I/O. Table F-2 shows the valid combinations of access modes for files and the relationship of each combination to the file organizations supported by VAX-11 PL/I.

Table F-2: File Access Attributes

Attributes Specified	Attributes Implied	Valid Devices and File Organizations	Usage
PRINT	STREAM OUTPUT	Any output device or file except indexed	Individual data values are written with PUT statements that convert the values to character strings and automatically format the strings into lines, or records. A PUT statement may fill part or all of one or more lines. Data conversion and alignment within lines may use the default processing provided by the PUT LIST form of the PUT statement or may be explicitly controlled by format specifications in the PUT EDIT form of the PUT statement. The output fields may be aligned to specific tab positions. The PAGESIZE and LINESIZE options may be specified to control the formatting of lines on pages. The ENDPAGE condition is signaled when the end-of-page is reached.

(Continued on next page)

Table F-2 (Cont.): File Access Attributes

Attributes Specified	Attributes Implied	Valid Devices and File Organizations	Usage
STREAM INPUT		Any input device or file except indexed	Individual data items are read by GET statements. A single GET statement may process all or part of one or more lines, or records. The format of an input field may be determined by the default processing provided by the GET LIST form of the GET statement or may be explicitly controlled by format specifications in the GET EDIT form of the GET statement.
STREAM OUTPUT		Any output device or file except indexed	This form of stream output is similar to that provided when PRINT is specified, except that tab positioning and page formatting are not provided. Moreover, when string values are written with the PUT LIST form of the PUT statement, they are enclosed in apostrophes. Files that are created with these attributes may be read back in using GET LIST statements when the file is opened with the STREAM and INPUT attributes.
SEQUENTIAL OUTPUT	RECORD	Any output device or file except indexed	Records may be added to the end of the file using WRITE statements. Each WRITE statement adds a single record to the file.
SEQUENTIAL INPUT	RECORD	Any input device or file	Records in the file are read using READ statements. Each statement reads a single record.
SEQUENTIAL UPDATE	RECORD	Relative, indexed, sequential disk [1]	READ statements read a file's records in order. PL/I maintains the current record, which is the record just read. This record may be replaced in a REWRITE statement. In a relative or indexed sequential file, the current record may also be deleted with a DELETE statement. Each statement processes a single record.
DIRECT OUTPUT	KEYED RECORD	Relative, indexed, sequential disk [1]	WRITE statements insert records into the file at positions specified by keys. Each statement inserts a single record.
DIRECT INPUT	KEYED RECORD	Relative, indexed, sequential disk [1]	READ statements specify records to be read randomly by key. Each statement reads a single record.
DIRECT UPDATE	KEYED RECORD	Relative, indexed, sequential disk [1]	READ, WRITE, and REWRITE statements specify records randomly by key. In a relative or indexed file, records may also be deleted by key.
KEYED SEQUENTIAL OUTPUT	RECORD	Relative, indexed, sequential disk [1]	WRITE statements insert records into the file at positions specified by keys. Each statement inserts a single record. This mode is identical to DIRECT OUTPUT.
KEYED SEQUENTIAL INPUT	RECORD	Relative, indexed, sequential disk [1]	READ statements access records in the file randomly by key or sequentially.
KEYED SEQUENTIAL UPDATE	RECORD	Relative, indexed, sequential disk [1]	Any record I/O operation is allowed except a WRITE statement that does not specify a key or a DELETE statement for a sequential disk file with fixed-length records.

1. The file must have fixed-length records.

■ Associating a PL/I File with a VAX/VMS File

The TITLE option of the OPEN statement specifies the name of the VAX/VMS file or device that is associated with the file. The name given in the TITLE option can be a VAX/VMS logical name or file specification or a PL/I variable whose value represents a VAX/VMS logical name or file specification. For example:

```
OPEN FILE (TAPEIO) TITLE('MT:');
```

This TITLE option specifies a magnetic tape device. **See** "TITLE Option." Additional information on file naming and logical names can be found in the *VAX-11 PL/I User's Guide.*

■ ENVIRONMENT Options

The ENVIRONMENT attribute can be used to specify properties of a file that are unique within the context of the VAX/VMS operating system. For example, you use the ENVIRONMENT attribute to specify the format of records in a file, the maximum record number for a relative file, and so on. You need to specify the ENVIRONMENT attribute only when you wish to take advantage of some special feature of the VAX/VMS file system, for example, if you want to define the number of buffers to use on input/output operations, or if the defaults applied to new files when they are created are not satisfactory.

For a list of these options, **see** "ENVIRONMENT Attribute." Complete details on the meanings of the options are given in the *VAX-11 PL/I User's Guide.*

FILE Option

The FILE option is specified in a stream or record I/O statement to designate the file upon which an operation is to be performed. The FILE option is required on all I/O operations except GET and PUT statements that access the default file constants SYSIN and SYSPRINT. The FILE option has the format:

FILE (file-reference)

file-reference

A reference to an identifier declared as a file constant, a scalar reference to a variable with the FILE attribute, or a function that returns a file value.

File Organization

A file organization defines the manner in which the data in a record file is arranged. The VAX-11 Record Management Services (RMS) support the following different file organizations:

- Sequential — a sequential file contains records that are arranged in serial order.

- Relative — a relative file contains numbered records that can be accessed by specifying the number.
- Indexed sequential — an indexed sequential file contains records that have one or more key fields and indexes that provide access to the records by key specification.

Operations on these files are normally performed using record I/O statements. Stream I/O statements can be used for any of these files in which all of the data is ASCII. Operations on files of each type are described individually, below. For a general discussion of the access modes that can be applied to each file organization, **see** "File Description Attributes and Options."

For complete details and examples of using various file organizations in VAX-11 PL/I, see the *VAX-11 PL/I User's Guide.*

■ Sequential Files

In VAX-11 PL/I, the term "sequential file" applies to the physical organization of the records in the file, and not to the manner in which the records will be accessed. The records can contain ASCII data or non-ASCII data and may be accessed using record I/O or stream I/O statements.

The records in a sequential file may have any of the following record formats:

- Variable length
- Fixed length
- Variable length with a fixed-length control area

In a sequential file with variable-length records, records may or may not be of the same length. This is the default record format for sequential files.

The properties and uses of sequential files with variable-length records with a fixed-length control area are described in the *VAX-11 PL/I User's Guide.*

To create a sequential file with fixed-length records, the ENVIRONMENT options FIXED_LENGTH_RECORDS and MAXIMUM_RECORD_SIZE must be specified. A sequential disk file with fixed-length records may be accessed randomly. In this case, the key is the relative record number of the record in the file, with the first record in the file being relative record number one.

■ Relative Files

A relative file contains a set of numbered records with numbers in the range of 1 to a maximum record number. A relative file has a fixed-length slot for each possible record number; not all slots need be filled at any one time. The size of each slot is set at the length of the maximum record size when the file is created.

Each record in the file has a unique number. Inserting and deleting records does not change the numbers of the other records.

Records may be accessed randomly or sequentially. Random access of a given record is performed by specifying the record number as a key in the KEY or KEYFROM option of a record I/O statement.

When a relative file is created, the maximum number of records that can be written to the file can be specified with the ENVIRONMENT option MAXIMUM_RECORD_NUMBER. If no maximum number is specified, there is no maximum; that is, the file may be of any size and the record numbers are not checked when new records are added.

■ Indexed Sequential Files

An indexed sequential file contains records that have a specifically defined structure and indexes. The structure of all records in the file is defined in terms of one or more key fields, each of which has a position in the record and a data type; no two records may have the same key. The key fields are determined when the file is created.

The file has an index for each key field. Records in the file can be accessed randomly by specifying a KEY or KEYFROM option that tells the value of a key. For example:

```
READ FILE(F) KEY('ABC') INTO (X);
```

This READ statement reads the record from the file F that has the character string ABC in the key field of the record.

In an I/O operation, PL/I automatically converts a key value specified in an I/O statement to the data type of the key value in the record.

When records in a file have more than one key field or index, there are a primary index and a number of alternate indexes. In the alternate indexes, duplicate instances of the same key are allowed. For example, in a key of names and addresses, a zip code field may be defined as an alternate key. Many records may have same value in the zip code key field.

The keys are numbered; the primary index is always numbered 0. To specify the index by which the record is to be located, you specify the INDEX_NUMBER option. For example:

```
READ FILE(F) KEY(12) INTO(X)
          OPTIONS (INDEX_NUMBER(2)) ;
```

Here, the READ statement uses the index numbered 2; the record with a key of 12 in this alternate index field is transferred into the variable X.

The INDEX_NUMBER option is necessary only to change indexes during file processing. By default, each operation uses the same index that was used for the most recent operation on the file. When a file is initially opened or when a WRITE statement specifies a KEYFROM option, the index number is set to the primary index, 0.

To access an indexed sequential file in PL/I, you can specify random or sequential access, or both. When an indexed sequential file is accessed sequentially, records are read based on the key values of the current index number.

If an index with alternate keys contains duplicate key values in the alternate keys, a random READ or DELETE operation accesses the first such record with the specified key. (Sequential processing can then be used to access the records with duplicate keys.) Records are always inserted into an indexed

sequential file based on the value of the primary key; thus, records that have duplicate alternate keys are inserted without respect to the values of the alternate keys.

FINISH Condition Name

The FINISH condition name can be specified in an ON, SIGNAL, or REVERT statement to designate a FINISH condition or a FINISH ON-unit.

PL/I signals the FINISH condition in the following contexts:

- When any procedure in the program executes the STOP statement
- When a procedure that specifies OPTIONS(MAIN) executes a RETURN statement, or, if the procedure does not execute a RETURN statement, when its corresponding END statement is executed
- When a program exits as a result of a call to the system procedures SYS$EXIT or SYS$FORCEX (Force Exit), or after the program is interrupted by an external CTRL key function
- When the SIGNAL FINISH statement signals the condition

The ways in which a PL/I program can be caused or forced to exit in the VAX/VMS environment are described in the *VAX-11 PL/I User's Guide*.

■ ON-Unit Completion

If a FINISH ON-unit does not execute a nonlocal GOTO, the program is terminated at the completion of the ON-unit.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement."

FIXED Attribute

The FIXED attribute indicates that the variable so declared is an arithmetic value with a fixed number of fractional digits. Such variables are called fixed-point (as opposed to floating-point) variables because the decimal point is fixed in relation to the representation of the value.

When you specify the FIXED attribute in a DECLARE statement, you can specify either the BINARY or DECIMAL attribute to indicate a binary or decimal fixed-point variable. You can specify the precision, which is the number of decimal or binary digits used to represent values of the variable. With fixed-point decimal data, you can also specify a scale factor that indicates how much of the precision is used for fractional digits. For example, the attributes FIXED BINARY(31) define a variable that takes fixed-point binary values of 1 to 31 bits. The attributes FIXED DECIMAL(10,2) define a variable that takes fixed-point decimal values of up to 10 decimal digits, two of which are fractional. PL/I supplies default attributes for attributes that you do not specify.

You can use binary and decimal fixed-point data as follows:

- Ordinarily, you use fixed-point binary data to represent integers. However, you can also use fixed-point decimal data, which can represent larger absolute values. Fixed-point binary data must have a zero scale factor, that is, must have no fractional digits. The precision of a fixed-point binary variable must be in the range 1–31. **See** "Fixed-Point Binary Data."
- You use fixed-point decimal data whenever arithmetic values must be precise to a specified number of fractional digits. For a fixed-point decimal value, the precision must be in the range 1–31 (decimal digits). The scale factor, if specified, must be greater than or equal to zero and less than or equal to the specified precision. If the scale factor is omitted, zero is used (that is, a decimal integer variable is declared). **See** "Fixed-Point Decimal Data."

The default values given for unspecified related attributes are:

Attributes Specified	**Defaults Supplied**
FIXED	BINARY (31)
FIXED BINARY	(31)
FIXED DECIMAL	(10,0)

■ Restrictions

The FIXED attribute directly conflicts with all data type attributes except BINARY and DECIMAL.

FIXED Built-In Function

The FIXED built-in function converts an arithmetic or string expression x to a fixed-point arithmetic value with a specified precision p and, optionally, a scale factor q. The scale factor q must be a nonnegative integer and must be zero if x is binary; if q is omitted, it is assumed to be zero. The precision p must be greater than zero and less than or equal to 31.

The format of the function is:

```
FIXED(x,p[,q])
```

■ Returned Value

The result type is fixed-point binary or decimal, depending on whether x is binary or decimal. (If x is a bit string, the result type is fixed-point binary; if x is a character string, the result type is fixed-point decimal.)

The expression x is converted to a value v of the result type, following the usual rules (**see** "Conversion of Data" for details). The returned value is v with precision p and scale factor q. If q is omitted, the returned value has the converted precision of x and the scale factor zero (**see** "Expression" for details). FIXEDOVERFLOW is signaled if appropriate.

Fixed-Point Binary Data

The attributes FIXED BINARY are used to declare binary integers in PL/I. The BINARY attribute is implied by FIXED. The declaration of a single fixed-point binary variable is of the form:

DECLARE identifier FIXED [BINARY] [(precision)];

identifier

The name used to refer to the variable.

precision

An integer from 1 to 31, giving the number of bits used to represent values of the variable. If you do not supply the precision, the default is 31. Depending on the precision you specify, either eight bits (a byte), 16 bits (a word), or 32 bits (a longword) are allocated; the high-order bit is used to represent the sign of a value.

Because fixed binary variables have a maximum precision of 31, fixed binary integers can have values only in the range of -2,147,483,648 through 2,147,483,647. An attempt to calculate a binary integer outside this range, in a context that requires an integer value, signals the FIXEDOVERFLOW condition.

There is no form for a fixed-point binary constant, although constants of other computational types are convertible to fixed-point binary. A fixed-point binary variable is usually given values by assigning to it an expression of another computational type or another fixed-point binary variable. **See** "Constant" and "Conversion of Data."

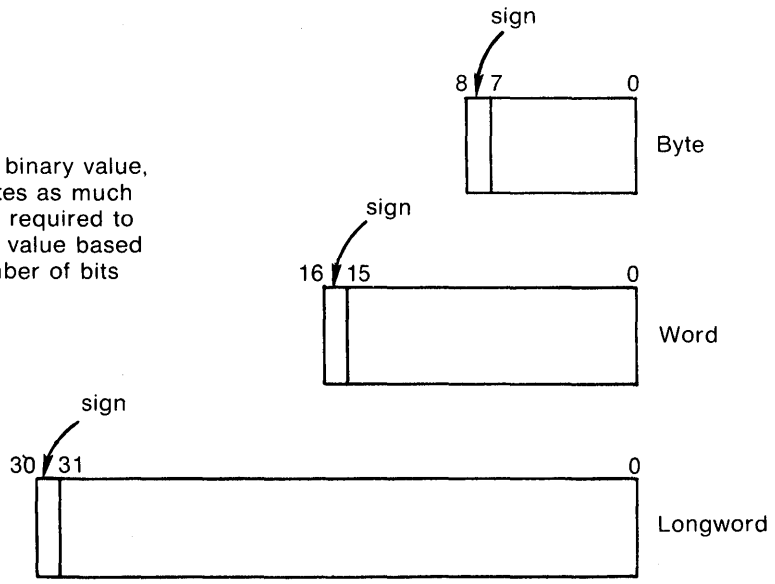
■ Restrictions

Some operations that are valid for other arithmetic data are invalid for fixed-point binary data:

- Fixed-point binary data cannot have a nonzero scale factor (that is, cannot have fractional digits). Consequently, you should not use the division operator (/) with two fixed-point binary operands, because the operation would normally result in fractional digits. Instead, use the DIVIDE built-in function.
- Operations that combine fixed-point decimal (including pictured) and fixed-point binary operands are performed in fixed-point binary in standard PL/I. Because fixed-point binary values cannot have fractional digits, you should not use a fixed-point binary operand in the same operation with a fixed-point decimal or pictured operand that has fractional digits. If you must perform such an operation, use the DECIMAL built-in function to convert the binary operand to decimal.

For more information, **see** "Integer Data" and "Fixed-Point Decimal Data."

■ Internal Representation of Fixed-Point Binary Data



For a fixed binary value, PL/I allocates as much space as is required to contain the value based on the number of bits needed.

Storage for fixed-point binary variables is always allocated in a byte, word, or longword. For any fixed-point binary value:

- If $1 \leq p \leq 7$, a byte is allocated.
- If $8 \leq p \leq 15$, a word is allocated.
- If $16 \leq p \leq 31$, a longword is allocated.

The binary digits of the stored value go from right to left in order of increasing significance; for example, bit 6 of a FIXED BINARY(7) value is the most significant bit and bit 0 is the least significant.

In all cases, the high-order bit (7, 15, or 31) is used to encode the sign.

Fixed-Point Decimal Data

Fixed-point decimal data is used in calculations where exact decimal values must be maintained, for example, in financial applications. Fixed-point decimal data with a scale factor of zero may also be used whenever integer data is required.

This discussion is divided into the following parts:

- Constants
- Variables
- Use in expressions
- Internal representation

■ Fixed-Point Decimal Constants

A fixed-point decimal constant can contain one or more of the decimal digits 0 through 9 with, optionally, a decimal point and/or a sign. If there is no decimal point, PL/I assumes that the decimal point is immediately to the right of the rightmost digit. Some examples of fixed-point decimal constants are:

```
12
4.56
12345.54
-2
.0004
01.
```

The precision (p) of a fixed-point decimal value is the total number of digits in the value. The scale factor (q) is the number of digits to the right of the decimal point, if any.

■ Fixed-Point Decimal Variables

The format of a declaration of a single fixed-point decimal variable is:

DECLARE identifier [FIXED] DECIMAL [(p[,q])];

identifier

The name to be used for the variable.

p

An integer constant giving the total number of decimal digits used to represent values of the variable. The value must be in the range:

$1 \leq p \leq 31$

q

An integer constant giving the number of fractional digits in values of the variable. The value must be in the range:

$0 \leq q \leq p$

If you omit p and q, the default values are p=10, q=0.

Some examples of fixed-point decimal declarations are:

```
DECLARE PERCENTAGE FIXED DECIMAL (5,2);
DECLARE TONNAGE FIXED DECIMAL (9);
```

■ Use in Expressions

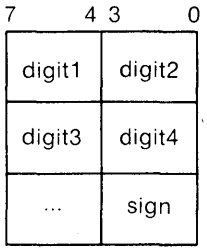
You cannot use fixed-point decimal data with a nonzero scale factor in calculations with binary integer variables. If you must use the two types of data together, use the DECIMAL built-in function to convert the binary value to a scaled decimal value before attempting an arithmetic operation. For example:

```
DECLARE I FIXED BINARY,
        SUM FIXED DECIMAL (10,2);

SUM = SUM + DECIMAL (I);
```

■ Internal Representation of Fixed-Point Decimal Data

Fixed decimal data is stored in packed decimal format. Each digit is stored in a half-byte, as illustrated below. The last half-byte contains, in bits 0 through 3, a value indicating the sign. Normally, the hexadecimal value 'C' indicates a positive value and the hexadecimal value 'D' indicates a negative value.



FIXEDOVERFLOW Condition Name

The FIXEDOVERFLOW condition name can be specified in an ON, SIGNAL, or REVERT statement to designate a fixed overflow condition or ON-unit.

PL/I signals the FIXEDOVERFLOW condition in the following circumstances:

- When the result of an arithmetic operation on a fixed-point decimal or binary integer value exceeds the maximum precision of the VAX-11 hardware. The maximum precision allowed for a fixed-point decimal or binary value is 31.
- When the source value of a fixed-point expression exceeds the precision of the target variable. For example, PL/I signals FIXEDOVERFLOW when a value that is not in the range -128–127 is assigned to a fixed-point binary variable with a precision of seven bits. Similarly, the condition is signaled if a value assigned to a picture variable has more integral digits than are specified by the picture specification.

The value resulting from an operation that causes this condition is undefined.

■ Value of ONCODE

There are two VAX-11 hardware exceptions that result in the FIXEDOVERFLOW condition. These are SS$_DECOVF (for a fixed-point decimal overflow) and SS$_INTOVF (for a fixed-point binary integer overflow). An ON-unit that receives control when FIXEDOVERFLOW is signaled can reference the ONCODE built-in function to determine which condition is actually signaled.

To define an ON-unit to respond to either of these errors specifically, use the VAXCONDITION keyword. For details on using the ONCODE built-in function and the VAXCONDITION condition name, see the *VAX-11 PL/I User's Guide*.

■ ON-Unit Completion

If the ON-unit does not transfer control elsewhere in the program, control returns to the point at which the condition was signaled.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement."

FLOAT Attribute

The FLOAT attribute indicates that a variable is a floating-point arithmetic item.

When you specify the FLOAT attribute in a DECLARE statement, you can specify either the BINARY or DECIMAL attribute and you can specify the precision. For a floating-point binary variable, the precision can be in the range of 1 through 113; for a floating-point decimal variable, the precision can be in the range of 1 through 34.

The default values given for unspecified related attributes are:

Attributes Specified	**Defaults Supplied**
FLOAT	BINARY (24)
FLOAT BINARY	(24)
FLOAT DECIMAL	(7)

■ Restrictions

The FLOAT attribute directly conflicts with all data type attributes except BINARY and DECIMAL.

FLOAT Built-In Function

The FLOAT built-in function converts a string or arithmetic expression x to floating point, with a specified precision. P must be an integer constant that is greater than zero and less than or equal to the maximum precision of the result type (34 for floating-point decimal, 113 for floating-point binary).

The format of the function is:

```
FLOAT(x,p)
```

■ Returned Value

The result type is floating-point binary or decimal, depending on whether x is a binary or decimal expression. (If x is a bit-string expression, the result type is floating-point binary; if x is a character-string expression, the result type is floating-point decimal.)

The expression x is converted to a value v of the result type, following the usual rules (**see** "Conversion of Data"). The value returned is v to the specified precision; UNDERFLOW or OVERFLOW is signaled if appropriate.

Floating-Point Data

The floating-point data types provide a way to express very large and very small numbers. For example, floating-point data types are used in scientific calculations.

All floating-point calculations are performed on values in one of the VAX binary floating-point formats. In general, the precision of the result is determined by the maximum precision of any operands in the operation. The numerical result of an operation is rounded to the result precision, so the results of most operations are approximate.

This discussion of floating-point data is divided into the following parts:

- Constants
- Variables
- Use in expressions
- Floating-point data formats
- Internal representation of floating-point data

■ Constants

A floating-point constant can contain one or more of the decimal digits 0 through 9 with an optional decimal point, followed by the letter E and from one to five decimal digits representing a power of 10. The floating-point value and the integer exponent can both be signed. The first portion of the value, to the left of the letter E, is called the mantissa.

Some examples of floating-point constants are:

```
2E10
-3E8
32E-8
.45632E16
```

The decimal precision of each of these values is the number of digits in the mantissa.

In VAX-11 PL/I, all floating-point constants are decimal.

■ Variables

The keyword FLOAT identifies a floating-point variable in a declaration. To declare a single floating-point binary variable, specify a DECLARE statement as follows:

DECLARE identifier FLOAT [BINARY] [(p)];

identifier

The name to be used for the variable.

p

The precision of the variable, that is, the number of digits to maintain in the mantissa. The precision must be an integer constant in the range 1–113. (If the compiler qualifier G_FLOAT is not used, the range is restricted to 1–53.) If you do not specify a precision, PL/I uses the default precision of 24.

To declare a decimal floating-point variable, specify:

DECLARE identifier FLOAT DECIMAL [(p)];

identifier

The name to be used for the variable.

p

The decimal precision, which must be an integer constant in the range 1–34. (If the compiler qualifier G_FLOAT is not used, the range is restricted to 1–15.) If you omit the precision, the default precision is 7.

Some examples of floating-point variables are:

```
DECLARE S FLOAT BINARY (16);
DECLARE X FLOAT DECIMAL (30);
```

Note that you can use either BINARY or DECIMAL to declare a floating-point value. Since the internal representation of floating-point variables is binary, it is recommended that you use BINARY FLOAT to declare variables (this is the default). In any event, you should declare all floating-point variables using the same base.

■ Using Floating-Point Data in Expressions

You can use both integer and scaled decimal constants freely in floating-point expressions. An arithmetic constant is always converted to the appropriate internal representation for use in a floating-point operation. The target type for the conversion depends on the context. In the following example:

```
DECLARE X FLOAT BINARY (53);
X = X + 1.3;
```

the constant 1.3 is converted to floating point when this expression is evaluated.

Such a conversion is normally done during compilation, although in some cases the constant is maintained in decimal until run time.

■ Floating-Point Data Formats

VAX-11 PL/I supports four types of floating-point values. Table F–3 summarizes the ranges of precision for each type.

Table F-3: VAX Floating-Point Types

Floating-Point Type	Sign Bits	Exponent Bits	Fractional Bits
F (single precision)	1	8	24
D (double precision)	1	8	53
G[1]	1	11	53
H[1]	1	15	113

1. Types G and H require a VAX-11 hardware option; types F and D are available on all VAX processors.

The PL/I compiler selects the appropriate VAX-11 floating-point type based on, first, the precision you specify and, second, a compile-time qualifier on the PLI command. The types are selected as shown in Table F-4.

Table F-4: Floating-Point Types Used by PL/I

Range of p (DECIMAL)	Range of p (BINARY)	Floating-Point Type
$1 \leq p \leq 7$	$1 \leq p \leq 24$	F
$8 \leq p \leq 15$	$25 \leq p \leq 53$	D or G[1]
$16 \leq p \leq 34$	$54 \leq p \leq 113$	H

1. D is used if possible, unless G is requested at compile time.

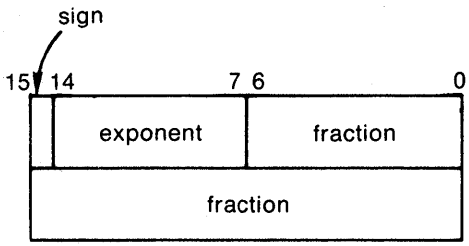
■ Internal Representation of Floating-Point Data

In all VAX floating-point formats, the value 0 is indicated by setting the sign bit and all exponent bits to zero. Effectively, this allows representation of, for example, a value with a 24-bit fraction and an eight-bit exponent in single precision, even though only 23 bits in the format are allocated for the fraction.

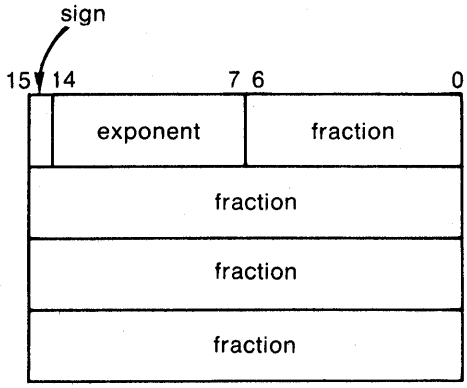
The double-precision and G-floating formats, as used by PL/I, have the same fractional precision; G-floating format allows an extra three bits for the exponent. Notice that the double-precision format has 56 bits available for the fraction, although only 53 bits are used by PL/I. If you specify a floating-point binary precision in the range 54–56, and you do not use the G_FLOAT compiler qualifier, the number is represented in double-precision format. (If the G_FLOAT qualifier is used, numbers with this range of precision are represented by the H-floating format.)

This small reduction in the precision of double-precision numbers is necessary so that the compiler does not select H-floating format on machines that lack the necessary hardware. The intent is to preserve the size of a structure containing double-precision data regardless of whether the G_FLOAT qualifier is used.

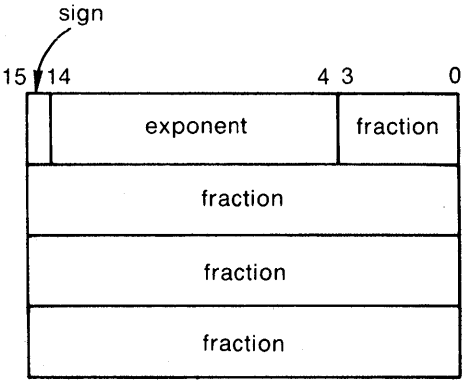
Single Precision



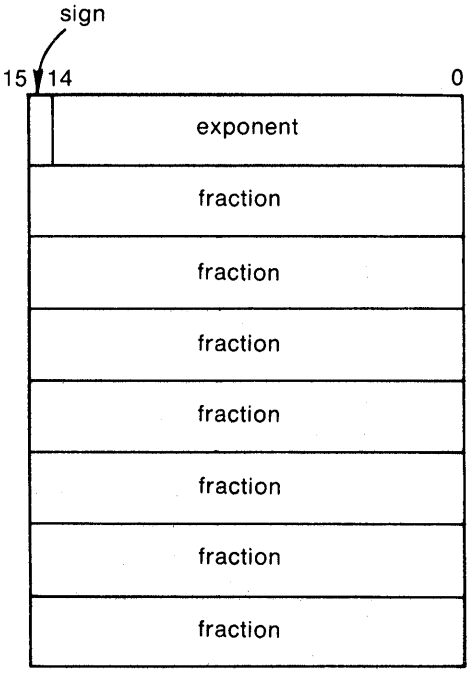
Double Precision



G-Floating



H-Floating



FLOOR Built-In Function

The FLOOR built-in function returns the largest integer that is less than or equal to an arithmetic expression x. Its format is:

FLOOR(x)

■ Returned Value

If x is a floating-point expression, the returned value is a floating-point value. If x is a fixed-point expression, the returned value is a fixed-point value with the same base as x and with the attributes:

precision = min(31,p–q+1)

scale factor = 0

where p and q are the precision and scale factor of x.

■ Examples

```
A = 3;
Y = FLOOR(A);          /* Y = 3.00 */

A = -3.32;
Y = FLOOR(A);          /* Y = -4.00 */
```

Format Items and Their Uses

This entry describes the formatting of input and output data in PL/I. Formatted data is transferred with the GET EDIT and PUT EDIT statements, which include a format specification made up of format items.

PL/I format items are categorized as follows:

- The data format items, A, B, E, F, and P, are used for input or output of data in various formats. A and B are used for character- and bit-string formats, respectively. E and F are used for floating- and fixed-point formats, respectively. P is used for input or output of data in a specified picture format. All data format items can be used with either the FILE or STRING option in edit-directed statements.
- The remote format item, R, is used to specify the label of a FORMAT statement, which contains a remote list of format items.
- The control format items, SKIP, LINE, PAGE, TAB, COLUMN, and X, are used to control the position in the input or output stream at which data is placed or from which it is acquired. Of the control format items, only X can be used with the STRING option in edit-directed statements.

The PL/I format items are summarized in Table F–5. Their general uses are discussed in this entry. Each format item also has its own entry in this manual; for example, **see** "A Format Item."

Table F-5: Summary of Format Items

Format Item	Use
A[(w)]	With GET EDIT, reads w characters from the input stream; with PUT EDIT, converts the value to be output to a w-character string and outputs the resulting string.
B[(w)]	With GET EDIT, reads w binary digits (0s and 1s) from the input stream; with PUT EDIT, the corresponding value is converted to a character string of length w, containing 0s and 1s, and written to the output stream. The B format item is equivalent to B1.
B1[(w)]	With GET EDIT, reads a character string of length w composed of the characters 0 and 1 from the input stream; with PUT EDIT, the corresponding value is converted to a character string of length w, containing 0s and 1s, and written to the output stream.
B2[(w)]	With GET EDIT, reads a character string of length w composed of the characters 0, 1, 2, and 3 from the input stream and converts it to a bit string; with PUT EDIT, converts w two-bit fields within the corresponding value to one of the characters 0, 1, 2, or 3, and writes the w-character string to the output stream.
B3[(w)]	With GET EDIT, reads a character string of length w composed of the characters 0, 1, 2, 3, 4, 5, 6, 7 from the input stream and converts it to a bit string; with PUT EDIT, converts w three-bit fields within the corresponding value to a string of the characters 0, 1, 2, 3, 4, 5, 6, or 7 and writes the w-character string to the output stream.
B4[(w)]	With GET EDIT, reads a character string of length w composed of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, or F from the input stream; with PUT EDIT, converts w four-bit fields within the corresponding value to a string of the characters 0 through F and writes the w-character string to the output stream.
COLUMN(position)	With GET EDIT, specifies the position at which reading of data is to proceed; with PUT EDIT, outputs spaces until the specified column position. May be used with files only.
E(w[,d])	With GET EDIT, converts a field of w characters from the input stream to a floating-point number; with PUT EDIT, converts a value to a w-character floating-point representation with d fractional digits in the mantissa and writes the w-character string to the output stream.
F(w[,d])	With GET EDIT, converts a field of w characters from the input stream to a fixed-point value; with PUT EDIT, converts a value to a w-character fixed-point representation with d fractional digits and writes the w-character string to the output stream.
LINE(number)	Valid for print files only. Specifies a line number, relative to the top of the page, at which output is to continue.

(Continued on next page)

Table F-5 (Cont.): Summary of Format Items

Format Item	Use
P 'picture'	With GET EDIT, acquires a character string from stream whose length is specified by the picture specification and signals ERROR if the string is not a pictured value; with PUT EDIT, converts an expression to a pictured value as specified by the picture and writes the pictured value to the output stream.
PAGE	Valid for print files only. Specifies that output is to be continued at the top of the next page.
R(label)	Indicates that format items are to be acquired from the FORMAT statement at the specified label.
SKIP[(linecount)]	With GET EDIT, continues reading after 'linecount' lines; with PUT EDIT, outputs 'linecount' blank lines and continues output. May be used with files only.
TAB[(n)]	Valid for print files only. Continues output at the nth tab stop relative to the current position.
X[(n)]	With GET EDIT, ignores n characters in the input stream; with PUT EDIT, places n spaces in the output stream. May be used with either files or character strings.

■ Data Format Items

The data format items refer to a field of characters in the stream. Each data format item specifies the width of the field in characters and either the manner in which the field is used to represent a value (output) or the manner in which the characters in the field are to be interpreted (input). Because the representation or interpretation is under control of the format items, certain symbols used in the stream with GET LIST and PUT LIST are not used with GET EDIT or PUT EDIT:

- Strings input by the GET EDIT statement should not be enclosed in apostrophes unless the apostrophes are intended to be part of the string. Strings output by PUT EDIT are not enclosed in apostrophes.
- Bit strings input by the GET EDIT statement should not be enclosed in apostrophes nor followed by the radix factors B, B1, B2, B3, or B4. These factors are not added by the PUT EDIT statement on output.
- The comma and space characters are not interpreted as data separators on input. On output, values are not automatically separated by spaces.

The following guidelines apply to errors and mismatches that occur between the actual data values and the fields specified by data format items:

- On input, the ERROR condition is signaled if the field of characters cannot be interpreted as required by the format item.
- On output, strings are left-justified in the specified field, and numeric data is right-justified. Truncation occurs if the field is too narrow to contain the necessary characters. Strings are truncated on the right and numeric data on the left.

■ Format Specifications

In the GET EDIT, PUT EDIT, and FORMAT statements, format items are used singly or in combination to form format specifications. A format specification can have three forms:

format-item
iteration-factor format-item
iteration-factor(format-specification,...)

The iteration factor is an integer that repeats the following format item or the following list of format specifications. If the iteration factor precedes a single format item that is not parenthesized, they must be separated by a space. For example, the statement:

```
PUT EDIT (A) (F(5,2));
```

specifies a five-character field containing decimal digits, two of which are fractional. Used by itself as a format specification, this item specifies one such field. To specify two such fields, precede the item with the iteration factor 2:

```
PUT EDIT (A,B) (2 F(5,2));
```

As shown above by the third form, an iteration factor can also repeat an entire list of format specifications, as in:

```
PUT EDIT ( (A(I) DO I = 1 TO 10) ) /* 10 array elements */
( 2( F(5,2),2(F(7,2),E(8)) ) ); /* 10 format items */
```

Expanded into individual format items, the above specification is:

F(5,2),F(7,2),E(8),F(7,2),E(8),F(5,2),F(7,2),E(8),F(7,2),E(8)

In general, data listed in the GET EDIT or PUT EDIT statement is matched to the expanded list of data format items, working from left to right, until the end of the input-target or output-source list is reached. Matching occurs only between input/output data and data format items; control format items are executed if and only if they are encountered while the matching is in progress.
See also "Format-Specification List."

Format-Specification List

Format-specification lists are used in GET EDIT, PUT EDIT, and FORMAT statements to control the conversion of data between the program and the input or output stream, and to precisely control positioning within the input or output stream. This entry describes the syntax of format-specification lists and the manner in which a format list is processed to acquire or transmit data.

■ Rules for Use

This section briefly describes rules and constraints for format-specification lists. For a general discussion of format items, **see** "Format Items and Their Uses." Each format item is defined in detail, for both input and output, in an individual entry (for instance, **see** "F Format Item").

- A GET EDIT or PUT EDIT statement must include one and only one format-specification list and also one and only one list of input targets or output sources. The input-target or output-source list must immediately follow the keyword EDIT and must be immediately followed by the format-specification list.
- The same set of data format items is used for both input and output. The F and E format items are used for I/O in fixed-point and floating-point formats, respectively. The A and B format items are used for I/O in character-string and bit-string formats, respectively. The P format item is used for input and output of data, with the format specified by a picture contained in the format item.
- Of the control format items, only X can be used when the input or output stream is a character string.
- Unlike the statement options PAGE, LINE, and SKIP, the format items PAGE, LINE, and SKIP are executed in the order in which they occur.

■ How Edit-Directed Operations Are Performed

This section describes the manner in which format items are matched to input targets or output sources. **See also** "Examples" at the end of this entry.

All edit-directed input and output statements contain the following syntax:

```
EDIT (input-target,...) (format-specification,...)
```

or

```
EDIT (output-source,...) (format-specification,...)
```

Each format specification is one of the following:

- A single control or data format item.
- A construct containing an iteration factor followed by one or more format items (for an explanation of iteration factors, **see** "Format Items and Their Uses").
- A remote (R) format item, which specifies the label of a FORMAT statement. Effectively, the entire format-specification list in the FORMAT statement is acquired and inserted at the position of the R format item.

Each input target is one of the following:

- A variable reference, which can be to a scalar or aggregate variable of any computational data type
- One of these constructs:
  1. (input-target,... DO reference=expression
     [TO expression][BY expression][WHILE(expression)])
  2. (input-target,... DO reference=expression
     [REPEAT expression] [WHILE (expression)])

Each output source is one of the following:

- Any expression with a computational value, including references to scalar or aggregate variables of any computational type
- A construct containing a DO specification, as shown for input targets

When PL/I performs an edit-directed operation, it examines the list of input targets or output sources, beginning with the first in the list. If the target or source is an array, the array is expanded in row-major order to form an ordered list of individual data items. If the target or source is a structure, the structure is expanded in the order of its declaration to form a list of individual items. If the target or source contains a DO specification, the item or items that precede the DO keyword are expanded in the preceding manner, and an ordered list of individual items is then created as per the DO specification.

Within a single target or source, items at the deepest level of parentheses are processed first.

Given a list of one or more data items contained in the first target or source, PL/I processes the data items from left to right. Beginning with the leftmost data item, and for each subsequent item, PL/I executes format items until the data item has been either assigned a value from the input stream or converted to a character representation and placed in the output stream. Control format items are therefore executed in the order in which they occur in the format-specification list. With the first target or source, the execution of format items begins with the leftmost format item in the format-specification list. If the end of the format-specification list is reached, PL/I returns to the leftmost format item and continues.

When all items contained in the first target or source have been processed, PL/I operates on the next target or source. The target or source is evaluated, and PL/I then examines the format-specification list, beginning where the previous operation stopped.

This processing continues until all data items in the input-target or output-source list have been processed, at which point the edit-directed statement terminates. If this occurs while PL/I is in the middle of the list of format items, the format items to the right are not executed.

■ Examples

The following examples show typical edit-directed operations. All cases shown are for input (GET EDIT), but the operations for PUT EDIT are similar. The simple cases are with input targets that are scalar variable references. The next cases shown are with aggregate references. The last cases shown are with DO specifications.

Simple Cases

These are cases in which the input targets are scalar variables.

```
GET EDIT (A,B,C,D) (A(12),F(5,2),F(6,2),A(14));
```

Acquire four values from the input stream: a 12-character string, a five-digit fixed-point decimal number, a six-digit fixed-point decimal number, and a 14-character string; assign these values, with any

necessary conversions, to the target variables A, B, C, and D, respectively. (For details of the conversions to the targets' types, **see** "Conversion of Data.")

```
GET EDIT (A,B,C,D) (A(12));
```

Acquire four 12-character strings and assign them (with conversions, if necessary) to the targets A, B, C, and D.

```
GET EDIT (A,B,C,D) (A(12), 2 F(5,2), A(14));
```

Acquire a 12-character string, two fixed-point decimal numbers, and a 14-character string, in that order, and assign them to A, B, C, and D. (Embedded spaces can be used in format lists, as elsewhere, for clarity; the space is required between "2" and "F(5,2)".)

```
GET EDIT (A,B,C,D,E) ( 2( A(12),A(14) ), A(20) );
```

Acquire, in order, a 12-character string, a 14-character string, another 12-character string, another 14-character string, and a 20-character string; assign the strings, in that order, to A, B, C, D, and E.

```
GET EDIT (A,B,C,D,E) ( 2( A(12),A(14) ), SKIP, A(20) );
```

Same operation as previous example, but acquire the 20-character string from the next line.

Aggregates

These cases use input targets that are references to array and structure variables.

```
GET EDIT (A) ( 2( A(12),A(14) ), A(20) );
```

where A is an array of five elements or a structure with five scalar members.

Expand A to a list of individual data items. Then acquire, in order, a 12-character string, a 14-character string, another 12-character string, another 14-character string, and a 20-character string; assign the strings, in that order, to the elements A(1) through A(5) (if an array) or to the five members of structure A in the order in which the members are declared.

```
GET EDIT (A,B) ( 2( A(12),A(14) ), A(20) );
```

where both A and B are aggregates with five elements or members.

For A, perform the same operation as the previous example, and then repeat the operation for B, using the same format list each time. Since

there are five format items specified, and the aggregates both have five elements or members, strings of the same length are acquired for corresponding elements of A and B.

```
GET EDIT (NAME) (SKIP,A(20),SKIP,A(80));
```

where NAME is a structure declared as

```
DECLARE 1 NAME
  2 FIRST CHARACTER(20) VARYING,
  2 LAST CHARACTER(80) VARYING;
```

Skip to the next line and acquire a 20-character string. Assign the string to NAME.FIRST. Skip to the next line and acquire an 80-character string. Assign that string to NAME.LAST.

```
GET EDIT (A,B) ( 2( A(12),A(14) ), SKIP, A(20) );
```

where both A and B are four-element arrays.

From the current line, execute A(12), A(14), A(12), and A(14), in that order, and assign the results to A(1)–A(4). Skip to the next line, and then execute A(20), A(12), A(14), and A(12), in that order, and assign the results to B(1)–B(4); the list of data items is now exhausted, so do not execute SKIP a second time.

DO Specifications

These examples use input targets that include DO specifications. The DO specifications control the assignment of input values to variables that are arrays and based structures.

```
GET EDIT ( (B(I) DO I=10 TO 4 BY -2) , B(1) )
 ( 2( A(12),A(14) ), A(20) );
```

where B is a 10-element array. (Notice that the parentheses surrounding the first input target are in addition to the parentheses surrounding the entire input-target list.)

Execute the format items A(12), A(14), A(12), and A(14) in that order, and assign the resulting strings to elements B(10), B(8), B(6), and B(4), respectively. Execute A(20) and assign the result to B(1).

```
GET EDIT ( ( (A(I,J) DO J=1 TO 10) DO I=1 TO 20) )
 (F(5),F(6));
```

where A is a two-dimensional array of 20 rows and 10 columns.

Two hundred decimal integers are acquired and assigned to the array elements in the order A(1,1), A(1,2),...,A(20,10). Elements with odd-numbered columns receive five-digit integers, and those with even-numbered columns, six-digit integers. Since the DO specifications specify row-major order, the same operation is performed by:

```
GET EDIT (A) (F(5),F(6));
```

Since row-major order is the default, nested DO specifications are generally used to change the order in which values are assigned.

The example is also identical with

```
DO I = 1 TO 20;
DO J = 1 TO 10;
GET EDIT(A(I,J)) (F(5),F(6));
END;
END;
```

Compared with a DO construct in the input-target list, however, the use of nested DO groups is much less efficient in execution speed. In addition, the identity is not generally true for all stream input/output statements. For instance, the statement

GET SKIP EDIT(input-target,...) (format-specification,...);

has different effects in the two cases. If it occurs in a pair of nested DO groups, as shown previously, the SKIP option is executed on each iteration of the innermost DO group. If instead the DO specifications are in the input-target list, the SKIP option is executed only once, and before any other input processing is performed.

```
GET EDIT ( ( CURRENT->PERSON.NAME
   DO CURRENT = FIRST
   REPEAT CURRENT->PERSON.NEXT
   WHILE (CURRENT ^= NULL)
      )

   ) (A(80)) ;
```

where CURRENT and FIRST are pointers and PERSON is a based structure declared as:

```
DECLARE /* Based structure for list elements: */
 1 PERSON BASED,
  /* Pointer to next element: */
  2 NEXT POINTER,
  2 NAME CHARACTER(80) VARYING;

DECLARE /* NULL function and pointers to first and
 current list elements: */
 NULL BUILTIN,
 (FIRST,CURRENT) POINTER;
```

The GET EDIT statement acquires 80-character strings from the input stream and assigns each to a list member PERSON.NAME. On the first input operation, the 80-character string is assigned to FIRST->PERSON.NAME. On subsequent iterations of the DO specification, the "next-pointer," PERSON.NEXT, is assigned to CURRENT before the input operation. Before each input operation, including the first, the WHILE clause tests to determine whether the end of the queued list has been reached (indicated by the null pointer).

The DO REPEAT construct is generally used in this type of application. It is advisable to provide a WHILE clause in this or any DO REPEAT construct, to be sure that the operation has a defined termination. However, the WHILE clause is not required.

FORMAT Statement

The FORMAT statement describes a remote format-specification list to be used by GET EDIT or PUT EDIT statements. The FORMAT statement and remote (R) format item are useful when the same format specification is used by a large number of GET EDIT and/or PUT EDIT statements. In this case, a change to the format specification can be made in the single FORMAT statement, rather than in each GET or PUT statement.

The form of the FORMAT statement is:

label: FORMAT (format-specification,...);

label

A valid PL/I label, required on a FORMAT statement. This label is specified in the GET EDIT or PUT EDIT statement that contains a remote format item, R, in its format-specification list.

format-specification

A list of one or more format items that match corresponding input targets in a GET EDIT statement, or output sources in a PUT EDIT statement. For further information, **see** "Format-Specification List" and "Format Items and Their Uses."

FREE Statement

The FREE statement releases the storage that was allocated for a based variable. The format of the FREE statement is:

FREE variable-reference ;

variable-reference

A reference to the based variable whose storage is to be released.

If you do not explicitly free the storage acquired for a based variable, the storage is not freed until the program terminates.

If you free a variable that is explicitly associated with a pointer, the pointer variable becomes invalid and must not be used to reference storage.

■ Examples

```
FREE LIST;
FREE P->INREC;
```

These statements release the storage acquired for the based variable LIST and for the allocation of INREC pointed to by the pointer P.

```
ALLOCATE STATE SET (STATE_PTR);
   .
   .
FREE STATE;
```

This FREE statement releases the storage for the based variable STATE and makes the value of STATE_PTR undefined.

FROM Option

The FROM option is specified on a REWRITE or WRITE statement to designate the variable whose contents are to be written to a record file. This option is specified in the format:

FROM (variable-reference)

variable-reference

A reference to a variable whose contents are to be written to the record file.

For example:

```
WRITE FILE (STATE_FILE) FROM (STATE_BUFFER);
```

This WRITE statement performs a sequential output operation to the file STATE_FILE. The contents of the variable STATE_BUFFER are used to create a new record at the end of the file.

See "REWRITE Statement" and "WRITE Statement."

Function

A function is a procedure that returns a scalar value. A function receives control when its name is referenced in the context of an expression. There are two types of function:

- PL/I built-in functions
- User-written functions

The PL/I built-in functions are available in all programs and generally need not be declared. (**See also** "Built-In Function" and "BUILTIN Attribute.")

A user-written function must:

- Contain the RETURNS option on the PROCEDURE statement.
- Specify a value on the RETURN statement that terminates the procedure. The value specified must be of a data type that is valid for conversion to the data type specified on the RETURNS option.

For example:

```
ADDER: PROCEDURE (X,Y) RETURNS (FLOAT);
DECLARE (X,Y) FLOAT;
        RETURN (X+Y);
        END;
```

This function has two parameters, X and Y. They are floating-point binary variables declared within the function. When this function is invoked by a function reference, it must be passed two arguments to correspond to these parameters. It returns a floating-point binary value representing the sum of the arguments it is passed.

■ Function Reference

The format of a function reference is:

entry-name ([argument,...])

entry-name

The name of an entry constant or variable used to invoke the function. (**See** "Procedure" and "Entry Data.")

argument,...

One or more arguments to be passed to the function. If specified, the arguments must correspond to the parameters specified in the PROCEDURE or ENTRY statement that identifies the entry name of the function.

Arguments must be enclosed in parentheses. Multiple arguments must be separated by commas.

For example, the function ADDER may be referenced as follows:

```
TOTAL = ADDER(5,6);
```

Arguments for a function must be separated by commas. An argument can be an expression of any data type.

If a function has no parameters, you must specify a null argument list; otherwise, the compiler treats the reference as a reference to an entry constant. Specify a null argument list as in this example:

```
GETDATE = TIME_STAMP();
```

This assignment statement contains a reference to the function TIME_STAMP, which has no parameters. This rule applies to PL/I built-in functions as well; however, if you declare a PL/I built-in function explicitly with the BUILTIN attribute, you need not specify the empty argument list.

For more information, **see** "Built-In Function" and "Procedure."

G

GET Statement

The GET statement acquires data from an input stream, which is either a stream file or a character-string expression. The input file may be a file declared with the STREAM attribute or the default file SYSIN, commonly associated with the user's default input device. (**See also** "Terminal Input/Output.")

This entry describes the syntax and options of GET statements. For a detailed description of the execution of a GET statement, **see** "Stream Input/Output."

The GET statement has several forms. They are summarized in Figure G–1 and described in this section.

```
GET EDIT (input-target,...) (format-specification,...)

    [ FILE (file-reference) ]
    [ OPTIONS (option,...) ]
    [ SKIP [(expression)] ] ;

GET LIST (input-target,...)

    [ FILE (file-reference) ]
    [ OPTIONS (option,...) ]
    [ SKIP [(expression)] ] ;

GET SKIP [(expression)]

    [ FILE (file-reference) ] ;

GET STRING (expression)

    { EDIT (input-target,...) (format-specification,...) } ;
    { LIST (input-target,...)                            }

    Options

    NO_ECHO
    NO_FILTER
    PROMPT (expression)
    PURGE_TYPE_AHEAD
```

Figure G–1: Forms of the GET Statement

■ GET EDIT

The GET EDIT statement acquires fields of character-string data from an input stream, which can be a stream file or a character-string expression. The stream file may be a declared file or the default file SYSIN. GET EDIT converts the character strings under control of a format specification and assigns the resulting values to a specified list of input targets (variables). It also allows input of characters from selected positions in the input stream.

The form of the GET EDIT statement is:

```
GET   EDIT (input-target,...) (format-specification,...)
[  FILE(file-reference)                    ]
[       [SKIP[(expression)]]               ]
[       [OPTIONS(option,...)]              ]
[                                          ]
[  STRING(expression)                      ]
     ;
```

input-target

The names of one or more variables to be assigned values from the input stream.

The input targets must be separated by commas.

An input target has the following forms:

1. reference

   where the reference is to a scalar or aggregate variable of any computational type. If the reference is to an array, data is assigned to array elements in row-major order. If the reference is to a structure, data is assigned to structure members in the order of their declaration.

2. (input-target,... DO reference=expression
   [TO expression][BY expression][WHILE(expression)])

   where the input target may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of input target are in addition to the parentheses surrounding the entire input list.

3. (input-target,... DO reference=expression
   [REPEAT expression][WHILE (expression)])

   where the input target may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of input target are in addition to the parentheses surrounding the entire input list.

For a discussion of the matching of format items to input targets, and of the use of DO specifications, **see** "Format-Specification List."

format-specification

A list of format items to control the conversion of data items in the input list. Format items can be data format items, control format items, or remote format items. For each variable name in the input-

target list, there is a corresponding data format item in the format-specification list that specifies the width of the field and controls the data conversion. (**See** "Format-Specification List" and "Format Items and Their Uses.")

FILE(file-reference)

An option specifying that the input stream is a file; the reference is to a declared file variable or constant. If neither the FILE option nor the STRING option is specified, PL/I assumes the file SYSIN. This file is associated with the default system input file SYS$INPUT.

If a file is specified and is not currently open, PL/I opens it with the attributes STREAM and INPUT. The UNDEFINEDFILE condition is signaled if the file cannot be opened.

STRING(expression)

An option specifying that the input stream is a character-string expression. The STRING option cannot be used with the FILE option, nor can it be used with the OPTIONS or SKIP option.

SKIP [***(expression)***]

An option that advances the input file a specified number of lines before processing the input list. May be used only with the implied or explicit FILE option. The expression, if specified, indicates the number of lines to advance; if it is omitted, the default is to skip to the next line. The SKIP option is always executed first, before any other input or positioning of the input file, and regardless of its position in the statement.

OPTIONS (option,...)

An option that specifies one or more of the following options. May be used only with the default or explicit FILE option. The options must be separated by commas and enclosed in parentheses.

NO_ECHO
NO_FILTER
PROMPT (string-expression)
PURGE_TYPE_AHEAD

The options are described fully in the *VAX-11 PL/I User's Guide*.

■ Examples

```
GET EDIT (FIRST,MID_INITIAL,LAST)
  (A(12),A(1),A(20));
```

Reads the next three character strings from the default stream input file (SYSIN) and assigns the strings to FIRST, MID_INITIAL, and LAST, respectively.

```
GET EDIT (SOCIAL_SECURITY) (A(12))
  FILE (SOCIAL) SKIP (12) ;
```

Opens (if closed) the stream file SOCIAL, advances 12 lines, reads the first 12 characters of the line, and assigns the characters to the variable SOCIAL_SECURITY.

```
GET EDIT (N, (A(I) DO I=1 TO N))
 (F(4),SKIP,100 F(10,5));
```

where the dimension of A is less than or equal to 100. The value of N is read from the input stream using the format item F(4). The process then skips to the next line (record). N elements are then read into the array A. Each element is read using the format item F(10,5).

```
GET EDIT (NAME.FIRST,NAME.LAST)
 (A(10),X(3),A(20))
 STRING('Philip    A. Rothberg          ');
```

Assigns 'Philip△△△△' to the structure member NAME.FIRST, skips the middle initial, period, and space, and assigns 'Rothberg△△△△△△△△△△△△' to NAME.LAST.

For more examples, **see** "Format-Specification List."

■ GET LIST

The GET LIST statement acquires character-string data from an input stream, which may be a stream file or a character-string expression. The stream file may be a declared file or the default file SYSIN. The acquired character strings are assigned to input targets named in the GET LIST statement, with the character strings being converted automatically to the targets' data types.

Use the GET LIST statement to read "unformatted" data from a stream file or character string. Because it is not necessary to place the input data in specific columns, GET LIST is useful for acquiring data from a terminal.

The form of the GET LIST statement is:

```
GET   LIST (input-target,...)
[  FILE(file-reference)                ]
[       [SKIP[(expression)]]           ]
[       [OPTIONS(option,...)]          ]
[                                      ]
[  STRING(expression)                  ]
    ;
```

input-target

The names of one or more variables to be assigned values from the input stream.

The input targets must be separated by commas.

An input target has the following forms:

1. reference

   where the reference is to a scalar or aggregate variable of any computational type. If the reference is to an array, data is assigned to array elements in row-major order. If the reference is to a structure, data is assigned to structure members in the order of their declaration.

2. (input-target,... DO reference=expression
   [TO expression][BY expression][WHILE(expression)])

   where the input target may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of input target are in addition to the parentheses surrounding the entire input list.

3. (input-target,... DO reference=expression
   [REPEAT expression][WHILE (expression)])

   where the input target may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of input target are in addition to the parentheses surrounding the entire input list.

The entry "Format-Specification List" shows how to use DO specifications with GET EDIT, and their use with GET LIST is comparable in most respects.

FILE(file-reference)

An option specifying that the input stream is a file; the reference is to a declared file variable or constant. If neither the FILE option nor the STRING option is specified, PL/I assumes the file SYSIN. This file is associated with the default system input file SYS$INPUT.

If a file is specified and is not currently open, PL/I opens it with the attributes STREAM and INPUT. The UNDEFINEDFILE condition is signaled if the file cannot be opened.

STRING(expression)

An option specifying that the input stream is a character-string expression. The STRING option cannot be used with the FILE option, nor can it be used with the OPTIONS or SKIP option.

Note that, as with list-directed input from a file, input fields must be separated by a space or comma. (**See** "Examples" below.)

SKIP [***(expression)***]

An option that advances the input file a specified number of lines before processing the input list. May be used only with the implied or explicit FILE option. The expression, if specified, indicates the number of lines to advance; if it is omitted, the default is to skip to the next line. The SKIP option is always executed first, before any other input or positioning of the input file, and regardless of its position in the statement.

OPTIONS (option,...)

An option specifies one or more of the following options. May be used only with the default or explicit FILE option. The options must be separated by commas and enclosed in parentheses.

NO_ECHO
NO_FILTER
PROMPT (string-expression)
PURGE_TYPE_AHEAD

The options are described fully in the *VAX-11 PL/I User's Guide.*

■ How to Specify Input Data

The items to be read into the input targets are separated by a space or a single comma. Multiple spaces are treated as a single space, and a comma may be surrounded by spaces. The following rules apply:

- No items can be split across lines unless the split occurs inside a quoted string.
- Character strings do not have to be enclosed in apostrophes unless they contain a space or comma or are written on more than one line. When a character string is enclosed in apostrophes, n apostrophes within the string are written as n+1 apostrophes; for instance, to input the word *isn't*, write `isn''t`.
- When a line begins with a comma or when two commas appear in the line without intervening nonspace characters, the item in the input-target list corresponding to that item is not updated. The target retains whatever value it contained before GET LIST was executed.
- Every input field, including the last input field in a line, must be terminated by a space or comma. On input from a terminal, a space is appended to the last input field when a carriage return is typed (unless ENVIRONMENT(IGNORE_LINE_MARKS) is used or the carriage return is inside a quoted string).
- Input fields are also terminated by the end-of-file (FILE option) or end-of-string (STRING option) unless the end is encountered inside a quoted string.
- If an input request from GET LIST encounters a null record, the corresponding input target is nulled. That is, the null character string ('') is assigned, with appropriate conversion, to the input target. A null input record means a null record in a file or, if the input is from a terminal, a carriage return with no other input. **See** "Terminal Input/Output" for examples. If ENVIRONMENT(IGNORE_LINE_MARKS) is used for the input file, record terminators such as the carriage return are ignored, and the GET LIST statement waits until the input request is satisfied.
- The ERROR condition is signaled whenever a data item in the stream cannot be converted to the data type of the corresponding item in the input-target list.
- The ENDFILE condition is signaled if the end of the file is encountered during file input. The ERROR condition is signaled if the expression in the STRING option does not contain enough characters to complete processing of the input-target list.

■ Examples

```
GETS: PROCEDURE OPTIONS(MAIN);

DECLARE NAME CHARACTER(80) VARYING;
DECLARE AGE FIXED;
DECLARE (WEIGHT,HEIGHT) FIXED DECIMAL(5,2);
DECLARE SALARY PICTURE '$$$$$$V.$$';
DECLARE DOSAGE FLOAT;
```

```
DECLARE INFILE STREAM INPUT FILE;
DECLARE OUTFILE PRINT FILE;

GET FILE(INFILE)
 LIST(NAME,AGE,WEIGHT,HEIGHT,SALARY,DOSAGE);

PUT FILE(OUTFILE)
 LIST(NAME,AGE,WEIGHT,HEIGHT,SALARY,DOSAGE);
END GETS;
```

If the file INFILE.DAT contains the following data:

```
'Thomas R. Dooley',33,150,60,5,87,15000,50,5E-6,
```

then the program GETS writes the following output to OUTFILE.DAT:

```
Thomas R. Dooley 33 150.60 5.87 $15000.50 4.9999999E-06
```

In the input file (INFILE.DAT) the string 'Thomas R. Dooley' was surrounded by apostrophes so that the spaces between words would not be interpreted as field separators.

```
GSTR: PROCEDURE OPTIONS(MAIN);

DECLARE STREXP CHARACTER(80) VARYING;
DECLARE (A,B,C,D,E) FIXED;
DECLARE OUTFILE STREAM OUTPUT FILE;

OPEN FILE(OUTFILE) TITLE('GSTR.OUT');

STREXP = '1,2,3,4,5';
GET STRING(STREXP) LIST(A,B,C,D,E);
PUT FILE(OUTFILE) LIST(A,B,C,D,E);

END GSTR;
```

The program GSTR writes the following output to GSTR.OUT:

```
1       2       3       4       5
```

For other examples, **see** "Terminal Input/Output."

■ GET SKIP

The GET SKIP statement positions the input file at the start of a new line. This format of the GET statement is:

GET [FILE(file-reference)] SKIP [(expression)] ;

file-reference

The name of the file to be advanced one or more lines. If no file is specified, PL/I assumes the default file SYSIN. This file is associated with the default system input file SYS$INPUT.

If a file is specified and is not currently opened, PL/I opens it with the attributes STREAM and INPUT.

expression

An integer expression giving the number of lines to advance; the default is one line.

GLOBALDEF Attribute

The GLOBALDEF attribute declares an external variable or an external file constant. It can optionally control the program section in which the data is allocated.

The format of the GLOBALDEF attribute is:

GLOBALDEF [(psect-name)]

psect-name

The name of a program section. A program section name can contain up to 31 alphanumeric characters, including a dollar sign ($) or underline (_). The first character cannot be a numeric (0 through 9).

If you do not specify a program section name, PL/I places the definition for the name in the default program section associated with the variable.

The GLOBALDEF attribute implies the EXTERNAL attribute. The GLOBALDEF attribute also implies STATIC except when used for file constants.

For complete details on using the GLOBALDEF attribute to declare global external symbols, **see** the *VAX-11 PL/I User's Guide.*

■ Restrictions

- The GLOBALDEF attribute conflicts with the GLOBALREF and INTERNAL attributes.
- It cannot be used with ENTRY constants.
- Only one procedure in a program may declare an external variable with the GLOBALDEF attribute.

GLOBALREF Attribute

The GLOBALREF attribute indicates that the declared name is a global symbol defined in an external procedure.

The GLOBALREF attribute implies the EXTERNAL attribute. The corresponding name must be declared in another procedure with the GLOBALDEF attribute or, if the external procedure is written in another programming language, with its equivalent in that language.

For complete details on using the GLOBALREF attribute to declare global external symbols, **see** the *VAX-11 PL/I User's Guide.*

■ Restrictions

- The GLOBALREF attribute conflicts with the INITIAL, GLOBALDEF, and INTERNAL attributes.
- If GLOBALREF is specified with the FILE attribute, no other file description attributes can be specified.

GOTO Statement

The GOTO statement causes control to be transferred to a labeled statement in the current procedure or any outer procedure. The format of the GOTO statement is:

{ GOTO / GO TO } label-reference ;

label-reference

A label constant or an expression that, when evaluated, yields a label value. A label value denotes a statement in the program and a block activation. (**See** "Label.")

The specified label cannot be the label of an ENTRY, FORMAT, or PROCEDURE statement. The label reference specified in a GOTO statement can be any of the following:

- An unsubscripted label constant. For example:

```
GOTO ALPHA;
   .
   .
ALPHA:
```

- A subscripted label constant, for which the subscript is specified with an integer constant or a variable expression. For example:

```
GOTO PROCESS(1);
   .
   .
PROCESS(1):
```

- A label variable that, when evaluated, yields a label value. For example:

```
DECLARE PROCESS LABEL VARIABLE;
   .
PROCESS = BILLING;
   .
GOTO PROCESS;
```

- A subscripted label variable that, when evaluated, yields a label value. For example:

```
DECLARE X(5) LABEL;
(1) = NEXT;

GOTO X(1);
```

In the case of a label variable, the resulting label value must designate an existing block activation. (This will always be true for a label constant.) If the designated block activation is the current block activation, the GOTO statement effects a local GOTO. No special processing occurs.

■ Nonlocal GOTO

If the specified label value is not in the current block, the GOTO statement is considered a nonlocal GOTO. The following can occur:

- The current block, and any blocks intervening between it and the block containing the label value, are released. This rule applies to procedure blocks and to begin blocks.

- If a GOTO statement transfers control out of a procedure that is invoked in a function reference, the statement containing the function reference is not evaluated further.

See also "Procedures — Terminating Procedures."

■ Examples

```
ON ERROR GOTO ERROR_MESSAGE;
```

The GOTO statement provides a transfer address for the current procedure when the ERROR condition is signaled.

```
DECLARE PROCESS(5) LABEL VARIABLE;
   .
   .
GOTO PROCESS(2);
```

The GOTO statement evaluates the label reference and transfers control to the label constant corresponding to the second element of the array PROCESS. PROCESS consists of label variables.

For more information, **see** "Label."

HBOUND Built-In Function

The HBOUND built-in function returns a fixed-point binary integer that is the upper bound of a specified dimension of an array. Its format is:

HBOUND(reference,dimension)

reference

The name of an array variable.

dimension

An integer constant indicating a dimension of the specified array.

See "Array — Array-Handling Functions" for an example.

I

IDENT Option

The PROCEDURE statement accepts the option IDENT, which places an identifying character string in the upper left corner of the listing file and in the object file as the module's "version" for the linker. The option format is:

OPTIONS(IDENT(string)[,option,...])

string

A character-string constant giving the identifying label for the listing. Only the first 31 characters of the string are placed in the object module.

option

Other procedure options.

Identifier

An identifier is a user-supplied name for a procedure, a statement label, or a variable that represents a data item. The rules for forming identifiers are:

- An identifier can have from 1 to 31 characters.
- An identifier can consist of any of the following characters:
  - The alphabetic letters A through Z and a through z. PL/I converts all lowercase letters to uppercase when it compiles a source program. Thus, the identifiers abc, ABC, Abc, and so on all refer to the same identifier.
  - The numeric digits 0 through 9.
  - The underline character (_).
  - A dollar sign character ($).
- An identifier cannot contain any blanks.
- An identifier must begin with an alphabetic letter, a dollar sign ($), or an underline (_).

Some examples of valid identifiers are:

```
STATE
total
FICA_PAID_YEAR_TO_DATE
ROUND1
SS$_UNWIND
```

IF Statement

The IF statement tests an expression and performs a specified action if the result of the test is true. The format of the IF statement is:

IF test-expression THEN action [ELSE action]

test-expression

Any valid expression that yields a scalar bit-string value. If any bit of the value is 1, then the test expression is true; otherwise, the test expression is false.

action

Any of the following:

- Any unlabeled statement except a DECLARE, FORMAT, PROCEDURE, END, or ENTRY statement
- An unlabeled DO-group or begin block

The IF statement evaluates the test expression. If the expression is true, the action specified following the keyword THEN is executed. Otherwise, the action, if any, specified following the ELSE keyword is executed.

■ Examples

```
IF A < B THEN BEGIN;
```

The begin block following this statement is executed if the value of the variable A is less than the value of the variable B.

```
IF ^SUCCESS THEN
          CALL PRINT_ERROR;
     ELSE
          CALL PRINT_SUCCESS;
```

The IF statement defines action to be taken if the variable SUCCESS has a false value (the THEN clause) and the action to be taken otherwise (the ELSE clause).

For details on the syntax of specifying expressions, **see** "Expression."

■ Nested IF Statements

The action specified in a THEN or an ELSE clause may be another IF statement.

An ELSE clause is matched with the nearest preceding IF/THEN that is not itself matched with a preceding ELSE. For example:

```
IF ABC
   THEN IF XYZ
           THEN GOTO GBH;
      ELSE GOTO THESTORE;
   ELSE GOTO HOME;
```

In the above example, the first ELSE clause is executed if ABC is true and XYZ is false. The second ELSE clause is executed if ABC is false.

In some cases, proper matching of IF and ELSE may require a null statement as the target of an ELSE. For example:

```
IF ABC
   THEN IF XYZ THEN GOTO HOME;
         ELSE;
   ELSE GOTO THESTORE;
```

In this example, the ELSE GOTO THESTORE statement is executed if ABC is false.

%INCLUDE Statement

The %INCLUDE statement incorporates text from other files into the current source file during compilation. It can occur anywhere in a PL/I source file; it need not be within a procedure. The format of the %INCLUDE statement is:

$$\%\text{INCLUDE}\left\{\begin{array}{l}\text{'file-spec'}\\ \text{module-name}\end{array}\right\};$$

file-spec

A file specification enclosed in apostrophes. The name is subject to logical name translation and the application of default values by the VAX/VMS system.

module-name

The 1- to 31-character name of a text module in a library of INCLUDE files and/or other text modules. The name of the library containing the module must be specified in the PLI command when the source program is compiled.

For details on the specification of files and libraries to be included in a PL/I compilation, **see** the *VAX-11 PL/I User's Guide*.

■ Examples

```
%INCLUDE 'SUM.PLI';
```

This statement copies the contents of the file SUM.PLI into the current file during compilation.

```
%INCLUDE SYSTEM_PROCEDURES;
```

This statement includes a module from a text module library. The library containing the module SYSTEM_PROCEDURES must be present on the command that compiles this program.

■ Restrictions

The maximum depth to which %INCLUDE statements can be nested is four.

INDEX Built-In Function

The INDEX built-in function returns a fixed-point binary integer that indicates the position of a specified substring within a string. The value returned

indicates the position of the leftmost occurrence of the substring within the string. If the substring is not found, or if the length of either argument is zero, the INDEX function returns zero.

The format of the function is:

INDEX(string,substring)

string

The string to search for the given substring. It can be either a character-string or bit-string expression.

substring

The substring to locate. It must have the same string data type as the string argument.

■ Examples

```
DECLARE RESULT FIXED BINARY(31),
 NEW_STRING CHARACTER(80);
RESULT = INDEX('ABCDEF','DEF');
 /* RESULT equals 4
 (DEF begins at fourth position) */
RESULT = INDEX('SHARP FORTUNE','R');
 /* RESULT equals 4
 (leftmost occurrence of R at
 fourth position) */
NEW_STRING = '315-54-3159';
IF INDEX(NEW_STRING,'-') = 4 THEN
 GO TO SOCIAL_SECURITY;
 /* Expression is TRUE */
.
.
.
```

INITIAL Attribute

The INITIAL attribute provides an initial value for a declared variable. The format of the INITIAL attribute is:

INITIAL (initial-element[,initial-element...])

initial-element

A construct that supplies a value for the initialized variable. The value must be valid for assignment to the initialized variable. If the initialized variable is an array, a list of initial elements separated by commas is used to initialize individual elements. The number of initial elements must be one for a scalar variable and must not exceed the number of elements of an array variable. Each initial element must be one of the following forms:

- string-constant
- (iteration-factor) (string-constant)
- [(iteration-factor)] arithmetic-constant
- [(iteration-factor)] scalar-reference

- [(iteration-factor)] (scalar-expression)
- [(iteration-factor)] *

The iteration factors are nonnegative integer-valued expressions that specify the number of successive array elements to be initialized with the following value. (Notice that a string constant must be parenthesized if it is used with an iteration factor.)

The asterisk form specifies that the corresponding array elements are to be skipped during the initialization.

Some examples are:

```
DECLARE RATE FIXED DECIMAL (2,2) STATIC INITIAL (.04);

DECLARE SWITCH BINARY STATIC INITIAL ('1'B);

DECLARE BELL_CHAR BINARY STATIC INITIAL ('07'B4);

DECLARE OUTPUT_MESSAGE CHARACTER(20) STATIC
            INITIAL ('GOOD MORNING');

DECLARE (A INITIAL ('A'), B INITIAL ('B'),
      C INITIAL ('C')) STATIC CHARACTER;

DECLARE QUEUE_END POINTER STATIC INITIAL(NULL());
```

■ Restrictions

- The INITIAL attribute must not be specified for a structure variable. Instead, initialize individual members of the structure.
- The INITIAL attribute must not be specified for a variable or member of a variable that has any of the following attributes:

FILE	LABEL	parameter
BASED	ENTRY	DEFINED

- You cannot specify the INITIAL attribute for a member of a structure unless the entire structure was declared with the STATIC or AUTOMATIC attribute.
- If the initialized variable is STATIC, only constants and references to the NULL built-in function are allowed. These may be used with a constant iteration factor and may be enclosed in parentheses.
- Variables and functions (except for parameters) occurring in an initial element must not be declared in the same block as the variable being initialized.

INPUT Attribute

The INPUT file description attribute indicates that the associated file is to be an input file, that is, the file represents an external source of data.

Specify the INPUT attribute on a DECLARE statement for a file constant or on an OPEN statement to access the file for reading.

The INPUT attribute may be specified with either the STREAM or RECORD attribute. For a stream file, INPUT indicates that the file will be accessed using GET statements. For a record file, INPUT indicates that the file will be accessed using only READ statements.

For example:

```
DECLARE INFILE RECORD INPUT;

OPEN FILE(INFILE);
READ FILE(INFILE) INTO(RECORD_BUFFER);
```

These statements declare, open, and access the first record in the input file INFILE.

For a description of the attributes that may be applied to files and the effects of combinations of these attributes, **see** "File Description Attributes and Options."

The INPUT attribute may be supplied by default for a file, depending on the context of its opening. **See** "Opening a File."

■ Restrictions

The INPUT attribute conflicts with the OUTPUT, UPDATE, and PRINT attributes and with any data type attribute other than FILE.

Input/Output Processing

PL/I provides extensive facilities for the transmission of data between variables in a PL/I program and RMS files or communication devices such as terminals. There are two basic types of input/output in PL/I:

- In stream I/O, the external data (which can be an RMS file or a device) is treated as a stream of ASCII characters divided into fields delimited by spaces, tabs, or commas, or by other field specifications. Stream I/O is performed by the GET and PUT statements. These statements also perform conversion between the internal representation of data and the ASCII representation of the data.
- In record I/O, an operation transmits an entire record. Record I/O is performed by the READ, WRITE, DELETE, and REWRITE statements. These statements can be used to process files with the sequential, relative, and indexed sequential file organizations.

Each of these types of input/output is described individually in this manual (**see** "Stream I/O" and "Record I/O"). For an overview of how to declare and reference files in PL/I, **see** the entry "File."

Integer Data

Integer data is used for values that can be expressed in integers, for example, counters, array subscripts, record numbers, and so on.

■ Constants

An integer constant can contain one or more of the decimal digits 0 through 9 and, optionally, a sign. Some examples of integer constants are:

```
1
245
-88
```

All integer constants are decimal.

■ Variables

Integer variables can be declared as fixed-point binary or fixed-point decimal with a zero scale factor.

The format of a declaration of a fixed binary integer variable is:

DECLARE identifier FIXED [BINARY] [(p)];

identifier
The name to be used for the variable.

p
An integer constant representing the precision, that is, the number of binary digits used to represent values of the variable. The precision must be in the range 1–31. If you do not specify a precision, PL/I uses the default precision of 31.

Because fixed binary variables have a maximum precision of 31, fixed binary integers can have values only in the range of -2,147,483,648 through 2,147,483,647. An attempt to calculate a binary integer outside this range, in a context that requires an integer value, signals the FIXEDOVERFLOW condition.

Specify a fixed-point decimal integer variable as follows:

DECLARE identifier [FIXED] DECIMAL [(p)];

identifier
The name to be used for the variable.

p
The precision of the variable in decimal digits. The maximum precision you can specify for a fixed-point decimal variable is 31. If you omit the precision, 10 is the default.

For the internal representation and other details of binary and decimal integers, **see** "Fixed-Point Binary Data" and "Fixed-Point Decimal Data."

■ Restricted Integer Expressions

A restricted integer expression is one that yields only integral results and has only integral operands. Such an expression must use only the addition (+), subtraction (-), and multiplication (*) operators. Division in a restricted integer expression must be performed with the DIVIDE built-in function. In

VAX-11 PL/I, a restricted integer expression may be used in certain contexts (such as the specification of array bounds) where an integer constant is ordinarily used.

INTERNAL Attribute

The INTERNAL attribute limits the scope of an identifier to the block in which the identifier is declared and its dynamic descendents. The format of the INTERNAL attribute is:

```
{INTERNAL}
{INT     }
```

You need use the INTERNAL attribute only to explicitly declare the scope of a file constant as internal. File constants, by default, have the EXTERNAL attribute. All other variables are internal by default.

■ Restrictions

The INTERNAL attribute directly conflicts with the EXTERNAL, GLOBALDEF, and GLOBALREF attributes.

Internal Procedure

An internal procedure is a procedure whose text is contained within that of another procedure. The name of the internal procedure is declared by its use as the label of the PROCEDURE statement.

See "Procedure."

Internal Representation of PL/I Data

This entry describes the internal representations used by VAX-11 PL/I for PL/I data types. For additional information, refer to the entries on individual data types; for example, **see** "Bit-String Data."

■ Internal Representation of Bit Data

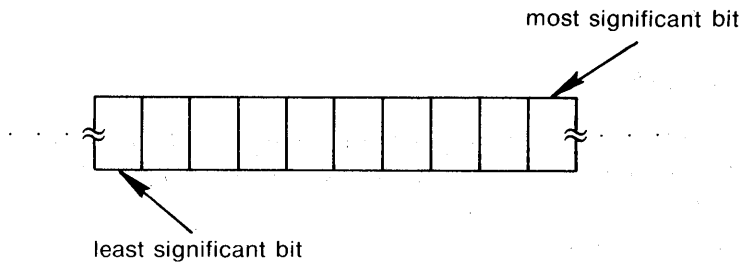
The way that PL/I allocates storage for a bit-string variable depends on whether the variable is declared with the ALIGNED attribute.

In this discussion, the term "most significant bit" means the leftmost bit in an external representation of the string, as, for example, when the string is output by the PUT LIST statement. The "least significant bit" is the rightmost bit in the external representation.

Unaligned Bit Strings

An unaligned bit string is stored beginning at an arbitrary bit location in storage; this location is the location of the most significant bit. The

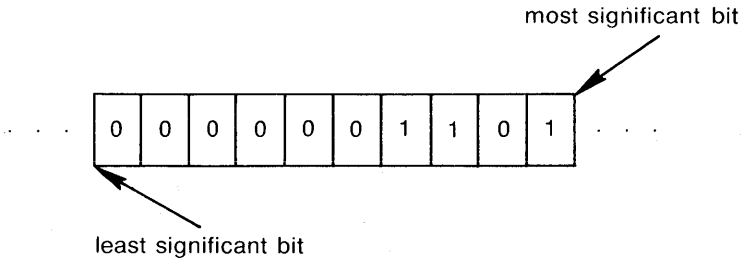
subsequent, less significant, bits are stored in progressively higher locations in memory, as shown here:



The following programming sequence illustrates how a value for an unaligned bit-string variable is stored:

```
DECLARE ABIT BIT (10);
ABIT = '1011'B;
```

After the assignment, the variable appears in storage like this:



Aligned Bit Strings

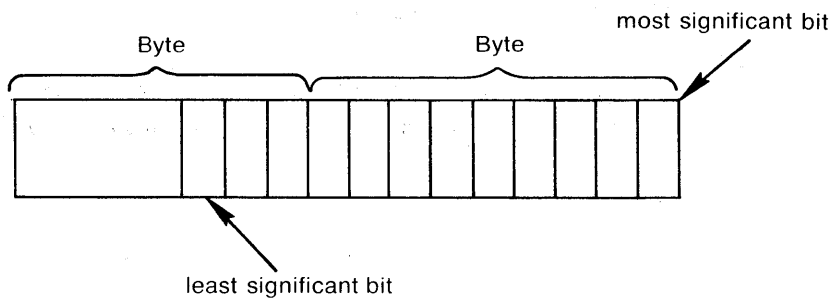
PL/I allocates storage for an aligned bit-string variable on a byte boundary and allocates an integral number of bytes. The number of bytes to allocate is calculated as follows:

ceil(n/8)

where n is the length specified for the bit string.

Beginning at bit 0 (the lowest memory location) of the lowest allocated byte, the bit string is stored like unaligned bit-string data; that is, the beginning bit is used to hold the most significant bit in the string. Less significant bits are stored in progressively higher memory locations. Unused bits are set to zeros each time the bit-string variable is assigned a value.

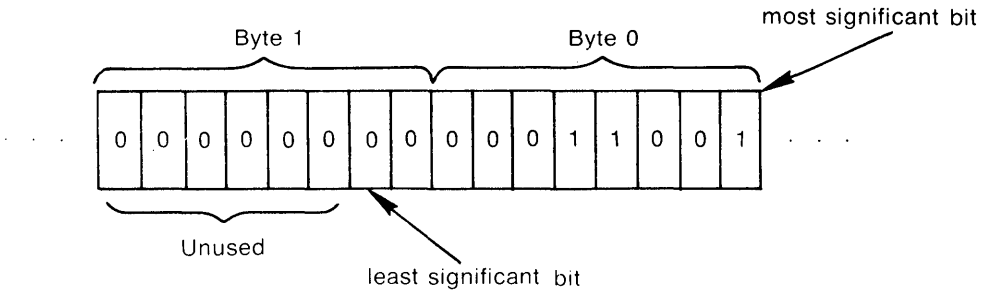
The representation is as follows:



The following programming sequence illustrates how values are stored for aligned bit strings:

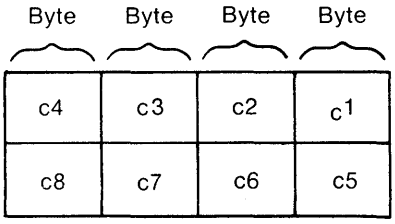
```
DECLARE ABIT BIT (10) ALIGNED;
ABIT = '10011'B;
```

In this example, the variable ABIT is aligned. When it is assigned the value 10011, its storage appears as follows:

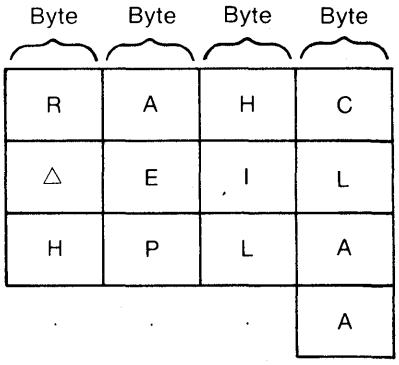


■ Internal Representation of Character Data

PL/I stores fixed-length character-string data from right to left, with each character occupying a byte of storage, as shown here:

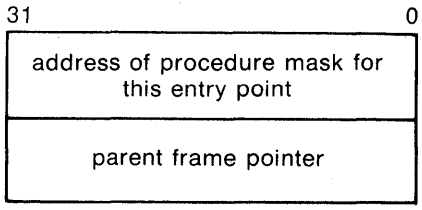


For example, a character string whose value is 'CHARLIE△ALPHA' appears as follows in storage:



Varying-length strings are stored in a number of bytes equal to n+2, where n is the declared maximum length. The two additional bytes contain, in the first two byte addresses, the current length of the value in bytes.

■ Internal Representation of Variable Entry Data



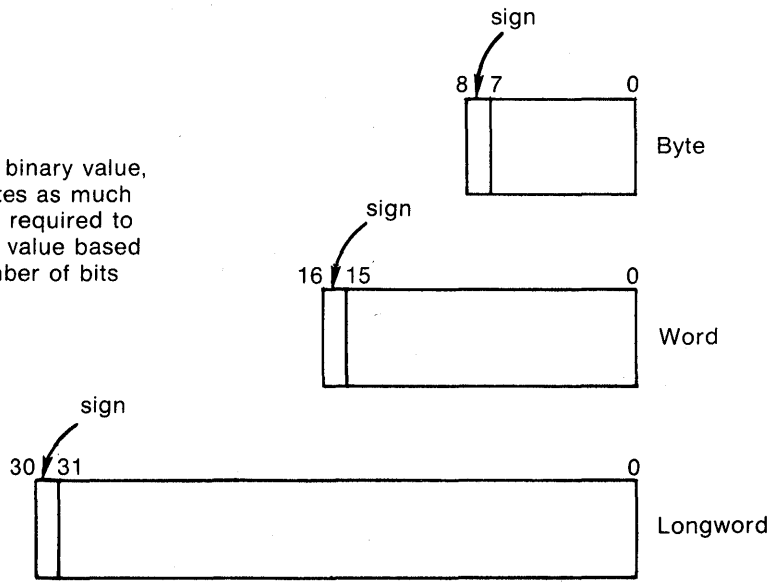
■ Internal Representation of File Data

A PL/I file, or file constant, is represented internally by a file control block. A file control block is an internal data structure maintained by PL/I.

A file variable is represented internally as a longword that contains a pointer to a file control block. The value of the file variable, when evaluated, is the address of the file control block for the file with which the variable is currently associated.

■ Internal Representation of Fixed-Point Binary Data

For a fixed binary value, PL/I allocates as much space as is required to contain the value based on the number of bits needed.



Storage for fixed-point binary variables is always allocated in a byte, word, or longword. For any fixed-point binary value:

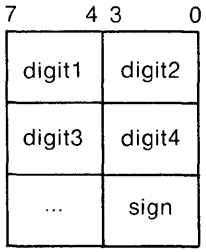
- If $1 \leq p \leq 7$, a byte is allocated.
- If $8 \leq p \leq 15$, a word is allocated.
- If $16 \leq p \leq 31$, a longword is allocated.

The binary digits of the stored value go from right to left in order of increasing significance; for example, bit 6 of a FIXED BINARY(7) value is the most significant bit and bit 0 is the least significant.

In all cases, the high-order bit (7, 15, or 31) is used to encode the sign.

■ Internal Representation of Fixed-Point Decimal Data

Fixed decimal data is stored in packed decimal format. Each digit is stored in a half-byte, as illustrated below. The last half-byte contains, in bits 0 through 3, a value indicating the sign. Normally, the hexadecimal value 'C' indicates a positive value and the hexadecimal value 'D' indicates a negative value.



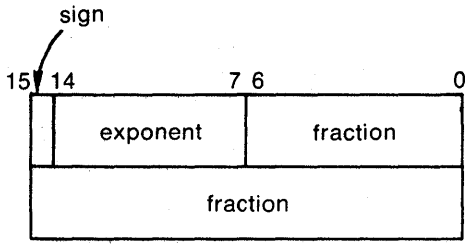
■ Internal Representation of Floating-Point Data

In all VAX floating-point formats, the value 0 is indicated by the sign bit and all exponent bits being zero. Effectively, this allows representation of, for example, a value with a 24-bit fraction and an eight-bit exponent in single precision, even though only 23 bits in the format are allocated for the fraction.

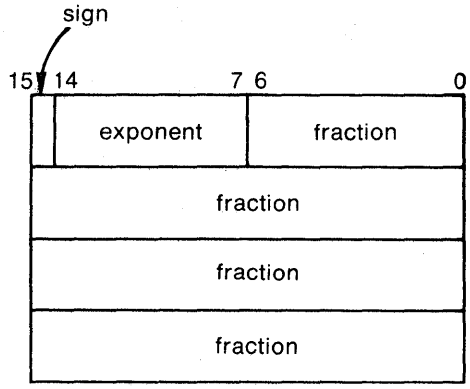
The double-precision and G-floating formats, as used by PL/I, have the same fractional precision; G-floating format allows an extra three bits for the exponent. Notice that the double-precision format has 56 bits available for the fraction, although only 53 bits are used by PL/I. If you specify a floating-point binary precision in the range 54–56, and you do not use the G_FLOAT compiler qualifier, the number is represented in double-precision format. (If the G_FLOAT qualifier is used, numbers with this range of precision are represented by the H-floating format.) This small reduction in the precision of double-precision numbers is necessary so that the compiler does not select H-floating format on machines that lack the necessary hardware. The intent

is that the size of a structure containing double-precision data is preserved regardless of whether the G_FLOAT qualifier is used.

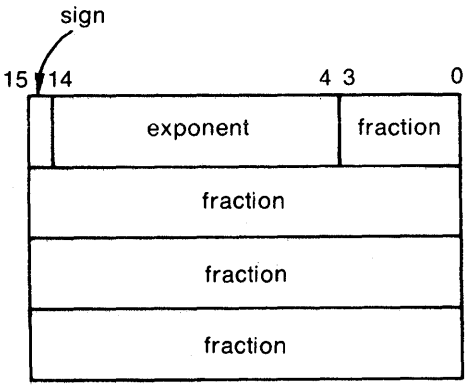
Single Precision



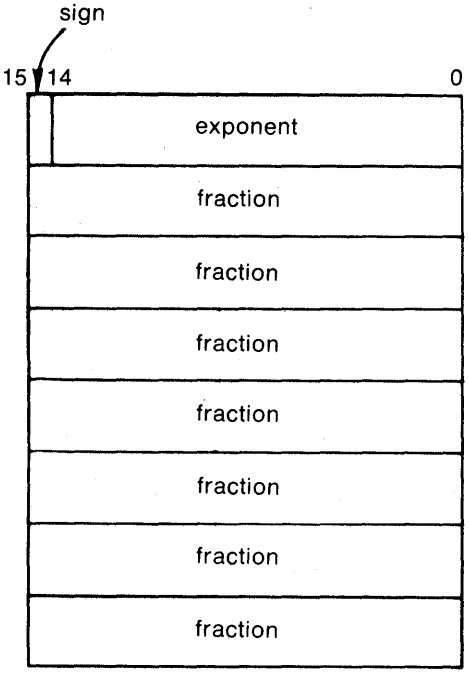
Double Precision



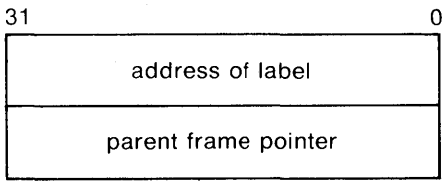
G-Floating



H-Floating



■ Internal Representation of Variable Label Data



■ Internal Representation of Pointer Data

A pointer occupies a longword (32 bits) of storage and represents a virtual memory address.

Internal Variable

An internal variable is a variable that is known only within the block in which it is defined and within all contained blocks. By default, PL/I gives all variables the internal attribute.

See "Block" and "Scope of Names."

INTO Option

The INTO option is specified in a READ statement to designate a variable into which the contents of a record from a record file are to be copied. This option is specified in the format:

INTO (variable-reference)

variable-reference

A reference to a variable into which the contents of the record are to be copied.

For example:

```
READ FILE (INFILE) INTO (RECORD_BUFFER) ;
```

This READ statement reads the next sequential record in the file INFILE and copies the contents of the record into the variable RECORD_BUFFER.

See "READ Statement."

Iteration Factor

An iteration factor is a syntactical method of requesting a specific operation or function more than once. In most cases, an iteration factor is an integer constant that specifies the number of times to repeat a particular item.

Iteration factors are allowed in the following contexts in a VAX-11 PL/I program:

- Format-specification lists (**see** "Format-Specification List")
- Initialization of array elements (**see** "INITIAL Attribute" and "Array")
- Picture character specifications (**see** "Picture")

K

Key

A key is a value that identifies a specific record in a relative file, in a sequential disk file with fixed-length records, or in an indexed sequential file. The nature of the key depends on the file's organization, as follows:

- If the file is a relative file or a sequential disk file with fixed-length records, the key is a fixed binary value indicating the relative record number of the record.
- If the file is an indexed sequential file, the key specifies a key that is contained within a record. The data type of the key and its location within the record are as specified when the file was created.

See individual entries for the READ, WRITE, DELETE, and REWRITE statements for details on how these statements interpret and use keys. For details on defining key fields for the creation of an indexed sequential file, see the *VAX-11 PL/I User's Guide*.

KEY Condition Name

The KEY condition name can be specified in an ON, SIGNAL, or REVERT statement to designate a key error condition or ON-unit for a specific file.

The format of the KEY condition name is:

KEY (file-reference)

file-reference

A reference to the file constant or file variable for which the ON-unit is to be established. If the name of a file variable is specified, the variable must be resolved to the name of a file constant when the condition is signaled.

PL/I signals the KEY condition during an operation on a keyed file when an error occurs in processing a key. Some examples of errors for which PL/I signals the KEY condition are:

- The record indicated by the specified key cannot be found.
- The key specification required conversion from one data type to another and the conversion is not valid.
- The key is not correctly specified.
- The key of a relative file exceeds the maximum record number specified when the file was created.

An ON-unit established to handle the KEY condition can obtain information about the condition by invoking the following built-in functions:

- The ONFILE built-in function — returns the name of the file being processed when the condition was signaled.
- The ONCODE built-in function — returns the specific condition value associated with the error.
- The ONKEY built-in function — returns the key value that caused the condition to be signaled.

■ ON-Unit Completion

If the ON-unit does not execute a nonlocal GOTO, control returns to the statement immediately following the statement that caused the KEY condition.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement."

KEY Option

The KEY option may be specified in a READ, REWRITE, or DELETE statement to indicate a specific record in a file that is opened with the KEYED attribute. This option is specified in the format:

KEY (expression)

expression

An expression giving the key value that identifies the record of interest.

For example:

```
DELETE FILE (CUST_ACCT) KEY (NAME) ;
```

This DELETE statement deletes the record in the file CUST_ACCT that has the key value represented by the variable NAME.

KEYED Attribute

The KEYED file description attribute indicates that records in the specified file may be accessed randomly. The KEYED attribute implies the RECORD attribute.

Specify KEYED in a DECLARE statement to identify a file or in an OPEN statement to open the file. For a description of the attributes that may be applied to files and the effects of combinations of these attributes, **see** "File Description Attributes and Options."

■ Restrictions

The KEYED attribute conflicts with the STREAM attribute and with any data type attributes other than FILE.

KEYFROM Option

The KEYFROM option may be specified in a WRITE statement to write a record to a file opened with the KEYED attribute. The KEYFROM option designates the key associated with the record. This option is specified in the format:

KEYFROM (expression)

expression

An expression giving the key value that indicates the record of interest. The specified value must have one of the computational data types.

For example:

```
WRITE FILE (EMPLOYEE_REC) FROM (EMP_DATA)
      KEYFROM (EMPLOYEE_NUMBER) ;
```

This WRITE statement writes a record to the file EMPLOYEE_REC by specifying the KEYFROM option.

See "WRITE Statement."

KEYTO Option

The KEYTO option may be specified in a READ statement to obtain the key associated with a record that was read sequentially. This option is specified in the format:

KEYTO (variable-reference)

variable-reference

A reference to a computational variable to be assigned the value of the key.

For example:

```
READ FILE (STATE_FILE) INTO (STATE_BUFFER)
          KEYTO (SAVE_NAME) ;
```

This READ statement reads the next sequential record in the file STATE_FILE into the variable STATE_BUFFER. The key associated with the record that is read is copied into the variable SAVE_NAME.

See "READ Statement."

Keyword

A keyword is a name that has a special meaning to PL/I when used in a specific context. In PL/I, keywords identify:

- Statements — for example, DECLARE, READ, or END
- Attributes — for example, DECIMAL, CHARACTER, or FILE
- Options — for example, KEYFROM, SKIP, or REPEAT

PL/I recognizes keywords when they appear in the correct context. You can also use keywords as identifiers. For example:

```
DECLARE DECLARE FIXED BINARY (6);
```

In the above statement, PL/I interprets the first occurrence of DECLARE as the keyword DECLARE because of its position in the statement. It interprets the second occurrence of DECLARE as an identifier because of its position.

■ Abbreviating Keywords

You can abbreviate some PL/I keywords. The valid abbreviations for PL/I keywords are given with the keyword description and in Appendix A.

L

Label

A label identifies a statement so that it can be referred to elsewhere in the program, for example, as the target of a GOTO statement. A label precedes a statement. It consists of any valid identifier terminated by a colon. Some examples are:

```
TARGET: A = A + B;
READ_LOOP: READ FILE (TEXT) INTO (TEMP);
```

These statements contain the implicit declarations of the names TARGET and READ_LOOP as label constants.

No statement can have more than one label. A statement can, however, be preceded by any number of labeled null statements. For example:

```
A:;
B: DO I = 1 TO 5;
```

Other statements in the program can refer to the DO statement in the above example by specifying either label A or label B.

A name occurring as a statement label is implicitly declared as a label constant. It has the attributes LABEL and constant. Label constants may not be explicitly declared.

■ Label Array Constants

Any label constant except the label of a PROCEDURE or FORMAT statement can have a single subscript. Subscripts must be specified using integer constants; the constant must appear in parentheses following the label name. For example:

```
PART(1):
  .
  .
PART(2):
  .
```

When labels are written as shown here, the unscripted label name represents the implicit declaration of a label array constant. In this example, the array is named PART. This array is treated as if it were declared within the block containing the subscripted labels. Elements of this array may be referenced in GOTO statements that specify a subscript, for example:

```
GOTO PART(I);
```

where I is a variable whose value represents the subscript of the element of PART that is the label to be given control.

Within a single block, the same subscript value may not be used in two different subscripted references with the same name. For example, the label array constant

```
PART(1):
```

can be used only once in a block. However, the subscript values are not constrained to be in any particular order or to be consecutive. For example, you can use the array constants PART(1) and PART(3) without using PART(2).

If a name is used as a label array constant in two or more different blocks, each declaration of the name is treated as an internal declaration. For example:

```
LIST(2): RETURN;
BEGIN;
   GOTO LIST (ELEMENT);
   LIST(1):
   LIST(3):
   END;
```

In the preceding example, the value of ELEMENT cannot result in control passing to the RETURN statement labeled LIST(2) in the containing block. The subscripted LIST labels in the begin block restrict the scope of the name to that block. (**See** "Scope of Names" for a further illustration of the scope of internal names.)

■ Label Values

Whenever a reference to a label constant is interpreted, the result is a label value. A label value has two components:

- The first component designates the statement identified by the label constant.
- The second component designates an activation of the block in which the label was declared (that is, to which the labeled statement belongs). If the label belongs to the current block, this block activation is the current block activation. If the label belongs to a containing block, the activation is found on the chain of parent block activations ending on the current block. (For additional details on block activations, **see** "Block.")

The statement

GOTO label-reference;

transfers control to the designated statement in the designated block activation. If the target block activation is different from the block activation in

which the GOTO statement is executed, then this is a nonlocal GOTO. For example:

```
DECLARE LV LABEL; /* LABEL variable */
  .
  .
LV = L; /* assigns a bound label value to LV */

BEGIN;
  .
  .
GOTO LV; /* nonlocal GOTO */
END;

L: RETURN;
```

Operations on label values are restricted to the operators = and ^=, for testing the equality or inequality of two values. Two values are equal if they refer to the same statement in the same block activation.

Any reference to a label value after its block activation ceases to exist is an error with unpredictable results.

■ Label Variables

When an identifier is explicitly declared with the LABEL attribute, it acquires the VARIABLE attribute by default. Such a variable can be used to denote different label values during the execution of the program. For example:

```
DECLARE PROCESS LABEL;
    .
    .
    IF CODE THEN
       PROCESS = BILLING;
    ELSE
       PROCESS = CHARGE;
    .
    .
GOTO PROCESS;
```

When the GOTO statement evaluates the reference to the label PROCESS, the result is the current value of the variable. The GOTO statement transfers control to either of the labels BILLING or CHARGE, depending on the current value of the Boolean variable CODE.

Label variables may also be given values by passing label values as arguments and by returning a label value as the value of a function (although the latter usage may lead to programming errors that are difficult to diagnose). For example:

```
CALL COMPUTER(ERROR_EXIT, YVAL, XVAL);

    .
    .
ERROR_EXIT:
```

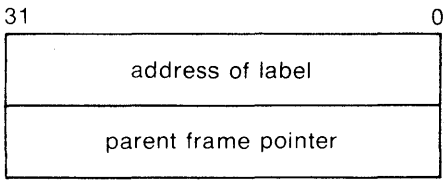
In this example, the actual argument that is passed for ERROR_EXIT is a dummy argument whose value consists of:

- The location in memory of the statement labeled ERROR_EXIT
- A pointer to the stack frame for the block in which the CALL statement is executed

■ Restrictions

- Any statement in a PL/I program can be labeled except:
  - A DECLARE statement
  - A statement beginning an ON-unit, THEN clause, or ELSE clause.
- Labels on PROCEDURE, ENTRY, and FORMAT statements are not considered statement labels and may not be used as the targets of GOTO statements.
- An identifier occurring as a label in a block may not be declared in that block (except as a structure member), and may not occur in a parameter list of that block.

■ Internal Representation of Variable Label Data



LABEL Attribute

The LABEL attribute declares a label variable; it indicates that values given to the variable will be statement labels.

■ Restrictions

You cannot specify the LABEL attribute with any other data type attribute or with any file description attributes.

LBOUND Built-In Function

The LBOUND built-in function returns a fixed-point binary integer that is the lower bound of a specified dimension of an array. Its format is:

LBOUND(reference,dimension)

reference

The name of the array variable.

dimension

An integer constant indicating the dimension of the specified array.

See "Array — Array-Handling Functions" for an example.

Length Attribute

The length attribute is applied to character-string and bit-string data. The length of a string is the number of characters or bits in the string, or the maximum length of the string if the string has the CHARACTER VARYING attributes.

For the rules for specifying the length of a character- or bit-string variable, **see** "Extent."

LENGTH Built-In Function

The LENGTH built-in function returns a fixed-point binary integer that is the number of characters or the number of bits in a specified character- or bit-string expression. If the string is a varying-length character string, the function returns its current length. The format of the function is:

LENGTH(string)

LINE Format Item

The LINE format item sets a print file to a specific line. It can be used only with print files and the PUT EDIT statement. If necessary, blank lines are inserted between the current file position and the specified line, and subsequent output begins on the specified line.

The LINE format item identifies an absolute line position on the current output page; to specify a line position relative to the current line, **see** "SKIP Format Item."

The form of the LINE format item is:

LINE(w)

w

An integer that specifies a line on the current page, where line 1 is the first line.

When the LINE format item is executed, the current line is determined. The current line is 1 if the file is at the beginning of a new page. Otherwise, the current line is n+1, where n is the number of complete lines already on the page. The position in the file is then changed as follows:

- If line w is the current line, and the file is either at the beginning of a new line or at the beginning of a new page, then no operation is performed.
- If line w is beyond the current line and is less than or equal to the current page size, then the file is positioned at line w, and the lines between the current line and line w are filled with blank lines. (**See also** "PAGESIZE Option.")
- If line w is at or before the current line, the current line is not beyond the current page size, and the file is not at the beginning of a line or page, then the remainder of the page (the portion between the current line and the current page size) is filled with blank lines, and the ENDPAGE condition is signaled. The same actions occur when the current line is less than or equal to the page size and w is greater than the page size.
- Otherwise (for instance, when w is zero), the file is positioned at the beginning of a new page, and the page number is incremented by 1.

LINE Option

The LINE option is used with the PUT statement to output data to a specific line in a print file. The output file is positioned at the beginning of the specified line.

The LINE option refers to an absolute line position relative to the beginning of the current page. To refer to a line position relative to the current line, use the SKIP option.

For further information on the LINE and SKIP options, **see** "PUT Statement."

LINENO Built-In Function

The LINENO built-in function returns a FIXED BINARY(15) integer that is the current line number of the referenced print file. Its format is:

LINENO(reference)

If the referenced print file is closed, the returned value is the last value from the previous opening. If the file was never opened, the returned value is zero.

LINESIZE Option

The LINESIZE option specifies the maximum number of characters that can be output in a single line when the PUT statement writes data to a file with the STREAM and OUTPUT attributes. The format of the LINESIZE option is:

LINESIZE(expression)

expression

A fixed-point binary expression in the range 1–32767, giving the number of characters per line.

The value specified in the LINESIZE option is used as the output line length for all subsequent output operations on the stream file, overriding the system default line size.

The default line size is as follows:

- If the output is to a physical record-oriented device, such as a line printer or terminal, the default line size is the width of the device.
- If the output is to a print file, the default line size is 132.
- If the output is to a nonrecord device (magnetic tape), the default line size is 510.

The line size is used by output operations to determine whether output will be placed on the current line or on the next line.

LIST Option

The LIST option is used with the GET and PUT statements to perform list-directed input or output to a stream file or character-string expression.

When the LIST option is used with the GET statement, strings of ASCII characters are acquired from the stream file and assigned to a list of input targets (variables). Conversions to the data types of the input targets, if necessary, are performed automatically. If the end of the input file is encountered, the ENDFILE condition is signaled. For additional details, **see** "GET Statement — GET LIST" and "Stream Input/Output."

When the LIST option is used with the PUT statement, a list of output sources (expressions) is evaluated and converted automatically to strings of ASCII characters, which are then written out. If the output file is a print file, character strings are not enclosed in apostrophes, and all output items are separated by tabs. All output data is appended at the current end-of-file. For additional details, **see** "PUT Statement — PUT LIST" and "Stream Input/Output."

List Processing

Linked lists or queues are processed in PL/I by using based variables and pointers or offsets. The principal language facilities are:

- The BASED, POINTER, AREA, and OFFSET attributes
- The ADDR, NULL, POINTER, and OFFSET built-in functions
- The ALLOCATE and FREE statements, and the DO REPEAT form of the DO statement
- The locator qualifier (->)

Each of these elements is described under its own entry in this manual. This section provides examples of simple procedures that create and process a linked list.

Figure L–1 illustrates a simple program that reads data from a terminal and constructs a linked list from the data.

```
MAKE_LIST: PROCEDURE;

DECLARE (FIRST, CURRENT, SAVE) POINTER;
DECLARE 1 LIST BASED,
        2 NEXT POINTER,
        2 DATA CHARACTER(120) VARYING;
DECLARE PRINT_LIST ENTRY(POINTER,FIXED BINARY);
DECLARE NULL BUILTIN;

/* declare a bit variable to test for end of stream input */
/* and set an ON-unit to finish processing                */
DECLARE EOF BIT(1) STATIC INITIAL('0'B);
ON ENDFILE(SYSIN) EOF = '1'B;

        FIRST = NULL;                   /* initialize queue head */
        DO WHILE (^EOF);
                ALLOCATE LIST SET(CURRENT);     /* get storage */
            GET LIST (CURRENT->DATA);           /* get data */

            IF FIRST = NULL THEN                /* first time through */
                FIRST = CURRENT;                /* set queue head */
            ELSE                                /* all other times */
                SAVE->NEXT = CURRENT;           /* set forward pointer */

            CURRENT->NEXT = NULL;               /* set forward pointer */
            SAVE = CURRENT;     /* save pointer to this allocation */

            END;

        CALL PRINT_LIST(FIRST,120);
        RETURN;

END MAKE_LIST;
```

Figure L–1: Creating a Linked List

Figure L–2 illustrates using pointers to step through a linked list and to print the data in each list element. The example in this figure uses the REPEAT option of the DO statement to modify the value of the pointer used to access each element in the list. This example may also be applied to a linked list within an area. Based variables in an area are referenced by offset values that indicate the locations of the variables with respect to the beginning of the area.

```
PRINT_LIST: PROCEDURE (QUEUE_HEAD,DATA_LENGTH);

DECLARE QUEUE_HEAD POINTER,                 /*start of queue*/
        DATA_LENGTH FIXED BINARY(31);       /*length of data*/
DECLARE 1 LIST BASED(P),                    /* structure of queue elements*/
          2 NEXT POINTER,
          2 DATA CHARACTER(DATA_LENGTH);
DECLARE P POINTER;
DECLARE NULL BUILTIN;

/* start output at queue head, repeat with
next pointers */
DO P = QUEUE_HEAD REPEAT P->LIST.NEXT
/* until end of list (null) encountered */
         WHILE (P ^= NULL);
         PUT SKIP LIST(P->LIST.DATA);
END;

RETURN;
END PRINT_LIST;
```

Figure L–2: Processing a Linked List

Locator Qualifier

A locator qualifier is an operator that specifies the storage associated with a based variable. The locator qualifier consists of the two symbols:

```
->
```

No blanks are allowed between the minus sign (–) and greater than symbol (>).

The format for specifying a locator-qualified reference to a variable is:

```
locator -> based-variable
```

locator

One of the following:

- The name of a pointer whose current value represents the storage associated with a variable.
- The name of an offset variable that was declared with an area reference and whose current value represents the storage of a based variable within the area.
- Any other pointer-valued expression, such as a reference to the POINTER or ADDR built-in function.

based-variable

The name of the based variable whose storage is to be referenced.

You must use a locator qualifier when you refer to a based variable for which more than one allocation of storage may exist. For example:

```
DECLARE NAMES (10) CHARACTER (20) BASED,
        (CLASS_PTR, GRADE_PTR) POINTER;

ALLOCATE NAMES SET (CLASS_PTR);
ALLOCATE NAMES SET (GRADE_PTR);
```

Any reference to the array NAMES in this example must specify the pointer associated with the the storage allocated for the variable, as shown below:

```
CLASS_PTR -> NAMES(1) = 'JONES';
```

This assignment statement refers to the storage allocated for the array NAMES that is pointed to by the pointer CLASS_PTR. The assignment sets the first element of the array to the string JONES.

You must also use a locator qualifier to associated a based variable with the storage of another variable. For example:

```
DECLARE DATA CHARACTER(10) BASED,
        DP POINTER,
        LINE CHARACTER(10);

LINE = 'string';
DP = ADDR(LINE);
PUT LIST( DP->DATA );
```

The locator qualifier in this PUT statement associates the based variable DATA with the storage occupied by the variable LINE, pointed to by the pointer DP. For more information, **see** "Based Variable," "Offset," and "Pointer."

LOG Built-In Function

The LOG built-in function returns a floating-point value that is the base *e* (natural) logarithm of an arithmetic expression x. The computation is performed in floating point. The expression x must be greater than zero after its conversion to floating point.

The format of the function is:

LOG(x)

LOG10 Built-In Function

The LOG10 built-in function returns a floating-point value that is the base 10 logarithm of arithmetic expression x. The computation is performed in floating point. The expression x must be greater than zero after its conversion to floating point. The format of the function is:

LOG10(x)

LOG2 Built-In Function

The LOG2 built-in function returns a floating-point value that is the base 2 logarithm of an arithmetic expression x. The computation is performed in floating point. The expression x must be greater than zero after its conversion to floating point. The format of the function is:

LOG2(x)

Logical Operator

The logical operators perform logical operations on one or two operands. The operands of the AND and OR operators must be bit-string expressions. The operand of the NOT operator can be a bit-string expression or a single relational operator. All relational expressions result in bit-string values of length one, and they may therefore be used as operands in logical operations.

Except when the NOT operator is used as the prefix of a relational operator, the result of a logical operation is always a bit string.

Logical operations are performed on their operands bit by bit. If bit-string operands are not the same length, PL/I extends the smaller of the operands on the right (that is, in the direction of the least significance) with zeros to match the length of the larger operand. This length is always the length of the result.

There are two infix operators and one prefix operator.

The prefix operator is:

Operator	Operation
^	Logical NOT. In a logical NOT operation, the value of the operand is complemented; that is, a 1 bit becomes a 0 and a 0 bit becomes a 1. The value of a relational expression is also complemented; that is ^(A < B) is equivalent to (A >= B).

The infix operators are:

Operator	Operation
&	Logical AND. In a logical AND operation, two operands are compared. If both corresponding bits are 1, the result is 1; otherwise, the result is 0.
| or !	Logical OR. In a logical OR operation, two operands are compared. If either or both of two corresponding bits are 1, the result is 1; otherwise the result is 0. (The | and the ! characters can be used interchangeably.)

You can define additional operations on bit strings with the BOOL built-in function.

Logical expressions may not be completely evaluated in some cases. If the result of the total expression can be determined from the value of one or more individual operands, the evaluation may be terminated. For example, in the expression:

```
A & B & C & D & E
```

evaluation may stop when any operand or the result of any operation is a bit string containing all zeros.

■ Examples

```
DECLARE (BITA,BITB,BITC) BIT(4);
BITA = '0001'B;
BITB = '1001'B;
BITC = ^BITA;
       /* BITC equals '1110'B */
BITC = BITA | BITB;
       /* BITC equals '1001'B */
BITC = BITA & BITB;
       /* BITC equals '0001'B */
BITC = ^(BITA & BITB);
       /* BITC equals '1110'B */
BITC = ^(BITA > BITB);
       /* BITC equals '1000'B (true) */
```

In the last assignment statement, the relational expression yields '1'B; when this value is assigned to BITC, a BIT(4) variable, the value is padded with zeros and becomes '1000'B.

M

MAIN Option

The MAIN option can be specified with the OPTIONS keyword on the PROCEDURE statement. It indicates that the specified entry name is the primary invocation point of the program. It is specified as follows:

entry-name: PROCEDURE OPTIONS (MAIN);

One, and only one, procedure in a program can specify the MAIN option. If no procedure specifies the MAIN option, the invocation point of the program is the first procedure in the image. (For details on binding procedures into an executable program image, **see** the *VAX-11 PL/I User's Guide.*)

VAX-11 PL/I provides a default ON-unit for the procedure declared with the MAIN option. **See** "ON Conditions and ON-Units."

Main Procedure

The main procedure in a program is the procedure declared with the MAIN option. Execution of a PL/I program begins with the main procedure.

See also "MAIN Option" and "Procedure."

MAX Built-In Function

The MAX built-in function returns the larger of two arithmetic expressions x and y. The format of the function is:

MAX(x,y)

■ Returned Value

The expressions x and y are converted to their derived type before the operation is performed (for a discussion of derived types, **see** "Expression"). If the derived type is floating point, the value returned is also floating point, with the larger precision of the two converted arguments. If the derived type is fixed point, the returned value is a fixed-point value with the base of the derived type and with the attributes:

precision = min(31,max(px-qx,py-qy)+max(qx,qy))

scale factor = max(qx,qy)

where px,qx and py,qy are the converted precisions and scale factors of x and y.

MIN Built-In Function

The MIN built-in function returns the smaller of two arithmetic expressions x and y. Its format is:

MIN(x,y)

■ Returned Value

The expressions x and y are converted to their derived type before the operation is performed (for a discussion of derived types, **see** "Expression"). If the derived type is floating point, the value returned is also floating point, with the larger precision of the two converted arguments. If the derived type is fixed point, the returned value is a fixed-point value with the base of derived type and with the attributes:

precision = min(31,max(px-qx,py-qy)+max(qx,qy))

scale factor = max(qx,qy)

where px,qx and py,qy are the converted precisions and scale factors of x and y.

MOD Built-In Function

The MOD built-in function returns, for an arithmetic expression x and non-negative arithmetic expression y, the value r that equals x modulo y. That is, r is the smallest positive value that must be subtracted from x to make the remainder exactly divisible by y. (For the result when y is negative, **see** "Returned Value" below.)

The format of the function is:

MOD(x,y)

■ Returned Value

The expressions x and y are converted to their derived type before the operation is performed (**see** "Expression" for a discussion of derived types).

If the derived type is fixed-point binary or unscaled fixed-point decimal, then the result precision is the precision of the second operand.

If the derived type is floating point, the returned value is an approximation in floating point, with the larger precision of the two converted arguments.

The value returned is:

u-w*floor(u/w)

where u and w are the arguments x and y, respectively, after conversion to their derived type. If w is zero, u is converted to the precision described below, which may signal FIXEDOVERFLOW.

If x and y are fixed-point expressions, the returned value is a fixed-point value with the attributes:

precision = min(31,pw–qw+max(qu,qw))

scale factor = max(qu,qw)

where qu is the scale factor of u, pw is the precision of w, and qw is the scale factor of w. The FIXEDOVERFLOW condition is signaled if:

pw–qw + max(qu,qw) > 31

■ Examples

```
MODEX: PROCEDURE OPTIONS(MAIN);

DECLARE OUTMOD PRINT FILE;

ON FIXEDOVERFLOW PUT FILE(OUTMOD)
      SKIP LIST('FIXEDOVERFLOW signaled');

PUT FILE(OUTMOD) SKIP LIST(MOD(28,128));
PUT FILE(OUTMOD) SKIP LIST(MOD(130,128));
PUT FILE(OUTMOD) SKIP LIST(MOD(-28,128));
PUT FILE(OUTMOD) SKIP LIST(MOD(4.5,.758));
PUT FILE(OUTMOD) SKIP LIST(MOD(-4.5,.758));
PUT FILE(OUTMOD) SKIP LIST(MOD(1.5E-3,-1.4E-3));
PUT FILE(OUTMOD) SKIP LIST(MOD(28,0));

END MODEX;
```

The program MODEX writes the following output to OUTMOD.DAT:

```
    28
     2
   100
 0.710
 0.048
-1.3E-03

FIXEDOVERFLOW signaled      8
```

The last PUT statement attempts to take MOD(28,0). The constants 28 and 0 are both fixed-point decimal expressions, with precisions (2,0) and (1,0), respectively. Therefore, the attributes of the returned value are determined to be FIXED DECIMAL, with:

precision = min(31,1–0+max(0,0)) = 1

scale factor = max(0,0) = 0

Although 28 mod 0 is 28, MOD(28,0) signals FIXEDOVERFLOW because 28 cannot be represented in the result precision. (The value of the function is therefore undefined.)

Multiplication

The asterisk character (*) indicates a multiplication operation in an expression; the result is the product of the operands. Both operands must be arithmetic or picture data.

■ Conversion of Operands

If both operands have the same base, precision, and scale, so has the result of the operation. The compiler converts operands of different data types as follows:

- If one operand has the FLOAT attribute and the other has the FIXED attribute, the fixed-point operand is converted to floating point before the operation.
- If one operand is FIXED DECIMAL and the other is FIXED BINARY, the fixed-point binary operand is converted to fixed-point decimal. However, the compiler issues a warning message to that effect.

The precision of the values resulting from conversion of operands is described under "Expression."

■ Precision of the Result

Floating-Point Operands

The result has the maximum of the converted precisions of the operands.

Fixed-Point Decimal Operands

If (p,q) and (r,s) represent the converted precisions and scale factors of the two operands, the resulting precision and scale factor are:

precision: min(31,p+r+1)
scale factor: q+s

Fixed-Point Binary Operands

If (p) and (r) represent the converted precisions of the two operands, the resulting precision is:

min(31,p+r+1)

NOT Operator

The ^ (circumflex) character is the logical NOT operator in PL/I. In a logical NOT operation, the value of a bit is reversed. If a bit is 1, the result is 0; if a bit is 0, the result is 1.

The NOT operator can be used on expressions that yield bit-string values (bit-string, relational, and logical expressions).

It can also be used to negate the meanings of the relational operators (<, >, =). For example:

```
IF A ^> B THEN ...
   /* equivalent to IF A <= B THEN ... */
```

The result of a logical NOT operation on a bit-string expression is a bit-string value. For example:

```
DECLARE (BITA, BITB) BIT (4);
BITA = '0011'B;
BITB = ^BITA;
```

The resulting value of BITB is '1100'B.

The NOT operator can test the falsity of an expression in an IF statement. For example:

```
IF ^(MORE_DATA) THEN ...
```

See "Logical Operator" and "Operator."

NULL Built-In Function

The NULL built-in function returns a null pointer value. Its format is:

NULL()

■ Example

```
IF NEXT_POINTER = NULL() THEN CALL TERM;
```

The IF statement checks whether the pointer variable NEXT_POINTER is null; if so, it executes the CALL statement. For more information, **see** "Based Variable," "List Processing," and "Pointer."

Null Statement

The null statement performs no action. Its format is:

```
;
```

The most common use for the null statement is as the target statement of a THEN or ELSE clause in an IF statement, or as an action in an ON-unit. For example:

```
ON ENDPAGE(SYSPRINT);
```

The null statement can also be used to declare two labels for the same executable statement, as in:

```
LABEL1:  ;  LABEL2:  statement ...
```

O

Offset

An offset is a value indicating the location of a based variable within an area relative to the beginning of the area. An offset variable must be declared with the OFFSET attribute.

When an area is transmitted or assigned, the offset values associated with variables within the area remain valid.

■ Offset Assignment

Offset variables are given values by assignment from existing offset values or from conversion of pointer values. The OFFSET built-in function converts a pointer value to an offset value. PL/I also automatically converts a pointer value to an offset value, and vice versa, in an assignment statement. The following assignments are valid:

1. pointer-variable = pointer-value ;
2. offset-variable = offset-value ;
3. pointer-variable = offset-variable ;
4. offset-variable = pointer-value ;

In (2), any area references are ignored in the assignment; therefore, the offset value and variable can refer to different areas. In (3) and (4), the offset variable must have been declared with an area reference.

■ Offset Variables in Expressions

Expressions containing offset variables are restricted to the following operators:

Operator	Meaning
=	Equal
^=	Not equal

For more information on offset values, **see** "Area." Specific details on how to allocate variables within areas and how to determine the offsets of these variables are contained in the *VAX-11 PL/I User's Guide.*

OFFSET Attribute

The OFFSET attribute declares a variable that will be used to reference a based variable within an area. Its format is:

OFFSET [(area-reference)]

area-reference

The name of a variable with the AREA attribute. The value of the offset variable will be interpreted as an offset within the specified area.

Example

```
DECLARE MAP_SPACE AREA (4096),
        MAP_START OFFSET (MAP_SPACE),
        MAP_LIST(100) CHARACTER(80) BASED (MAP_START);
```

These declarations define an area named MAP_SPACE, an offset value that will contain offset values within that area, and a based variable whose storage is located by the value of the offset variable MAP_START.

Restrictions

The area reference must be omitted if the OFFSET attribute is specified within a returns descriptor, parameter declaration, or a parameter descriptor. The OFFSET attribute conflicts with all other data type attributes.

OFFSET Built-In Function

The OFFSET built-in function converts a pointer to an offset relative to a designated area. If the pointer is null, the result is null. The format of the function is:

OFFSET(pointer,area)

pointer

A reference to a pointer variable whose current value either represents the location of a based variable within the specified area or is null.

area

A reference to a variable declared with the AREA attribute. If the specified pointer is not null, it must designate a storage location within this area.

Example

```
DECLARE MAP_SPACE AREA (2048),
        START OFFSET (MAP_SPACE),
        QUEUE_HEAD POINTER;
    .
    .
START = OFFSET (QUEUE_HEAD,MAP_SPACE);
```

The offset variable START is associated with the area MAP_SPACE. The OFFSET built-in function converts the value of the pointer to an offset value.

ON Conditions and ON-Units

An ON condition is any one of several named conditions whose occurrences during the execution of a program interrupt the program. When an ON condition occurs, or is signaled, a statement or sequence of statements, called an ON-unit, is executed.

This discussion of ON conditions has the following subheadings:

- Summary of ON conditions
- Default PL/I ON-unit
- Establishment of ON-units
- Contents of an ON-unit
- Search for ON-units
- Completion of ON-units

■ Summary of ON Conditions

Most, but not all, ON condition names are associated with errors. The types of condition for which you can establish ON-units are grouped in the categories listed below.

- Conditions that occur during input/output operations. The ON-units you can define for these conditions are:
  - ENDFILE, to take action when the end-of-file occurs during reading a file
  - ENDPAGE, to take action when the last line on a page is printed
  - KEY, to take action when an error occurs accessing a record by key
  - UNDEFINEDFILE, to respond to any file-specific errors that can occur during the opening of a file
- Conditions that indicate arithmetic conditions related to hardware violations. The ON-units you can define for these conditions are:
  - FIXEDOVERFLOW, to respond when integer or fixed-point decimal values become too large to be expressed
  - OVERFLOW, to respond when floating-point values become too large to be expressed
  - UNDERFLOW, to respond when floating-point values become too small to be expressed
  - ZERODIVIDE, to respond when the divisor in a division operation has a value of zero
- General classes of exceptional conditions. The ON-units you can define are:
  - ANYCONDITION, to respond to all conditions for which no specific ON-unit is established in the current block
  - ERROR, to respond to language- and run-time-specific errors
  - FINISH, to respond when a STOP statement is executed
  - VAXCONDITION, to respond to operating-system-specific condition values

Each condition is described individually in this manual under its own heading.

■ Default PL/I ON-Unit

PL/I defines a default ON-unit for the procedure that is designated as the main procedure. This default ON-unit performs the following actions:

- If the signal is the ENDPAGE condition, the default PL/I handler executes a PUT PAGE for the file, and then continues the program at the point at which ENDPAGE was signaled.
- If the signal is the ERROR condition and the severity is fatal, the default handler signals the FINISH condition. Then, one of the following occurs:
  - If a FINISH ON-unit is found, it is given a chance to execute. If it executes a nonlocal GOTO or signals another condition, program execution continues.
  - If no FINISH ON-unit is found, or if a FINISH ON-unit completes execution by handling the condition, then PL/I resignals the condition to the default VAX/VMS condition handler. This handler prints a message, displays a traceback, and terminates the program.
- If the signal is any condition other than ENDPAGE or ERROR with a fatal severity, the default PL/I handler signals the ERROR condition with the severity of the original condition. Then, one of the following occurs:
  - If an ERROR ON-unit is found, it is executed. If it completes execution by handling the condition, the program continues.
  - If an ERROR ON-unit is not found, the default PL/I handler resignals the condition. If this resignal results in return of control to the system, the default VAX/VMS condition handler prints a message and a traceback. If the error is a fatal error, the default handler terminates the program; if the error is nonfatal, the program continues.

■ Establishment of ON-Units

An ON-unit is established for a specific ON condition following the execution of an ON statement that specifies that condition name. For example:

```
ON ENDFILE (ACCOUNTS) GOTO CLOSE_FILES;
```

This ON statement defines an ON-unit for an ENDFILE (end-of-file) condition in the file specified by the name ACCOUNTS. The ON-unit consists of a single statement, a GOTO statement.

After an ON-unit is established by an ON statement for a condition, it remains in effect for the activation of the current block and all its dynamically descendent blocks, unless one of the following occurs:

- Another ON statement is specified for the same condition in a descendent block. The ON-unit established within the descendent block remains in effect as long as the descendent block is active.
- A REVERT statement is executed for the specified condition. A REVERT statement nullifies the most recent ON-unit for the specified condition.

- Another ON statement is specified for the same condition within the current block. Within the same block, an ON statement for a specific condition cancels the previous ON-unit.
- The block or procedure within which the ON-unit is established terminates. When a block exits, any ON-units it has established are canceled.

■ Contents of an ON-Unit

An ON-unit can consist of a single simple statement, a group of statements in a begin block, or a null statement.

Simple Statements in ON-Units

The following ON statement specifies a single statement in the ON-unit:

```
ON ERROR GOTO WRITE_ERROR_MESSAGE;
```

This ON statement specifies a GOTO statement that transfers control to the label WRITE_ERROR_MESSAGE in the event of the ERROR condition.

A simple statement must not be labeled and must not be any of the following:

DECLARE	FORMAT	RETURN
DO	IF	
END	ON	
ENTRY	PROCEDURE	

Begin Blocks in ON-Units

An ON-unit can also consist of a sequence of statements in a begin block, for example:

```
ON ENDFILE (SYSIN) BEGIN;
   CLOSE FILE (TEMP);
   CALL PRINT_STATISTICS(TEMP);
   END;
```

This ON-unit consists of CLOSE and CALL statements that request special processing when the end-of-file condition occurs during reading of the default system input file, SYSIN.

If a BEGIN statement is specified for the ON-unit, the BEGIN statement must not be labeled. The begin block can contain any statement except a RETURN statement.

Null Statements in ON-Units

A null statement specified for an ON-unit indicates that no processing is to occur when the condition occurs. Program execution continues as if the condition had been handled. For example:

```
ON ENDPAGE(SYSPRINT);
```

This ON-unit causes PL/I to continue output on a terminal regardless of the number of lines that have been output.

■ Search for ON-Units

When a condition is signaled during the execution of a PL/I procedure, PL/I searches for an ON-unit to respond to the condition. It first searches the current block, that is, the block in which the condition occurs. If no ON-unit exists in this block for the specific condition, it searches the block that activated the current block (its "parent"), and then the block that activated that block, and so on.

PL/I executes the first ON-unit it finds, if any, that can handle the specified condition. If no ON-unit for the specific condition is found, the default PL/I condition handling is performed.

Figure O–1 illustrates a program with ON-units established at several levels of block activation and describes the sequence in which the ON-units are located.

For more information on blocks and block activation, **see** "Block." For a more detailed explanation of the search for ON-units and a description of how PL/I ON-units relate to condition-handling routines that may be written in other programming languages, **see** the *VAX–11 PL/I User's Guide.*

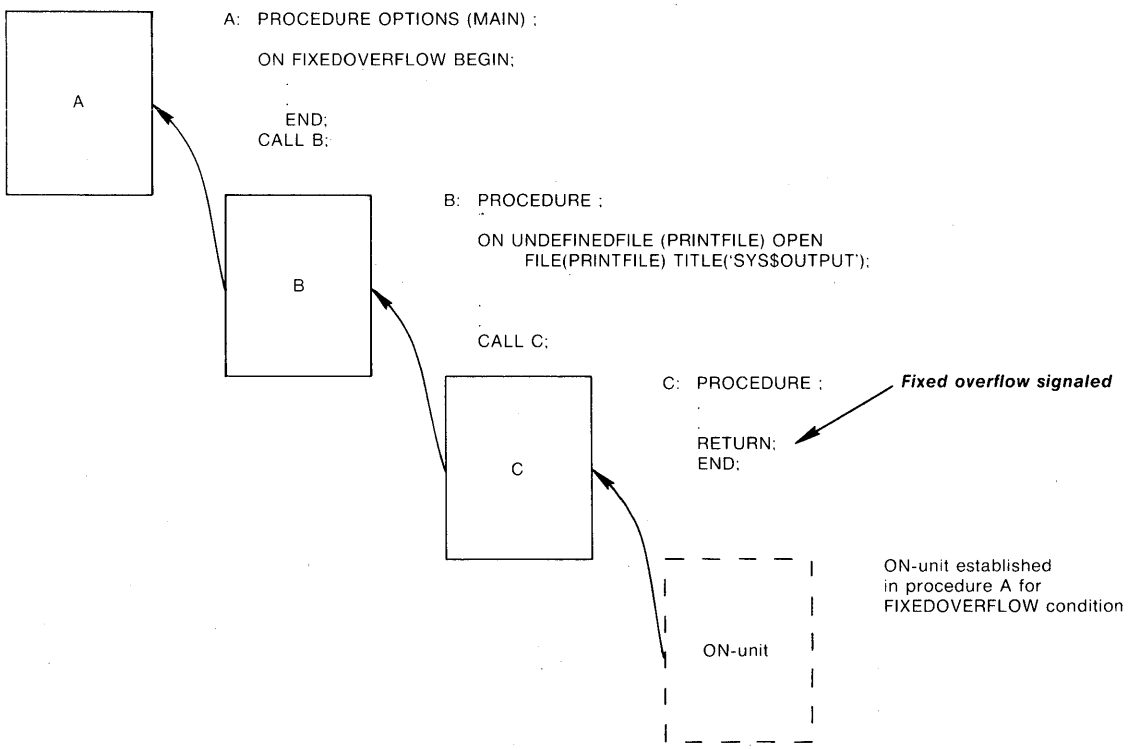


Figure O–1: Search for ON-Units

■ Completion of ON-Units

PL/I executes an ON-unit as if the unit were a procedure having no parameters; that is, it creates a block activation for the ON-unit and links it to the block in which the condition occurred. The ON-unit can complete its execution in any of the following ways:

- If the ON-unit executes a nonlocal GOTO statement, or if it invokes a subroutine or function that executes a nonlocal GOTO, program control is transferred to that statement and continues sequentially at that point in the program.
- If the ON-unit executes a STOP statement, then the FINISH condition is signaled. If no FINISH ON-unit exists, the program is terminated.
- An ON-unit can use the RESIGNAL built-in subroutine to request that PL/I continue to search for an ON-unit to handle a specific condition. For a description of this built-in subroutine and an explanation of the effects of a nonlocal GOTO and resignaling in the VMS environment, **see** the *VAX-11 PL/I User's Guide*.
- Normal completion of any ON-unit except an ERROR ON-unit or a FINISH ON-unit executed as a result of image exit results in return of control either to the statement that caused the condition or to the statement immediately following the statement that caused the condition.

Descriptions of each ON condition in this manual indicate the action that PL/I takes on completion of an ON-unit associated with the condition.

ON Statement

The ON statement defines the action to be taken when a specific condition is signaled during the execution of a program. The format of the ON statement is:

```
ON  condition-name  on-unit ;
```

condition-name

The name of the specific condition for which an ON-unit is established. There is a keyword name associated with each condition. The conditions are summarized in Table O–1; each condition is described individually, as an entry in this manual.

on-unit

The action to take when the specified condition is signaled. An ON-unit can be any single, unlabeled statement except DECLARE, DO, END, ENTRY, FORMAT, IF, ON, PROCEDURE, or RETURN. It can also be an unlabeled begin block.

If no ON-unit is established for a particular condition, the default PL/I ON-unit, if any, is executed.

For information on ON-units and the default PL/I ON-unit, **see** "ON Conditions and ON-Units."

Table O–1: Summary of ON Conditions

Condition Name	Usage
ANYCONDITION	Handles any condition not specifically handled by another ON-unit
ENDFILE	Handles end-of-file condition for a specified file
ENDPAGE	Handles end-of-page for a specified file with PRINT attribute
ERROR	Handles miscellaneous error conditions and conditions for which no specific ON-unit exists
FINISH	Handles program exit when the main procedure executes a RETURN statement, when any block executes a STOP statement, or when the program exits due to an error that is not handled by an ON-unit
FIXEDOVERFLOW	Handles fixed-point decimal and integer overflow exception conditions
KEY	Handles any error involving the key when using keyed access to a specified file
OVERFLOW	Handles floating-point overflow exception conditions
UNDEFINEDFILE	Handles any errors opening a specified file
UNDERFLOW	Handles floating-point underflow exception conditions
VAXCONDITION	Handles a specifically signaled condition value
ZERODIVIDE	Handles divide-by-zero exception conditions

ONARGSLIST Built-In Function

The ONARGSLIST built-in function returns a pointer to the location in memory of the argument list for an exception condition. If the ONARGSLIST built-in function is referenced in any context outside of an ON-unit, it returns a null pointer. Its format is:

ONARGSLIST()

The format of the argument list and the information available to an ON-unit from the argument list are described in the *VAX–11 PL/I User's Guide.*

ONCODE Built-In Function

The ONCODE built-in function returns a fixed-point binary integer that is the status value of the most recent run-time error that signaled the current ON condition. The function may be used in any ON-unit to determine the specific error that caused the condition. If the function is used within any context outside an ON-unit, it returns a zero. Its format is:

ONCODE()

For details on the condition values returned by ONCODE and examples of using the ONCODE built-in function, **see** the *VAX–11 PL/I User's Guide.*

ONFILE Built-In Function

The ONFILE built-in function returns the name of the file constant for which the current file-related condition was signaled. Its format is:

```
ONFILE()
```

This built-in function can be used in an ON-unit established for any of the following conditions:

- An ON-unit for the KEY, ENDFILE, ENDPAGE, and UNDEFINEDFILE conditions
- A VAXCONDITION ON-unit established for input/output errors that can occur during file processing
- An ERROR ON-unit that receives control as a result of the default PL/I action for file-related errors, which is to signal the ERROR condition

■ Returned Value

The returned value is a varying-length character string. If referenced outside an ON-unit or within an ON-unit that is executed as a result of a SIGNAL statement, the ONFILE function returns a null string.

ONKEY Built-In Function

The ONKEY built-in function returns the key value that caused the KEY condition to be signaled during an I/O operation to a file that is being accessed by key. Its format is:

```
ONKEY()
```

This built-in function can be used in an ON-unit established for these conditions:

- The KEY, ENDFILE, or UNDEFINEDFILE conditions
- An ERROR ON-unit that receives control as a result of the default PL/I action for the KEY condition, which is to signal the ERROR condition

■ Returned Value

The returned key value is a varying-length character string. If referenced outside an ON-unit, or within an ON-unit executed as a result of the SIGNAL statement, the ONKEY built-in function returns a null string.

OPEN Statement

The OPEN statement explicitly opens a PL/I file with a specified set of attributes that describe the file and the method for accessing it. The format of the OPEN statement is:

```
OPEN FILE(file-reference)
        [file-description-attribute ...] ;
```

file-reference

A reference to the file to be opened. If the file is already open, the OPEN statement has no effect.

file-description-attribute ...

The attributes of the file. The attributes specified are merged with the permanent attributes of the file specified in its declaration, if any. Then, default rules are applied to the union of these sets of attributes to complete the set of attributes in effect for this opening.

The attributes and options you can specify on the OPEN statement are:

DIRECT	PRINT
ENVIRONMENT(option,...)	RECORD
INPUT	SEQUENTIAL
KEYED	STREAM
LINESIZE(expression)	TITLE(expression)
OUTPUT	UPDATE
PAGESIZE(expression)	

Each of these attributes is described in its own entry. For a summary of the valid combinations of these attributes and their meanings, **see** "File Description Attributes and Options." Merging of attributes and default attributes supplied are described under "Opening a File."

■ Examples

```
DECLARE INFILE FILE;

OPEN FILE (INFILE);
```

Neither the file's declaration nor its open specify any file description attributes. PL/I applies the default attributes STREAM and INPUT. If any statement other than a GET is used to process this file, the ERROR condition is signaled.

```
DECLARE STATE_FILE FILE KEYED;

OPEN FILE(STATE_FILE) UPDATE;
   .
   .
CLOSE FILE(STATE_FILE);
OPEN FILE(STATE_FILE) INPUT SEQUENTIAL;
```

The file STATE_FILE is declared with the KEYED attribute. On the first OPEN statement that specifies this file, it is given the UPDATE attribute and opened for updating; that is, READ, WRITE, REWRITE, and DELETE statements may be used to operate on records in the file. The KEYED attribute implies the SEQUENTIAL attribute; thus records in the file may be accessed sequentially or by key.

The second OPEN statement specifies the INPUT and SEQUENTIAL attributes. During this opening, the file may be accessed by sequential and keyed READ statements; REWRITE, DELETE, and WRITE statements may not be used.

```
DECLARE COPYFILE FILE OUTPUT;
OPEN FILE(COPYFILE) TITLE('COPYFILE.DAT');
```

The file constant COPYFILE is opened for output. Each time this program is run, it creates a new version of the file COPYFILE.DAT.

Opening a File

A file is opened explicitly by an OPEN statement or implicitly by a READ, WRITE, REWRITE, DELETE, PUT, or GET statement issued for a file that is not open. In either case, opening a file in PL/I has the following effects:

- The permanent attributes, if any, specified in the DECLARE statement of a file constant, if any, are merged with the attributes specified in the OPEN statement, if any, or with the attributes implied by the context of the opening. (For example, if no attributes are specified for a file in its declaration, and the first reference to the file is a GET statement, PL/I opens the file with the INPUT and STREAM attributes.) The rules that PL/I follows in applying default attributes are described below, under "Establishing the File's Attributes."
- The merged attributes apply to the file for the duration of this opening only. When the file is closed, only its permanent attributes remain in effect.
- The file specification of the file is determined, using the value of the TITLE option.
- If the file already exists, it is located and its attributes are checked for compatibility with the attributes specified or implied by the open.
- If the file does not exist, and if the attempted access does not require that the file exist, PL/I creates a new file using the attributes specified or implied to determine its organization.
- If the open is successful, the file is positioned.

Each of these steps is described in more detail below. If an error occurs during the opening of a file, the UNDEFINEDFILE condition is signaled. (**See** "UNDEFINEDFILE Condition Name.")

■ Establishing the File's Attributes

The file description attributes specified by the opening context are merged with the file's permanent attributes. Duplicate specification of the same attribute is allowed only for an attribute that does not specify a value.

If the set of attributes is not complete, it is augmented with implied attributes. Table O–2 summarizes the attributes that may be added to an incomplete set.

Table O–2: File Description Attributes Implied at Open Time

Attribute	Implied Attributes
DIRECT	RECORD KEYED
KEYED	RECORD
PRINT	STREAM OUTPUT
SEQUENTIAL	RECORD
UPDATE	RECORD

If the set of attributes is still not complete, PL/I uses the steps below to complete the set:

1. If neither STREAM nor RECORD is present, STREAM is supplied.
2. If neither INPUT, nor OUTPUT, nor UPDATE is present, INPUT is supplied.
3. If RECORD is specified, but neither SEQUENTIAL nor DIRECT is present, SEQUENTIAL is supplied.
4. If the file is associated with the external file constant SYSPRINT, and the attributes STREAM and OUTPUT are present but the attribute PRINT is not, PRINT is supplied.
5. If the set contains the LINESIZE option, it must contain STREAM and OUTPUT. If it contains these attributes and does not contain LINESIZE, the default system line size value is supplied.
6. If the set contains the PAGESIZE option, it must contain PRINT. If PRINT is present but PAGESIZE is not, the default system page size is supplied.
7. If the set does not contain TITLE, a default option TITLE(name) is supplied, where name is the name of the file constant associated with the file.

The completed set of attributes applies only for the current opening of the file. The file's permanent attributes, specified in the declaration of the file, are not changed.

■ Determining the File Specification

PL/I uses the value of the TITLE option to determine the file specification, that is, the actual name of the file or device on which the I/O is to be performed. The determination of the file specification depends on the following system-specific functions:

1. The value of the TITLE option may be a logical name or a portion of it may contain a logical name. In either case, the logical name is translated. If the resulting name is a logical name, that name is also translated, to a maximum of ten translations.

2. After logical name translation, VAX-11 PL/I applies default values, if any, specified in the DEFAULT_FILE_NAME option of the ENVIRONMENT attribute list.
3. If the file specification is still not complete, system defaults are applied to the incomplete portions of the file specification. Defaults are provided for node, device, directory, file type and version number. If a file name is not specified, PL/I uses the default name supplied in the TITLE option.

The rules for logical name translation and for the application of system defaults are described in detail in the *VAX-11 PL/I User's Guide.*

■ Accessing an Existing File

An open accesses an existing file if the file specified by the TITLE option actually exists and if the following attributes are present:

- The file is opened for INPUT or UPDATE.
- The file is opened with the OUTPUT attribute and with the ENVIRONMENT (APPEND) option.

Whenever PL/I accesses an existing file, the file's organization is checked for compatibility with the PL/I attributes specified. If any incompatibilities exist, the UNDEFINEDFILE condition is signaled.

■ Creating a File

An open creates a new file if the following are all true:

- The OUTPUT attribute is specified.
- The TITLE option, after logical name translation and the application of system defaults, specifies a mass storage device, for example, a disk or a tape.
- The ENVIRONMENT(APPEND) option is not specified.

The organization and record format of a new file can be specified by ENVIRONMENT options. If no ENVIRONMENT options are given, the new file's organization is determined as follows:

- If the KEYED attribute is present, PL/I creates a relative file with the maximum record size of 512 bytes and the maximum record number of 0.
- If the PRINT attribute is present, PL/I creates a sequential stream file with variable-length records, no maximum record length, and a fixed-control field used by PL/I to store carriage control information.
- If neither KEYED nor PRINT is specified, PL/I creates a sequential file with variable-length records and no maximum record size.

When a file is opened with the RECORD and OUTPUT attributes, only WRITE statements may be used to access the file. If the file has the KEYED attribute as well, the WRITE statements must include the KEYFROM option.

■ File Positioning

When PL/I opens a file, the initial positioning of the file depends on the type of file (record or stream), the access mode, and certain ENVIRONMENT options.

For a definition of the file positioning information for record files, **see** "Record Input/Output." For a definition of file positioning information for stream files, **see** "Stream Input/Output."

Operator

An operator is a symbol that requests a unique operation. It may be a prefix operator or an infix operator.

■ Prefix Operator

A prefix operator precedes a single operand. The prefix operators are the unary plus (+), the unary minus (–), and the logical not (^).

- The plus sign can prefix an arithmetic value or variable. However, it does not change the sign of the operand.
- A minus sign reverses the sign of an arithmetic operand.
- The ^ prefix operator performs a logical NOT operation on a bit-string operand.

Some examples of expressions containing prefix operators are:

```
A = +55;
B = -88;
BITC = ^BITB;
```

■ Infix Operator

An infix operator appears between two operands. It indicates the operation to be performed on the operands. PL/I has infix operators for arithmetic operations, logical operations, relational (comparison) operations, and string concatenations. Some examples of expressions containing infix operators are:

```
RESULT = A / B;
IF NAME = FIRST_NAME || LAST_NAME THEN GOTO NAME_OK;
```

An expression can contain both prefix and infix operators, for example:

```
A = -55 * +88;
```

Prefix and infix operators can be applied to expressions by using parentheses for grouping.

■ Operands

The expressions on which an operation is performed are called operands. All operators must yield scalar values. Therefore, operands may not be arrays or structures. The data type that you can use for an operand in a specific operation depends on the operator:

- Arithmetic operators must have arithmetic operands; if the operands are of different arithmetic types, they are converted to a single type before the operation (**see also** "Expression").
- Logical operators must have bit-string operands.
- Relational operators must have two operands of the same type (arithmetic, bit string, or character string).
- The concatenation operator must have two bit-string operands or two character-string operands.

The categories of operator and the operator characters are listed in Table O-3.

Table O-3: Operators

Category	Symbol	Operation
Arithmetic operators	+	Addition or prefix plus
	-	Subtraction or prefix minus
	/	Division
	*	Multiplication
	**	Exponentiation
Relational (or comparison) operators	>	Greater than
	<	Less than
	=	Equal to
	^>	Not greater than
	^<	Not less than
	^=	Not equal to
	>=	Greater than or equal to
	<=	Less than or equal to
Bit-string (or logical) operators	^	Logical NOT
	&	Logical AND
	| or !	Logical OR
Concatenation operator	|| or !!	String concatenation

■ Precedence of Operations

A PL/I expression can consist of many subexpressions and operands. When an expression contains more than one operator, PL/I uses a defined set of rules to determine which operation to perform first, second, and so on. If the expression contains parentheses, PL/I evaluates expressions within the parentheses (according to the rules of priority) first and then uses the resulting value as a single operand. Unparenthesized operations of equal priority are performed in any order.

Table O–4 gives the priority of PL/I operators. In Table O–4, low numbers indicate high priority; that is, the exponentiation operator (**) has the highest priority and the OR operator (|), the lowest.

Table O–4: Precedence of Operations

Operator	Priority	Operator	Priority
**	1	>	5
+ (prefix)	1	<	5
- (prefix)	1	^>	5
^	1	^<	5
*	2	=, ^=	5
/	2	<=	5
+ (infix)	3	>=	5
- (infix)	3	&	6
||	4	|	7

OPTIONS Option

The OPTIONS keyword option is provided in PL/I statements to request special processing. Normally, this keyword is associated with options that are not part of the standard PL/I language.

The statements that have the OPTIONS keyword option are:

- The input/output statements GET, PUT, READ, WRITE, DELETE, and REWRITE
- The PROCEDURE statement
- The DECLARE statement with the ENTRY attribute

Statement options are specified within parentheses following the OPTION keyword. The individual options within an option list are separated by spaces.

For a list of the valid options, see the entry for the individual statement or attribute.

■ Example

```
APPLIC: PROCEDURE OPTIONS (MAIN,IDENT('APPLIC'));
```

The keywords MAIN and IDENT are options specified by the OPTIONS keyword of the PROCEDURE statement.

OR Operator

The | (vertical bar) or ! (exclamation point) character represents the logical OR operation in PL/I. In a logical OR operation, two bit-string operands are compared bit by bit. If the two operands are of different lengths, the shorter operand is converted to the length of the longer operand, and that is the length of the result. If either of two corresponding bits is 1, the resulting bit is 1; otherwise, the resulting bit is 0.

All relational expressions result in bit strings of length 1, and they may therefore be used as operands in an OR operation.

The result of the OR operation is a bit-string value. For example:

```
DECLARE (BITA, BITB, BITC) BIT (4) ;
BITA = '0011'B;
BITB = '1111'B;
BITC = BITA | BITB ;
```

The resulting value of BITC is '1111'B.

The OR operator can test whether one of the expressions in an IF statement is true, for example:

```
IF (LINENO(PRINT_FILE) < 60) |
   (MORE_DATA = YES) THEN ...
```

See also "Logical Operator."

OUTPUT Attribute

The OUTPUT file description attribute indicates that data is to be written to, and not read from, the associated external device or file.

Specify the OUTPUT attribute on a DECLARE statement for a file constant or on an OPEN statement to access the file for writing. You may specify the OUTPUT attribute with either the STREAM or RECORD attribute. For a stream file, OUTPUT indicates that the file will be accessed using PUT statements. For a record file, OUTPUT indicates that the file will be accessed using only WRITE statements.

For example:

```
DECLARE OUTFILE RECORD OUTPUT;

OPEN FILE(OUTFILE);
WRITE FILE(OUTFILE) FROM(RECORD_BUFFER);
```

These statements declare, open, and write a record to the output file OUTFILE.

For a description of the attributes that may be applied to files and the effects of combinations of these attributes, **see** "File Description Attributes and Options."

The OUTPUT attribute may be supplied by default for a file, depending on the context of its opening. **See** "Opening a File."

■ Restrictions

The OUTPUT attribute conflicts with the INPUT and UPDATE attributes and with any data type attributes other than FILE.

OVERFLOW Condition Name

The OVERFLOW condition name can be specified in an ON, REVERT, or SIGNAL statement to designate an ON condition or ON-unit for floating-point overflow conditions.

The exponent of a floating-point value is adjusted, if possible, to represent the value with the specified precision. The maximum precision allowed for a binary floating-point value is 113; the maximum precision of a decimal floating-point value is 34. PL/I signals the OVERFLOW condition when the result of an arithmetic operation on a floating-point value exceeds the maximum allowed exponent size of the VAX-11 hardware.

The value resulting from an operation that causes this condition is undefined.

■ ON-Unit Completion

Control returns to the point of the interruption.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement."

P Format Item

The picture format item—P—describes a field of characters in the input or output stream. The field can be an input field acquired with GET EDIT or an output field transmitted by PUT EDIT. With GET EDIT, the P format item acquires a pictured value from the input stream. With PUT EDIT, the P format item edits an output source to a specified picture format.

The form of the P format item is:

P 'picture'

'picture'

A picture of the same syntax as for the PICTURE data attribute. The syntax is summarized in "PICTURE Attribute." The field width is the total number of characters, exclusive of V, in the picture. For full details, **see** "Picture."

The interpretation of the P format item, for input and output, is given below. For a general discussion of format items, **see** "Format Items and Their Uses."

■ Input with GET EDIT

Used with the GET EDIT statement, the P format item acquires a pictured value (a field of characters) from the stream file, extracts its fixed-point decimal value, and assigns the value to an input target of any computational type. The picture describes a field of w characters, where w is the total number of picture characters in the picture, exclusive of the V character.

A string of w characters is acquired from the input stream and validated against the picture specified in the format item. The string is valid if it corresponds to an internal representation that would be created by the specified picture if the picture were used to declare a variable of type PICTURE. If the string is valid, its fixed-point decimal value is extracted and assigned to the input target. Any necessary conversion to the type of the input target is done automatically, following the usual rules (**see** "Conversion of Data"). If the string is not valid, the ERROR condition is signaled. **See** "Examples" below.

When no decimal point appears in the input stream item, the scale factor of the item is assumed to be the number of digit positions specified to the right of the V character in the picture. If no V character appears, the scale factor is zero.

■ Output with PUT EDIT

Used with the PUT EDIT statement, the P format item outputs a source of any computational type in the specified format. If necessary, the output source is first converted to a fixed-point decimal value, following the usual rules (**see** "Conversion of Data"). The fixed-point decimal value is then edited by the picture specified in the format item. The P format item therefore describes an output field of w characters, where w is the total number of characters in the picture, exclusive of the V character. If the output source is a pictured value, then its extracted fixed-point decimal value must be editable by the picture specified in the P format item. Otherwise, the ERROR condition is signaled.

■ Examples

The tables below show the relationship between the internal and external representations of numbers that are read or written with the P format item.

Input Examples

The "input stream" shown in the table is a field of characters beginning at the current position in the stream and continuing to the right. The target type is the type of the variable to which the input value is assigned.

Format Item	Input Stream	Target Type	Target Value
P '$$$,$$$,$$9V.99DB'	$10,987,654.00DB...	DECIMAL(10,2)	-10987654.00
P '$$$,$$$,$$9V.99DB'	△△△△△△△△$10.99△△...	DECIMAL(10,2)	10.99
P 'SSSSV.SSSSS'	△△-1.12345...	DECIMAL(8,5)	-1.12345
P 'SSSSV.SSSSS'	+100.12345...	DECIMAL(8,5)	100.12345
P 'SSSSV.SSSSS'	△100.12345...	DECIMAL(8,5)	[ERROR]
P 'SSSSV.SSSSS'	+1001.2345...	DECIMAL(8,5)	[ERROR]

The last two cases signal the ERROR condition. In the first case, the input field has a space instead of a plus symbol or minus symbol in the first position. In the second case, the input field has four digits to the left of the period, and the P format item specifies a maximum of three. The P format item in both cases uses "drifting strings" of S characters, and, if used to declare a picture variable, the specification could create several different character representations. However, the specification could not have created the last two input fields shown, and they are therefore invalid values, as described under "Input with GET EDIT" above.

Note also that in the second line in the table, the characters "$10.99" must be surrounded with the number of spaces shown. The drifting dollar signs and the comma insertion characters always specify either digits, the characters themselves, or spaces. Similarly, the characters "DB" in the picture specification specify either these characters or the same number of spaces. If the pictured input value did not contain these spaces, it would be invalid.

Output Examples

The output source value shown in the table is either a constant or the value of a variable that is written out with the associated format item.

Output Source Value	Format Item	Output Value
-12234	P '$$$$$$DB'	$12234DB
-12234	P 'SSSSSSV.SS'	-12234.00
-12.234	P 'T9V.999'	J2.234
-1.23456E3	P '-9999V.99'	-1234.56
-1.23456E3	P '+ZZZ9V.99'	△1234.56

PAGE Format Item

The PAGE format item is used with print files to begin a new page.

The form of the PAGE format item is:

PAGE

Subsequent output begins on line 1 of the next page, and the current page number for the print file is incremented by 1 (**see also** "PAGENO Built-In Function" and "Print File").

PAGE Option

The PAGE option is used with the PUT statement to advance a print file to the top of the next page before beginning output. The output file must be a print file, that is, declared with the PRINT attribute.

For further information, **see** "PUT Statement," "PRINT Attribute," and "Print File."

PAGENO Built-In Function

The PAGENO built-in function returns a FIXED BINARY(15) integer that is the current page number in the referenced print file. The print file must be open. The format of the function is:

PAGENO(reference)

PAGENO Pseudovariable

The PAGENO pseudovariable refers to the page number of the referenced print file. Assignment to the pseudovariable modifies the current page number. **See also** "Pseudovariable," for general rules. The format (in an assignment statement) is:

PAGENO(reference) = expression ;

reference

A reference to a file for which the page number is to be set. The file must be open and must be a print file.

PAGENO(reference) is a FIXED BINARY(15) variable; however, values assigned to it must not be negative.

PAGESIZE Option

The PAGESIZE option is used in the OPEN statement to specify the maximum number of lines that can be written to a print file without signaling the ENDPAGE condition. The format of the PAGESIZE option is:

PAGESIZE(expression)

expression

A fixed-point binary expression in the range 1–32767, giving the number of lines per page.

The value specified in the PAGESIZE option is used as the output page length for all subsequent output operations on the print file, overriding the system default page size. The default page size is as follows:

- If the logical name SYS$LP_LINES is defined, the default page size is the numeric value of SYS$LP_LINES - 6.
- If SYS$LP_LINES is not defined, or if its value is less than 30 or greater than 99, or if its value is not numeric, the default page size is 60.

During output operations, the ENDPAGE condition is signaled the first time that the specified page size is exceeded.

■ Restrictions

The PAGESIZE option is valid only for print files.

Parameter Attribute

A variable occurring in the parameter list of a PROCEDURE or ENTRY statement is considered to have the parameter attribute (which has no keyword). For more information, **see** "Parameters and Arguments."

Parameter Descriptor

See "ENTRY Attribute."

Parameters and Arguments

A PL/I procedure can invoke other procedures and can transmit values to and receive them from the invoked procedure. Values are transmitted to an invoked procedure by means of arguments written in the procedure invocation. Values are returned to the invoking procedure by means of parameters and also, in the case of functions, by specifying a value in the function's RETURN statement.

Arguments may be specified for a subroutine (invoked by a CALL statement) or for a function (invoked by a function reference). Subroutines and functions return values by different means:

- A subroutine may return values only via a list of parameters. A subroutine must not specify a return value in its RETURN statement, and the declaration of an external entry point must not include the RETURNS attribute if the entry point is to be invoked as a subroutine. Instead, you can return values by assigning them, within the invoked subroutine, to the variables listed as parameters. (**See also** "Argument Passing" below.)
- A function may return values via its parameter list and, in addition, must return a single value that becomes the value of the function reference in the invoking procedure; this value is specified in the function's RETURN statement. The attributes of this returned value are specified within the invoking procedure, in the function's PROCEDURE or ENTRY statement, or in the declaration of the external entry constant or entry variable used to invoke the function. (**See also** "RETURN Statement.")

Figure P–1 illustrates the relationship between arguments (specified on a CALL statement or function reference) and parameters (specified on a PROCEDURE statement).

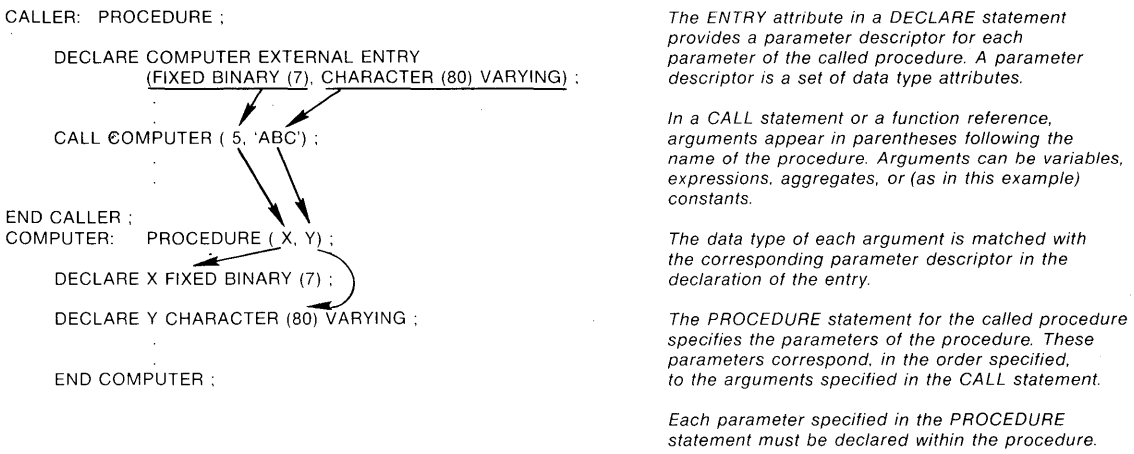


Figure P–1: Parameters and Arguments

■ Parameter List

A parameter is a variable that occurs in the parameter list of a PROCEDURE or ENTRY statement. When the entry point is invoked, each parameter in the parameter list is associated with an argument variable. Within the procedure invocation, any reference to the parameter is equivalent to a reference to the associated argument variable.

If the invoked entry point is external to the invoking procedure, the attributes of the parameters must be described in parameter descriptors, which are part of the declaration of the external entry point.

Procedures can have more than one entry point (**see** "Procedure"). Each entry point must have a parameter list if that entry point is to be invoked with an argument list. Multiple entry points in a procedure do not need to have identical parameters, but a reference to a parameter is valid only if the procedure was invoked via an entry point that specified that parameter.

■ Argument List

An argument is an expression or variable reference denoting a value to be passed to the invoked procedure. A procedure must be invoked with the same number of arguments as it has parameters. The maximum number of arguments that can be passed to a procedure is 253. The argument variable associated with a parameter, or "actual argument," may be a variable written in the argument list or a dummy argument created by the compiler. A dummy argument is created when the specified argument is a constant or expression and exists only for the duration of the procedure invocation. Therefore, references in the invoked procedure to the parameter associated with a dummy argument do not modify any storage in the invoking procedure. (For additional details, **see** "Argument Passing.")

An argument list consists of zero or more arguments specified in the invocation of a procedure, built-in function, or built-in subroutine. In the case of built-in functions, arguments are expressions that supply values to the built-in function, and the argument types must be those required by the specific function. In the case of user-defined procedures, arguments correspond to parameters defined on the PROCEDURE or ENTRY statement of the invoked procedure.

Arguments in an argument list must be separated by commas and enclosed in parentheses. For example:

```
CALL XYZ(STRING, 5, INDEX(ABC, STRING));
```

The CALL statement in this example invokes the procedure XYZ with an argument list consisting of three arguments:

- A variable named STRING.
- An integer constant, 5.
- A function reference. The INDEX built-in function is invoked with the variable arguments ABC and STRING. The value returned by INDEX is passed as the third argument to the procedure XYZ.

An empty argument list is required in the invocation of a user-defined function with no parameters. An empty argument list may be used in the invocation of a subroutine or built-in function that has no parameters. Examples:

```
X = F(); /* user-defined procedure-argument list required */
S = DATE(); /* built-in function-argument list optional */
CALL P(); /* subroutine-argument list optional */
```

■ Rules for Specifying Parameters

The general rules listed below for specifying parameters are followed by specific rules that pertain only to certain data types.

- A parameter must be declared explictly in a DECLARE statement (to give it a data type) within the invoked procedure. This declaration must not be as part of a structure.
- A parameter must not be declared with any of the following attributes:

AUTOMATIC	GLOBALREF	VALUE
DEFINED	INITIAL	
EXTERNAL	READONLY	
GLOBALDEF	STATIC	

- A maximum of 253 parameters can be specified for an entry point.
- The parameters of an external entry must be explicitly specified by parameter descriptors in the declaration of the entry constant. The parameters of a procedure that is invoked via an ENTRY variable must be specified by parameter descriptors in the ENTRY attribute of the variable's declaration. The parameters of an internal entry must not be declared. For details on entries and parameter descriptors, **see** "Entry."
- Each parameter must have a corresponding argument at the time of the procedure's invocation. PL/I matches the data type of the parameter with the data type of the corresponding argument and creates a dummy argument if they do not match. (**See** "Argument Passing.")

Array Parameters

If the name of an array variable is passed as an argument, the corresponding parameter descriptor or parameter declaration must specify the same number of dimensions as the argument variable. You can specify the bounds of a dimension using asterisks (*) or optionally signed integer constants. If the bounds are specified with integer constants, they must match exactly the bounds of the corresponding argument. An asterisk indicates that the bounds of a dimension are not known. (If one dimension contains an asterisk, all the dimensions must contain asterisks.) For example:

```
DECLARE SUMUP ENTRY ((*) FIXED BINARY);
```

This declaration indicates that SUMUP's argument is a one-dimensional array of fixed-point binary integers that can have any number of elements. Any one-dimensional array of fixed-point binary integers may be passed to this procedure.

All the data type attributes of the array argument and parameter must match.

Structure Parameters

If the name of a structure variable is passed as an argument, the corresponding parameter descriptor or declaration must be identical in terms of structure levels, members' sizes, and members' data types. Array bounds and string lengths can be specified with asterisks or with optionally signed integer constants. The level numbers do not have to be identical. The following example shows the parameter descriptor for a structure variable:

```
DECLARE SEND_REC ENTRY (1,
                        2 FIXED BINARY(31),
                        2 CHARACTER(40) VARYING,
                        2 PICTURE '999V99');
```

The written argument in the invocation of the external procedure SEND_REC must have the same structure and its members must have the same data types.

Structures are always passed by reference. They cannot be passed by dummy argument.

Character-String Parameters

If a character-string variable is passed as an argument, the corresponding parameter descriptor or parameter declaration can specify the length using an asterisk (*) or an optionally signed nonnegative integer constant. For example:

```
COPYSTRING: PROCEDURE (INSTRING,COUNT);
DECLARE INSTRING CHARACTER(*);
```

The asterisk in the declaration of this parameter indicates that the string may have any length.

Entry, File, and Label Constant Parameters

Entry, file, and label constants may be passed as arguments. The actual argument is a variable.

■ Argument Passing

The following paragraphs describe the precise rules that PL/I uses to determine how to pass an argument.

There are different rules for passing arguments to procedures written in PL/I and passing arguments to procedures written in other languages. This manual describes only the conventions for passing arguments to procedures that are written in PL/I. For complete rules and details on passing arguments to procedures written in other languages, **see** the *VAX-11 PL/I User's Guide*.

Number of Arguments

The number of arguments in the argument list must equal the number of parameters of the invoked entry point. The compiler checks that the count matches as follows:

- For an internal procedure, the compiler checks the number of arguments specified in the argument list against the number of parameters specified on the PROCEDURE or ENTRY statement for the internal procedure.
- For an external procedure, the compiler checks that the number of parameter descriptors in the parameter descriptor list of the ENTRY declaration matches the number of arguments specified in the procedure invocation. This argument checking can be overridden for an external procedure declared with the OPTIONS(VARIABLE) option; however, this option applies only to procedures that are not written in PL/I. **See** the *VAX-11 PL/I User's Guide* for information on how to use this option.

Actual Arguments

When a PL/I procedure is invoked, each of its parameters is associated with a variable determined by the corresponding written argument of the procedure call. This is the actual argument for this procedure invocation. This actual argument may be either:

- A reference to the written argument
- A dummy argument

The data type of the actual argument is the same as the data type of the corresponding parameter. When a written argument is a variable reference, PL/I matches the variable against the corresponding parameter's data type according to the rules given under the heading "Argument Matching," below. If it matches, the actual argument is the variable denoted by the written argument. That is, the parameter denotes the same storage as the written variable reference. If it does not match, the compiler creates a dummy argument and assigns the value of the written argument to the dummy argument.

Dummy Arguments

A dummy argument is a unique variable allocated by the compiler, and it exists only for the duration of the procedure invocation.

When the written argument is a constant or an expression, the actual argument is always a dummy argument. The value of the written argument is assigned to this dummy argument before the call. The data type of the written argument must be valid for assignment to the data type of the dummy argument.

Aggregate Arguments

An array, structure, or area argument must be a variable reference that matches the corresponding parameter. It may not be a reference to an unconnected array. A dummy argument is never created for an array or structure.

Argument Matching

A written argument that is a variable reference is passed by reference only if the argument and the corresponding parameter have identical data types. (For the definition of identical data types, **see** "Data and Data Types.")

- For an internal procedure, the attributes of the argument must match the attributes specified in the declaration of the parameter.
- For an external procedure or a procedure invoked via an ENTRY variable, the attributes specified in the ENTRY attribute parameter descriptor must match the attributes of the arguments.

When the compiler detects that a scalar variable argument does not match the data type of the corresponding parameter, it issues a warning message, creates a dummy argument, and associates the address of the dummy argument with the corresponding parameter. You can suppress the warning message and force the creation of a dummy argument if you enclose the argument in parentheses. For example, if a parameter requires a CHARACTER VARYING string and an argument is a CHARACTER nonvarying variable, you would enclose the variable in parentheses.

For string lengths and array bounds, an asterisk (*) in the parameter matches any expression. An integer constant matches only an integer constant with the same value.

Conversion of Arguments

When the data type of a written argument is suitable for conversion to the data type of the corresponding parameter descriptor, PL/I performs the conversion of the argument to a dummy argument using the rules described under "Conversion of Data."

Picture

Pictured data is used when you want to manipulate a quantity arithmetically and print or display its value using a special output format. This entry describes:

- Pictured variables — variables declared with the PICTURE data attribute
- Editing by picture — the process by which a value is assigned to a pictured variable or written out with the P format item
- Extracting values from pictured data — the process by which a pictured value is assigned to other variables or acquired with the P format item
- Picture characters — the special characters that make up a picture specification in the PICTURE attribute and in the P format item. "Picture Characters" gives a detailed description of each picture character. For a brief description of the characters and for the required picture syntax, **see** "PICTURE Attribute."

For a description of the P format item, **see** "P Format Item."

■ Pictured Variables

A pictured variable has the attributes of a fixed-point decimal variable, but values assigned to it are stored internally as character strings. Such a character string contains digits representing the variable's numeric value as well as such special symbols as the dollar sign. When the value of a pictured variable is written out by, for example, the PUT LIST statement, the internally stored character string is placed in the output stream. The value that appears on a line printer or terminal thus contains a fixed-point decimal number that has been "edited" with the requested special symbols.

The formatting possible with pictured data is useful in many applications, but pictured data is much less efficient than fixed-point decimal data for strictly computational use.

The numeric attributes of a pictured variable and its output format are both described in a picture specification, or simply, a picture. A simple picture looks like this in a DECLARE statement:

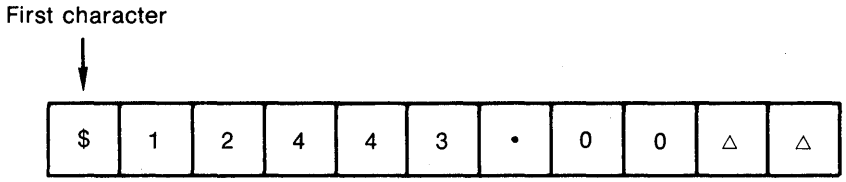
```
DECLARE CREDIT PICTURE '$99999V.99DB';
```

The variable CREDIT is declared as a pictured variable; its picture comprises the characters within the apostrophes.

The assignment

```
CREDIT = 12443.00;
```

stores the following data internally, as a character string:

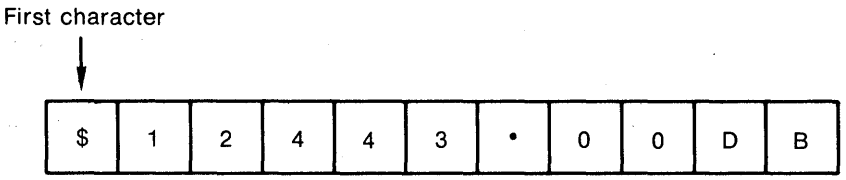


where △ represents a space.

The assignment

```
CREDIT = -12443.00;
```

stores the following data internally:



In situations that call for a character representation of a pictured data item (such as output with PUT LIST), this internal representation is used, including the nonnumeric characters. On output, the values assigned to CREDIT would look like this:

```
$12443.00 /* a positive value (credit) */

$12443.00DB /* a negative value (debit) */
```

■ Editing by Picture

Any computational value or expression may be assigned to a pictured variable, as long as it meets these two qualifications:

- The value either is a fixed-point decimal value or can be converted to a fixed-point decimal value (**see also** "Conversion of Data").
- The fixed-point decimal value can be represented with the precision and scale factor of the picture specified for the target pictured variable.

When a value is assigned to a pictured variable, the value is edited to construct a character string that meets the picture specification. Editing also occurs when a value is output with the PUT EDIT statement and the P format item. Editing was performed in the previous examples in which fixed-point decimal values were assigned to the pictured variable CREDIT.

Because a picture specifies a fixed-point decimal value, the FIXEDOVERFLOW condition is signaled in the same circumstances as for assignment of an expression to a FIXED DECIMAL variable.

In addition, two programming errors are common in assignments to pictured variables:

```
CREDIT = '$12443.00';
```

This example signals the ERROR condition, because the character string contains a dollar sign and is therefore not convertible to fixed-point decimal. The value assigned to CREDIT should be either '12443.00' or simply 12443.00, both of which result in the same value assigned to CREDIT.

If a negative value is assigned to a pictured variable, the picture must include one of the sign picture characters (such as DB). If, for example, the picture of CREDIT did not contain the DB characters, then the assignment

```
CREDIT = -12443.00;
```

would signal the FIXEDOVERFLOW condition, because the sign would be lost.

In some circumstances (for example, with the READ statement), it is possible to assign a value to a pictured variable that is not valid with respect to the variable's picture specification. In such cases, the VALID built-in function can be used to validate the contents of the variable. **See** "VALID Built-In Function."

■ Extracting Values from Pictured Data

When a pictured value is used in an arithmetic context (for example, when it is assigned to an arithmetic variable), the picture is used to extract the fixed-point decimal number from the character string that is the internal representation of the pictured value. Extraction also occurs when a pictured value is input with the GET EDIT statement and the P format item.

In the picture for CREDIT:

```
DECLARE CREDIT PICTURE '$99999V.99DB';
```

the 9 character specifies the position of a decimal digit; since the picture contains seven of these, the fixed-point decimal precision of CREDIT is 7.

The V character separates the integral and fractional digits; since there are two 9 characters to the right of the V, the scale factor of CREDIT is 2. The V character is unique among picture characters in that it specifies only a numeric property; it does not cause a decimal point (or any other character) to appear in the internal representation of CREDIT. Therefore, a period picture character (.) can be included after the V to ensure that the output value has a decimal point in the correct place.

The period and dollar sign are always inserted in the internal representation and the output value regardless of CREDIT's numeric value.

The picture character DB appears only when the value of CREDIT is less than zero; otherwise, two spaces appear in the indicated positions. The DB character also indicates that a value of CREDIT is numerically negative, so that if CREDIT is later assigned to an arithmetic variable, the variable will be given a negative value.

■ Picture Characters

The picture is a string made up of special characters. (For a full list of PL/I picture characters, **see** Table P–2 in "PICTURE Attribute.") An individual picture character, and its position in the picture, indicate the interpretation of an associated position in the pictured value. All picture characters are shown here in uppercase form, although the lowercase equivalents can be used identically.

The picture characters fall into three categories and are described below in this order:

- Characters that affect only the numeric interpretation of the value. The decimal place character (V) is the only one in this category.
- Characters that affect both the numeric interpretation and character representation of the value. These characters are:
  - The digit characters (9, Z, *, Y)
  - The encoded-sign characters (T, I, R)
  - The drifting characters ($, +, –, S)

- Characters that affect only the character representation of the value. These characters are:
  - The insertion characters (comma, period, slash, space)
  - The credit (CR) and debit (DB) characters

Any picture character that can appear more than once in a picture may be preceded by an iteration factor. The iteration factor must be a positive integer constant enclosed in parentheses. For example, the picture

```
'(4)9'
```

is the same as

```
'9999'
```

Decimal Place Character (V)

The V character shows the position of the "assumed" decimal point, or, in other words, the scale factor for the fixed-point decimal value. The V character has no effect on the internal representation of the pictured value and does not, therefore, cause a decimal point to appear in the internal representation. (The period insertion character is used for this purpose — **see** "Insertion characters" below.) The following additional rules apply to the V character:

- Only one V character may appear in a picture.
- If a picture does not contain the V character, a V character is assumed to be at the right end of the picture. That is, the pictured value has a scale factor of zero.
- When a fixed-point value is assigned to a pictured variable, the integral portion of the assigned value is described by the picture characters to the left of the V; the fractional portion of the assigned value is described by the picture characters to the right of the V. Note therefore:
  - If the assigned value has fewer integral digits than are indicated by the picture characters to the left, then the integral value of the pictured variable is extended on the left with zeros. If the assigned value has too many integral digits, the value of the pictured variable is undefined and the FIXEDOVERFLOW condition is signaled.
  - If the assigned value has fewer fractional digits than are indicated in the picture, then the fractional value of the pictured variable is extended on the right with zeros. If the assigned value has too many fractional digits, then the excess fractional digits are truncated on the right; no condition is signaled. Thus, if the V character is the last character in the picture or is omitted, assigned fixed-point values are truncated to integers.

Digit Characters (9, Z, *, Y)

All of these characters mark the positions occupied by decimal digits. The number of these characters present in a picture specifies the number of digits, or precision, of the fixed-point decimal value of the pictured variable. These characters also describe the internal, character representation of the digits; they allow zeros in a number to be represented either by the character 0 or by an alternate character. Specifically:

- The position occupied by 9 always contains a decimal digit, whether or not the digit is significant in the numeric interpretation of the pictured value.
- The position occupied by Z contains a decimal digit only if the digit is significant in the integral portion of the numeric interpretation; if the digit is an insignificant, or "leading," zero, it is replaced by a space in the internal representation.
  - The Z character must not appear in the same picture with the character *. It must not appear to the right of the characters 9, T, I, or R nor to the right of a drifting string (**see** "Drifting characters" below).
  - If the Z character appears to the right of the V character, then all digits to the right of the V must be indicated by Z characters. Fractional zeros are then suppressed only if all fractional digits are zero and all of the integral digits are suppressed; in that case, the internal representation contains only spaces in the digit positions.
- The position occupied by the * character functions identically with the Z character except that leading zeros are replaced in the internal representation by asterisks instead of spaces. The * character must not appear in the same picture as Z nor to the right of the characters 9, T, I, or R nor to the right of a drifting string (**see** "Drifting characters" below).
- The position occupied by the Y character contains a decimal digit only if the digit is not zero. All zeros in the indicated positions, whether significant or not, are replaced by spaces in the internal representation.

Encoded-Sign Characters (T, I, R)

The characters T, I, and R are digit characters that may be used wherever 9 is valid. One of these characters represents a digit that has the sign of the pictured value encoded in the same position.

Only one of these characters can be used in a picture.

An encoded-sign character cannot be used in a picture that contains an S, +, -, CR, or DB (described below).

The meanings of the characters are as follows:

- The T character indicates that the position contains an encoded minus sign if the numeric value is less than zero and an encoded plus sign if the numeric value is greater than or equal to zero. These encoded-sign digits are represented internally and in output by the ASCII characters shown in Table P–1.
- The I character indicates an encoded plus sign if the numeric value is greater than or equal to zero. Otherwise, the position contains an ordinary digit.
- The R character indicates an encoded minus sign if the numeric value is less than zero. Otherwise, the position contains an ordinary digit.

Table P–1 shows the ASCII characters used to indicate digits with encoded signs. In the table, the notation +digit means the digit with an encoded plus sign, and -digit means the digit with an encoded minus sign. The characters in Table P–1 are used in the internal representation of a pictured value and must be used for input of an encoded-sign digit from a stream file.

Table P–1: ASCII Representation of Encoded-Sign Digits

Digit	ASCII Character	Digit	ASCII Character
+0	{	-0	}
+1	A	-1	J
+2	B	-2	K
+3	C	-3	L
+4	D	-4	M
+5	E	-5	N
+6	F	-6	O
+7	G	-7	P
+8	H	-8	Q
+9	I	-9	R

Drifting Characters ($, +, -, S)

The drifting characters can be used to indicate digits, and they also indicate a symbol to be inserted in the internal representation. The inserted symbol then appears when, for example, a pictured value is written out by PUT LIST.

- The dollar sign ($) causes a dollar sign to be inserted.
- The plus sign (+) causes a plus sign to be inserted if the numeric value is greater than or equal to zero.

- The minus sign (-) causes a minus sign to be inserted if the numeric value is less than zero.
- The S character causes a plus sign to be inserted if the numeric value is greater than or equal to zero and a minus sign if the value is less than zero.

If one of these characters is used alone in the picture, it marks the position at which a special symbol or space is always inserted, and it has no effect on the value's numeric interpretation. In this case, the character must appear either before or after all characters that specify digit positions.

However, if a series of *n* of these characters appears, then the rightmost *n*-1 of the characters in the series also specify digit positions. If the digit is a leading zero, the leading zero is suppressed, and the leftmost character "drifts" to the right; in the internal representation, the character appears either in the position of the last drifting character in the series or immediately to the left of the first significant digit, whichever comes first. Used this way, the *n*-1 drifting characters also define part of the numeric precision of the pictured variable, since they describe at least some of the positions occupied by decimal digits. The following additional rules apply to drifting characters:

- A drifting string is a series of more than one of the same drifting character. If a drifting string appears in the picture, it must be the only drifting string; the other drifting characters can be used only singly and therefore designate insertion characters and not digits.
- The characters Z and * cannot appear to the right of a drifting string.
- A digit position cannot be specified (for instance, with a 9) to the left of a drifting string.
- A drifting string can contain the V character and one of the insertion characters (defined below). The following additional rules apply to insertion characters that are embedded in a drifting string:
  - If the drifting string contains an insertion character, the insertion character is inserted in the internal representation only if a significant digit appears to its left. In the position of the insertion character, a space appears if the the leftmost significant digit is more than one position to the right; the drifting symbol appears if the next position to the right contains the leftmost significant digit.
  - If the drifting string contains a V character, all digit positions to the right of the V (the fractional digits) must also be part of the drifting string. In this case, insignificant fractional digits are suppressed if and only if all integral and fractional digits are zeros; if so, they are replaced by spaces in the internal representation. If any digit is not zero, all fractional digits appear as actual digits.
  - Any insertion characters that are immediately to the right of a drifting string are considered part of the drifting string.

Insertion Characters

The insertion characters indicate that characters are inserted in the internal representation of the pictured value. They are inserted between digits. The insertion characters are the comma (,), period (.), slash (/), and the space (B). The B character indicates that a space is always inserted at the indicated position.

The drifting characters also function as insertion characters when used singly (that is, not as part of a drifting string).

The following rules describe the actual characters inserted by the comma, period, and slash insertion characters.

- In general, the insertion character itself is inserted in the internal representation of the pictured value. In particular, this is true if the insertion character is the first character in the picture, or if all the picture characters to its left are characters that do not specify decimal digits.
- If zero suppression occurs, the insertion character is inserted only in these cases:
  - A significant digit appears immediately to the left of the insertion character.
  - The V character appears immediately to the left, and the fractional part of the numeric value contains significant digits.
- If the position preceding the insertion character is occupied by an asterisk or drifting string and the preceding position is taken by a leading zero, then the preceding character also indicates the character to be inserted in the position of the insertion character. If, however, the preceding position is taken by a leading zero and does not have an asterisk or drifting string, then the insertion character's position is a space in the internal representation of the pictured value.
- To guarantee that the decimal point is in the same position in both the numeric and character interpretations, the V and period characters must be immediately adjacent. Note, however, that if the period precedes the V, then it is suppressed if there are no significant integral digits, even though all the fractional digits are significant. This property can make fractions appear to be integers when the internal (character) value is displayed. Consequently, the period should immediately follow the V character; the period will then be in the correct location and will appear whenever any fractional digit is significant.
- Other insertion characters, such as the comma, can be used to separate the integral and fractional portions of a number. However, the comma should not be used with GET LIST input, because a comma is used in that context to separate different data items in the input stream.

Credit (CR) and Debit (DB) Characters

These picture characters are always specified in the pairs CR and DB. If either of these character pairs is included, the character pair appears in the internal representation if the numeric value is less than zero. In each case, the associated positions in the internal representation contain two spaces if the numeric value is greater than or equal to zero.

The characters are always inserted with the same case used in the picture; if the lowercase form cr is used in the picture, lowercase letters are inserted in the pictured value; if the combination Cr is used, then Cr is inserted.

The credit and debit characters cannot be used in the same picture, nor can they be used in the same picture with any other character that specifies the sign of the value (that is, S, +, –, and the encoded-sign characters). In addition, they must appear to the right of all picture characters that specify digits.

PICTURE Attribute

The PICTURE attribute is used to declare a pictured variable. Pictured variables have fixed-point decimal attributes, but values of the variable are stored internally as character strings. The character string contains decimal digits representing the numeric value of the variable, plus special editing symbols described in the picture.

The PICTURE attribute conflicts with the FIXED, FLOAT, DECIMAL, and BINARY attributes.

The format of the PICTURE attribute is:

{ PICTURE / PIC } 'picture'

picture

The picture is a string of picture characters that define the representation of the variable.

These characters are described in Table P–2. A brief description of picture syntax, and examples, follow Table P–2. For precise definitions of picture characters, **see** "Picture."

Table P–2 shows the uppercase form of picture characters; lowercase letters can also be used.

Table P-2: Picture Characters

Character	Meaning
9	Decimal digit, including leading zeros
Z	Decimal digit with leading-zero suppression
*	Decimal digit with asterisk for leading zero
Y	Decimal digit with space for any zero
V	Position of assumed decimal point
(n)	Iteration factor for subsequent character
T	Position of digit and encoded plus sign or minus sign
I	Position of digit and encoded plus sign if number ≥ 0
R	Position of digit and encoded minus sign if number < 0
.	Position at which decimal point is inserted
,	Position at which comma is inserted
/	Position at which slash is inserted
B	Position at which space is inserted
$	Position[s] of [drifting] dollar sign
+	Position[s] of [drifting] plus sign if number ≥ 0
–	Position[s] of [drifting] minus sign if number < 0
S	Position[s] of [drifting] plus sign or minus sign
CR	Positions at which 'CR' is inserted if number < 0
DB	Positions at which 'DB' is inserted if number < 0

■ Picture Syntax

After all its iterations are expanded and all its insertion characters are removed, a picture must satisfy the following syntax rules (the notation character... indicates a series of the same character, with no embedded characters).

picture:

'[left-part]center-part[right-part]'

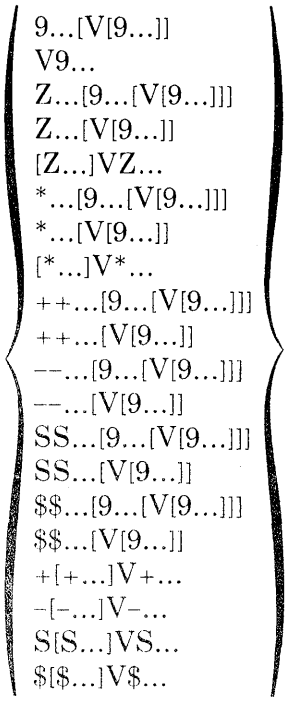
left-part:

{ $ / + / – / S }

right-part:

{ $ / + / – / S / CR / DB }

center-part:



NOTE

The character Y, T, I, or R may appear wherever 9 is valid, with the following restrictions. Only one character T, I, or R may appear in a picture. A picture may not contain T, I, or R if it also contains S, +, -, CR, or DB.

■ Examples

Valid Pictures

```
'S99V.99'
```

The picture specifies a signed fixed-point number with p=4, q=2. The sign of the number is always included in its representation, in the first position. A period is inserted at the position of the assumed decimal point.

```
'****99'
```

The picture specifies a six-digit integer, with the first four leading zeros replaced by asterisks.

```
'****V.**'
```

The picture specifies a fixed-point number with p=6,q=2. The first four leading zeros are replaced by asterisks in the integral portion. Both fractional digits always appear unless all six digits are 0. A period is inserted at the position of the assumed decimal point.

```
'ZZ99V.99'
```

The picture specifies a fixed-point number with p=6,q=2. The first two digits in the integral portion are replaced with spaces if they are zeros. Two digits always appear on either side of the decimal point.

```
'(4)SV.99'
```

The picture specifies a fixed-point number with p=5,q=2. (The iteration factor 4 specifies a string of four S characters, one of which specifies a sign and three of which specify digits.) A plus (+) or minus (-) symbol is inserted to the immediate left of the first significant integral digit, or to the left of the decimal point if no integral digit is significant. Any insignificant integral digits are replaced with spaces or with the sign symbol.

```
'ZZZ,ZZZV.99'
```

The picture specifies a fixed-point number with p=8,q=2. If the integral portion has four or more significant digits, a comma is inserted between the third and fourth; otherwise, both the leading zeros and the comma are suppressed. The decimal point always appears followed by two fractional digits.

```
'ZZZ.ZZZV,99'
```

The picture specifies a fixed-point number with p=8,q=2. If the integral portion has four or more significant digits, a period is inserted between the third and fourth; otherwise, both the leading zeros and the period are suppressed. The decimal point (indicated by a comma) always appears followed by two fractional digits.

```
'ZZZ/ZZZ/ZZZ'
```

The picture specifies a fixed-point number with p=9,q=0. A slash is inserted between the three-digit groups unless the digit preceding the slash is a suppressed zero.

Invalid Pictures

```
'999ZZZZV.99'
```

The picture is invalid because a 9 occurs to the left of Z.

```
'$$$---99V.99'
```

The picture is invalid because it contains two drifting strings ('$$$' and '---').

```
'(4)-V.ZZZ'
```

The picture is invalid because fractional digits in this case must be pictured either with a drifting minus sign or with 9s.

Pointer

A pointer is a variable whose value represents the location in memory of another variable or data item.

All pointers must be declared with the POINTER attribute before they can be referenced in a BASED attribute or an ALLOCATE statement with the SET option. For example:

```
DECLARE X POINTER,
        BUFFER CHARACTER(80) BASED (X);
```

The variable X is given the POINTER attribute. Then, it is used as the target pointer in another declaration, which defines a buffer to be based on X. Pointers are used to qualify references to based variables, that is, variables for which storage is explicitly allocated at run time by the ALLOCATE statement. For example:

```
DECLARE LIST_POINTER POINTER;
DECLARE 1 LIST_STRUCTURE BASED,
          2 FORWARD_PTR POINTER,
          2 MEMBER_NAME CHAR(20) VAR;

ALLOCATE LIST_STRUCTURE SET (LIST_POINTER);
LIST_POINTER -> LIST_STRUCTURE.MEMBER_NAME = 'newname';
```

When these statements are executed, the ALLOCATE statement allocates storage for a variable LIST_STRUCTURE and sets the pointer LIST_POINTER to the address in memory of the allocated storage. This dynamically created variable is called an allocation of the variable LIST_STRUCTURE.

In the assignment statement, the locator qualifier (->) and the identifier LIST_POINTER distinguish this allocation of LIST_STRUCTURE from allocations created by other ALLOCATE statements, if any.

■ Pointer Variables in Expressions

Expressions containing pointer variables are restricted to the following relational operators:

Operator	Meaning
=	Equal
^=	Not equal

For example, to test whether a pointer is null, that is, to to determine whether it is currently pointing to valid storage, you can write the following statement:

```
IF LIST_POINTER = NULL() THEN
   DO;
```

The NULL built-in function, referenced in this example, always returns a null pointer value.

Pointer variables can be used in simple assignment statements that assign a pointer value to a pointer variable. For example:

```
LIST_POINTER_1 = LIST_POINTER_2;

LIST_END = NULL();
```

A pointer variable can also be used as the source or target in an assignment statement involving an offset variable or offset value. **See** "Offset."

■ Internal Representation of Pointer Data

A pointer occupies a longword (32 bits) of storage and represents a virtual memory address.

For more information, **see** "ALLOCATE Statement," "Based Variable," "FREE Statement," "List Processing," "Locator Qualifier," "Offset," and "Storage Classes."

POINTER Attribute

The POINTER attribute indicates that the associated variable will be used to identify locations of data. The format of the POINTER attribute is:

{ POINTER }
{ PTR }

■ Restrictions

The POINTER attribute conflicts with all other data type attributes.

POINTER Built-In Function

The POINTER built-in function returns a pointer to the location identified by the referenced offset and area. Its format is:

POINTER (offset,area)

offset

A reference to an offset variable whose current value either represents the offset of a based variable within the specified area or is null.

area

A reference to a variable that is declared with the AREA attribute and with which the specified offset value is associated.

■ Returned Value

The returned value is of type POINTER. If the offset value is null, the result is null.

■ Example

```
DECLARE MAP_SPACE AREA (2048),
        START OFFSET (MAP_SPACE),
        P POINTER;
        .
        .
P = POINTER (START,MAP_SPACE);
```

The POINTER built-in function converts the value of the offset variable START in the area MAP_SPACE to a pointer value.

POSITION Attribute

The POSITION attribute specifies the character or bit position in a defined variable's base at which the defined variable begins. Its format is:

POSITION (expression)

expression

An integer expression that specifies a position in the base. A value of one indicates the first character or bit.

■ Restrictions

The POSITION attribute may be specified only in connection with DEFINED and only when the defined variable satisfies the rules for string overlay defining (**see also** "Defined Variable").

Precedence

The precedence, or priority, of operators defines the order in which expressions are evaluated when they contain more than one operator.

PL/I defines the precedence of arithmetic operators with respect to each other and with respect to other types of operators. In general, the rules for precedence produce "expected" results without the need for parenthesized expressions. However, for details **see** "Expression" and "Operator."

Precision Attribute

The precision attribute applies to binary and decimal data; the precision of an item is the number of decimal or binary digits used to represent a value. The precision of an arithmetic variable can be specified in any of the following formats, depending on the numeric base of the data item:

BINARY [FIXED] [(precision)]
[BINARY] FLOAT [(precision)]
DECIMAL [FIXED] [(precision[,scale-factor])]
DECIMAL FLOAT [(precision)]

In each case, the precision is the number of bits or decimal digits used to represent values of the variable. Only fixed-point decimal data has a scale

factor. The scale factor specifies that all values of the fixed-point decimal variable are "scaled" by the factor 10^{-q}, where q is the specified scale factor; in other words, all values have q fractional digits. The scale factor must be less than or equal to the precision specified for the fixed-point decimal variable, and it must be greater than or equal to zero. Fixed-point binary data must always have a zero scale factor; only integers can be represented in that data type.

The precision of a floating-point data item is the number of decimal or binary digits in the mantissa of the floating-point representation.

■ Restrictions

The ranges of values you can specify for the precision for each arithmetic data type, and the defaults applied if you do not specify a precision, are summarized as follows:

Data Type Attributes	Precision	Scale Factor	Default Precision
BINARY FIXED	$1 \le p \le 31$	—	31
BINARY FLOAT	$1 \le p \le 113$	—	24
DECIMAL FIXED	$1 \le p \le 31$	$\le p$	10
DECIMAL FLOAT	$1 \le p \le 34$	—	7

If no scale factor is specified with DECIMAL FIXED, the default is zero.

■ Precision of Expressions

The precision of the result of an expression is determined by the precisions and data types of the variables and constants used in the expression, and by the rules governing the specific operation being performed by the expression.

For the rules governing the conversion of operands in an expression, **see** "Expression — Conversion of Operands." The conversion of operands in an expression produces converted operands of the same data type but with individual precisions. These individual precisions are then used to determine the precision of the result, which depends on the operation being performed. For example, **see** "Subtraction."

PRINT Attribute

The PRINT attribute is used to declare a print file. The file SYSPRINT, used as the default output by PUT statements, is also a print file.

Print files are stream output files with special formatting characteristics (**see** "Print File"). The PRINT attribute implies the OUTPUT and STREAM attributes.

■ Restrictions

The PRINT attribute conflicts with the INPUT, RECORD, UPDATE, KEYED, SEQUENTIAL, and DIRECT attributes.

Print File

A print file is a stream output file that is intended for output on a terminal, line printer, or other output device. Any stream output file can be declared a print file by use of the PRINT attribute. The default stream output file, SYSPRINT, is also a print file.

The following list describes the special features of print files as opposed to ordinary stream output files (**see also** "Stream Input/Output"):

- Character strings are not enclosed in apostrophes on list-directed output.
- List-directed output data items are separated by tabs instead of spaces. Tab stops occur at eight-column increments beginning with column 1. With the PUT EDIT statement and the TAB format item, you can begin output at a specified tab stop.
- A record is kept internally of the current line in a print file. The LINENO built-in function returns the current line number for a specified file. This function allows you to keep track of the number of lines being written to a file and to decide where page advances should occur.
- Print files are divided into both lines and pages. A record is kept internally of the number of lines per page. You can specify a page size when the print file is created (**see** "PAGESIZE Option").
- During output of data to a print file, the ENDPAGE condition is signaled when the output exceeds the page size.
- New pages are started by the PUT PAGE statement, the PAGE format item, and certain other format items. Each of these operations increments the current page number by one. The PAGENO built-in function returns the current page number from a print file. This function allows you to keep track of the number of pages being written to a file. You can set the current page number to a specific value by assigning the value to the PAGENO pseudovariable.
- If the print file is a terminal, the output is written to the terminal at the conclusion of each PUT statement.
- A print file is created with PRN-format carriage control. PRN format is efficient for both terminals and line printers because blank lines do not require individual records. (PRN format is discussed in the *VAX-11 Record Management Services Reference Manual.*)
- Print files usually cannot be read properly with GET LIST or GET EDIT.

Procedure

A procedure is the basic executable program unit in PL/I. It consists of a sequence of statements, headed by a PROCEDURE statement and terminated by an END statement, that define an executable set of program instructions. The two types of procedure that can be invoked by another procedure during its execution are:

- Subroutines, which must be invoked with a CALL statement. Subroutines return values to the invoking procedure only by means of their parameter

lists; they must not include an expression in their RETURN statements and must not include a RETURNS option on their PROCEDURE or ENTRY statements.

- Functions, which must be invoked by a function reference. A function reference can appear in place of a scalar value in any appropriate context in a PL/I statement. A function returns to the invoking procedure a single value that becomes the value of the function reference in the invoking procedure. Functions may also return values via their parameter lists. Functions must include a RETURNS option to describe the attributes of the returned value and must specify an expression in their RETURN statements.

Each type of procedure can be passed data or information from the invoking procedure by means of an argument list.

A procedure may have multiple entry points, and it is permissible for some entry points to be subroutine entry points and some to be function entry points. In this case, the procedure is treated as a subroutine or function in accordance with the entry point through which it is invoked. Remember, though, that when a procedure is invoked as a function, any RETURN statement executed in the procedure must specify a return value.

■ External and Internal Procedures

An internal procedure is one whose text is contained within another block. An external procedure is one whose text is not contained in any other block.

The source text of an external procedure can be separately compiled.

The primary coding differences between internal and external procedures are:

- Before an external procedure can be invoked (except via an entry variable), its name must be declared within the procedure that invokes it. The DECLARE statement for the external entry name must also provide a list of parameter descriptors that give the data type(s) of the procedure's parameters, if any, and the DECLARE statement must provide a RETURNS attribute if the procedure is a function.

  Internal procedures must not be explicitly declared. The procedure name is implicitly declared by its occurrence in the PROCEDURE or ENTRY statement of the internal procedure.

- External procedures can reference the same variable only if the variable is declared with the EXTERNAL attribute in all procedures that reference it.

  An internal procedure, on the other hand, can reference internal variables declared in any procedure in which it is contained.

- Any procedure can call an external procedure.

  An internal procedure can be called only by the procedure that contains it or by other procedures at the same level of nesting within the containing procedure. The only exception is invocation via an entry variable.

Figures P–2 and P–3 illustrate invoking internal and external procedures.

```
MAINP: PROCEDURE OPTIONS (MAIN);
    .
    .
    COMPUTE: PROCEDURE;
        .
        .
        ADD_NUMBERS: PROCEDURE;

            .
            .
        END;
        .
        .
    END;
    .
    .
    PRINT_REPORT: PROCEDURE;
            .
            .
    END;

    .
    .
END;
```

Figure P-2: Invoking Internal Procedures

In Figure P-2, the procedures COMPUTE and PRINT_REPORT are internal to the procedure MAINP, and the procedure ADD_NUMBERS is internal to the procedure COMPUTE. MAINP can invoke the procedures COMPUTE and PRINT_REPORT, but not ADD_NUMBERS. COMPUTE and PRINT_REPORT can invoke one another. ADD_NUMBERS can call COMPUTE and PRINT_REPORT. (**See also** "Scope of Names.")

```
WINDUP: PROCEDURE;
        .
        .
    DECLARE PITCH EXTERNAL ENTRY (CHARACTER(15) VARYING,
                                  FIXED BINARY(7) );
        .
        .
    CALL PITCH (PLAYER_NAME,NUMBER_OF_OUTS);
        .
        .
```

Figure P-3: Invoking an External Procedure

In Figure P-3, the procedure WINDUP declares the procedure PITCH with the EXTERNAL and ENTRY attributes. The text of the procedure PITCH is in another source program that is separately compiled.

For information on compiling and linking together separately compiled procedures, **see** the *VAX-11 PL/I User's Guide*.

■ Terminating Procedures

Subroutines and functions can be terminated in the following ways:

- A RETURN statement

  A RETURN statement provides a normal termination for a subroutine or function. For a function, a RETURN statement must specify a return value.

- A STOP statement

  A STOP statement ends the entire program execution. It does not pass a return value.

- An END statement

  If an END statement closes the procedure block of a subroutine before a RETURN or STOP statement is executed, the END statement has the same effect as RETURN. A function cannot be terminated without a RETURN statement.

- A nonlocal GOTO statement

  A GOTO statement that transfers control to a label that is outside the current block terminates a subroutine or a function. The label specified on the GOTO statement must be known within the block that contains the GOTO statement, and the block containing the specified label must be active when the GOTO is executed.

■ Passing Arguments to Subroutines and Functions

You specify arguments for a subroutine or function by enclosing the arguments in parentheses following the procedure or entry point name. Arguments correspond to parameters specified on the PROCEDURE or ENTRY statement of the invoked procedure. For example, a procedure call can be coded as follows:

```
CALL COMPUTER (A,B,C);
```

The variables A, B, and C in this example are arguments to be passed to the procedure COMPUTER. The procedure COMPUTER might have a parameter list like this:

```
COMPUTER: PROCEDURE (X, Y, Z);
DECLARE (X,Y,Z) FLOAT;
```

The parameters X, Y, and Z, specified in the PROCEDURE statement for the subroutine COMPUTER, are the parameters of the subroutine. PL/I establishes the equivalence of the arguments A, B, and C with the parameters X, Y, and Z.

For more information, **see** "Parameters and and Arguments."

■ Entry Points

The entry points of a procedure are the points at which it can be invoked. One entry point is specified by the PROCEDURE statement that begins the procedure block. Additional entry points may be specified with ENTRY statements

in the procedure block. ENTRY statements are allowed anywhere except within a begin block, an ON-unit, or a DO group (except a simple, noniterative DO group).

The labels used on PROCEDURE and ENTRY statements implicitly declare entry constants. (**See also** "Entry Data" and "ENTRY Statement.") The scope of these declarations is internal if the PROCEDURE and ENTRY statements appear in internal procedures and external if they appear in external procedures.

Note that the declaration of an entry name is made in the block containing the procedure to which the entry point belongs. For example:

```
P: PROCEDURE;

Q: PROCEDURE
DECLARE E FIXED BINARY;
E: ENTRY;
END Q;
```

The entry names E and Q are declared in the procedure P. Within the procedure Q, E is declared as a fixed-point binary variable.

An entry point can be invoked by using the appropriate entry constant as the reference in a CALL statement or function reference. Invoking an entry point enters a procedure at the specified point and activates the procedure block that contains the entry point.

If the CALL statement or function reference invokes an entry point in an external procedure, the entry constant must be declared with the ENTRY attribute, as in Figure P–3 above. The declaration of an external constant must also describe the parameters for that entry point, if any. For example:

```
DECLARE PITCH ENTRY (CHARACTER(*), FIXED BINARY(15));
```

The identifier PITCH is declared as an entry constant. When the procedure containing this declaration is linked to other procedures, one of the external procedures must define an entry point named PITCH, either as the label of a PROCEDURE statement or as the label of an ENTRY statement.

The data type attributes in the parentheses (known as "parameter descriptors") are the data types of the parameters that are defined elsewhere for the entry point PITCH. Arguments of these types must be supplied when PITCH is invoked. **See also** "Parameters and Arguments" and "ENTRY Attribute."

If PITCH is to be used to invoke a function, the DECLARE statement must also include a RETURNS attribute to describe the attributes of the returned value, as in:

```
DECLARE PITCH ENTRY (CHARACTER(*), FIXED BINARY(15))
  RETURNS(FIXED);
```

Within the scope of this DECLARE statement, the entry constant PITCH must be used in a function reference. The function reference will invoke the external entry point, and a returned fixed-point binary value will become the value of the function reference.

■ Multiple Entry Points

A procedure can be entered at more than one point. Only one entry point can be specified by a PROCEDURE statement; additional entry points are declared with ENTRY statements.

The rules governing the declaration of multiple entry points are:

- A particular parameter need not be specified in all of a procedure's entry points (including the point defined by the PROCEDURE statement). However, a reference to the parameter is valid only if the procedure was invoked via one of the entries specifying the parameter.
- In a procedure that has multiple entry points, a RETURN statement must be compatible with the entry point by which the procedure was invoked. If the entry point does not have a RETURNS option, the RETURN statement must not specify a return value. If the entry point has a RETURNS option, the RETURN statement must specify a return value that is valid for conversion to the data type specified in the RETURNS option. (In addition, if the entry point does not have the RETURNS option, it must be invoked as a "subroutine"—that is, with the CALL statement.)
- An ENTRY statement is not executable. If control reaches it sequentially, control simply continues on to the next statement.

The following example shows a procedure with two alternate entry points:

```
QUEUES: PROCEDURE(ELEMENT,QUEUE_HEAD);
    .
    .
ADD_ELEMENT: ENTRY(ELEMENT);
    .
    .
REMOVE_ELEMENT: ENTRY(ELEMENT);
```

This procedure can be entered by CALL statements that reference QUEUES, ADD_ELEMENT, or REMOVE_ELEMENT. If it is invoked at QUEUES, it must be passed two parameters. At either of the entries ADD_ELEMENT or REMOVE_ELEMENT, it must be passed only one parameter.

When it is entered at either alternate entry point, the entire block beginning at QUEUES is activated, but execution begins with the first executable statement following the entry point.

■ Recursive Procedures

In VAX-11 PL/I, any procedure may be invoked recursively — that is, by a statement within itself or within a dynamically descendent block (**see also** "Block"). A recursive invocation of a procedure is similar to any invocation; a recursive invocation creates a new block activation, allocates new storage for automatic variables, and so forth.

In standard PL/I, the RECURSIVE option must be used on a PROCEDURE statement if the procedure is to be invoked recursively. In VAX-11 PL/I, the

RECURSIVE option is needed only for program documentation, since all procedures can be recursive.

Procedure Block

A procedure block defines a unit of a PL/I program. The block begins with a PROCEDURE statement and ends with an END statement. The OPTIONS(MAIN) option identifies the main procedure that is activated when the program begins. A procedure block can be activated only by a CALL statement or a function reference unless it is the main procedure. The CALL statement or function reference can activate the procedure block by invoking either the label of its PROCEDURE statement or the label of an ENTRY statement within the procedure.

For information on procedure block activation, **see** "Block." For a definition and examples of procedures, **see** "Procedure" and "PROCEDURE Statement."

PROCEDURE Statement

The PROCEDURE statement defines the beginning of a procedure block and specifies the parameters, if any, of the procedure. If the procedure is invoked as a function, the PROCEDURE statement also specifies the data type attributes of the value that the function returns to its point of invocation.

The PROCEDURE statement may denote the beginning of an internal or external subroutine or function. The format of the PROCEDURE statement is:

```
entry-name: { PROCEDURE } [ (parameter,...) ]
            { PROC      }
            [ OPTIONS (option,...) ]
            [ RECURSIVE ] ;
            [ RETURNS (returns-descriptor) ]
```

entry-name

A 1- to 31-character identifier denoting the entry label of the procedure. The label cannot be subscripted. The PROCEDURE statement declares the entry name as an entry constant. The scope of the name is INTERNAL if the procedure is internal, and EXTERNAL if the procedure is external.

parameter,...

One or more parameters that the procedure expects when it is activated, separated by commas. Each parameter specifies the name of a variable declared in the procedure headed by this PROCEDURE statement. The parameters must correspond, one to one, with arguments specified for the procedure when it is invoked with a CALL statement or in a function reference. **See also** "Parameters and Arguments" for details.

OPTIONS (option,...)

An option that specifies one or more options, separated by commas. The valid options are:

IDENT (string)

An option specifying a character-string constant giving the identifying label for the listing and the module's version for the linker. Only the first 31 characters of the string are placed in the object module.

MAIN

An option specifying that the named procedure is the initial procedure in a program. The identifier of the procedure is the primary entry point for the program. The MAIN option is not allowed on internal procedures, and only one procedure in a program can have the MAIN option.

UNDERFLOW

An option that requests that the run-time system signal underflow conditions when they occur. By default, the run-time system does not signal these conditions. **See also** "UNDERFLOW Condition Name."

RECURSIVE

An option that indicates (for program documentation) that the procedure will be invoked recursively, that is, that it will be activated while it is currently active. In standard PL/I, the RECURSIVE option must be specified for a procedure to be invoked recursively. However, in VAX-11 PL/I, all procedures may be invoked recursively, and the RECURSIVE option is ignored by the compiler.

RETURNS (returns-descriptor)

An option that specifies that the procedure can be invoked only by a function reference and specifies the attributes of the function value returned. **See** "RETURNS Attribute and Option" for syntax and details.

RETURNS must be specified for functions. It is invalid for procedures that are invoked by CALL statements.

For general information on procedures, **see** "Procedure."

Program Structure

A PL/I program consists of a series of statements, which perform the following:

- Define the data to be used for program input and output
- Define the operations to be performed on the data during the execution of the program
- Control the environment within which the program executes
- Define the order of execution or control flow for a program

A statement comprises user-specified identifiers, constants, and PL/I keywords, separated by blanks, comments, and punctuation marks. Statements themselves can be organized into structural sequences of groups or blocks. Figure P–4 illustrates the structure of a PL/I program.

```
SAMPLE: PROCEDURE OPTIONS(MAIN);                    A PROCEDURE is the basic executable program unit.

DECLARE (X,Y,Z) FIXED,
        MESSAGE CHARACTER(80),                      The declarations of variables in a procedure are usu-
        CALC ENTRY (FLOAT) RETURNS(FLOAT),          ally, but not necessarily, placed at the beginning of the
        TOTAL FLOAT;                                procedure.

        X = 0;
        PUT SKIP LIST(MESSAGE);                     Executable statements are placed following variable
                                                    declarations.

FINISH: PROCEDURE;                                  Internal procedures may be placed anywhere.
DECLARE TEXT (5) CHARACTER(20);

END FINISH;                                         All procedures must terminate with END statements.
END SAMPLE;
```

Figure P–4: Structure of a PL/I Program

■ Source Program Format

The source text of a PL/I program is freeform. As long as you terminate every statement with a semicolon (;), individual statements can begin in any column, spill over onto additional lines, or be written with more than one statement to a line.

Individual keywords or identifiers of a statement cannot be split onto more than one line, however. Only a character string constant (which must be enclosed in apostrophes) can spill over onto more than one line.

PL/I programs are easier to read and to comprehend if you follow a standard pattern in formatting. For example:

- Write source statements with no more than one statement per line.
- Use indention to show the nesting level of blocks and DO-groups.

For information on the punctuation marks used in PL/I statements, **see** "Punctuation Marks." For information on blocks, **see** "Block."

Pseudovariable

VAX-11 PL/I has the pseudovariables PAGENO, STRING, SUBSTR, and UNSPEC.

A pseudovariable can be used, in certain assignment contexts, in place of an ordinary variable reference. For example:

```
SUBSTR(S,2,1) = 'A' ;
```

assigns the character 'A' to a one-character substring of S, beginning at the second character of S.

A pseudovariable can be used wherever the following three conditions are true:

1. The syntax specifies a variable reference.
2. The context is one that explicitly assigns a value to the variable.
3. The context does not require the variable to be addressable.

The principal contexts in which pseudovariables are used are:

- The left side of an assignment statement
- The input target of a GET statement

Note that a pseudovariable cannot be used in an argument list. For example:

```
CALL P(SUBSTR(S,2,1));
```

Here, SUBSTR is interpreted as a built-in function reference, not as a pseudovariable. The actual argument passed to procedure P is a dummy argument containing the second character of string S.

Punctuation Marks

PL/I recognizes punctuation marks in statements. The punctuation marks serve to:

- Specify arithmetic or relational operations to be performed on expressions in a statement
- Delimit and separate identifiers, keywords, and constants in PL/I statements

For example, in the statement shown below, the equals sign (=), representing the assignment statement, the addition operator (+), and the semicolon (;) are valid punctuation:

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

These punctuation marks separate the identifiers A, B, and C and define the operation to be performed.

Whenever you use a punctuation mark in a PL/I statement, you can precede or follow the character with any number of spaces. For example, the following two statements are equivalent:

```
DECLARE (A,B) FIXED DECIMAL (7,0);
DECLARE(A,B)FIXED DECIMAL(7,0);
```

In the second statement, the spaces preceding and following parenthetical expressions are omitted; the parentheses themselves are sufficient to distinguish elements in the statement. The only space required in this statement is the space that separates the two keywords FIXED and DECIMAL.

Table P–3 summarizes the punctuation marks that PL/I recognizes. Note that operators consisting of two characters (for example, ** and >=) must be entered without intervening spaces in a PL/I program.

Table P–3: Punctuation Marks Recognized by PL/I

Category	Symbol	Meaning to PL/I
Arithmetic operators	+	Addition or prefix plus
	-	Subtraction or prefix minus
	/	Division
	*	Multiplication
	**	Exponentiation
Relational (or comparison) operators	>	Greater than
	<	Less than
	=	Equal to
	^>	Not greater than
	^<	Not less than
	^=	Not equal to
	>=	Greater than or equal to
	<=	Less than or equal to
Logical operators	^	Logical NOT
	&	Logical AND
	| or !	Logical OR
Concatenation operator	|| or !!	String concatenation
Separators	,	Delimits elements in a list
	;	Terminates a PL/I statement
	.	Separates identifiers in a structure name; specifies a decimal point
	:	Terminates a procedure name or a statement label
	( )	Enclose lists and extents; define the order of evaluation of expressions; separate statement and option names from specific keywords
	'	Delimit character strings and bit strings
Locator qualifier	->	Pointer resolution

■ Spaces, Tabs, and Line-End Characters

In addition to punctuation marks, PL/I accepts spaces, tabs, and line-end characters between identifiers, constants, and keywords.

The line-end character is a valid punctuation mark between items in a PL/I statement except when it is embedded in a string constant. In a string constant, the line-end character is ignored. For example:

```
A = 'THIS IS A VERY LONG STRING THAT MUST BE CONTI
NUED ON MORE THAN ONE LINE IN THE SOURCE FILE' ;
```

This assignment statement gives the variable A the value of the specified character-string constant. (The line-end character in the constant is ignored.)

PUT Statement

The PUT statement transfers data from the program to the output stream. The output stream may be either a stream file or a character-string variable. The output file may be a declared file or the default file SYSPRINT.

This entry describes the syntax and options of PUT statements. For a detailed description of the execution of a PUT statement, **see** "Stream Input/Output."

The PUT statement has several forms. These forms are summarized in Figure P–5 and described individually below.

```
PUT EDIT (output-source,...) (format-specification,...)

    [ FILE (file-reference) ]
    [ LINE (expression) ]
    [ OPTIONS (option) ]
    [ PAGE ]
    [ SKIP [(expression)] ] ;

PUT LINE (expression)

    [ FILE (file-reference) ] ;

PUT LIST (output-source,...)

    [ FILE (file-reference) ]
    [ OPTIONS (option) ]
    [ PAGE ]
    [ SKIP [(expression)] ] ;

PUT PAGE

    [ FILE (file-reference) ] ;

PUT SKIP [(expression)]

    [ FILE (file-reference) ] ;

PUT STRING (reference)

    { EDIT (output-source,...) (format-specification,...) } ;
    { LIST (output-source,...)                            }
```

Option

CANCEL_CONTROL_O

Figure P–5: Forms of the PUT Statement

■ PUT EDIT

The PUT EDIT statement takes output sources (variables and expressions) from the program, converts the results to characters under control of a format specification, and places the resulting character strings in the output stream. The output stream is either a stream file or a character-string variable.

With PUT EDIT, the format of the output data is controlled by the program.

The form of the PUT EDIT statement is:

```
PUT   EDIT (output-source,...) (format-specification,...)
┌ FILE(file-reference)                    ┐
      [PAGE] [LINE(expression)]
       [SKIP[(expression)]]
       [OPTIONS(option)]
└ STRING(reference)                       ┘
  ;
```

output-source

A construct that specifies one or more expressions to be placed in the output stream.

The output sources must be separated by commas.

An output source has the following forms:

1. expression

   where the expression is of any computational type, including a reference to a scalar or aggregate variable. If the reference is to an array, data is output from array elements in row-major order. If the reference is to a structure, data is output from structure members in the order of their declaration.

2. (output-source,... DO reference=expression
   [TO expression][BY expression][WHILE(expression)])

   where the output source may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of output source are in addition to the parentheses surrounding the entire output-source list.

3. (output-source,... DO reference=expression
   [REPEAT expression][WHILE (expression)])

   where the output source may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of output source are in addition to the parentheses surrounding the entire output-source list.

   For a discussion of the matching of format items to output sources, and of the use of DO specifications, **see** "Format-Specification List."

format-specification

A list of format items to control the conversion of data items in the output list. Format items can be data format items, control format items, or remote format items. For each variable name in the output-source list, there is a corresponding data format item in the format-specification list that specifies the width of the output field and controls the data conversion. (**See** "Format-Specification List" and "Format Items and Their Uses.")

FILE(file-reference)

An option that specifies that the output stream is a stream file; the reference is to a declared file variable or constant. If neither the FILE option nor the STRING option is specified, PL/I uses the default file SYSPRINT; this print file is associated with the default system output file SYS$OUTPUT, which in turn is generally associated with the user's terminal.

If a file is specified, and it is not currently open, PL/I opens the file with the attributes STREAM and OUTPUT.

PAGE

An option that advances the output file to a new page before any data is transmitted. The PAGE option may be used only with implied or explicit print files. The file is positioned at the beginning of the next page, and the current page number is incremented by 1. The PAGE, LINE, and SKIP options are always executed, in that order, before any other output or file-positioning operations. The page size is either the default value or the specific value that you have established for the file (**see** "PAGESIZE Option"). The PAGESIZE option may be used only with print files.

LINE (expression)

An option that advances the output file to a specified line. The LINE option may be used only with implied or explict print files. The expression must yield an integer i. Blank lines are inserted in the output file such that the next output data appears on the ith line of a page.

If the file is currently positioned at the beginning of line i, no operation is performed by the LINE option.

If the file is currently positioned before line i, and i is less than or equal to the page size, then blank lines are inserted following the current line until line i is reached.

If the file is currently positioned at or beyond line i, and the file is not at the beginning of line i, then the remainder of the page (the portion between the current line and the current page size) is filled with blank lines. The ENDPAGE condition is signaled.

When the LINE option is used within an ENDPAGE ON-unit, it causes a skip to the next page.

SKIP [***(expression)***]

An option that advances a specified number of lines from the current line. The SKIP option may be used only with the implied or explicit FILE option. The expression must yield an integer i, which must not

be negative and must be greater than zero except for print files. If the expression is omitted, i equals one.

If the file is not a print file, i–1 blank lines are inserted following the current line, and subsequent output of data begins at the beginning of (current line)+i.

If the file is a print file, i=0 causes a return to the beginning of the current line. If i is greater than zero, and either the current line exceeds the page size or the page size is greater than or equal to the current line plus i, then i–1 blank lines are inserted. Otherwise, the remainder of the current page is filled with blank lines, and the ENDPAGE condition is signaled.

On output devices with the space-suppression feature, SKIP(0) can be used to cause overprinting, underlining, and so forth. For further information on lines and pages in stream files, **see** "Stream Input/Output" and "Print File."

OPTIONS (CANCEL_CONTROL_O)

A statement option that may be included only with the implied or explicit FILE option. The option is described fully in the *VAX-11 PL/I User's Guide*.

STRING(reference)

An option that specifies that the output stream is the referenced character-string variable. The STRING option cannot be used in the same statement with FILE, OPTIONS, PAGE, LINE, or SKIP.

■ Examples

```
PUTE: PROCEDURE OPTIONS(MAIN);

DECLARE SOURCE FIXED DECIMAL(7,2);

DECLARE OUTFILE PRINT FILE;

OPEN FILE(OUTFILE) TITLE('PUTE.OUT');

SOURCE = 12345.67;

PUT SKIP FILE(OUTFILE) EDIT(SOURCE) (F(8,2));
PUT SKIP FILE(OUTFILE) EDIT(SOURCE) (E(13));
PUT SKIP FILE(OUTFILE) EDIT(SOURCE) (A);
PUT SKIP FILE(OUTFILE) EDIT('American: ',SOURCE)
      (A,P'ZZ,ZZZV.ZZ');
PUT SKIP FILE(OUTFILE) EDIT('European: ',SOURCE)
      (A,P'ZZ.ZZZV,ZZ');

END PUTE;
```

The program PUTE writes the following output to PUTE.OUT:

```
12345.67
 1.234567E+04
  12345.67
American: 12,345.67
European: 12.345,67
```

■ PUT LINE

The PUT LINE statement advances a print file to a specified line. Its format is:

```
PUT [ FILE (file-reference) ] LINE (expression) ;
```

file-reference

A reference to the file to which the statement applies. The file must be a print file.

If the FILE option is not specified, PL/I uses the default file SYSPRINT. This print file is associated with the default system output file SYS$OUTPUT, which in turn is generally associated with the user's terminal.

expression

An expression giving a line in the print file, relative to the top of the current page. The expression must yield an integer i.

If the file is currently positioned at the beginning of line i, no operation is performed by the LINE option. If the file is currently positioned before line i, and i is less than or equal to the page size, then blank lines are inserted following the current line until line i is reached.

If the file is currently positioned at or beyond line i, and the file is not at the beginning of line i, then the remainder of the page (the portion between the current line and the current page size) is filled with blank lines. The ENDPAGE condition is signaled.

When the PUT LINE statement is used within an ENDPAGE ON-unit, it causes a skip to the next page.

■ PUT LIST

The PUT LIST statement specifies a list of output sources (variables and expressions) whose results are converted to character strings and transmitted to the output stream. If the output file is a print file, the output character strings are separated by tabs. Otherwise, the strings are separated by spaces.

With PUT LIST, the conversion of the output sources and formatting of the output data are automatic.

The form of the PUT LIST statement is:

```
┌  PUT   LIST (output-source,...)   ┐
   FILE(file-reference)
        [PAGE] [LINE(expression)]
         [SKIP[(expression)]]
         [OPTIONS(option)]

└  STRING(reference)                 ┘
     ;
```

output-source

A construct that specifies one or more expressions to be placed in the output stream.

The output sources must be separated by commas.

An output source has the following forms:

1. expression

   where the expression is of any computational type, including a reference to a scalar or aggregate variable. If the reference is to an array, data is output from array elements in row-major order. If the reference is to a structure, data is output from structure members in the order of their declaration.

2. (output-source,... DO reference=expression
   [TO expression][BY expression][WHILE(expression)])

   where the output source may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of output source are in addition to the parentheses surrounding the entire output-source list.

3. (output-source,... DO reference=expression
   [REPEAT expression][WHILE (expression)])

   where the output source may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of output source are in addition to the parentheses surrounding the entire output-source list.

FILE(file-reference)

An option that specifies that the output stream is a file; the reference is to a declared file variable or constant. If neither the FILE option nor the STRING option is specified, PL/I uses the default file SYSPRINT; this print file is associated by default with the system output file SYS$OUTPUT.

If a file is specified, and it is not currently open, PL/I opens the file with the attributes STREAM and OUTPUT.

PAGE

An option that advances the output file to a new page before any data is transmitted. The PAGE option may be used only with implied or explicit print files. The file is positioned at the beginning of the next page, and the current page number is incremented by one. The PAGE, LINE, and SKIP options are always executed, in that order, before any other output or file-positioning operations. The page size is either the default value or the specific value that you have established for the file (**see** "PAGESIZE Option"). The PAGESIZE option may be used only with print files.

PL/I does not skip automatically to a new page; the PAGE option must be used to perform this function.

LINE (expression)

An option that advances to a specified line in the output file. The LINE option may be used only with implied or explicit print files. The expression must yield an integer i.

If the file is currently positioned at the beginning of line i, no operation is performed by the LINE option.

If the file is currently positioned before line i, and i is less than or equal to the page size, then blank lines are inserted following the current line until line i is reached.

If the file is currently positioned at or beyond line i, and the file is not at the beginning of line i, then the remainder of the page (the portion between the current line and the current page size) is filled with blank lines. The ENDPAGE condition is signaled.

When the LINE option is used within an ENDPAGE ON-unit, it causes a skip to the next page.

SKIP [***(expression)***]

An option that advances a specified number of lines from the current line. The SKIP option may be used only with the implied or explicit FILE option. The expression must yield an integer i, which must not be negative and must be greater than zero except for print files. If the expression is omitted, i equals one.

If the file is not a print file, i–1 blank lines are inserted following the current line, and subsequent output of data begins at the beginning of (current line)+i.

If the file is a print file, i=0 causes a return to the beginning of the current line. If i is greater than zero, and either the current line exceeds the page size or the page size is greater than or equal to the current line plus i, then i–1 blank lines are inserted. Otherwise, the remainder of the current page is filled with blank lines, and the ENDPAGE condition is signaled.

On output devices with the space-suppression feature, SKIP(0) can be used to cause overprinting, underlining, and so forth. For further information on lines and pages in stream files, **see** "Stream Input/Output" and "Print File."

OPTIONS (CANCEL_CONTROL_O)

The only valid statement option for PUT statements is CANCEL_CONTROL_O. The option is described fully in the *VAX-11 PL/I User's Guide*.

STRING(reference)

An option that specifies that the output stream is the referenced character-string variable. The STRING option cannot be used in the same statement with FILE, OPTIONS, PAGE, LINE, or SKIP.

■ Examples

```
PUTL: PROCEDURE OPTIONS(MAIN);

DECLARE I FIXED BINARY,
       F FLOAT,
       P PICTURE '99V.99',
       S CHAR(10);

DECLARE INFILE STREAM INPUT FILE;
DECLARE OUTFILE PRINT FILE;

OPEN FILE(INFILE) TITLE('PUTL.IN');
OPEN FILE(OUTFILE) TITLE('PUTL.OUT');

GET FILE(INFILE) LIST (I,F,P,S);
PUT FILE(OUTFILE) SKIP LIST (I,F,P,S);

END PUTL;
```

If the file PUTL.IN contains the following data:

```
2,3.54,22.33,'A string'
```

then the program PUTL writes the following output to PUTL.OUT:

```
 2 3.5400000E+00 22.33 A string
```

For print files, each output item is written at the next tab position. Floating-point values are represented in floating-point notation. Character values are not enclosed in apostrophes.

■ PUT PAGE

The PUT PAGE statement positions the output file at the start of a new page. This statement is valid only for print files, that is, files that have been opened with the PRINT attribute.

The form of the PUT PAGE statement is:

PUT [FILE(file-reference)] PAGE;

file-reference

A reference to a print file that is to be advanced to the next output page. If no file is specified, PL/I assumes the default file SYSPRINT. This file is associated with the default system output file SYS$OUTPUT.

■ Example

```
PUT FILE(REPORT) PAGE SKIP LINE(2);
```

The PUT statement advances the file REPORT to the beginning of the next page, advances to line 2, and skips to the beginning of the next line (3).

■ PUT SKIP

The PUT SKIP statement positions the output file at the start of a new line.

The form of the PUT SKIP statement is:

PUT [FILE(file-reference)] SKIP [(expression)] ;

file-reference

A reference to the file to which the SKIP option applies. If no file is specified, PL/I assumes the file SYSPRINT. This file is associated with the default system output file SYS$OUTPUT.

If a file is specified, and it is not currently open, PL/I opens the file with the attributes STREAM and OUTPUT.

expression

An expression giving the number of lines to advance. The expression must yield an integer i, which must not be negative and must be greater than zero except for print files. If the expression is omitted, i equals one.

If the file is not a print file, i–1 blank lines are inserted following the current line, and subsequent output of data begins at the beginning of (current line)+i.

If the file is a print file, i=0 causes a return to the beginning of the current line. If i is greater than zero, and either the current line exceeds the page size or the page size is greater than or equal to the current line plus i, then i–1 blank lines are inserted. Otherwise, the remainder of the current page is filled with blank lines, and the ENDPAGE condition is signaled.

On output devices with the space-suppression feature, SKIP(0) can be used to cause overprinting, underlining, and so forth. For further information on lines and pages in stream files, **see** "Stream Input/Output" and "Print File."

R Format Item

The R format item specifies the label of a FORMAT statement from which some or all of a format specification is obtained by a GET EDIT or PUT EDIT statement.

The form of the R format item is:

R (label)

label

The label of a FORMAT statement within the same block as the GET EDIT or PUT EDIT statement. If the item occurs in a recursive procedure, the R item and FORMAT statement must occur in the same recursion.

Although the FORMAT statement can contain another R format item, these restrictions must be observed:

- The FORMAT statement cannot designate its own label with an R format item.
- The FORMAT statement cannot begin a chain of remote format items that leads back to the original FORMAT statement.

■ Examples

```
RFRM: PROCEDURE OPTIONS(MAIN);

DECLARE SYSIN STREAM INPUT FILE;
DECLARE SYSPRINT PRINT FILE;
DECLARE SALARY PICTURE '$$$$$$$$9V.99';
DECLARE (FIRST,MID,LAST) CHARACTER(80) VARYING;
DECLARE 1 HIRING,
        2 DATE CHARACTER(20) VARYING,
        2 EXPERIENCE FIXED,
        2 SALARY PICTURE '$$$$$$$$9V.99';

OPEN FILE(SYSIN) TITLE('RFRM.IN');
OPEN FILE(SYSPRINT) TITLE('RFRM.OUT');

GET EDIT(SALARY,FIRST,MID,LAST,DATE,EXPERIENCE,HIRING.SALARY)
        (F(8,2),R(PERSONNEL_FORMAT));

PUT SKIP LIST(LAST||', '||FIRST||' '||MID||':',
        'Hired '||DATE||' at '||HIRING.SALARY);
PUT SKIP LIST(EXPERIENCE||' years prior experience');
PUT SKIP LIST('Present salary: '||SALARY);

PERSONNEL_FORMAT: FORMAT(R(NAME),A(20),SKIP,F(2),X,F(8,2));
NAME: FORMAT(3(SKIP,A(80)));

END RFRM;
```

If the file RFRM.IN contains the following data:

```
△25005.50
Thomasina
A.
Delacroix
6△July△1976
△2△15003.65
```

then the following output is written to the print file RFRM.OUT:

```
Delacroix,△Thomasina△A.:△△△△△△△△Hired△6△July△1976△at△△△△$15003.60△
△△△△△△△△△△△△△2△years△prior△experience△
Present△salary:△△△△$25005.50△
```

RANK Built-In Function

The RANK built-in function returns a fixed-point binary integer that is the ASCII code for the designated character. The precision of the returned value is 7. The format of the function is:

RANK(character)

character

Any expression yielding a one-character value.

■ Examples

```
CODE = RANK('A'); /* CODE = 65 */
CODE = RANK('a'); /* CODE = 97 */
CODE = RANK('$'); /* CODE = 36 */
```

See "ASCII Character Set" for a list of the ASCII characters and their corresponding numeric codes.

READ Statement

The READ statement reads a record from a file, either the next record or a record specified by the KEY option. The file must have either the INPUT or the UPDATE attribute.

■ Format

The format of the READ statement is:

READ FILE (file-reference)

{ INTO (variable-reference) }
{ SET (pointer-variable) }

[KEY (expression)]
[KEYTO (variable-reference)]

[OPTIONS (option,...)] ;

file-reference

A reference to the file from which the record is to be read. If the file is not currently open, PL/I opens the file with the implied attributes RECORD and, if the file does not have the UPDATE attribute, INPUT. The implied attributes are merged with the attributes specified in the file's declaration. (**See also** "Opening a File.")

INTO (variable-reference)

An option that specifies that the contents of the record are to be assigned to the specified variable name. The variable must be an addressable variable.

If the variable has the VARYING attribute and the file does not have the attribute ENVIRONMENT(SCALARVARYING), the entire record is treated as a string value and assigned to the variable; if the record is longer than the variable, it is truncated and the ERROR condition is signaled. For any other type of variable, the record is simply copied into the variable's storage. If the record is not exactly the same size as the target variable, as much of the record as will fit is copied into the variable and the ERROR condition is signaled.

SET (pointer-variable)

An option that specifies that the record should be read into a buffer allocated by PL/I and the specified pointer variable be assigned the value of the location of the buffer in storage.

This buffer remains allocated until the next operation on the file but no longer. Therefore, neither the pointer value nor the buffer should be used after the next operation on the file. The only valid use of the buffer during a subsequent I/O operation is in a REWRITE statement. In this case, the record can be rewritten from the buffer before the buffer is deallocated.

KEY (expression)

An option that specifies that the record to be read is to be located using the key specified by the expression. The file must have the KEYED attribute. The key value must have a computational data type.

The nature of the key depends on the file's organization, as follows:

- If the file is a relative file or a sequential disk file with fixed-length records, the key is a fixed binary value indicating the relative record number of the record to be read.
- If the file is an indexed sequential file, the key specifies a key that is contained within a record. The data type of the key and its location within the record are as specified when the file was created.

The value of the specified expression is converted to the data type of the key. If no record with the specified key exists in the file, or if the value specified is not valid for conversion to the data type of the key, the KEY condition is signaled.

KEYTO (variable-reference)

An option that specifies that the key of the record being read is to be assigned to the designated variable. The value of the key is converted

from the data type implied by the file's organization to the data type of the variable. The variable must have a computational data type but cannot be an unaligned bit string or an aggregate consisting entirely of unaligned bit strings.

KEYTO can be specified only for a file that has both the KEYED and SEQUENTIAL attributes. It conflicts with the KEY option.

OPTIONS (option,...)

An option that specifies one or more of the READ statement options listed below, separated by commas.

FIXED_CONTROL_TO (variable-reference)
INDEX_NUMBER (expression)
MATCH_GREATER
MATCH_GREATER_EQUAL
RECORD_ID (expression)
RECORD_ID_TO (variable-reference)

These options are described fully in the *VAX-11 PL/I User's Guide*.

■ File Positioning Following a READ Statement

If the file is accessed sequentially, the READ statement reads the file's next record. If the next record position is at the end-of-file, the ENDFILE condition is signaled.

After a successful read, the file's current record position denotes the record that was just read. The next record position denotes the following record or, if there is no following record, end-of-file.

If any error occurs other than an incorrect record size, the current record becomes undefined and the next record is the same as it was before the read was attempted.

■ Examples

The program COPY, below, illustrates reading a sequential file with variable-length records into a character-string with the VARYING attribute and writing the records to a new sequential output file.

```
COPY: PROCEDURE;
DECLARE INREC CHARACTER(80) VARYING,
        ENDED BIT(1) STATIC INIT('0'B),
        (INFILE,OUTFILE) FILE;

      OPEN FILE (INFILE) RECORD INPUT
               TITLE('RECFILE.DAT');
      OPEN FILE (OUTFILE) RECORD OUTPUT
               TITLE('COPYFILE.DAT');

      ON ENDFILE(INFILE) ENDED = '1'B;
```

```
        READ FILE(INFILE) INTO (INREC);
        DO WHILE (^ENDED);
                WRITE FILE (OUTFILE) FROM (INREC);
                READ FILE (INFILE) INTO (INREC);
                END;
        CLOSE FILE(INFILE);
        CLOSE FILE(OUTFILE);
        RETURN;
        END;
```

The procedure COPY uses a DO-group to read the records in the file sequentially until the end-of-file is reached. It uses the ON statement to establish the action to take when the end-of-file occurs: it sets the bit ENDED to '1'B so that the DO-group will not be executed again.

The VARYING character-string variable INREC has a maximum length of 80 characters. If any record in the file is more than 80 characters, the ERROR condition is signaled. If no ERROR ON-unit exists, the program exits.

The next example shows a keyed READ statement to access a record in an indexed sequential file:

```
DECLARE 1 STATE,
          2 NAME CHARACTER(30),
          2 CAPITAL,
            3 NAME CHARACTER(20),
          ,
          ,
          2 SYMBOLS,
            3 FLOWER CHARACTER(30),
            3 BIRD CHARACTER(30),
        STATE_FILE FILE,
        INPUT_NAME CHARACTER(30) VARYING;
    ,
    ,
OPEN FILE(STATE_FILE) KEYED;
     PUT SKIP LIST('State?');
     GET LIST(INPUT_NAME);
     READ FILE(STATE_FILE) INTO(STATE) KEY(INPUT_NAME);
     PUT SKIP
     LIST('The flower of',STATE.NAME,'is the',FLOWER);
```

In this example, the file STATE_FILE is opened for keyed access and the READ statement specifies the key of interest in the KEY option. The value for this option is determined at run time by a GET statement. In the READ statement, the contents of a record from the file STATE_FILE are read into the structure STATE.

The next example illustrates accessing a relative file sequentially with READ statements and obtaining the key value of each record, that is, the relative record number.

```
PRINT_DATA: PROCEDURE OPTIONS(MAIN);

DECLARE 1 EMPLOYEE BASED (EP),
        2 NAME,
            3 LAST CHAR(30),
            3 FIRST CHAR(20),
            3 MIDDLE_INIT CHAR(1),
          2 DEPARTMENT CHAR(4),
          2 SALARY FIXED DECIMAL (6,2),
       EP POINTER,
       EMP_FILE FILE;

DECLARE EOF BIT(1) STATIC INIT('0'B),
       NUMBER FIXED BIN(31);

       ON ENDFILE(EMP_FILE) EOF = '1'B;
       OPEN FILE(EMP_FILE) INPUT SEQUENTIAL KEYED;

       READ FILE(EMP_FILE) SET(EP) KEYTO(NUMBER);
       DO WHILE (^EOF);
          PUT SKIP LIST('EMPLOYEE',NUMBER,
               NAME.FIRST,NAME.LAST,MIDDLE_INIT);
          READ FILE(EMP_FILE) SET(EP) KEYTO(NUMBER);
          END;
       CLOSE FILE(EMP_FILE);

END;
```

In this example, the records in the file EMP_FILE are arranged according to employee numbers. Each employee number corresponds to a relative record number in the file. READ statements read records into the based structure EMPLOYEE and set the pointer EP to the location of the allocated buffer. The READ statements specify the KEYTO option to obtain the record number of each record. The procedure prints the employee numbers and names. When the last record has been read, the program closes the input file and exits.

READONLY Attribute

The READONLY attribute can be applied to any static computational variable whose value does not change during the execution of the program.

When you specify READONLY in conjunction with the declaration of a static variable, the PL/I compiler allocates storage for the variable based on the fact

that its value does not change. A static variable with the READONLY attribute can be given an initial value with the INITIAL attribute.

The READONLY attribute is described in detail in the *VAX-11 PL/I User's Guide*.

■ Restrictions

- The READONLY attribute can only be applied to static computational variables. The variables must be declared with the EXTERNAL, STATIC, GLOBALREF, or GLOBALDEF attributes.
- The value of a variable with the READONLY attribute must not be modified. An attempt to modify a variable declared with the READONLY attribute will result in a run-time error.
- The READONLY attribute conflicts with the ENTRY, FILE, LABEL, POINTER, and VALUE attributes.

RECORD Attribute

The RECORD file description attribute indicates that data in an input or output file consists of separate records and that the file will be processed by record I/O statements.

The RECORD attribute is implied by the DIRECT, SEQUENTIAL, KEYED, and UPDATE attributes.

You can specify this attribute in a DECLARE statement for a file constant or in the OPEN statement that accesses the file. For a description of the attributes that may be applied to files and the effects of combinations of these attributes, **see** "File Description Attributes and Options."

■ Restrictions

The RECORD attribute conflicts with the STREAM and PRINT attributes.

Record Input/Output

Record input/output is performed by the READ, WRITE, DELETE, and REWRITE statements. In record I/O, each I/O statement processes an entire record. (In stream I/O, more than one line or record can be processed by a single statement; **see** "Stream Input/Output" for details.) Table R–1 summarizes the PL/I file description attributes that apply to record I/O. For an overview of how to declare and reference files in PL/I, **see** the entry "File."

Table R-1: Access Modes for Record Files

Attributes Specified	Attributes Implied	Valid Devices and File Organizations	Usage
SEQUENTIAL OUTPUT	RECORD	Any output device or file except indexed	Records may be added to the end of the file using WRITE statements. Each WRITE statement adds a single record to the file.
SEQUENTIAL INPUT	RECORD	Any input device or file	Records in the file are read using READ statements. Each statement reads a single record.
SEQUENTIAL UPDATE	RECORD	Relative, indexed, sequential disk[1]	READ statements read a file's records in order. PL/I maintains the current record, which is the record just read. This record may be replaced in a REWRITE statement. In a relative or indexed sequential file, the current record may also be deleted with a DELETE statement. Each statement processes a single record.
DIRECT OUTPUT	KEYED RECORD	Relative, indexed, sequential disk[1]	WRITE statements insert records into the file at positions specified by keys. Each statement inserts a single record.
DIRECT INPUT	KEYED RECORD	Relative, indexed, sequential disk[1]	READ statements specify records to be read randomly by key. Each statement reads a single record.
DIRECT UPDATE	KEYED RECORD	Relative, ndexed, sequential disk[1]	READ, WRITE, and REWRITE statements specify records randomly by key. In a relative or indexed file, records may also be deleted by key.
KEYED SEQUENTIAL OUTPUT	RECORD	Relative, indexed, sequential disk[1]	WRITE statements insert records into the file at positions specified by keys. Each statement inserts a single record. This mode is identical to DIRECT OUTPUT.
KEYED SEQUENTIAL INPUT	RECORD	Relative, indexed, sequential disk[1]	READ statements access records in the file randomly by key or sequentially.
KEYED SEQUENTIAL UPDATE	RECORD	Relative, indexed, sequential disk[1]	Any record I/O operation is allowed except a WRITE statement that does not specify a key or a DELETE statement for a sequential disk file with fixed-length records.

1. The file must have fixed-length records.

■ Position Information for a Record File

When a record file is open, PL/I maintains the following position information:

- The next record, for files with the SEQUENTIAL INPUT or SEQUENTIAL UPDATE attributes. The next record designates the record that will be accessed by a READ statement that does not specify the KEY option. The next record may contain end-of-file.
- The current record, for a file with the UPDATE attribute. The current record designates either of the following:
  - the record that will be modified by a REWRITE statement that does not specify the KEY option
  - the record that will be deleted by a DELETE statement that does not specify the KEY option

  The value of the current record may be undefined.

When a file is opened the current record is undefined and the next record designates the first record in the file or, if the file is empty, end-of-file.

After a sequential read, the current record designates the record just read. The next record indicates the following record or, if there are no more records, the end-of-file.

After a keyed I/O statement, that is, an I/O statement that specifies the KEY or KEYFROM option, the current record and next record are set as follows:

Statement	Current Record	Next Record
READ	X	X+1
WRITE	X	X+1
REWRITE	X	X+1
DELETE	undefined	X

where X is the record specified by key and X+1 is the next record or, if there are no more records, the end-of-file.

RECURSIVE Option

The RECURSIVE option may be specified on a PROCEDURE or ENTRY statement to indicate (for program documentation) that the procedure will invoke itself. For example:

```
HANOI: PROCEDURE (T1, T2, T3, RINGS)
         RECURSIVE;
```

In standard PL/I, the RECURSIVE option is required for a recursive procedure. However, in VAX-11 PL/I all procedures may be invoked recursively, and the RECURSIVE option is ignored by the compiler. For more information, **see** "Procedure."

Reference

In this manual, the term *reference* means a reference to a named constant or variable. This entry gives the complete syntax for references and explains in detail how a reference is interpreted. Because of the flexibility of PL/I, this explanation is complex and is probably of interest to only a few programmers. For general information on how to write references to accomplish a specific operation, **see**, for example, "Built-In Function," "Expression," and "Procedure."

■ Syntax of References

The complete syntax of a reference is:

```
[locator-qualifier] [structure-qualifier]... identifier
 [(subscript-list)] [(argument-list)]
```

The referenced identifier is the declared name of the constant or variable.

The locator qualifier has the form:

```
reference->
```

where reference is a reference to a pointer variable, a pointer-valued function, or an offset variable that was declared wtih a base area. (**See** "Based Variable" and "Offset.")

The structure qualifier has the form:

```
identifier [(subscript-list)] .
```

where identifier is the name of a structure declaration containing (at some level) a declaration of the referenced identifier.

The subscript list is a list of integer-valued expressions separated by commas. The argument list is either empty or is a list of expressions, separated by commas, that determine the arguments of a procedure or built-in function. If, ignoring structure qualifiers, only one of the subscript list or argument list is included, the listed items are interpreted as subscripts or arguments depending on the declaration of the referenced identifier.

■ Complete Interpretation of a Reference

Complete interpretation of a reference follows the sequence of steps given below.

1. *Determine the initial block, B, of interpretation.* This is the block in which the search for the referenced declaration starts. The initial block, B, is always the block in which the reference textually occurs. This is usually

the current block, but it may be a parent block when references from declarations are to be interpreted. For example:

```
P: PROC;
  DCL X(N) FIXED BASED(R);
.
.
Q: PROC;
X(1)=0;
```

When the assignment statement X(1)=0 is executed, the reference X(1) is interpreted in the block Q. However, interpreting X(1) requires interpreting the references N and R in X's declaration, and this is done in block P, the block of X's declaration.

2. *Find the referenced declaration, D, and the block to which it belongs.* This block becomes the block, B, for further interpretation. To find the declaration, make a qualifying list containing the identifiers in the structure qualifiers (if any) and the identifiers in the reference, taken in left-to-right order. Search the declarations in block B for any declaration whose complete list of qualifying names matches the reference's qualifying list, as follows:

   (a) If the reference's qualifying list of names is the same as the declaration's list, the reference completely matches the declaration. In this case, the declaration is the reference's governing declaration; no further searching of declarations is done.

   (b) If the reference's qualifying list is a sublist (in order) of the declaration's, and the last occurring identifiers are the same, then the reference is partially qualified. If the reference does not completely match any declaration as in 2(a), and it does partially match exactly one declaration in B, then that declaration is the governing declaration. If the reference does not completely match as in 2(a), and it partially matches two or more declarations in B, then the reference is ambiguous, and the compiler issues an error message.

   (c) If the reference does not match any declaration in B, B is replaced by its immediate parent block, and the search for matching declarations is performed again. This process continues until a match is found or there is no parent block (the outermost block has been searched). In the latter case, if the identifier in the reference is SYSIN or SYSPRINT, or the name of a built-in function, the compiler creates an appropriate declaration in the external procedure. Otherwise, it issues an error message.

For example, suppose the block being searched contains only the following structure declaration:

```
DECLARE 1 STATE,
  2 NAME CHAR(20) VAR,
  2 POPULATION FIXED,
  2 CAPITAL,
          3 NAME CHAR(30) VAR,
          3 POPULATION FIXED,
  2 SYMBOLS,
          3 FLOWER CHAR(20),
          3 BIRD CHAR(20);
```

The references STATE, STATE.NAME, and STATE.CAPITAL.NAME match completely. The references NAME and POPULATION are ambiguous. The reference CAPITAL.NAME partially matches exactly one declaration.

3. *Find the block activation, BA, associated with the block B.* If B is the current block, BA is the current block activation. Otherwise, BA is found by searching the chain of parent block activations that ends at the current block. BA is used to determine the value of a reference to an automatic variable, the actual argument associated with a parameter, the extents of automatic or defined variables, and the block-activation component of a label or entry value when a label or entry constant is interpreted.
4. *Evaluate the locator qualifier.* If the reference contains a locator qualifier, evaluate it to obtain a pointer value. In this case, the reference must be to a based variable or to a member of a based structure. If the reference is to a based variable or to a member of a based structure, and if the reference does not contain a locator qualifier, the level-1 variable must have been declared:

   BASED(reference)

   and that reference is evaluated as a locator qualifier. Note that the pointer value obtained must satisfy the rules given in "Based Variable — Pointer Values."
5. *Evaluate the base reference and position.* If the reference is to a defined variable, its base reference and POSITION attribute (if any) are evaluated.
6. *Determine all extents of the referenced variable.* The extents (if any) are given in the declaration. Those which are not constant are determined as follows:
   (a) If the variable is automatic or defined, the extents were evaluated at the time of block activation and the resulting values saved at that time. These saved values are used.
   (b) If the variable is a parameter, the extents were passed along with the argument to which the parameter corresponds.
   (c) If the variable is a based variable, the extents are evaluated now. This includes all extent expressions in the referenced declaration and all array bounds of containing structures.

7. *Interpret subscripts.* This step depends on the total dimensionality of the referenced declaration, D; that is, the number of dimensions in D itself plus the number in each containing structure declaration. The subscripts are evaluated as follows:

   (a) All subscripts in the structure qualifiers (if any) are gathered together in one list (in order). If the reference itself contains both a subscript list and an argument list [for example, S.Y(1,1,1,)(7)], those subscripts are added to the list. If the reference contains a single list, which could be either subscripts or arguments, its elements are treated as subscripts and added to the list unless the number of subscripts already collected equals D's dimensionality. If the single list is not interpreted as a subscript list, it is an argument list. Note that an empty argument list is never interpreted as a subscript list.

   (b) The complete list of subscripts is now compared with the dimensionality of D and with each declaration of an array of structures containing D. The number of subscripts must be zero or equal to the total dimensionality of one of these declarations. If it is not, the compiler issues an error message. The array properties of the reference are then determined as follows:

      (i) If the number of subscripts equals the total dimensionality of D, this is not an array reference.

      (ii) If D is an array declaration and the number of subscripts equals the inherited dimensionality of D (that is, D's total dimensionality minus the dimensionality of D itself), then this is a connected array reference.

      (iii) If the number of subscripts is less than the inherited dimensionality of D, then the reference is an unconnected array reference.

   For example, consider this declaration and the following series of references:

```
DECLARE 1 S(5),
        2 A(10,20),
          3 X(50) FLOAT,
          3 Y ENTRY(FIXED),
          3 Z FLOAT,
        2 B ENTRY(FLOAT,FLOAT);
```

   S.A(1,1)

   This reference is invalid, because the number of subscripts is too large for S and too small for S.A.

   S(1).A(1,1).Y(3)
   S.A.Y(1,1,1)(3)

   These are equivalent. The value of S.A.Y(1,1,1) is an entry value. The entry is invoked with the argument list (3).

   S(1).A(1,1).X

   This is a reference to a connected one-dimensional floating-point array whose bounds are (1:50).

S.A.X(3,10,20,2)
This is a reference to a floating-point variable that is an element of the array S.A.X.

S(1).A.X
This is a reference to an unconnected array. It is three dimensional, with bounds (1:10,1:20,1:50).

(c) All subscript values must lie within the corresponding bounds. If the compiler option CHECK is used, all subscript values are checked either at compile time or at run time. If CHECK is not in effect, some constant subscripts may still be checked at compile time.

8. *Invoke the procedure.* If the reference contains an argument list or was the reference in a CALL statement, the referenced procedure is invoked with the specified arguments. (**See also** "Procedure.") In this case, the reference must not be an array reference and must have data type ENTRY. If the reference is to an entry variable, the procedure is invoked using the current value of the variable. Note that the ENTRY attribute and RETURNS attribute (if any) in the declaration D are used to interpret the argument list and to determine if this is a function or a procedure invocation.

Relational Operator

The relational, or comparison, operators test the relationship of two operands; the result is always a Boolean value (that is, a bit string of length one). If the comparison is true, the resulting value is '1'B; if the comparison is false, the resulting value is '0'B. The relational operators are all infix operators. They are:

Operator	Operation
<	Less than
^<	Not less than
<=	Less than or equal to
=	Equal to
^=	Not equal to
>=	Greater than or equal to
>	Greater than
^>	Not greater than

Relational operators compare any of the following data types: arithmetic (decimal or binary); bit-string; character-string; and entry, pointer, label, or file data. Specific results of operations on each type of data are elaborated below. The following general rules apply:

- All operands must be scalars.
- Both operands must be arithmetic, or they must have the same data type.

■ Arithmetic Comparisons

Arithmetic and picture operands are compared algebraically. If the operands have a different base, scale, or precision, PL/I converts them according to the rules for arithmetic operand conversion (**see** "Expression").

■ Bit-String Comparisons

When two bit strings are compared, they are compared bit by bit from the most significant bit to the least significant bit (as represented by PUT LIST). If the operands have different lengths, PL/I extends the smaller operand with zeros in the direction of the least significance. Null bit strings are always equal.

■ Character-String Comparisons

When two character strings are compared, they are compared character by character in a left to right order. The comparison is based on the ASCII collating sequence. The ASCII value for each character is given in Table A–1 in the entry "ASCII Character Set."

Note that in the collating sequence:

- Uppercase letters are less than any lowercase letters.
- Numeric characters are less than any letters.

If the operands are not the same length, PL/I extends the smaller operand on the right with blanks for the comparison. Either or both of the strings can have the attribute VARYING; PL/I uses the current length of a varying character string when it makes the comparison.

■ Comparing Noncomputational Data

Only the following operators are valid, or meaningful, for comparisons of any of the noncomputational data types entry, file, label, offset, and pointer:

Operator	Operation
=	Equal
^=	Not equal

The results of the comparisons provide the information indicated below for each data type.

Entry Data

Two entry values are equal if they identify the same entry point in the same block activation of a procedure.

File Data

Two values defined with the FILE attribute are equal if they identify the same file constant.

Label Data

Two label values are equal if they identify the same statement in the same block activation.

A label that identifies a null statement is not equal to the label of any other statement.

Pointer Data

Two pointer values are equal if they identify the same storage location or if they are both null.

Offset Data

Two offset values are equal if they identify the same storage location or if they are both null.

REPEAT Option

The REPEAT option may be specified in a DO statement to specify values to be assigned to the control variable. The input-target and output-source lists of GET and PUT statements can also have a DO construct with the REPEAT option. The REPEAT option is most often used to step though a list that is linked by pointer or offset values. For example:

```
DO P = LIST_HEAD REPEAT P->LIST_ELEMENT.NEXT
       WHILE (P ^= NULL()) ;
```

For more information, **see** "DO Statement," "GET Statement," "List Processing," and "PUT Statement."

%REPLACE Statement

The %REPLACE statement specifies that an identifier is a constant of a given value. It may be used anywhere within a procedure or anywhere in a PL/I source file.

Beginning at the point at which a %REPLACE statement is encountered, PL/I replaces all occurrences of the specified identifier with the specified constant value, until the end of compilation.

The format of the %REPLACE statement is:

%REPLACE identifier BY constant-value ;

identifier
: Any valid PL/I identifier. The identifier must not be the name of a declared variable and can appear in only one %REPLACE statement in a source program.

constant-value
: Any valid character-string, bit-string, or arithmetic constant.

Integer constants that are given values by %REPLACE statements are valid in constant expressions. For example:

```
%REPLACE PREFIX BY 8;

DECLARE BUFFER CHARACTER( 80 + PREFIX);
```

When the program containing these lines is compiled, the variable BUFFER is declared with a length of 88 characters.

RESIGNAL Built-In Subroutine

The RESIGNAL built-in subroutine is used in an ON-unit to "pass" a signaled condition, so that the run-time system will attempt to locate another ON-unit to handle the condition. The format of the RESIGNAL built-in subroutine is:

CALL RESIGNAL();

■ Bit-String Comparisons

When two bit strings are compared, they are compared bit by bit from the most significant bit to the least significant bit (as represented by PUT LIST). If the operands have different lengths, PL/I extends the smaller operand with zeros in the direction of the least significance. Null bit strings are always equal.

■ Character-String Comparisons

When two character strings are compared, they are compared character by character in a left to right order. The comparison is based on the ASCII collating sequence. The ASCII value for each character is given in Table A–1 in the entry "ASCII Character Set."

Note that in the collating sequence:

- Uppercase letters are less than any lowercase letters.
- Numeric characters are less than any letters.

If the operands are not the same length, PL/I extends the smaller operand on the right with blanks for the comparison. Either or both of the strings can have the attribute VARYING; PL/I uses the current length of a varying character string when it makes the comparison.

■ Comparing Noncomputational Data

Only the following operators are valid, or meaningful, for comparisons of any of the noncomputational data types entry, file, label, offset, and pointer:

Operator	Operation
=	Equal
^=	Not equal

The results of the comparisons provide the information indicated below for each data type.

Entry Data

Two entry values are equal if they identify the same entry point in the same block activation of a procedure.

File Data

Two values defined with the FILE attribute are equal if they identify the same file constant.

Label Data

Two label values are equal if they identify the same statement in the same block activation.

A label that identifies a null statement is not equal to the label of any other statement.

Pointer Data

Two pointer values are equal if they identify the same storage location or if they are both null.

Offset Data

Two offset values are equal if they identify the same storage location or if they are both null.

REPEAT Option

The REPEAT option may be specified in a DO statement to specify values to be assigned to the control variable. The input-target and output-source lists of GET and PUT statements can also have a DO construct with the REPEAT option. The REPEAT option is most often used to step though a list that is linked by pointer or offset values. For example:

```
DO P = LIST_HEAD REPEAT P->LIST_ELEMENT.NEXT
       WHILE (P ^= NULL()) ;
```

For more information, **see** "DO Statement," "GET Statement," "List Processing," and "PUT Statement."

%REPLACE Statement

The %REPLACE statement specifies that an identifier is a constant of a given value. It may be used anywhere within a procedure or anywhere in a PL/I source file.

Beginning at the point at which a %REPLACE statement is encountered, PL/I replaces all occurrences of the specified identifier with the specified constant value, until the end of compilation.

The format of the %REPLACE statement is:

%REPLACE identifier BY constant-value ;

identifier
: Any valid PL/I identifier. The identifier must not be the name of a declared variable and can appear in only one %REPLACE statement in a source program.

constant-value
: Any valid character-string, bit-string, or arithmetic constant.

Integer constants that are given values by %REPLACE statements are valid in constant expressions. For example:

```
%REPLACE PREFIX BY 8;

DECLARE BUFFER CHARACTER( 80 + PREFIX);
```

When the program containing these lines is compiled, the variable BUFFER is declared with a length of 88 characters.

RESIGNAL Built-In Subroutine

The RESIGNAL built-in subroutine is used in an ON-unit to "pass" a signaled condition, so that the run-time system will attempt to locate another ON-unit to handle the condition. The format of the RESIGNAL built-in subroutine is:

CALL RESIGNAL();

When an ON-unit has determined that it cannot or should not respond to a condition, RESIGNAL permits the ON-unit to pass the signal along.

This subroutine is not provided in the standard PL/I language. It is provided specifically for use in the VAX/VMS operating system environment. For complete details on condition handling in VAX/VMS, **see** the *VAX-11 PL/I User's Guide*.

RETURN Statement

The RETURN statement terminates execution of the current procedure. The format of the RETURN statement is:

RETURN [(return-value)] ;

return-value

The value to be returned to the invoking procedure. If the current procedure was invoked by a function reference, a return value must be specified. If the current procedure was invoked by a CALL statement, a return value is invalid.

A return value can be any scalar arithmetic, bit-string, or character-string expression; it can also be an entry, pointer, or label expression or other noncomputational expression. The return value must be valid for conversion to the data type specified in the RETURNS option of the function.

The actual action taken by the RETURN statement depends on the context of the procedure activation, as follows:

- If the current procedure is the main, or only, active procedure, the RETURN statement terminates the program.
- If the current procedure was activated by a CALL statement, the next executable statement in the calling procedure is executed.
- If the current procedure was activated by a function reference, control returns to continue the evaluation of the statement that contained the function reference.
- If the RETURN statement is executed in a begin block, the effect is to return from the containing procedure.

■ Restrictions

The RETURN statement must not be immediately contained in an ON-unit or in a begin block that is immediately contained in an ON-unit.

RETURNS Attribute and Option

The RETURNS option must be specified on the PROCEDURE or ENTRY statement if the corresponding entry point is invoked as a function. (**See also** "Procedure.") The RETURNS attribute is specified with the ENTRY attribute, to give the data type of a value returned by an external function. The format of the option or attribute is:

RETURNS (returns-descriptor)

returns-descriptor

One or more attributes that describe the value returned by the function to its point of invocation. The returned value becomes the value of the function reference in the invoking procedure. The attributes must be separated by spaces except for attributes (precision, for example) that are enclosed in parentheses.

■ Restrictions

The data types you can specify for a returns descriptor are restricted to scalar elements of either computational or noncomputational types. Areas are not allowed.

The extent of a character-string value may be specified as an asterisk (*), to indicate that the string may have any length; in this case, VARYING must not be specified. Otherwise, extents must be specified using unsigned decimal integer constants.

The RETURNS option and RETURNS attribute must not be used for procedures that are invoked by the CALL statement.

The attributes specified in a returns descriptor in a RETURNS attribute must correspond to those specified in the RETURNS option of the PROCEDURE statement or ENTRY statements in the corresponding procedure. For example:

```
CALLER: PROCEDURE OPTIONS (MAIN);
      DECLARE COMPUTER ENTRY (FIXED BINARY)
      RETURNS (FIXED BINARY); /* RETURNS attribute */
      DECLARE (TOTAL,A,B) FIXED BINARY;
        .
        .
      TOTAL = COMPUTER (A+B);
```

The first DECLARE statement declares an entry constant named COMPUTER. COMPUTER will be used in a function reference to invoke an external procedure, and the function reference must supply a fixed-point binary argument. The invoked function returns a fixed-point binary value, which then becomes the value of the function reference.

The function COMPUTER contains:

```
COMPUTER: PROCEDURE (X)
  RETURNS (FIXED BINARY); /* RETURNS option */
DECLARE (X, VALUE) FIXED BINARY;
  .
  .
RETURN (VALUE); /* RETURN statement */
```

In the PROCEDURE statement, COMPUTER is declared as an external entry constant, and the RETURNS option specifies that the procedure returns a fixed-point binary value to the point of invocation. The RETURN statement specifies that the value of the variable VALUE is returned by COMPUTER. If the data type of the returned value does not match the data type specified in the RETURNS option, PL/I converts the value to the correct data type according to the rules given under "Conversion of Data."

REVERT Statement

The REVERT statement cancels an ON-unit established for a specified condition in the current block. The format of the REVERT statement is:

REVERT condition-name ;

condition-name

Specifies the keyword name associated with the condition for which the ON-unit is to be reverted. It must be one of the keyword names listed below. Each of these conditions is described under its own entry in this manual.

Condition Names

ANYCONDITION
ENDFILE (file-reference)
ENDPAGE (file-reference)
ERROR
FINISH
FIXEDOVERFLOW
KEY (file-reference)
OVERFLOW
UNDEFINEDFILE (file-reference)
UNDERFLOW
VAXCONDITION (expression)
ZERODIVIDE

If no ON-unit is established for the specified condition for the current block, the REVERT statement has no effect. When the REVERT statement is executed for a specific condition for which an ON-unit exists, then:

- If a previous block activation specified an ON-unit for the indicated condition, that ON-unit will be executed if the condition is signaled.
- If no previous block activation specified on ON-unit for the specified condition, the default PL/I condition handling is reestablished.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement."

REWRITE Statement

The REWRITE statement replaces a record in a file, either the current record or the record specified by the KEY option. The file must have the UPDATE attribute. The format of the REWRITE statement is:

REWRITE FILE (file-reference)
[FROM (variable-reference) [KEY (expression)]]
[OPTIONS (option,...)] ;

file-reference

A reference to the file which contains the record to be replaced. If the file is not open, it is opened with the implied attributes RECORD and UPDATE; these attributes are merged with the attributes specified in the file's declaration. (**See also** "Opening a File.")

FROM (variable-reference)

An option giving the variable whose value is to be used to rewrite the specified record. The variable must be an addressable variable. (**See** "Variable — Addressable Variable.")

If the FROM option is not specified, there must be a currently allocated buffer from an immediately preceding READ statement that specified the SET option, and this file must have the SEQUENTIAL attribute. In this case, the record is rewritten from the buffer containing the record that was read.

If the variable has the VARYING attribute and the file does not have the attribute ENVIRONMENT(SCALARVARYING), the REWRITE statement writes only the current value of the varying string into the specified record. In all other cases, the REWRITE statement writes the variable's entire storage.

KEY (expression)

An option specifying that the record to be rewritten is to be located using the key specified by expression. The file must have the KEYED attribute. The expression must have a computational data type. The FROM option must be specified.

The nature of the key depends on the file's organization, as follows:

- If the file is a relative file or a sequential disk file with fixed-length records, the key is a fixed binary value indicating the relative record number of the record to be rewritten.
- If the file is an indexed sequential file, the key specifies a key that is contained within a record. The data type of the key and its location within the record are as specified when the file was created. The primary key field in the record cannot be modified.

The value of the specified expression is converted to the data type of the key. If no record with the specified key exists, if the value specified is not valid for conversion to the data type of the key, or if the primary key in a record in an indexed sequential file has been modified, the KEY condition is signaled.

OPTIONS (option,...)

An option giving one or more of the REWRITE statement options listed below, separated by commas:

FIXED_CONTROL_FROM(variable-reference)
INDEX_NUMBER (expression)
MATCH_GREATER
MATCH_GREATER_EQUAL
RECORD_ID (expression)
RECORD_ID_TO (variable-reference)

These options are described fully in the *VAX-11 PL/I User's Guide*.

■ File Positioning

The next record position is set to denote the record immediately following the record that was rewritten or, if there is no following record, end-of-file.

The current record is set to designate the record just rewritten.

■ Examples

The procedure NEW_SALARY, below, updates the salary field in a relative file containing employee records. The procedure receives two input parameters: the employee number and the new salary. The employee number is the key value for the records in the relative file.

```
NEW_SALARY: PROCEDURE (EMPLOYEE_NUMBER,PAY);

DECLARE EMPLOYEE_NUMBER FIXED DECIMAL(5,0),
     PAY FIXED DECIMAL (6,2);

DECLARE 1 EMPLOYEE,
       2 NAME,
            3 LAST CHAR(30),
            3 FIRST CHAR(20),
            3 MIDDLE_INIT CHAR(1),
          2 DEPARTMENT CHAR(4),
          2 SALARY FIXED DECIMAL (6,2),
     EMP_FILE FILE;

     OPEN FILE(EMP_FILE) DIRECT UPDATE;
     READ FILE(EMP_FILE) INTO(EMPLOYEE)
          KEY (EMPLOYEE_NUMBER);
     EMPLOYEE.SALARY = PAY;
     REWRITE FILE(EMP_FILE) FROM(EMPLOYEE)
          KEY(EMPLOYEE_NUMBER);
     CLOSE FILE(EMP_FILE);
     RETURN;
     END;
```

In this example, the KEY option is specified in the READ statement that obtains the record of interest and in the REWRITE statement that replaces the record, with its new information, in the file. The FROM and KEY options must both be specified on the REWRITE statement.

The sample program CHANGE_HEADER, below, changes the contents of the first record in the sequentially organized file TITLE_PAGE. The file consists of 80-byte, fixed-length records.

```
CHANGE_HEADER: PROCEDURE OPTIONS(MAIN);

DECLARE TITLE_PAGE FILE SEQUENTIAL UPDATE,
        INREC CHARACTER(80) BASED(P),
        P POINTER;

     OPEN FILE(TITLE_PAGE);
     READ FILE(TITLE_PAGE) SET(P);

     INREC = 'Summary of Courses for Fall 1980';
     REWRITE FILE(TITLE_PAGE);
     CLOSE FILE(TITLE_PAGE);
     RETURN;
END;
```

In this example, the READ statement specifies the SET option. The input record is read into a buffer, INREC, that is a based character-string variable. The assignment statement modifies the buffer and the REWRITE statement

rewrites the record. Because the REWRITE statement does not specify a FROM option, PL/I uses the contents of the buffer to rewrite the current record in the file (that is, the record that was just read).

ROUND Built-In Function

The ROUND built-in function rounds a fixed-point decimal expression to a specified number of decimal places. Its format is:

ROUND(expression,position)

expression

An arithmetic expression that yields a fixed-point decimal value with a nonzero scale factor or a pictured value with fractional digits.

position

A nonnegative integer constant specifying the number of decimal places in the rounded result.

■ Returned Value

Where the arguments are an expression of type FIXED DECIMAL(p,q) and a position k, the returned value is the rounded value with the attributes:

precision: max(1,min(p–q+k+1,31))

scale factor: k

The rounded value is

ROUND(x,k) = sign(x) * (10^{-k}) * floor(abs(x) * (10^{k}) + .5)

■ Examples

```
A = 1234.567;
Y = ROUND(A,1);      /*   Y = 1234.6    */

Y = ROUND(A,0);      /*   Y = 1235      */

A = -1234.567;
Y = ROUND(A,2);      /*   Y = -1234.57  */
```

Scope of Names

The scope of a declaration of a name is that region of the program in which the name is known. A declaration of a name is known in:

- The block in which it is declared
- Any blocks contained within the declaring block, so long as the name is not redeclared in the contained block

Two declarations of the same name denote distinct objects unless both specify the EXTERNAL attribute. All EXTERNAL declarations of a particular name denote the same variable or constant, and all must agree as to the properties of the variable or constant. Note that EXTERNAL is supplied by default for declarations of ENTRY and FILE constants. It must be specified explicitly for variables.

Figure S–1 illustrates the scope of internal names.

```
MAINP: PROCEDURE OPTIONS (MAIN);

DECLARE (X, Y, VALUE) FIXED;
|
|   ALPHA: PROCEDURE;
|   |
|   |   BETA: BEGIN;
|   |   DECLARE VALUE FLOAT;
|   |   |
|   |   |     GOTO ERROR;
|   |   |
|   |   END BETA;
|   |
|   |   ERROR:
|   END ALPHA;
|
|   CALC: PROCEDURE;
|   DECLARE (SUM,TOTAL) FLOAT;
|   |
|   END CALC;
END MAINP;
```

Name	Scope
MAINP	MAINP, ALPHA, BETA, and CALC
X, Y	MAINP, ALPHA, BETA, and CALC
VALUE (MAINP)	MAINP, ALPHA, and CALC
VALUE (BETA)	BETA
ALPHA	MAINP, BETA, and CALC
BETA	ALPHA
ERROR	ALPHA, BETA
CALC	MAINP, ALPHA
SUM, TOTAL	CALC

Figure S–1: Scope of Internal Names

SEQUENTIAL Attribute

The SEQUENTIAL file description attribute indicates that records in the file will be accessed in a sequential manner. The format of the SEQUENTIAL attribute is:

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

If you specify SEQUENTIAL, the RECORD attribute is implied.

Specify the SEQUENTIAL attribute in a DECLARE statement for a file constant or in the OPEN statement that accesses the file.

The SEQUENTIAL attribute can be applied to files with sequential, relative, or indexed sequential file organizations.

■ Restrictions

The SEQUENTIAL attribute conflicts with the DIRECT, STREAM, and PRINT attributes.

SET Option

The SET option may be specified in an ALLOCATE or READ statement. In an ALLOCATE statement, it sets a pointer variable to the memory location of storage acquired for a based variable. In a READ statement, it sets a pointer variable to the location of the input buffer.

■ Examples

```
ALLOCATE X SET (P);
READ FILE (STATE) SET (READBUF);
```

See also "ALLOCATE Statement" and "READ Statement."

SIGN Built-In Function

The SIGN built-in function returns 1, -1, or 0, indicating whether an arithmetic expression is positive, negative, or zero, respectively. The returned value is a fixed-point binary integer. The format of the function is:

```
SIGN(expression)
```

SIGNAL Statement

The SIGNAL statement causes a specified condition to be signaled. The format of the SIGNAL statement is:

```
SIGNAL condition-name ;
```

condition-name

The name of the condition to be signaled. It must be one of the keywords listed below. Each of these conditions is described under its own entry.

Condition Names

ANYCONDITION
ENDFILE (file-reference)
ENDPAGE (file-reference)
ERROR
FINISH
FIXEDOVERFLOW
KEY (file-reference)
OVERFLOW
UNDEFINEDFILE (file-reference)
UNDERFLOW
VAXCONDITION (expression)
ZERODIVIDE

Most conditions occur as a result of a hardware trap or fault, or as a result of signaling by PL/I run-time procedures. The SIGNAL statement may be used within a program as a general-purpose communication technique.

In particular, the VAXCONDITION keyword lets you specify unique programmer-defined values as well as operating-system-specific values. The VAXCONDITION keyword is described briefly in its own entry and in more detail in the *VAX-11 PL/I User's Guide*.

For details on condition handling, **see** "ON Conditions and ON-Units."

SIN Built-In Function

The SIN built-in function returns a floating-point value that is the sine of an arithmetic expression x, where x is an angle in radians. The sine is computed in floating point. The format of the function is:

SIN(x)

SIND Built-In Function

The SIND built-in function returns a floating-point value that is the sine of an arithmetic expression x, where x represents an angle in degrees. The sine is computed in floating point. The format of the function is:

SIND(x)

SINH Built-In Function

The SINH built-in function returns a floating-point value that is the hyperbolic sine of an arithmetic expression x. The hyperbolic sine is computed in floating point. The format of the function is:

SINH(x)

SKIP Format Item

The SKIP format item sets a stream file to a new position relative to the current line. It may be used with input and output files.

The form of the SKIP format item is:

SKIP [(w)]

w

An integer giving the number of lines to be skipped; w must not be negative and must be greater than zero except for print files. If it is omitted, a value of one is assumed.

If w is one or is omitted, the file is positioned at the beginning of the next line. If w is greater than one, w–1 lines are skipped on input, but the ENDFILE condition is signaled if the end of the file is encountered first. On output, w–1 blank lines are inserted. In both cases, the new position is the beginning of (current line)+w.

■ Use with Print Files

If w is zero, the file is repositioned at the beginning of the current line, allowing overprinting of the line. If w is greater than zero, and either the current line exceeds the page size or the page size is greater than or equal to the current line plus w, then w–1 blank lines are inserted. Otherwise, the remainder of the page (the portion between the current line and the page size) is filled with blank lines, and the ENDPAGE condition is signaled.

SKIP Option

The SKIP option is used with the GET and PUT statements to advance the stream file to a new line before beginning a data transfer.

The SKIP option specifies a line number relative to the current line. In some cases, this line number can be 0, which causes a return to the beginning of the current line. With the PUT statement, the option SKIP(0) allows overprinting of a line in a PRINT file.

For further information, **see** "GET Statement," "PUT Statement," and "Stream Input/Output."

Space

A space (or blank) character can be used to separate elements in a PL/I statement. You must use spaces to separate keywords and identifiers that are not separated by other delimiters. For example:

```
DECLARE A FIXED BINARY;
```

Spaces are required between the keyword DECLARE and the identifier A, between the identifier A and the keyword FIXED, and between the keywords FIXED and BINARY.

You can insert spaces preceding or following any other type of delimiter to improve the readability of the source text. For example:

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

None of the spaces in the above statement is required.

You cannot, however, insert spaces within identifiers, between two characters that function as one operator (for example >=), or in constants other than character-string constants.

SQRT Built-In Function

The SQRT built-in function returns a floating-point value that is the square root of an arithmetic expression x. The square root is computed in floating point. After its conversion to floating point, x must be greater than or equal to zero.

The format of the function is:

SQRT(x)

Statement

A statement is the basic element of a PL/I procedure. Statements are used to:

- Define and identify the structure of the program and the data that it acts upon
- Request specific action to be performed on data
- Control the flow of execution in a program

All PL/I statements are included in this manual under individual entries. The description of each statement gives its syntax, abbreviation, if any, and options.

Table S–1, at the end of this entry, provides a summary of PL/I statements.

■ Statement Formats

The general format of a PL/I statement consists of an optional statement label, the body of the statement, and a required terminator, the semicolon (;).

The body of the statement consists of user-specified identifiers, literal constants, or PL/I keywords. Each element must be properly separated, either by special characters that punctuate the statement or by spaces or comments.

■ Statement Labels

A label identifies a statement so that it can be referred to elsewhere in the program, for example, as the target of a GOTO statement. A label precedes a statement; it consists of any valid identifier terminated by a colon. Some examples are:

```
TARGET: A = A + B; READ_LOOP: READ FILE (TEXT) INTO (TEMP);
```

No statement can have more than one label.

For more information on labels and rules for specifying them, **see** "Label."

■ Simple Statements

A simple statement contains only one action to be performed. There are three types of simple statements:

- Keyword statements
- Assignment statements
- Null statements

Keyword Statements

Keyword statements are identified by the PL/I keyword that requests a specific action. Some examples of keyword statements are:

```
READ FILE (A) INTO (B);
GOTO LOOP;
DECLARE PRICE PICTURE'$$$99V.99';
```

In the above examples, READ, GOTO, and DECLARE are keywords that identify these statements to PL/I.

Assignment Statements

PL/I identifies an assignment statement by syntax: an assignment statement consists of two identifiers separated by an equals sign (=). Some examples are:

```
TOTAL = TOTAL + PRICE;
COUNTER = 0;
```

Null Statements

A null statement consists of only a semicolon; it indicates that PL/I is to perform no operation. Some examples are:

```
;

IF A < B THEN GOTO COMPUTE;
     ELSE ;
```

The IF statement above illustrates a common use of the null statement, that is, as the target of an ELSE clause.

■ Compound Statements

A compound statement contains more than one PL/I statement within the statement body; it is terminated by the semicolon that terminates the final statement. The PL/I compound statements are:

- IF statement
- ON statement

Some examples are:

```
ON ENDFILE (SYS$INPUT) GOTO FINISH;
IF (A + B) < (D + E) THEN C = A*D;
```

■ Begin Blocks and DO-Groups

A begin block is a group of statements begun by a BEGIN statement and ended by an END statement:

```
BEGIN; statement [...] END;
```

A begin block can generally be used wherever a single statement is valid — for example, as an ON-unit. Begin blocks can also define variables that are local, or internal, to the begin block. **See also** "Begin Block."

A DO-group is a group of statements begun by a DO statement and ended by an END statement. For example:

```
DO WHILE(A<B) statement [...] END;
```

DO-groups provide control over, or "conditionalize," the execution of statements in the group (whereas statements in a begin block are always executed when the BEGIN statement is executed).

If the DO statement has a WHILE option (a "DO WHILE" statement), the statements in the group are executed if and only if a specified expression is true. When the closing END statement is reached, the entire group of statements is reiterated if the WHILE expression is still true.

The DO statement can also have TO, BY, and REPEAT options that assign new values to a control variable on successive iterations. These options are used to reiterate the group a given number of times and to assign new values to variables used in the group's statements. For details, **see** "DO Statement" and "DO-Group."

■ Summary of Statements by Function

The PL/I statements can be grouped in the categories listed below.

Data Definition and Assignment Statements

The DECLARE statement defines variable names:

DECLARE identifier [attribute ...];

The assignment statement gives a value to a variable:

reference = expression;

Input/Output Statements

These statements identify files and data formats and perform input and output operations:

CLOSE	GET	READ
DELETE	OPEN	REWRITE
FORMAT	PUT	WRITE

Program Structure Statements

These statements define the organization of the program into procedures, blocks, and groups:

BEGIN	ENTRY
DO	PROCEDURE
END	null

Flow Control Statements

These statements change or interrupt the normal sequential flow of execution in a PL/I program:

CALL ON SIGNAL
GOTO RETURN STOP
IF REVERT

Storage Allocation Statements

These statements acquire and control the use of storage in a PL/I program:

ALLOCATE
FREE

Source File Modification Statements

These statements cause the PL/I compiler to include additional text in the source program at compile time or to change the values of constant identifiers at compile time:

%INCLUDE
%REPLACE

Table S-1: Summary of PL/I Statements

Statement	Use
Assignment	Evaluates an expression and gives its value to an identifier
null	Specifies no operation
ALLOCATE	Allocates storage for a based variable
BEGIN	Denotes the beginning of a block of statements to be executed as a unit
CALL	Transfers control to a subroutine or external procedure
CLOSE	Terminates association of a file control block with an input or output file
DECLARE	Defines the variable names and identifiers to be used in a PL/I program and specifies the data attributes associated with them
DELETE	Removes an existing record from a file
DO	Denotes the beginning of a group of statements to be executed as a unit
END	Denotes the end of a block or group of statements begun with a BEGIN, DO, or PROCEDURE statement
ENTRY	Specifies an alternate point at which a procedure can be invoked
FORMAT	Specifies the format of data that is being read or written with GET EDIT and PUT EDIT statements and defines the conversion, if any, to be performed
FREE	Releases storage of a based variable
GET	Obtains data from an external stream file or from a character-string expression
GOTO	Transfers control to a labeled statement

Table S-1 (Cont.): Summary of PL/I Statements

Statement	Use
IF	Tests an expression and establishes actions to be performed based on the result of the test
%INCLUDE	Copies the text of an external file into the source file at compilation time
ON	Establishes the action to be performed when a specified condition is signaled
OPEN	Establishes the association between a file control block and an external file
PROCEDURE	Specifies the point of invocation for a program, subroutine, or user-defined function
PUT	Transfers data to an external stream file or to a character-string variable
READ	Obtains a record from a file
%REPLACE	Assigns a constant value to an identifier at compile time
RETURN	Gives back control to the procedure from which the current procedure was invoked
REVERT	Cancels the effect of the most recently established ON-unit
REWRITE	Replaces a record in an existing file
SIGNAL	Causes a specific condition to be signaled
STOP	Halts the execution of the current program
WRITE	Copies data from the program to an external record file

STATIC Attribute

The STATIC attribute specifies the way that PL/I is to allocate storage for a variable. Static storage is allocated when an external procedure is loaded into memory and is not released until the procedure terminates.

The STATIC attribute is implied by the EXTERNAL attribute.

■ Restrictions

- The STATIC attribute directly conflicts with the BASED, parameter, and DEFINED attributes.
- The STATIC attribute cannot be applied to members of structures, parameters, or descriptions in an ENTRY or RETURNS attribute.

For more information on STATIC and on other storage-class attributes, **see** "Storage Classes."

STOP Statement

The STOP statement terminates execution of the program. The format of the STOP statement is:

```
STOP ;
```

The STOP statement terminates the program regardless of the current block activation. The STOP statement signals the FINISH condition and closes all open files. If the main procedure has the RETURNS attribute, no return value is obtainable.

Storage Classes

The storage class to which a variable belongs determines whether PL/I allocates storage for it at compile time or dynamically during the execution of the program. This entry summarizes the storage classes of variables in PL/I programs. For more information on the attributes that define the class to which a variable belongs, see the individual entries for the attributes. For more information on how the linker arranges variables in an executable image, **see** the *VAX-11 PL/I User's Guide.*

■ Automatic Storage

The default storage class attribute for PL/I variables is AUTOMATIC. PL/I does not allocate storage for an automatic variable until the block that declares the variable is activated. When the block is deactivated, the storage is released. For example:

```
CALC: BEGIN;
DECLARE TEMP FIXED BINARY (31);
   .
   .
END;
```

Each time the block labeled CALC is activated, storage is allocated for the variable TEMP. When the END statement is executed, the block is deactivated, and all storage for TEMP and all other automatic variables is released. The value of TEMP becomes undefined.

The storage requirements of an automatic variable are evaluated each time the block is activated. Thus, an extent may be specified as follows:

```
DECLARE STRING_LENGTH FIXED;
   .
   .
COPY: BEGIN;
DECLARE TEXT CHARACTER(STRING_LENGTH);
```

When this begin block is activated, the extent of TEXT is evaluated. The variable is allocated storage depending on the value of STRING_LENGTH, which must have a valid value.

■ Static Storage

A static variable is allocated storage when the program is activated, and it exists for the duration of the program. A variable has the static attribute if it is declared with any of the attributes STATIC, EXTERNAL, GLOBALDEF, or GLOBALREF.

Static arrays and strings must be declared with constant extents.

If a block that declares a static variable is entered more than once during the execution of the program, the value of the static variable remains valid. For example:

```
UNIQUE_ID: PROCEDURE RETURNS (FIXED BINARY(31));
DECLARE ID STATIC INTERNAL FIXED INITIAL (0);
        ID = ID + 1; /* Increment ID */
        RETURN (ID);
        END;
```

The function UNIQUE_ID declares the variable ID with the STATIC attribute and specifies an initial value of zero for it. The variable is initialized to this value when the program is activated. The storage for the variable is preserved, and the function returns a different integer value each time it is referenced.

A variable that has the STATIC attribute can also have external scope: that is, its definition and value can be accessed by any other procedure that declares it with the STATIC and EXTERNAL attributes. For more information, **see** "External Variable."

■ Based Variables

The BASED attribute defines a variable whose storage is accessed by means of a pointer. When you declare a based variable, you provide PL/I with a description of the data that will be accessed by the variable. The actual data must be referenced by a pointer that contains the address of the storage location of the data. For example:

```
DECLARE BUFFER CHARACTER(80) BASED (BUF_PTR),
        LINE CHARACTER(80),
        BUF_PTR POINTER;

BUF_PTR = ADDR(LINE);
```

The declaration of the variable BUFFER does not result in the allocation of any storage for the variable. Rather, PL/I associates the declaration of the variable with the pointer variable BUF_PTR. During the execution of the program, the value of the pointer variable is set to the location (address) in storage of the variable LINE. This effectively associates the description of the variable BUFFER with the actual data value of the variable LINE.

A based variable can be associated with a storage location using the ADDR built-in function, as in the preceding example; with the ALLOCATE statement; with a locator-qualified reference to the based variable; with the SET option of the READ statement; or by explicit allocation within an area. For more information on processing based variables, **see** the entries for those items or the entry "Based Variable."

■ Defined Variables

When you use the DEFINED attribute in the declaration of a variable, PL/I associates the description of the variable in the declaration with the storage allocated for the variable on which the declaration is defined. For example:

```
DECLARE NAMES(10) CHARACTER(5) DEFINED (LIST),
        LIST(10) CHARACTER(5);
```

In this example, the variable NAMES is a defined variable; its data description is mapped to the storage occupied by the variable LIST. Any reference to NAMES or to LIST is resolved to the same location in memory.

With certain defined variables, the POSITION attribute can be used to specify the position in the base variable at which the definition begins. For more information, **see** "Defined Variable."

■ Parameter Storage Class

A parameter variable is a variable that is declared in a procedure and that receives a value when the procedure is invoked. For example:

```
FUNC: PROCEDURE (X);
   DECLARE X FIXED BINARY;
```

In this example, X is implicitly declared a parameter variable because its name appears in the parameter list of the PROCEDURE statement. PL/I does not allocate storage for X, but rather uses storage associated with the actual argument specified when the procedure is invoked.

For more information on parameters, **see** "Parameters and Arguments."

Storage Sharing

Variables that have any of the attributes BASED, DEFINED, or parameter may share physical storage locations with one or more other variables.

A based variable is not allocated any storage when it is declared. Instead, storage is either located by a locator-qualified reference to the variable or allocated by the ALLOCATE statement. The BASED attribute thus allows you to describe the characteristics of a variable, which can then be located by a reference that qualifies the variable's name with any valid pointer value. Based variables are useful when the program must control the allocation of storage for several variables with identical attributes. The creation and processing of a queued, or linked, list is a common case. For full details on based variables and valid pointer values, **see** "Based Variable."

A defined variable uses the storage of a previously declared variable, which is referenced in the DEFINED attribute. The referenced variable is known as the base of the defined variable. The base can be a character- or bit-string variable, in which case the technique is called string overlay defining. When the base is a string variable, the POSITION attribute can also be specified for the defined variable, giving the position within the base variable's storage at which the overlay defining begins. Defined variables are useful when the program must refer to the same storage by different names. For full details, **see** "Defined Variable."

Parameters of a procedure share storage with their associated arguments. The associated argument is either a variable written in the argument list or a dummy variable allocated by the compiler. When the written argument is a variable, the sharing of storage by the parameter and argument allows a procedure to "return" values to the invoking procedure by changing the value of the parameter. For instance, a function can return values in this manner, in addition to the value specified in its RETURN statement. For details, **see** "Parameters and Arguments" and "Procedure."

STREAM Attribute

The STREAM file description attribute indicates that the file consists of ASCII characters and that it will be processed using GET and PUT statements.

The STREAM attribute is implied by the PRINT attribute. It is also supplied by default for a file that is implicitly opened with a GET or PUT statement.

Specify the STREAM attribute in a DECLARE statement for a file identifier or in the OPEN statement that opens the file.

■ Restrictions

The STREAM attribute directly conflicts with the RECORD, KEYED, DIRECT, SEQUENTIAL, and UPDATE attributes.

Stream Input/Output

Stream input/output, or stream I/O, is one of the two general kinds of I/O performed by PL/I (**see also** "Record Input/Output"). Stream input and output are performed by the statements GET and PUT, respectively. Both statements can perform either list-directed or edit-directed operations.

In record I/O, only one record of a file is processed by each READ or WRITE statement. In stream I/O, more than one record or line can be processed by a single statement, and, conversely, multiple statements can process a single line or record.

Successive GET statements acquire their input from the same line or record until all the characters in the line have been read, unless the program explicitly skips to the next line. When necessary, a single GET statement will read multiple lines to satisfy its input-target list. A single input data item may not cross a line unless it is a character string enclosed in apostrophes or unless the ENVIRONMENT option IGNORE_LINE_MARKS is in effect for the input file. This option produces stream input operations that concur exactly with standard PL/I. However, the option is usually not necessary; most programs produce the expected results without it. (For more information on ENVIRONMENT options, **see** the *VAX-11 PL/I User's Guide.*)

Successive PUT statements write their output to the same line or record until the line size is reached, or unless the program explicitly skips to a new line. A single PUT statement will write as many records as necessary to satisfy its output-source list. Any single data item that will not fit on the current line is split across lines.

This entry describes the following aspects of stream I/O:

- "Input by the GET Statement" gives a detailed description of the execution of GET statements. **See also** the entries "GET Statement" and "Terminal Input/Output."
- "Output by the PUT Statement" gives a detailed description of the execution of PUT statements. **See also** the entries "PUT Statement," "Print File," and "Terminal Input/Output."
- "Processing and Positioning of Stream Files" describes the characteristics and use of stream files with the GET and PUT statements.
- "Processing and Positioning of Character Strings" describes the characteristics and use of character-string expressions with GET STRING and PUT STRING statements.
- "Examples" gives general examples that use stream I/O statements. Examples are also given in the entries "GET Statement," "PUT Statement," "Terminal Input/Output," and in the entries for most format items.

■ Input by the GET Statement

When a GET statement is executed, the first action is to evaluate the FILE option, if there is one. For example, if the statement is:

```
GET FILE(INFILE) LIST(A);
```

then PL/I looks for an existing file referenced by INFILE. The following actions are taken:

- If INFILE is a reference to an existing file, and the file is not open, it is opened implicitly with the attributes STREAM and INPUT. Note that if INFILE is declared as a STREAM INPUT file but was not opened explicitly with the TITLE option, then INFILE is assumed to be a logical name defined by the user or, if no logical name was defined, an existing file named 'INFILE.DAT'.
- If INFILE is not associated with a file, or if the associated file does not exist, or if for any reason the associated file cannot be opened, the UNDEFINEDFILE condition is signaled.

If the statement has a STRING option instead of a FILE option, the reference in the STRING option is evaluated.

If the statement has neither a FILE option nor a STRING option, it is taken to refer to the default file constant SYSIN. SYSIN is declared by default with the STREAM INPUT attributes, and it is normally used for input from a terminal. **See also** "Terminal Input/Output."

If the input stream is a file, the next action is to execute the SKIP option, if there is one. For details, **see** "Processing and Positioning of Stream Files" below, or the entry "GET Statement." The SKIP option cannot be used with the STRING option. Note that a GET statement can perform a SKIP operation even if it performs no data input. For example, the statement:

```
GET FILE(INFILE) SKIP(2);
```

repositions the file referenced by INFILE to the second line following the current line in the file.

A GET statement that has the EDIT or LIST option performs input from the stream to a list of input targets, which must be variables of computational data types. If the input target is an aggregate variable, then input is assigned to each element of the aggregate; input values are assigned to array elements in row-major order and to structure members in the order of their declaration. An input target can also contain a DO construct that further controls the assignment; for details, **see** "GET Statement." Since a stream consists only of ASCII characters, and the input targets are not necessarily character-string variables, an input field must be selected from the input stream for each target and must be converted, if necessary, to the type of the target.

In edit-directed (GET EDIT) statements, the selection and assignment of the input field are controlled by a format item that corresponds to the input target. In the default case, which applies to terminal input and to input from most stream files, a data format item assumes that the end of the input field has occurred if it encounters the end of a record in an input file or the end of a line when the input is from a terminal.

For example, a common technique for reading lines of varying length from a terminal is to deliberately use a format item that specifies a field wider than the column width of the terminal. An example is shown in the entry "X Format Item." If a carriage return is typed in response to an input request for GET EDIT, or if the end of a record is immediately encountered, the requested field width is filled with spaces and assigned to the input target under the control of the corresponding format item. (Note that all spaces will cause an error for B format items.) However, if the input stream is a character-string expression (GET STRING), the ERROR condition is signaled if the format item causes the end of the input string to be reached in the middle of an input field. If the input stream is a file declared or opened with ENVIRONMENT (IGNORE_LINE_MARKS), the search for characters to complete the input field simply continues at the next record.

Details on the matching of format items to input targets are given in the entry "Format-Specfication List — How Edit-Directed Operations Are Performed." The execution of individual format items is described in individual entries — **see,** for example, "F Format Item." IGNORE_LINE_MARKS, and other ENVIRONMENT options, are described in the *VAX-11 PL/I User's Guide.*

In list-directed (GET LIST) statements, an input field is acquired by examining the input stream to find the next character that is not a space character. The following actions are taken depending on the next character that is found:

- If the next nonspace character is an apostrophe, the input field is assumed to contain a bit-string or character-string constant, in the same format as used to write a string constant in a program. The constant is acquired and may span the end of a record or line. However, the ERROR condition is signaled if the end of the file is reached before the terminating apostrophe is found; if the input stream is a character-string expression rather than a file, the ERROR condition is signaled if the end of the string is reached. The apostrophes and B suffix are removed from the constant, and any double apostrophe within a character-string constant is changed to a single apostrophe. (If the field contains a bit-string constant in base 4, octal, or hexadecimal radix, its binary equivalent is found.) The resulting character- or bit-string value is then assigned to the corresponding input target. If the input

target is not of the same data type, the input value is converted according to the usual rules (**see** "Conversion of Data").

- If the next nonspace character is a comma, and the previous operation on the input file was by GET LIST, and the previous input field was terminated by a space, carriage return, or end-of-record, the scan continues. If the next nonspace character is a comma, and the previous nonspace character was also a comma, the corresponding input target is skipped; the input target retains whatever value it had before the GET LIST statement.
- If the input line or record is empty (that is, a carriage return or end-of-record is encountered immediately after the beginning of a line), the corresponding input target is nulled. That is, the null character string '' is assigned to the input target with appropriate type conversion. However, if the input file was opened with ENVIRONMENT(IGNORE_LINE_MARKS), the carriage return or end-of-record is ignored.
- Otherwise, the next nonspace character is neither a comma nor an apostrophe. The input field is then assumed to begin with this character and to be terminated by the next space, comma, carriage return or end-of-record [if ENVIRONMENT(IGNORE_LINE_MARKS) was not used], end-of-file (if the input stream is a file), or end-of-string (if the input stream is a character string). All the characters in the field are acquired and assigned, with appropriate type conversion, to the input target.

If the GET LIST statement attempts to read a file after its last input field has been read, or if it attempts to read an empty file, the ENDFILE condition is signaled. If the GET LIST statement attempts to read a character string after its last field has been read, or if it attempts to read a null string, the ERROR condition is signaled.

■ Output by the PUT Statement

When a PUT statement is executed, the first action is to evaluate the FILE or STRING option, if there is one. If the statement has a file option, the referenced file is either opened or created with the attributes STREAM and OUTPUT, if it is not already open. If the file referenced in a statement such as:

```
PUT FILE(OUTFILE) LIST(A);
```

was not previously declared or opened with the TITLE option, the reference (here, OUTFILE) is assumed to be a logical name defined by the user or, if no logical name is defined, an existing file named 'OUTFILE.DAT'. If a STRING option is present instead, the referenced character-string variable is assigned the null character string.

If neither the FILE option nor STRING option is present, the output stream is assumed to be the default file SYSPRINT.

If the output stream is a file, the next action is to execute any of the options PAGE, LINE, and SKIP that occur in the statement, in that order. The output stream must be a file if any of these options are included, and it must be a print file if LINE or PAGE is included. Note that a PUT statement can

contain one or more of these options even if it performs no data output. For example, the statement:

```
PUT FILE(OUT) PAGE LINE(20);
```

skips to a new page in the file referenced by OUT (which must be a print file), moves to line 20 of the file, and then terminates.

If, however, the statement also has a LIST or EDIT option, it then writes out a list of output sources, which must be variables, constants, or other expressions of computational data types. If the output source is a reference to an aggregate variable, all the variable's elements are written out; array elements are written out in row-major order, and structure members are written out in the order of their declaration. (For more information on output sources, **see** "PUT Statement.") Since a stream consists only of ASCII characters, each output source is converted to a character string before being written out, as follows:

- If the PUT statement is list directed, the output source is converted according to the usual rules for converting a computational value to a character string (**see** "Conversion of Data").
- If the PUT statement is edit directed, the output source is converted as specified by a corresponding format item. For details, **see** the entries for individual format items or "Format Items and Their Uses."
- If the output stream is a character-string variable or file with the attributes STREAM OUTPUT (but not PRINT), the statement is list directed, and the output source is of type CHARACTER, the output source value is surrounded by apostrophes, and any apostrophe within the value is replaced by a double apostrophe.
- If the output source is of type BIT, and the statement is list directed, the converted output source is surrounded by apostrophes, and the letter 'B' is appended.

The converted output source is then written to the output stream, as follows:

- If the statement is list directed and the output stream is a file with the attributes STREAM OUTPUT (but not PRINT), then the converted output source is written beginning at the end of the file and followed by a single space. If the output stream is a print file, the converted output source is written out beginning at the end of the file, after enough spaces have been written out to move to the next tab stop. In either case, if the converted output source does not fit on the remainder of the current line, as much as possible is written on the current line, and the rest is written on the next line. The ENDPAGE condition may be signaled if the output stream is a print file. For more information on print files, **see** "Print File."
- If the statement is edit directed, the exact number of characters specified by the format item is written out, and no space follows. As much output as possible is written on the remainder of the current line, and it is continued, if necessary, on the next line. Any additional positioning, such as on tab stops in a print file, is performed by control format items (**see** "Format Items and Their Uses").

- If the output stream is a character-string variable, the output process is identical to that with a STREAM OUTPUT file except that the first output source written out by a PUT statement is placed at the beginning of the variable's storage, and any previous value in the variable is erased. Note that the X format item, which can be used with PUT STRING, performs positioning by writing out spaces, not by "skipping" characters in the previous value of the variable. Note also that list-directed output to a character variable, followed by list-directed output of the variable itself, can result in a proliferation of apostrophes in the value finally written to a file (**see** "Examples," below).

■ Processing and Positioning of Stream Files

This section discusses the processing and positioning of the stream when the stream is a file. A stream file is a file of ASCII text, divided into lines. For every stream file used in a program, PL/I maintains the following information:

- The locations of the beginning and end of the file. On input operations, the ENDFILE condition is signaled on the first attempt to read past the end of the file.
- For output files, the maximum number of ASCII characters in a line, or the line size. The line size is either a default value or the specific value you have established for the file (**see** "LINESIZE Option"). The line size is used to determine when to skip to the next line (for example, **see** "X Format Item — Output with PUT EDIT"). On input, a single data item cannot cross a line unless it is a character string enclosed in apostrophes or unless the file was opened with ENVIRONMENT(IGNORE_LINE_MARKS). On output, data items are continued on the next line.
- The current position in the file. Essentially, this is the point in the file at which the last input or output operation stopped. It is the exact character position (sometimes in the middle of a line) at which the next output item is written or from which the next input item is read.

Input operations can begin at any position from the current position onward. The default is the current position. To acquire data from a different position, you can:

- Use the SKIP option of the GET statement to advance by a specified number of lines before reading data.
- Use control format items to move to a specified position before reading data. With the GET statement, control format items are restricted to SKIP (same operation as the SKIP option), COLUMN (advance to a specified character position), and X (advance by a specified number of character positions from the current position). Note that the control format items, unlike the SKIP option, are executed during, not before, the input of data. **See also** "Format Items and Their Uses." The control format items can signal the ENDFILE and ERROR conditions if the end-of-file is encountered.
- Close and then reopen the file, which sets the current position to the first character in the file.

Because stream files are sequential files, output operations always place data at the end of the file. You can do the following additional formatting of output with any stream output file:

- Use the SKIP option of the PUT statement to skip lines following the current position. If the current position is the beginning of a line, the SKIP option inserts null lines in the file between the current position and the position of the next output. The SKIP option can reposition the file even though no data is output.
- Use the control format items to advance to a specified line or character position, or to a new page. The control format items are COLUMN (move to a specified character position), SKIP (same effect as the SKIP option), and X (skip a specified number of characters following the current position). As with the input case, control format items are executed only during the output of data; if only part of the format list is used, the excess control format items are ignored.

If the output file is a print file (that is, has the attributes STREAM, OUTPUT, and PRINT or is the default file SYSPRINT), the following additional information is maintained for the file:

- The current page number. The first output to a print file is written to page 1. The current page number is incremented by the PAGE option, the PAGE format item, and, in some circumstances, by the LINE option and LINE format item. The current page number can be evaluated for a specified print file with the PAGENO built-in function. It can also be set to a new value by assigning a value to the PAGENO pseudovariable.
- The page size. This is an integer that specifies the number of lines on a page. The page size is either the default value or the specific number that you have established for the print file (**see** "PAGESIZE Option"). When the last line on a page is filled, the first attempt to write (or position the file) beyond that position signals the ENDPAGE condition. The ENDPAGE condition is signaled only on the first such attempt; if no ON-unit is established for the condition, a PUT PAGE is executed. For example, the ON-unit for the ENDPAGE condition can write a trailer at the bottom of the current page, or a header at the top of the next page, before printing a new page of data.
- The current line number. This is an integer specifying the line currently being used for output, relative to the top of the page. The first line on the page is line 1. The LINENO built-in function can evaluate the current line of a specified print file. The LINE option of the PUT statement, and the LINE format item, can reposition the file to a specified line.
- Position of tab stops. Tab stops always occur at eight-column increments on every line of a print file, beginning with column 1. The TAB format item can reposition a print file to a specified tab stop relative to the current position.

Terminals should always be declared as print files when used for output. **See** "Terminal Input/Output."

■ Processing and Positioning of Character Strings

If the input or output stream is a character string, the processing is similar to the processing of files, but the positioning options are more limited:

- Input can begin at either the beginning of the string or at a specified character position. The ERROR condition is signaled if the end of the string is encountered. Only the X format item can be used for positioning.
- The first output by a PUT statement always occurs at the beginning of the string, and subsequent output by the same statement follows the previous output. The ERROR condition is signaled if the maximum length of the string is exceeded. Only the X format item can be used for positioning.

On input, the value of the character-string expression specified in the STRING option must include commas or spaces to separate input fields, as with any stream input. For an example, **see** "GET Statement — GET LIST."

■ Examples

```
LOI: PROCEDURE OPTIONS(MAIN);

DECLARE (I,J) FIXED BINARY;

GET LIST(I);
GET LIST(J);

PUT SKIP LIST('I=',I);
PUT LIST('J=',J);

END LOI;
```

The input data for the two GET statements may appear on the same line:

```
3,4(RET)
```

Because the first PUT statement contains a SKIP option, the output begins on a new line. The second PUT statement does not contain a SKIP option, so the output appears on the same line as that of the first statement:

```
I=          3 J=      4
```

For another example showing terminal input and output, **see** "Terminal Input/Output."

```
PUTSTR: PROCEDURE OPTIONS(MAIN);

DECLARE SOURCE CHARACTER(80) VARYING;

DECLARE OUTFILP PRINT FILE;

SOURCE = 'Old string';

PUT FILE(OUTFILP) LIST(SOURCE);

PUT FILE(OUTFILP) EDIT(SOURCE) (A);

PUT STRING(SOURCE) LIST('New string');
```

```
PUT FILE(OUTFILP) LIST(SOURCE);

PUT FILE(OUTFILP) EDIT(SOURCE) (A);

END PUTSTR;
```

The program PUTSTR writes the following output to the print file OUTFILP.DAT:

```
Old string Old string   'New string'  'New string'
```

The last two strings are surrounded by apostrophes because the apostrophes were added by the PUT STRING statement.

```
PUTSTR: PROCEDURE OPTIONS(MAIN);

DECLARE SOURCE CHARACTER(80) VARYING;

DECLARE OUTFILS STREAM OUTPUT FILE;

SOURCE = 'Old string';

PUT FILE(OUTFILS) LIST(SOURCE);

PUT FILE(OUTFILS) EDIT(SOURCE) (X,A);

PUT STRING(SOURCE) LIST('New string');

PUT FILE(OUTFILS) LIST(SOURCE);

PUT FILE(OUTFILS) EDIT(SOURCE) (X,A);

END PUTSTR;
```

This version of PUTSTR writes the following output to the stream file OUTFILS.DAT:

```
'Old△string'△△Old△string''New△string''△'△△'New△string'△
```

Here, every PUT LIST has added a new pair of apostrophes to the output value. First, the characters "Old string" are assigned to SOURCE. When SOURCE is written out with PUT LIST, the characters are surrounded by apostrophes (because OUTFILS is not a print file) and written out followed by a space:

```
'Old string'△
```

The following PUT EDIT statement writes out a space (because of the X format item) followed by the exact characters in SOURCE:

```
△Old string
```

Then, the PUT STRING statement writes the characters "New string" to SOURCE; here, SOURCE behaves like a stream output file, and the resulting value in SOURCE is:

```
'New string'△
```

Now, when SOURCE is written out by another PUT LIST statement, every apostrophe in SOURCE's value is replaced by two apostrophes and the resulting value is again surrounded by apostrophes and written out followed by a space:

```
''''New string''△'△
```

When, instead, SOURCE is written out by PUT EDIT, no additional apostrophes are added, and the output is:

```
△'New string'△
```

where the initial space was created by the X format item, and the terminating space was already in the value of SOURCE.

STRING Built-In Function

The STRING built-in function concatenates the elements of an array or structure and returns the result. Elements of a string array are concatenated in row-major order. Members of a structure are concatenated in the order in which they were declared.

The format of the STRING built-in function is:

STRING(reference)

reference

A reference to a variable that is suitable for bit-string or character-string overlay defining. Briefly, a variable is suitable if it consists entirely of characters or bits, and these characters or bits are packed into adjacent storage locations, without gaps. (For a precise definition, see "Defined Variable.")

■ Returned Value

The string returned is of type CHARACTER or BIT, depending on whether the reference is suitable for character- or bit-string overlay defining. The length of the string is the total number of characters or bits in the base reference.

■ Examples

```
STRING_BIF_EXAMPLE: PROCEDURE;
DECLARE NEW_NAME CHARACTER(40);
DECLARE 1 FULL_NAME,
        2 FIRST_NAME CHARACTER(10),
        2 MIDDLE_INITIAL CHARACTER(3),
        2 LAST_NAME CHARACTER(27) ;
FIRST_NAME = 'MABEL';
MIDDLE_INITIAL = 'S.';
LAST_NAME = 'MERCER';
NEW_NAME = STRING(FULL_NAME);
      /* NEW_NAME =
   'MABEL△△△△△S.△MERCER△△△△△△△△△△△△△△△△△△△△'
      where △ is a space       */
END STRING_BIF_EXAMPLE;
```

String Handling

VAX-11 PL/I provides the following facilities for handling strings. Each is described in its own entry in this manual.

- The concatenation operator (¦ ¦ or !!), which concatenates two strings
- The bit-string operators AND (&) and OR (¦ or !), which perform logical operations on two bit-string operands
- The bit-string operator NOT (^), which complements the bits in the string
- The built-in functions
  - BIT, which converts an expression to a bit string
  - BOOL, which specifies a "truth table" to be used in comparing two bit strings and returns the resulting bit string
  - BYTE, which returns the ASCII character corresponding to a given integer code
  - CHARACTER, which converts an expression to a character string
  - COLLATE, which returns a string of the ASCII characters in collating sequence
  - COPY, which replicates a bit or character string and concatenates the replications into a single string
  - DATE, which returns a character string giving the date
  - INDEX, which returns the position at which a specified substring is found in a specified bit or character string
  - LENGTH, which returns the current length of a bit or character string
  - RANK, which returns the ASCII code for a given character
  - STRING, which concatenates an array or structure of strings into a single string
  - SUBSTR, which returns a specified portion of a bit or character string
  - TIME, which returns a character string giving the current time of day
  - TRANSLATE, which replaces occurrences of a specified character with a new character
  - UNSPEC, which returns, as a bit string, the internally coded form of a scalar expression
  - VERIFY, which compares two character strings and returns the position of a mismatched character
- Character and bit-string assignments, such as NAME = 'HAROLD', STATUS = '0001011'B
- Character- and bit-string relational expressions, such as
  IF 'ARTHUR' < 'HAROLD' THEN . . .
- The GET STRING and PUT STRING statements, for transferring data between character strings and program variables

- The STRING pseudovariable, which assigns parts of a string to an array or structure
- The SUBSTR pseudovariable, which replaces a specified substring with a specified character-string expression

STRING Option

The STRING option is used with the GET and PUT statements to perform data transfers from or to a character-string variable in the program instead of an external file.

The STRING option can be used with either the LIST option or the EDIT option, depending on whether type conversions are to be automatic or under program control.

In most respects, stream I/O to a character-string expression is performed as if the string were a file with the attributes STREAM and, as appropriate, INPUT or OUTPUT.

The GET STRING statement acquires a string from a character-string variable and assigns it to one or more input targets. If more than one input target is listed, the characters in the string should include any punctuation (comma or space separators, apostrophes) that would be required if the character string were in an external file.

The PUT STRING statement evaluates a list of output sources (expressions), converts the results to characters if necessary, and assigns the concatenated results to a character-string variable declared in the program. The concatenated results include any punctuation (space separators, apostrophes) that would result if the character string were being sent to a STREAM OUTPUT file. For example, apostrophes are added to character-string output, and every output value is followed by a space.

For further details, **see** "GET Statement" and "PUT Statement."

STRING Pseudovariable

The STRING pseudovariable interprets a suitable reference as a reference to a fixed-length string. By using it, you can modify an entire aggregate with a single string assignment or assign it to a pictured variable as if it were a character-string variable. The format of the pseudovariable (in an assignment statement) is:

```
STRING(reference) = expression;
```

reference

A reference to a variable that is suitable for character-string (or bit-string) overlay defining. The length of the pseudovariable is equal to the total number of characters (or bits) in the scalar or aggregate denoted by the reference. This length must be less than or equal to the maximum length for character-string (or bit-string) data.

Assignment to the STRING pseudovariable modifies the entire storage denoted by the reference.

■ Examples

```
STRING_PSD_EXAMPLE: PROCEDURE;
DECLARE 1 NAME,
        2 FIRST CHARACTER(10),
        2 MIDDLE_INITIAL CHARACTER(3),
        2 LAST CHARACTER(10);
STRING(NAME)='FRANKLIN D. ROOSEVELT';
      /* NAME.FIRST = 'FRANKLIN D';
         NAME.MIDDLE_INITIAL = '. R';
         NAME.LAST = 'OOSEVELT '; */
END STRING_PSD_EXAMPLE;
      .
      .
DECLARE 1 FLAGS,
      2 (A,B,C) BIT(1);
STRING(FLAGS) = '0'B; /* sets all three flags false */
      .
      .
DECLARE P PICTURE 'Z.ZZZV,ZZDB';
GET EDIT (STRING(P)) (A(10));
      /* assigns 10 characters from SYSIN to P,
      without conversion */
```

Structure

A structure is a data aggregate consisting of one or more members. The members may be scalar data items, arrays of scalar data items, structures, or arrays of structures, and different members may have different data types.

A structure declaration defines a structure variable by means of level numbers. For example:

```
DECLARE 1 TRANSACTION,
          2 PART_NUMBER,
            3 FACTORY CHARACTER (3),
            3 ITEM CHARACTER (5);
```

The level number 1 indicates that TRANSACTION is a structure variable. TRANSACTION is the name of the entire, or "major," structure. The higher numbers 2 and 3 indicate that the associated identifiers are the names of members of the structure TRANSACTION or its "minor" structure, PART_NUMBER.

The following sections define the rules for specifying level numbers and attributes for members in a structure.

■ Level Numbers for Structures

You must precede each variable in the structure declaration with a level number, indicating the position of the variable in the structure. The following rules apply:

- The level number of the major structure must be 1.
- Level numbers must be specified using decimal integer constants.

- A level number must be separated from its associated variable name by at least one space or tab character.
- Level numbers after level 1 can be any integer values, as long as each level number is equal to or greater than the level number of the preceding level. (There can be only one level 1.)
- Each identifier in the structure must be separated from the declaration of the previous identifier by a comma.
- Substructures at the same logical level of nesting do not have to have the same level number.
- The deepest possible logical level is 15.
- The largest possible level number constant is 32767.
- A substructure at level n contains all following items declared with level numbers greater than n, up to but not including the next item declared with a level number less than or equal to n.

■ Attributes for Structure Variables

Within a structure, only members at the lowest level of each substructure can be declared with data type attributes. Additional rules for specifying attributes for the various components of a structure are listed below.

- Only the following attributes are valid for the major structure name:

AUTOMATIC	EXTERNAL
BASED	INTERNAL
DEFINED	STATIC

- The major structure, or a minor structure, or any member of the structure can be dimensioned: that is, there can be arrays of structures and structures whose members are arrays. **See** "Arrays of Structures."
- Member names cannot have any of the following attributes:

AUTOMATIC	GLOBALREF
BASED	READONLY
DEFINED	STATIC
EXTERNAL	VALUE
GLOBALDEF	

- If a structure has the STATIC attribute, the extents of all members (that is, lengths for character- and bit-string variables, dimensions for array variables, and area extents) must be specified using optionally signed decimal integer constants.

■ Structure-Qualified References

To refer to a structure in a program, you use the major structure name, minor structure names, and individual member names. Member names need not be unique even within the same structure. To refer to names of members or minor structures, you must ensure only that the reference uniquely identifies the minor structure name or member. You can qualify the variable name by preceding it with the name(s) of higher-level variable(s) in the structure; names in this format, called a qualified reference, must be separated by periods (.).

The following sample structure definition illustrates the rules for identifying names of variables within structures:

```
DECLARE 1 STATE,
          2 NAME CHARACTER (20),
          2 POPULATION FIXED (10),
          2 CAPITAL,
            3 NAME CHARACTER (30),
            3 POPULATION FIXED (10,0),
          2 SYMBOLS,
            3 FLOWER CHARACTER (20),
            3 BIRD CHARACTER (20);
```

The rules for selecting and specifying variable names for structures are as follows:

- The name of the major structure is subject to the rules for the scope of variables in a program.
- The name of any minor structure or member in a structure can be qualified by the names of higher-level members in the structure. The variable names must be specified from left to right in order of increasing level numbers, separated by periods. The members of the sample structure, completely qualified, are:

  STATE.NAME
  STATE.POPULATION
  STATE.CAPITAL.POPULATION
  STATE.CAPITAL.NAME
  STATE.SYMBOLS.FLOWER
  STATE.SYMBOLS.BIRD

- Names of minor structures or members within structures do not have to be qualified if they are unique within the scope of the name. The following names in the sample structure can be referred to without qualification (as long as there are no other variables with these names):

  CAPITAL
  SYMBOLS
  FLOWER
  BIRD

- Intermediate qualification names can be omitted if the reference remains unambiguous. The following references to members in the sample structure are valid:

  STATE.FLOWER
  STATE.BIRD

If a name is ambiguous, the compiler cannot resolve the reference and issues a message. In the example, the names POPULATION and NAME are ambiguous.

■ Initializing Structures

A structure can be initialized by giving the INITIAL attribute to its members. Not all members need be initialized. For example:

```
DECLARE 1 COUNTS,
          2 FIRST FIXED BIN(15) INITIAL(0),
          2 SECOND FIXED BIN(15),
          2 THIRD (5) FIXED BIN(15) INITIAL (5(1));
```

The first and third members of the structure COUNTS are initialized.

The INITIAL attribute cannot be applied, however, to a major or a minor structure name.

■ Using Structure Variables in Expressions

You can specify the name of a major or minor structure in an assignment statement only if the source expression and the target variable are identical in size and structure and all corresponding members have the same data types.

■ Passing Structure Variables as Arguments

A structure variable can be passed as an argument to another procedure. The relative structuring of the structure variable specified as the argument and the corresponding parameter must be the same. The level numbers do not have to be identical. The following example shows the parameter descriptor for a structure variable:

```
DECLARE SEND_REC ENTRY (1,
                        2 FIXED BINARY(31),
                        2 CHARACTER(40),
                        2 PICTURE '999V99');
```

The written argument in the invocation of the external procedure SEND_REC must have the same structure and its corresponding members must have the same data types.

When structures are passed as arguments, they must match the corresponding parameters. They cannot be passed by dummy argument. For information on arguments and argument passing, **see** "Parameters and Arguments."

Subroutine

A subroutine is a procedure that is invoked by another procedure by means of a CALL statement. The subroutine may be internal or external to the procedure that calls it. **See** "Procedure."

SUBSTR Built-In Function

The SUBSTR built-in function returns a specified substring from a string. Its format is:

SUBSTR(string,position[,length])

string

A bit- or character-string expression.

position

An integer expression that indicates the position of the first bit or character in the substring. The position must be greater than or equal to one and less than or equal to LENGTH(string) + 1.

length

An integer expression that indicates the length of the substring to be extracted. If not specified, length is:

LENGTH(string)–position+1

which extracts the substring beginning at the indicated position and ending at the end of the string.

The length must satisfy the condition:

$0 \leq \text{length} \leq \text{LENGTH(string)} - \text{position} + 1$

■ Returned Value

The returned substring is of type BIT (length) or CHARACTER(length, depending on the type of the string argument. If the length argument is zero, the result is a null string.

■ Examples

```
DECLARE (NAME,LAST_NAME) CHARACTER(20),
        START FIXED BINARY(31);

NAME = 'ISAK DINESEN';
        /* NAME =
'ISAK△DINESEN△△△△△△△△' */

START = INDEX(NAME,' ')+1;
        /* START = 6 */

LAST_NAME = SUBSTR(NAME,START);
        /* default length = LENGTH(NAME)-START+1 =15 */
        /* LAST_NAME = 'DINESEN△△△△△△△△△△△△△△' */
```

SUBSTR Pseudovariable

The SUBSTR pseudovariable refers to a substring of a specified string variable reference. (**See also** "Pseudovariable" for general rules.) Assignment to the pseudovariable modifies only the substring. The format of the pseudovariable (in an assignment statement) is:

SUBSTR(reference,position[,length]) = expression;

reference

A reference to a bit- or character-string variable. If the reference is to a varying-length character string, the substring defined by the position and length arguments must be within the current value of the string. Assignment to the SUBSTR pseudovariable does not change the length of a varying string.

position

An integer expression indicating the position of the first bit or character in the substring. The position must be greater than or equal to LENGTH(reference)+1.

length

An integer expression that indicates the length of the substring. If not specified, length has the value:

length = LENGTH(reference)–position+1

which specifies the substring beginning at the indicated position and ending at the end of the string. The length must satisfy the condition:

$0 \leq \text{length} \leq \text{LENGTH(reference)} - \text{position} + 1$

Note that

```
SUBSTR(r,p,1) = v;
```

is equivalent to

```
r = SUBSTR(r,1,p-1)||v||SUBSTR(r,p+1);
```

■ Examples

```
DECLARE (NAME,NEW_NAME) CHARACTER(20) VARYING;

NAME = 'ISAK DINESEN';
NEW_NAME = NAME;
SUBSTR(NEW_NAME,4) = 'AC NEWTON' ;
/* NEW_NAME = 'ISAAC△NEWTON' */
```

Subtraction

The minus sign character (–) indicates a subtraction operation in an expression; the result is the difference between the operands. Both operands must be arithmetic or picture data.

■ Conversion of Operands

If both operands have the same base, precision, and scale, so has the result of the operation. The PL/I compiler converts operands of different data types as follows:

- If one operand has the FLOAT attribute and the other has the FIXED attribute, the fixed-point operand is converted to floating point before the operation.
- If one operand is FIXED DECIMAL and the other is FIXED BINARY, the fixed-point binary operand is converted to fixed-point decimal. However, the compiler issues a warning message to that effect.

The precision of the values resulting from conversion of operands is described under "Expression."

■ Precision of the Result

Floating-Point Operands

The result has the maximum of the converted precisions of the operands.

Fixed-Point Decimal Operands

If (p,q) and (r,s) represent the converted precisions and scale factors of the two operands, the resulting precision and scale factor are:

precision: min(31,max(p–q,r–s)+max(q,s)+1)

scale factor: max(q,s)

Fixed-Point Binary

If (p) and (r) represent the converted precisions of the two operands, the resulting precision is:

min(31,max(p,r)+1))

SYSIN Default File

SYSIN is the default input file for GET statements. SYSIN is normally associated with a user's default input device (SYS$INPUT). For example:

```
GET LIST (A,B,C);
```

This GET statement does not include the FILE option. Thus, when the program containing this line is executed, this statement reads data from the file SYSIN.

For more information, **see** "GET Statement" and "Terminal Input/Output." For information on the relationship between the PL/I file SYSIN and the default input device, **see** the *VAX-11 PL/I User's Guide*.

SYSPRINT Default File

SYSPRINT is the default output file for PUT statements. Unless it is explicitly declared with other attributes, SYSPRINT has the attributes STREAM OUTPUT PRINT. (If you declare an external file constant named SYSPRINT with the STREAM and OUTPUT attributes, PRINT is added by the compiler.) SYSPRINT is normally associated with a user's default output device (SYS$OUTPUT). For example:

```
PUT LIST (A,B,C);
```

This PUT statement does not include the FILE option. Thus, when the program containing this line is executed, this statement writes data to the file SYSPRINT.

For more information, **see** "PUT Statement" and "Terminal Input/Output." For information on the relationship between the PL/I file SYSPRINT and the default output device, **see** the *VAX-11 PL/I User's Guide*.

T

TAB Format Item

The TAB format item sets a print file to a specified tab stop. It can be used only for output to print files. Within a line, tab stops always occur at columns (n*8)+1, where n equals 0, 1, 2, ... (That is, at columns 1, 9, 17, ...) The form of the TAB format item is:

TAB [(w)]

w

An integer that identifies the wth tab stop from the current position; w must not be negative. If w is zero, no operation is performed. If w is omitted, a value of one is assumed.

When the TAB format item is executed, the current column, cc, is determined. If the current position is the beginning of a line, page, or file, then cc = 0. Otherwise, cc is the column in the current line at which the next output character would appear. If, for example, seven symbols have already been written on a line, then the next output would appear at column 8, which is the current column. The file is then repositioned in one of the following ways:

- If there are at least w tab stops between cc+1 and the end of the line, then the file is moved to the wth tab stop from the current column, and the intervening positions are filled with spaces. The end of the line is at one column after the current line size, which is either the default value or the specific value that you have established for the file (**see** "LINESIZE Option").
- Otherwise (if there are fewer than w tab stops on the remainder of the current line), the file is skipped to the beginning of the next line and positioned at the first tab stop (column 1). If, before the skip operation, the current line was the last line on the page, the ENDPAGE condition is signaled, and the current line becomes (page size)+1. The page size is either the default value or the specific value that you have established for the file (**see** "PAGESIZE Option").

■ Examples

```
TAB: PROCEDURE OPTIONS(MAIN);

DECLARE OUT STREAM OUTPUT PRINT FILE;

OPEN FILE(OUT) LINESIZE(60);
```

```
PUT FILE(OUT) SKIP
              EDIT('12345678901234567890123456789O') (A);
PUT FILE(OUT) SKIP EDIT('COL1','?') (A,TAB(2),A);
PUT FILE(OUT) EDIT('!') (TAB(20),A);
PUT FILE(OUT) SKIP EDIT('*') (TAB(1),A);
PUT FILE(OUT) EDIT('abcdefg') (A); /* cc now = 17 */
PUT FILE(OUT) EDIT('&') (TAB(6),A);

END TAB;
```

The program TAB writes the following output to the print file OUT.DAT:

```
12345678901234567890123456789O
COL1            ?
!
        *abcdefg
&
```

The question mark appears in column 17, which is the second tab stop following the string 'COL1'. The exclamation point appears in column 1 of the next line because there were fewer than 20 tab stops on the remainder of the line. In the third PUT EDIT statement, the SKIP option first resets the current column to zero. When the TAB format item is executed, it must position the file to the first tab stop that is between column 1 (cc+1) and the end of the line; therefore, the file is positioned, and the asterisk appears, in column 9. Similarly, the fourth statement writes out the string 'abcdefg', after which the current column is 17, a tab stop. Since the line size has been established as 60, there are only five tab stops between cc+1 and the end of the line: 25, 33, 41, 49, and 57. Therefore, the format item TAB(6) in the last PUT EDIT statement causes a skip to the next line, and the ampersand appears in column 1.

TAN Built-In Function

The TAN built-in function returns a floating-point value that is the tangent of an arithmetic expression x, where x represents an angle in radians. The tangent is computed in floating point. After its conversion to floating point, x must not be an odd multiple of $\pi/2$.

The format of the function is:

TAN(x)

TAND Built-In Function

The TAND built-in function returns a floating-point value that is the tangent of an arithmetic expression x, where x represents an angle in degrees. The tangent is computed in floating point. After its conversion to floating point, x must not be an odd multiple of 90.

The format of the function is:

TAND(x)

TANH Built-In Function

The TANH built-in function returns a floating-point value that is the hyperbolic tangent of an arithmetic expression x. The hyperbolic tangent is computed in floating point. The format of the function is:

```
TANH(x)
```

Terminal Input/Output

This entry describes the PL/I features that perform input and output to a user's terminal.

In this discussion, and in most applications, the terminal is treated as a stream file. You can explicitly declare a stream file to be associated with a user's terminal. In addition, the stream input and output statements, GET and PUT, use the default PL/I files SYSIN and SYSPRINT, respectively, when no file reference is included in the statement. For general information on stream input and output, **see** "Stream Input/Output," "GET Statement," and "PUT Statement."

In VAX-11 PL/I, SYSIN is associated with the default system input file SYS$INPUT, which in turn is usually assigned to the user's terminal. The PL/I print file SYSPRINT is associated with the default system file SYS$OUTPUT, which, in interactive mode, is also assigned to the user's terminal. (For further information, **see** the *VAX-11 PL/I User's Guide.*)

The discussions and examples in this section use the GET and PUT statements for terminal input and output. The statements use the default files SYSIN and SYSPRINT instead of specific file references.

VAX-11 PL/I also provides statement options that may be useful in terminal input and output. For full details on the GET and PUT options, **see** the *VAX-11 PL/I User's Guide.*

■ Simple Input from a Terminal

This case covers the acquisition of one or more values from the terminal. A simple application of the GET LIST statement is the most expedient solution. Such a statement has the form:

```
GET LIST (input-target,...) ;
```

Because this statement has no reference to a specific file, the default file SYSIN (the terminal) is assumed. When this GET LIST statement is executed in a program, the program will pause until enough values are typed by the user to satisfy the input-target list.

The values typed by the user must be separated by carriage returns, spaces, or commas. The user must type at least one carriage return to send the typed line to the program. VAX-11 PL/I always appends a space to the end of any input line terminated by a carriage return unless the carriage return is inside

a quoted string. The appending of spaces can be disabled by the IGNORE_LINE_MARKS ENVIRONMENT option; **see** the *VAX-11 PL/I User's Guide*.

The input-target list must be enclosed in parentheses and input targets must be separated by commas. In the context of simple terminal input, the input targets are usually simple variable references. For example, the statement:

```
GET LIST (SALARY,CONTRIBUTION(42),PAYROLL.DEDUCTION) ;
```

acquires three character strings from the terminal. The strings are converted automatically to the target data types and assigned to the scalar variable SALARY, element 42 of the array CONTRIBUTION, and member DEDUCTION of the structure PAYROLL. There are several sequences with which the user can type the needed values, including:

```
15500,500,1200(RET)

15500(RET)500(RET)1200(RET)

15500,500(RET)1200(RET)
```

If a carriage return is typed in response to an input request from GET LIST, the null character string '' is assigned to the input target. If a carriage return is typed in response to an input request from GET EDIT, the requested field width is filled with spaces and assigned to the input target under control of the corresponding format item. (Note that an all-space field will cause an error for B formats.)

For full details on input targets, **see** "GET LIST Statement."

■ Simple Output to a Terminal

You can send data to a terminal with the PUT LIST statement. A simple form of PUT LIST is:

```
PUT LIST (output-source,...);
```

The output sources in simple cases are expressions, including variable references. The PUT LIST statement converts the results of the expressions to the appropriate character representations and sends the character strings to the terminal. For instance, the statement:

```
PUT LIST (A,B,C);
```

converts the values of the variables A, B, and C to character strings and sends the results to the terminal. In this simple case, the displayed strings are separated by tabs.

The file SYSPRINT, used as the default output stream by PUT LIST, is a print file, and the terminal has the characteristics of print files (**see** "Print File"). For example, the ENDPAGE condition is signaled when the terminal's page size is exceeded.

■ Examples

```
SIMPLE_INPUT: PROCEDURE OPTIONS (MAIN);
      /* Simple input from user's terminal */
DECLARE
      BADGE_NUMBER FIXED DECIMAL (5),
      SOCIAL_SECURITY_NUMBER CHARACTER(11) ;

GET LIST (BADGE_NUMBER,SOCIAL_SECURITY_NUMBER);

PUT LIST (BADGE_NUMBER,SOCIAL_SECURITY_NUMBER);

END SIMPLE_INPUT;
```

VAX-11 PL/I does not print a prompt character on the terminal when a program executes a GET or READ statement. Consequently, it is difficult to tell that a program is trying to read data unless the program executes an output statement containing a prompting message. The program SIMPLE_INPUT would be easier to use if the following statement appeared immediately before GET LIST:

```
PUT SKIP
LIST('Enter badge number,social security number:');
```

A carriage return does not occur automatically after the prompt, so the input can be entered on the same line. The completed line might be:

```
Enter badge number,social security number:7,116-40-0482 RET
```

The GET statement also has a PROMPT statement option that displays a prompt on the user's terminal. **See** the *VAX-11 PL/I User's Guide* for details.

```
TIN: PROCEDURE OPTIONS(MAIN);

DECLARE STRING CHAR(10) VARYING,
      I FIXED BINARY STATIC INITIAL(0),
      A FLOAT BINARY;
DECLARE EOF BIT STATIC INITIAL('0'B);

ON ENDFILE(SYSIN) EOF = '1'B;

DO WHILE(^EOF); /* stop when CTRL/Z is typed */
      PUT SKIP LIST('Enter string,integer,float>');
      GET LIST(STRING,I,A);

      PUT SKIP LIST(STRING,I,A);
END;
END TIN;
```

Here, the user is prompted to enter three values from the default file SYSIN. The three values are immediately written out to the default file SYSPRINT. This sequence continues until the prompt is answered with a CTRL/Z, which

signals the ENDFILE condition for SYSIN; the current values of the three variables are then written out, and the program terminates. A sample dialog with the program is as follows:

```
$ R TIN(RET)

Enter string,integer,float> JONES,27,3.75(RET)
JONES                  27    3.7500000E+00
Enter string,integer,float> JONES 27 3.75(RET)
JONES                  27    3.7500000E+00
Enter string,integer,float> JONES(RET)
27(RET)
3.75(RET)
JONES                  27    3.7500000E+00
Enter string,integer,float> DOOLEY(RET)
(RET)
3E-6(RET)
DOOLEY                 0     3.0000001E-06
Enter string,integer,float> ^Z
DOOLEY                 0     3.0000001E-06
$
```

Notice that input fields can be separated by commas, spaces, or carriage returns. Notice also that entering a blank line after 'DOOLEY' causes the program to "null" the value of I, setting it to zero.

■ Other Topics

The following list of topics may be of interest in terminal input and output applications:

- Use of GET STRING and certain built-in functions for string handling. **See** "GET Statement" and "String Handling."
- Use of GET EDIT and PUT EDIT to control the format of input or output data. **See** "GET Statement" and "PUT Statement."
- Use of PUT SKIP, PUT LINE, and PUT PAGE to create formatted displays. **See** "PUT Statement."
- Use of the OPTIONS keyword with GET and PUT to override default operations. **See** the *VAX-11 PL/I User's Guide.*

THEN Keyword

The THEN keyword is specified in an IF statement to define the action to be taken if a given expression is true. For example:

```
IF (A < B) THEN BEGIN ;
```

The action following the THEN keyword may be null. For more information, see the entry for the IF statement.

TIME Built-In Function

The TIME built-in function returns an eight-character string representing the current time of day in the form hhmmssxx, where:

hh is the current hour (00–23)
mm is the minutes (00–59)
ss is the seconds (00–59)
xx is hundredths of seconds (00–99)

The format of the TIME built-in function is:

TIME()

■ Returned Value

The time is returned as a string of type CHARACTER (8).

TITLE Option

The TITLE option is specified in an OPEN statement to designate the external file specification of the file to be associated with the PL/I file. The TITLE option can be specified only on the OPEN statement for a file. Its format is:

TITLE(expression)

expression

A character-string expression of up to 128 characters, representing an external file specification for the file.

The file specification can be any valid VAX/VMS file specification, device name, or logical name.

When the name given in the TITLE does not fully specify a VAX/VMS file or device, VAX-11 PL/I:

1. Performs logical name translation.
2. Applies default values given in the DEFAULT_FILE_NAME option of the ENVIRONMENT attribute.
3. Applies system defaults.

For complete details on how the file specification is interpreted, **see** the *VAX-11 PL/I User's Guide*.

TO Option

The TO option defines an end-value for a controlled DO statement specification. For example:

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

The DO-group following this statement executes until the value of I exceeds 10. **See** "DO Statement."

TRANSLATE Built-In Function

Given a character-string argument, the TRANSLATE built-in function replaces occurrences of an old character with a corresponding translation character and returns the resulting string. Its format is:

TRANSLATE(original,translation[,oldchars])

original

A character-string expression in which specific characters are to be translated.

translation

A character-string expression giving replacement characters for corresponding characters in oldchars.

If the translation is shorter than oldchars, the translation is padded on the right with spaces to the length of oldchars before any translation occurs. If the translation is longer than oldchars, its excess characters (on the right) are ignored.

oldchars

A character-string expression indicating which characters in the original are to be replaced. If oldchars is not specified, it defaults to COLLATE().

The following steps are performed for each character (beginning at the leftmost) in the original:

1. Let original(i) be the current character in the original string, and let result(i) be the corresponding character in the resulting string.
2. Search oldchars for the leftmost occurrence of original(i).
3. If oldchars does not contain original(i), then let result(i) equal original(i). Otherwise, let j equal the position of the leftmost occurrence of original(i) in oldchars, and let result(i) equal translation(j).
4. Return to step 1.

■ Returned Value

The string returned is of type CHARACTER(length), where length is the length of the original string. If the original string is a null string, the returned value is a null string.

■ Examples

```
TRANSLATE_XM: PROCEDURE OPTIONS(MAIN);

DECLARE NEWSTRING CHARACTER(80) VARYING;
DECLARE TRANSLATION CHARACTER(128);
DECLARE I FIXED;
DECLARE COLLATE BUILTIN;

/* translate space to '0': */
NEWSTRING = TRANSLATE('1 2','0',' ');
PUT SKIP LIST(NEWSTRING);
```

```
/* translate letter 'F' to 'E': */
NEWSTRING = TRANSLATE('BFFLZFBUB','E','F');
PUT SKIP LIST(NEWSTRING);

/* change case of letters in sentence */
TRANSLATION = COLLATE;

DO I=66 TO 91; /* replace upper with lower */
SUBSTR(TRANSLATION,I,1) = SUBSTR(COLLATE,I+32,1);
END;
DO I=98 TO 123; /* replace lower with upper */
SUBSTR(TRANSLATION,I,1) = SUBSTR(COLLATE,I-32,1);
END;
NEWSTRING =
TRANSLATE('THE QUICK BROWN fox JUMPS OVER THE LAZY dog',
TRANSLATION);
PUT SKIP LIST(NEWSTRING);

END TRANSLATE_XM;
```

The first reference translates the string '1 2' to '102'. The second reference translates 'BFFLZFBUB' to 'BEELZEBUB'. The third reference produces the new sentence:

```
'the quick brown FOX jumps over the lazy DOG'
```

TRUNC Built-In Function

The TRUNC built-in function changes all fractional digits in an arithmetic expression x to zeros and returns the resulting integer value. Its format is:

TRUNC(x)

■ Returned Value

If x is a floating-point expression, the returned value is a floating-point value. If x is a fixed-point expression, the returned value is a fixed-point value with the same base as x and with the attributes:

precision: min(31,p–q+1)

scale factor: 0

where p and q are the precision and scale factor of x.

UNDEFINEDFILE Condition Name

The UNDEFINEDFILE condition name can be specified in an ON, SIGNAL, or REVERT statement to designate an undefined file condition or ON-unit for a specific file. The format of the UNDEFINEDFILE condition name is:

UNDEFINEDFILE (file-reference)

file-reference

A reference to a file constant or file variable for which the ON-unit is established.

If the name of a file variable is specified, the variable must be resolved to the name of a file constant when the condition is signaled.

PL/I signals the UNDEFINEDFILE condition when a file cannot be opened. Some examples of errors that cause the UNDEFINEDFILE condition are:

- The value specified by the TITLE option is an invalid file specification.
- The file is opened for input or update and the specified file does not exist.
- An existing file is accessed with PL/I file description attributes that are inconsistent with the file's actual organization.
- Any file system-detected error prevents the file from being accessed.

The UNDEFINEDFILE condition lets you establish an ON-unit to provide processing when a file cannot be opened, for example, to provide a default file if no file is specified at run time.

```
X: PROCEDURE (FILENAME);
DECLARE FILENAME CHARACTER (128) VARYING;
DECLARE INPUT_FILE FILE INPUT;
    ON UNDEFINEDFILE (INPUT_FILE)
           OPEN FILE (INPUT_FILE)
           TITLE ('SYS$INPUT');
    OPEN FILE (INPUT_FILE) TITLE (FILENAME);
```

In this example, the procedure X expects a file specification string to be passed as an argument. If no argument is passed, or if the argument is not a valid file specification, the OPEN statement fails. The UNDEFINEDFILE ON-unit provides a default OPEN statement with the file specification SYS$INPUT.

An ON-unit established to handle the UNDEFINEDFILE condition can obtain information about the condition by invoking the following built-in functions:

- The ONFILE built-in function returns the name of the file being processed when the condition was signaled.
- The ONCODE built-in function returns the specific status value associated with the error.

■ ON-Unit Completion

The action taken on a normal return from the UNDEFINEDFILE condition depends on whether the file was opened explicitly or implicitly.

If the UNDEFINEDFILE condition was signaled following an explicit OPEN statement for a file, then the normal action following the ON-unit execution is for the program to continue. If the ON-unit does not transfer control elsewhere in the program, control returns to the statement following the OPEN statement that caused the condition to be signaled.

If the UNDEFINEDFILE condition was signaled during an implicit open attempt, the run-time system tests the state of the file. If the file is not open, the ERROR condition is signaled. If the file was opened by the ON-unit, execution of the input/output statement continues.

If an ON-unit receives control when an explicit OPEN results in the UNDEFINEDFILE condition, and the ON-unit does not handle the condition by opening the file or by transferring control elsewhere in the program, control returns to the statement following the OPEN. Then, if an attempt is made to access the file with an I/O statement, the UNDEFINEDFILE condition is signaled again when PL/I attempts the implicit open of the file. This time, PL/I signals the ERROR condition on completion of the ON-unit.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement."

UNDERFLOW Condition Name

The UNDERFLOW condition name can be specified in an ON, REVERT, or SIGNAL statement to designate a floating-point underflow condition or ON-unit.

PL/I signals the UNDERFLOW condition when the absolute value of the result of an arithmetic operation on a floating-point value is smaller than the minimum value that can be represented by the VAX–11 hardware.

The value resulting from an operation that causes this condition is set to zero.

This condition is signaled by PL/I only in procedures in which the UNDERFLOW option is enabled. (**See** "UNDERFLOW Option.")

■ ON-Unit Completion

Control is returned to the point of the interrupt, and execution continues with zero as the result of the operation.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement."

UNDERFLOW Option

The UNDERFLOW option may be specified using the OPTIONS keyword on a PROCEDURE statement. It specifies that the run-time system is to signal floating-point underflow conditions that occur during the execution of the procedure. (**See** "UNDERFLOW Condition.") The UNDERFLOW option affects the procedure on which it is specified and all defined descendents of that procedure. For example:

```
COMPUTE: PROCEDURE OPTIONS (UNDERFLOW);
```

In standard PL/I, the UNDERFLOW condition is signaled whenever an underflow occurs, and the UNDERFLOW option need not be specified. In VAX-11 PL/I, the UNDERFLOW option must be specified in each procedure for which underflow conditions are to be signaled.

UNSPEC Built-In Function

The UNSPEC built-in function returns a bit string representing the internal coded value of the referenced scalar variable. The variable can be a scalar variable of any type. The format of the function is:

UNSPEC(reference)

■ Returned Value

The returned value is a bit string whose length is the number of bits occupied by the referenced variable. This length must be less than or equal to the maximum length for bit-string data. The returned bit string contains the contents of the referenced variable's storage, the first bit in storage being the first bit in the returned value. The actual value is specific to VAX-11 PL/I and may differ from other PL/I implementations. Note that if the referenced variable is a binary integer (FIXED BINARY), the first bit in the returned value is the lowest binary digit.

■ Example

```
DECLARE X CHARACTER(2), Y BIT(16);

X = 'AB';
Y = UNSPEC(X);
      .
      .
DECLARE I FIXED BINARY(15);
I = 2;
PUT LIST(UNSPEC(I));
```

As a result of the first UNSPEC reference, Y contains the ASCII codes of 'A' and 'B'. The PUT LIST statement containing UNSPEC(I) prints the string

```
'0100000000000000'B
```

UNSPEC Pseudovariable

The UNSPEC pseudovariable interprets any reference to a scalar variable as a reference to a bit string. **See also** "Pseudovariable" for general rules. The format of the pseudovariable (in an assignment statement) is:

UNSPEC(reference) = expression;

reference

A reference to a scalar variable. The length of its storage in bits must be less than or equal to the maximum length for bit-string data.

In an assignment of the form:

UNSPEC(reference) = value;

the value is converted to a bit string if necessary and copied into the storage of the reference. The value is truncated or zero-extended as necessary to match the length of the storage.

■ Example

```
DECLARE X FIXED BINARY (15);
UNSPEC(X) = '110'B;
```

The use of the constant '110'b, which appears to be 6 in binary, actually assigns 3 to X. The two low-order bits of X (that is, X's first two bits of storage) are set; all other bits of X are cleared.

UPDATE Attribute

The UPDATE attribute is a file description attribute that indicates that the associated file is to be used for both input and output. The UPDATE attribute can be applied to relative files, indexed sequential files, and sequential disk files with fixed-length records.

Specify the UPDATE attribute on a DECLARE statement for a file constant or on an OPEN statement to access the file for update. The UPDATE attribute implies the RECORD attribute.

For a description of the attributes that may be applied to files and the effects of combinations of these attributes, **see** "File Description Attributes and Options."

The UPDATE attribute may be supplied by default for a file, depending on the context of its opening. **See** "Opening a File."

■ Restrictions

The UPDATE attribute directly conflicts with the INPUT, OUTPUT, STREAM, and PRINT attributes, and with any data type attribute other than FILE.

VALID Built-In Function

The VALID built-in function determines whether the argument x, a pictured variable, has a value that is valid with respect to its picture specification. A value is valid if it is any of the character strings that can be created by the picture specification. The function returns '0'B if x has an invalid value and '1'B if it has a valid value. The function can be used whenever a data item is read in with a record input (READ) statement, to ensure that the input data is valid. The format of the function is:

VALID(x)

x

A reference to a variable declared with the PICTURE attribute.

Note that pictured data is always validated (and thus, the VALID function is unnecessary) when it is read in with the GET EDIT statement and the P format item; the ERROR condition is signaled if the data does not conform to the picture given in the P format item. If GET LIST is used (or GET EDIT with a format item other than P), the input value is converted to conform to the pictured input target. (**See** "Conversion of Data" for details.)

■ Example

```
VALP: PROCEDURE OPTIONS(MAIN);

DECLARE INCOME PICTURE '$$$$$$V.$$';
DECLARE MASTER RECORD FILE;
DECLARE I FIXED;

DO I = 1 TO 2;
READ FILE(MASTER) INTO(INCOME);
IF VALID(INCOME) THEN;
      ELSE PUT SKIP LIST('Invalid input:',INCOME);
END;

END VALP;
```

If the file MASTER.DAT contains:

```
$15000.50
△50000.50
```

then the program VALP writes out:

```
Invalid input: △50000.50
```

The picture '$$$$$$V.$$' specifies a fixed-point decimal number of up to seven digits, two of which are fractional. To be valid, a pictured value must consist of nine characters; the first digit must be immediately preceded by a dollar sign, the number must contain a period before the fractional digits, and

each position specified by a dollar sign must contain either that sign, a digit, or a space. The second record in MASTER.DAT can be assigned by the READ statement because it has the correct size; however, the pictured value is invalid because it does not contain a dollar sign.

VALUE Attribute

The VALUE attribute is provided for calling non-PL/I procedures. For complete details on using the VALUE attribute, **see** the *VAX-11 PL/I User's Guide*.

The VALUE attribute serves two purposes:

1. It specifies, for global external variables, that the variable has a constant value that the compiler can use as an immediate value in generating instructions for the VAX-11 hardware. No storage is allocated for the variable. For this usage, VALUE must be specified in conjunction with the GLOBALREF or GLOBALDEF attribute.
2. It specifies, in a parameter descriptor in an ENTRY declaration, that the corresponding argument is to be passed using the VAX-specific convention for passing arguments by value. For this usage, VALUE must be specified in conjunction with one of the attributes ANY, FIXED BINARY, ENTRY, POINTER, or BIT(n) where n < 33.

Its format is:

```
VALUE { GLOBALDEF [ (psect-name) ] [INITIAL (value)] }
      { GLOBALREF                                    }
```

Variable

A variable is a named data item that can be assigned various values in the program. The converse of a variable is a constant, that is, a data item whose value cannot be changed.

Normally, a variable's value will change during the execution of the program. However, it is sometimes convenient to declare a static variable whose value will never change. For example:

```
DECLARE MONTHS (12) CHARACTER (12) VARYING
      STATIC INITIAL ('JANUARY', 'FEBRUARY', ...
                'DECEMBER') ;
```

The term *variable* is used in this manual to mean any of the following:

- A name declared as a variable
- The storage associated with such a name
- A reference to all or part of the storage, as in MONTHS(2)

■ Addressable Variable

It is required in some contexts, such as in argument lists of certain built-in functions, that a variable be addressable. A variable is addressable if it has the following properties:

1. It is not suitable for bit-string overlay defining; that is, it does not consist entirely of unaligned bit data. (**See** "Defined Variable" for a definition of string overlay defining.)
2. It is not an unconnected array. (**See** "Arrays of Structures.")
3. It is not declared with the VALUE attribute. (**See** "VALUE Attribute.")

These rules ensure that the variable can occupy contiguous storage beginning on a byte boundary. (Note that constants are never addressable in PL/I.)

VARIABLE Attribute

The VARIABLE attribute indicates that the associated identifier is a variable. VARIABLE is implied by all computational data type attributes and by all noncomputational attributes except FILE and ENTRY.

If you specify the FILE or ENTRY attribute in a DECLARE statement without the VARIABLE attribute, the defined object is assumed to be a file or entry constant.

The VARIABLE attribute is implied by the LABEL attribute. Label constants are declared only by use of the label identifier in the program; a label constant cannot be defined in a DECLARE statement.

See "Entry Data," "File," and "Label," for descriptions of variables of these data types.

■ Restrictions

The VARIABLE attribute is not valid in a returns descriptor or in a parameter descriptor.

VARIABLE Option

The VARIABLE option specifies that an external procedure can be invoked with argument lists of different lengths or that default arguments will not be specified in the invocation of an external procedure. It is specified in the declaration of an external entry as in the following example:

```
DECLARE SYS$FAO ENTRY (ANY) OPTIONS (VARIABLE) ;
```

This attribute is applicable only in the declaration of external procedures that are not written in PL/I. For complete details on using OPTIONS (VARIABLE), **see** the *VAX-11 PL/I User's Guide*.

■ Restrictions

The VARIABLE option is valid only in conjunction with the ENTRY attribute.

VARYING Attribute

The VARYING attribute indicates that a character-string variable does not have a fixed length, but that its length changes according to its current value. The format of the VARYING attribute is:

```
{VARYING}
{VAR    }
```

A length attribute must be specified in conjunction with VARYING, giving the maximum length allowed for the variable. The current length of a value of the variable is stored with the value, and the current length can be determined at any time with the LENGTH built-in function.

For example:

```
DECLARE STRING CHARACTER(80) VARYING;
```

This declaration indicates that the longest length the string can have is 80. The storage allocated for varying-length strings is two bytes longer than the maximum length declared. These first two bytes contain the current length of the string.

Note that special rules apply to reading and writing record files into and from variables that have the VARYING attribute. **See** the *VAX-11 PL/I User's Guide*.

■ Restrictions

The VARYING attribute directly conflicts with any data type attribute other than CHARACTER.

VAXCONDITION Condition Name

The VAXCONDITION condition name can be specified in an ON, SIGNAL, or REVERT statement. The VAXCONDITION condition name provides a way to signal and handle operating-system or programmer-specific condition values. The format of the VAXCONDITION condition name is:

VAXCONDITION (expression)

expression

An expression yielding a fixed binary value. The expression is evaluated when the ON statement is executed, not when the condition is signaled.

The VAXCONDITION condition name is provided specifically for PL/I procedures that interact with VAX/VMS operating-system routines. For details on using the VAXCONDITION condition name and the meanings of system- and user-defined values you can specify, **see** the *VAX-11 PL/I User's Guide*.

VERIFY Built-In Function

The VERIFY built-in function compares a string with a test string and verifies that all characters that appear in the string also appear in the test string. If not, the VERIFY built-in function returns a fixed-point binary integer that

indicates the position of the first character in the string that is not present in the test string. If each character in the string is also in the test string, the function returns the value zero.

The format of the function is:

VERIFY(string,test-string)

string

A character-string expression representing the string to verify.

test-string

A character-string expression containing the set of characters against which the string is verified.

■ Examples

```
STRING = 'HOW MUCH IS 1 PLUS 2';
ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ ';
A = VERIFY(STRING,ALPHABET);
```

In this example, the variable ALPHABET contains the 26 uppercase letters plus the space character. The function returns a value of 13, indicating the position of the first nonalphabetic and nonspace character in STRING. Since the test string can also be a constant, you can find the first nonspace character in any string by writing:

```
A = VERIFY(STRING,' ');

NEWSTRING = 'ALL LETTERS';
A = VERIFY(NEWSTRING,ALPHABET);
```

In this example, VERIFY returns a value of zero. All characters in the string NEWSTRING are present in the string ALPHABET.

W

WHILE Option

The WHILE option may be specified in a DO statement to define a condition that must be met for the following DO-group to execute. It has the format:

WHILE (expression)

expression

A bit-string expression of any length. If any bit in the expression is 1, the expression is considered "true."

For example:

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

The subsequent DO-group is executed while the value of the expression A<B is true.

For more information, **see** "DO Statement."

WRITE Statement

The WRITE statement adds a record to a file, either at the end of a file that has the SEQUENTIAL and OUTPUT attributes, or in a specified key position in a file that has the KEYED and OUTPUT attributes or the KEYED and UPDATE attributes. The format of the WRITE statement is:

WRITE FILE(file-reference) FROM (variable-reference)
[KEYFROM (expression)]
[OPTIONS (option,...)] ;

file-reference

A reference to the file to which the record is to be written. If the file is not currently open, the WRITE statement opens the file with the implied attributes RECORD, OUTPUT, and SEQUENTIAL; these attributes are merged with the attributes specified in the file's declaration. **See also** "Opening a File."

variable-reference

A reference to the variable containing data for the output record. The variable must be addressable.

If the variable has the VARYING attribute and the file does not have the attribute ENVIRONMENT(SCALARVARYING), the WRITE statement writes only the current value of the varying string into the specified record. In all other cases, the WRITE statement writes the entire storage of the variable. If the contents of the variable do not fit the specified record size, the WRITE statement outputs as much of the variable as will fit and the ERROR condition is signaled.

KEYFROM (expression)

An option specifying that the record to be written is to be positioned in the file according to the key specified by expression. The file must have the KEYED attribute.

The nature of the key depends on the file's organization, as follows:

- If the file is a relative file or a sequential disk file with fixed-length records, the key value is a fixed binary value indicating the relative record number of the record to be written.
- If the file is an indexed sequential file, the key specifies the record's primary key. PL/I inserts the key value specified into the correct key field in the record and sets the key number to the primary index.

The value of the specified expression is converted to the data type of the key. If no record with the specified key exists, or if the specified key value cannot be converted to the data type of the key, the KEY condition is signaled.

OPTIONS (option,...)

An option specifying one or more of the WRITE statement options listed below, separated by commas.

FIXED_CONTROL_FROM (variable-reference)
RECORD_ID_TO (variable-reference)

These options are described fully in the *VAX-11 PL/I User's Guide*.

■ File Positioning

If the file has the UPDATE attribute, the current record is set to designate the record just written, and the next record is set to designate the record following the record just written. If there is no such record following the record just written, the next record is set to end-of-file.

■ Examples

The program TRUNC, below, reads a file with variable-length records into a character-string with the VARYING attribute and creates a sequential output file in which each record has a fixed length of 80 characters.

```
TRUNC: PROCEDURE;
DECLARE  INREC CHARACTER(80) VARYING,
         OUTREC CHARACTER(80),
         ENDED BIT(1) STATIC INIT('0'B),
         (INFILE,OUTFILE) FILE;

       OPEN FILE (INFILE) RECORD INPUT
                TITLE('RECFILE.DAT');
       OPEN FILE (OUTFILE) RECORD OUTPUT
                TITLE('TRUNCFILE.DAT')
                ENVIRONMENT(FIXED_LENGTH_RECORDS,
                            MAXIMUM_RECORD_SIZE(80));
```

(Continued on next page)

```
ON ENDFILE(INFILE) ENDED = '1'B;

READ FILE(INFILE) INTO (INREC);
DO WHILE (^ENDED);
        OUTREC = INREC;
        WRITE FILE (OUTFILE) FROM (OUTREC);
        READ FILE (INFILE) INTO (INREC);
        END;
CLOSE FILE(INFILE);
CLOSE FILE(OUTFILE);
RETURN;
END;
```

The ENVIRONMENT attribute for the file OUTFILE specifies the record format and length of each fixed-length record.

When records are written to a file with fixed-length records, the variable specified in the FROM option must have the same length as the records in the target output file. Otherwise, the ERROR condition is signaled. Thus, in this example, each record read from the input file is copied into a fixed-length character-string variable for output.

Each time this program is executed, it creates a new version of the file TRUNCFILE.DAT.

The next example adds records to the existing relative file EMP_FILE. The file is organized by employee numbers, and each record occupies the relative record number in the file that corresponds to the employee number.

```
ADD_EMPLOYEE: PROCEDURE;

DECLARE 1 EMPLOYEE,
        2 NAME,
            3 LAST CHAR(30),
            3 FIRST CHAR(20),
            3 MIDDLE_INIT CHAR(1),
          2 DEPARTMENT CHAR(4),
          2 SALARY FIXED DECIMAL (6,2),
      EMP_FILE FILE;
DECLARE MORE_INPUT BIT(1) STATIC INIT('1'B),
      NUMBER FIXED DECIMAL (5,0);

      OPEN FILE(EMP_FILE) DIRECT UPDATE;

      DO WHILE (MORE_INPUT);
         PUT SKIP LIST('Employee Number');
         GET LIST (NUMBER);

         PUT SKIP LIST
          ('Name (Last, First, Middle Initial)');
         GET LIST
          (EMPLOYEE.NAME.LAST,EMPLOYEE.NAME.FIRST,
              EMPLOYEE.NAME.MIDDLE_INIT);

         PUT SKIP LIST('Department');
         GET LIST (DEPARTMENT);

         PUT SKIP LIST('Starting salary');
         GET LIST(EMPLOYEE.SALARY);
```

```
      WRITE FILE (EMP_FILE)
          FROM (EMPLOYEE) KEYFROM(NUMBER);

      PUT SKIP LIST('More?');
      GET LIST(MORE_INPUT);
      END;
   CLOSE FILE(EMP_FILE);
   RETURN;
   END;
```

In this example, the file is opened with the DIRECT and UPDATE attributes, since records will be written only by referring to a key number. Within the DO-group, the program prompts for data for each new record that will be written to the file. After the data is input, the WRITE statement specifies the KEYFROM option to specify the relative record number. The number itself is not a part of the record, but will be used to retrieve the record when the file is accessed for keyed input.

X

X Format Item

The X format item sets a stream file or character-string expression to a column relative to the current position. It is the only control format item that can be used with either the FILE or STRING option of GET EDIT and PUT EDIT. The form of the X format item is:

X [(w)]

w

An integer that specifies a number of consecutive character positions in the stream; w must not be negative. If w is zero, no operation is performed. If w is omitted (permissible only on output), its value is assumed to be one.

■ Input with GET EDIT

On input, the next w columns after the current column are skipped.

■ Output with PUT EDIT

On output, w spaces are inserted following the current column.

When the output stream is a file, and the end of the current line is reached, the output of spaces continues on the next line until w spaces have been output. The size of the current line is either the default value or the specific value you have established for the file (**see** "LINESIZE Option"). If the file is a print file, the ENDPAGE condition is signaled if the page size is reached; on normal return from the ENDPAGE on-unit, output of spaces continues at the top of the next page, until w spaces have been output.

If the output stream is a character-string variable, w spaces are written to the variable. The ERROR condition is signaled if the maximum length of the string is exceeded.

■ Examples

```
XFOR: PROCEDURE OPTIONS(MAIN);

DECLARE INLINE CHARACTER(80) VARYING;
DECLARE FIRSTWORD CHARACTER(80) VARYING;
DECLARE OUTFILE PRINT FILE;
DECLARE SPACE1 FIXED;

GET EDIT(INLINE) (A(1000)) OPTIONS(PROMPT('Line>'));

SPACE1 = INDEX(INLINE,' '); /* position of first wordbreak */

FIRSTWORD = SUBSTR(INLINE,1,SPACE1-1);

PUT STRING(FIRSTWORD) EDIT (FIRSTWORD,'-FIRST WORD TYPED')
(A,X(2),A);

PUT SKIP FILE(OUTFILE) LIST(FIRSTWORD);

END XFOR;
```

The GET EDIT statement in the program XFOR inputs a complete line from a user's terminal, after issuing the prompt 'Line>'. If the user responds as follows:

```
Line>beautiful losers(RET)
```

then the following output is written to OUTFILE.DAT:

```
beautiful -FIRST WORD TYPED
```

The X format item has correctly inserted two spaces between 'beautiful' and '—FIRST WORD TYPED'.

```
XFOR2: PROCEDURE OPTIONS(MAIN);

DECLARE INLINE CHARACTER(80) VARYING;
DECLARE OUTFILE2 PRINT FILE;

GET EDIT(INLINE) (X(10),A(1000))
  OPTIONS(PROMPT('Line>'));

PUT SKIP FILE(OUTFILE2) LIST(INLINE);
END XFOR2;
```

In the program XFOR2, the GET EDIT statement skips the first 10 characters typed after the prompt and then inputs the remainder of the line. If the user responds to the prompt as follows:

```
Line>ABCDEFGHIJKLMNOPQRSTUVWXYZ(RET)
```

then the following output is written to OUTFILE2.DAT:

```
KLMNOPQRSTUVWXYZ
```

The first 10 letters (A–J) have been ignored on input.

Z

ZERODIVIDE Condition Name

The ZERODIVIDE condition name can be specified in an ON, REVERT, or SIGNAL statement to designate a divide-by-zero condition or ON-unit.

PL/I signals the ZERODIVIDE condition when the divisor in a division operation has a value of zero. The value resulting from such an operation is undefined.

■ ON-Unit Completion

Control returns to the point of the interruption.

For more information, **see** "ON Conditions and ON-Units" and "ON Statement."

Appendix A
Alphabetic Summary of Keywords

A summary of the PL/I keywords follows. This alphabetic summary does not include the options for the ENVIRONMENT attribute nor the options for I/O statements, since these keywords are not standard PL/I. The ENVIRONMENT options are listed under the entry "ENVIRONMENT Attribute." Options for each I/O statement are listed under the entry for the statement.

Keyword	Abbreviation	Use
A		Format item
ABS		Built-in function
ACOS		Built-in function
ADDR		Built-in function
ALIGNED		Data attribute
ALLOCATE	ALLOC	Statement
ANY		Attribute
ANYCONDITION		ON condition name
AREA		Data attribute
ASIN		Built-in function
ATAN		Built-in function
ATAND		Built-in function
ATANH		Built-in function
AUTOMATIC	AUTO	Attribute
B		Format item
B1		Format item
B2		Format item
B3		Format item
B4		Format item
BASED		Attribute
BEGIN		Statement
BINARY	BIN	Data attribute, built-in function
BIT		Data attribute, built-in function
BOOL		Built-in function
BUILTIN		Attribute
BY		Option of DO statement
BYTE		Built-in function

Keyword	Abbreviation	Use
CALL		Statement
CEIL		Built-in function
CHARACTER	CHAR	Data attribute, built-in function
CLOSE		Statement
COLLATE		Built-in function
COLUMN	COL	Format item
COPY		Built-in function
COS		Built-in function
COSD		Built-in function
COSH		Built-in function
DATE		Built-in function
DECIMAL	DEC	Data attribute, built-in function
DECLARE	DCL	Statement
DEFINED	DEF	Attribute
DELETE		Statement
DESCRIPTOR		Built-in function
DIMENSION	DIM	Built-in function
DIRECT		File attribute
DISPLAY		Built-in subroutine
DIVIDE		Built-in function
DO		Statement, option of GET and PUT statements
E		Format item
EDIT		Option of GET and PUT statements
ELSE		Optional clause of IF statement
END		Statement
ENDFILE		Condition name
ENDPAGE		Condition name
ENTRY		Statement, data attribute
ENVIRONMENT		Option of FILE attribute, OPEN and CLOSE statements
ERROR		Condition name
EXP		Built-in function
EXTEND		Built-in subroutine
EXTERNAL	EXT	Attribute
F		Format item
FILE		Data attribute, option of I/O statements
FINISH		Condition name
FIXED		Data attribute, built-in function
FIXEDOVERFLOW		Condition name
FLOAT		Data attribute, built-in function
FLOOR		Built-in function
FLUSH		Built-in subroutine
FORMAT		Statement
FREE		Statement
FROM		Option of WRITE and REWRITE statements

Keyword	Abbreviation	Use
GET		Statement
GLOBALDEF		Attribute
GLOBALREF		Attribute
GOTO	GO TO	Statement
HBOUND		Built-in function
IF		Statement
%INCLUDE		Statement
INDEX		Built-in function
INITIAL	INIT	Attribute
INPUT		File attribute
INTERNAL	INT	Attribute
INTO		Option of READ statement
KEY		Option of READ, WRITE, REWRITE, and DELETE statements, Condition name
KEYED		File attribute
KEYFROM		Option of WRITE statement
KEYTO		Option of DELETE, READ, REWRITE, and WRITE statements
LABEL		Data attribute
LBOUND		Built-in function
LENGTH		Built-in function
LINE		Format item, Option of PUT statement
LINENO		Built-in function
LINESIZE		Option of OPEN statement
LIST		Option of GET and PUT statements
LOG		Built-in function
LOG10		Built-in function
LOG2		Built-in function
MAIN		Option of PROCEDURE and ENTRY
MAX		Built-in function
MIN		Built-in function
MOD		Built-in function
NULL		Built-in function
NXTVOL		Built-in subroutine
OFFSET		Data attribute, built-in function
ON		Statement
ONARGSLIST		Built-in function
ONCODE		Built-in function
ONFILE		Built-in function
ONKEY		Built-in function
OPEN		Statement
OPTIONS		Option of DELETE, GET, PROCEDURE, PUT, READ, REWRITE, WRITE statements, ENTRY attribute

Keyword	Abbreviation	Use
OUTPUT		File attribute
OVERFLOW		Condition name
P		Format item
PAGE		Format item, option of PUT statement
PAGENO		Built-in function, pseudovariable
PAGESIZE		Option of OPEN statement
PICTURE	PIC	Data attribute
POINTER	PTR	Data attribute, built-in function
POSITION		Attribute
PRINT		File attribute
PROCEDURE	PROC	Statement
PUT		Statement
R		Format item
RANK		Built-in function
READ		Statement
READONLY		Attribute
RECORD		File attribute
RECURSIVE		Option of PROCEDURE and ENTRY
REPEAT		Option of DO statement
%REPLACE		Source modification statement
RESIGNAL		Built-in subroutine
RETURN		Statement
RETURNS		Attribute, Option of PROCEDURE and ENTRY statements
REVERT		Statement
REWIND		Built-in subroutine
REWRITE		Statement
ROUND		Built-in function
SEQUENTIAL	SEQ	File attribute
SET		Option of ALLOCATE and READ
SIGN		Built-in function
SIGNAL		Statement
SIN		Built-in function
SIND		Built-in function
SINH		Built-in function
SKIP		Format item, Option of GET and PUT statements
SPACEBLOCK		Built-in subroutine
SQRT		Built-in function
STATIC		Attribute
STOP		Statement
STREAM		File attribute
STRING		Built-in function, pseudovariable, option of GET and PUT statements
SUBSTR		Built-in function, pseudovariable
SYSIN		Default input file
SYSPRINT		Default output file

Keyword	Abbreviation	Use
TAB		Format item
TAN		Built-in function
TAND		Built-in function
TANH		Built-in function
THEN		Keyword of IF statement
TIME		Built-in function
TITLE		Option of OPEN statement
TO		Option of DO statement
TRANSLATE		Built-in function
TRUNC		Built-in function
UNDEFINEDFILE		Condition name
UNDERFLOW		Condition name, option of PROCEDURE and ENTRY
UNSPEC		Built-in function, pseudovariable
UPDATE		File attribute
VALID		Built-in function
VALUE		Attribute
VARIABLE		Attribute, option of ENTRY attribute
VARYING	VAR	Data attribute
VAXCONDITION		Condition name
VERIFY		Built-in function
WHILE		Option of DO statement
WRITE		Statement
X		Format item
ZERODIVIDE		Condition name

Appendix B
Compatibility with Standard PL/I, Subset G

This appendix describes the differences between the VAX-11 implementation of PL/I and the definition of the PL/I General-Purpose Subset. The subset (X3.74) is a subset of ANSI X3.53-1976.

This appendix has the following sections:

- Section B.1 provides an overview of the G Subset.
- Section B.2 lists the extensions made to the language to provide enhancements for PL/I programs executing in the VAX-11 VMS operating system environment.
- Section B.3 lists features of full PL/I that were excluded from the G Subset but that have been incorporated in the implementation of VAX-11 PL/I.
- Section B.4 lists the implementation-defined values that are used in VAX-11 PL/I.

B.1 The G (General-Purpose) Subset

The G subset of PL/I was designed to be useful in scientific, commercial, and systems programming, especially on small and medium-size computer systems. Among the primary goals of the design of the subset were:

- To include features that were easy to learn and to use and to exclude features that were difficult to learn or prone to error
- To provide a subset that would be easily portable from one computer system to another
- To exclude features that were not often used and whose implementation greatly increased the complexity of the run-time support required by the compiler

The essential elements of the subset are described below. These descriptions are extracted from the definition of the subset standard in the report prepared by the ANSI Technical Committee X3J1-PL/I.

B.1.1 Program Structure

The G Subset includes a complete character set, with comments, identifiers, decimal arithmetic constants, and simple string constants.

Begin blocks and DO-groups are included in the subset. Each block or group in the program must be terminated with an END statement.

B.1.2 Program Control

The following program control statements are included in the subset: CALL, RETURN, IF, DO, GOTO, null, STOP, ON, REVERT, and SIGNAL.

The DO statement options supported are TO, BY, WHILE, and REPEAT.

An IF statement may contain unlabeled THEN and ELSE clauses.

An ON statement may specify a single condition. The condition names supported are ERROR, ENDFILE, ENDPAGE, FIXEDOVERFLOW, KEY, OVERFLOW, UNDEFINEDFILE, UNDERFLOW, and ZERODIVIDE.

B.1.3 Storage Control

The subset includes the assignment statement and the assignment of array and structure variables whose dimensions and data types match. The subset also permits aggregate promotion, that is, the assignment of a scalar expression to every element or member of an aggregate variable.

In the subset, only static variables may be initialized.

The ALLOCATE statement with the SET option and the FREE statement are included in the subset.

B.1.4 Input/Output

The I/O statements are:

- OPEN and CLOSE
- READ, WRITE, DELETE, and REWRITE for record I/O
- GET and PUT, with FILE, STRING, EDIT, LIST, PAGE, SKIP, and LINE options for stream I/O

The file attributes, specified in DECLARE or OPEN, are DIRECT, ENVIRONMENT, INPUT, KEYED, OUTPUT, PRINT, RECORD, SEQUENTIAL, STREAM, and UPDATE.

The FORMAT statement is included. The format items are E, F, P, A, B, X, R, PAGE, SKIP, COLUMN, TAB, and LINE

B.1.5 Attributes and Pictures

The DECLARE statement is included in the subset. All names must be declared, either by means of a DECLARE statement or by means of a label prefix.

The attributes supported are: ALIGNED, AUTOMATIC, BASED, BINARY, BIT, BUILTIN, CHARACTER, DECIMAL, DEFINED, DIRECT, ENTRY, ENVIRONMENT, EXTERNAL, FILE, FIXED, FLOAT, INITIAL, INPUT, INTERNAL, KEYED, LABEL, OPTIONS, OUTPUT, PICTURE, POINTER, PRINT, RECORD, RETURNS, SEQUENTIAL, STATIC, STREAM, UPDATE, VARIABLE, and VARYING.

The picture characters included are CR, DB, S, V, Z, 9, –, +, $, and *. The picture insertion characters (. , / B) are also included.

B.1.6 Built-In Functions and Pseudovariables

The built-in functions in the subset are: ABS, ACOS, ADDR, ASIN, ATAN, ATAND, ATANH, BINARY, BIT, BOOL, CEIL, CHARACTER, COLLATE, COPY, COS, COSD, COSH, DATE, DECIMAL, DIMENSION, DIVIDE, EXP, FIXED, FLOAT, FLOOR, HBOUND, INDEX, LBOUND, LENGTH, LINENO, LOG, LOG2, LOG10, MAX, MIN, MOD, NULL, ONCODE, ONFILE, ONKEY, PAGENO, ROUND, SIGN, SIN, SIND, SINH, SQRT, STRING, SUBSTR, TAN, TAND, TANH, TIME, TRANSLATE, TRUNC, UNSPEC, VALID, and VERIFY.

The pseudovariables are PAGENO, STRING, SUBSTR, and UNSPEC.

B.1.7 Expressions

The subset supports all infix and prefix operators, the locator qualifier, parenthesized expressions, subscripts, and function references. Implicit conversion from one data type to another is restricted to those contexts in which the conversion is likely to produce the desired results.

B.2 VAX–11 Extensions to the G Subset Standard

B.2.1 Procedure-Calling and Condition-Handling Extensions

The following extensions to PL/I were made to allow VAX–11 PL/I procedures to call procedures written in any other programming language that also supports the VAX–11 calling standard.

1. The attributes ANY and VALUE describe how data is to be passed to a called procedure.
2. The VARIABLE option for the ENTRY attribute permits a PL/I procedure to call a non-PL/I procedure with an argument list of variable length. It also permits a procedure to omit arguments in an argument list.
3. The DESCRIPTOR built-in function may be used to pass an argument by descriptor to a non-PL/I procedure.

The following new attributes provide storage classes for PL/I variables. These attributes permit PL/I programs to take advantage of features of the VAX–11 linker and to combine PL/I procedures with other procedures that use these storage classes.

1. The GLOBALDEF and GLOBALREF attributes let you define and access external global variables and optionally place all external global definitions in the same program section.
2. The READONLY attribute can be applied to a static computational variable whose value does not change.
3. The VALUE attribute defines a variable that is, in effect, a constant whose value is supplied by the linker.

The following extensions to ON condition handling provide support for condition handling in the VAX/VMS environment:

1. The ON statement supports the ANYCONDITION keyword. The ON-unit established by this keyword is executed when any condition occurs for which no explicit ON-unit exists.
2. The ON statement supports programmer-named conditions with the VAXCONDITION keyword.
3. The RESIGNAL built-in subroutine permits an ON-unit to keep a signal active.
4. The ONARGSLIST built-in function provides an ON-unit with access to the mechanism and signal arguments of an exception condition.

B.2.2 Support of VAX–11 Record Management Services

The options of the ENVIRONMENT attribute provide support for many of the features and control values of the VAX–11 Record Management Services (RMS). Additional extensions have been made to the PL/I language to augment this support, as described below.

1. The OPTIONS option is supported on the GET, PUT, READ, WRITE, REWRITE, and DELETE statements.
2. The following built-in subroutines provide file handling and control functions: DISPLAY, EXTEND, FLUSH, NXTVOL, REWIND, and SPACEBLOCK.

B.2.3 Miscellaneous Extensions

The RANK and BYTE built-in functions are supported.

B.3 Full PL/I Features Supported

The items listed below are features that are explicitly excluded from the subset standard but that have been implemented in VAX–11 PL/I. These features all exist in full PL/I.

1. The ENTRY statement is supported.
2. The ENVIRONMENT option is supported on the CLOSE statement.
3. The picture characters Y, T, I, and R are supported, and pictures may include iteration factors.
4. RETURNS (CHARACTER(*)) is valid.
5. The FINISH condition is supported.
6. A REWRITE statement need not specify the FROM option if the most recent I/O operation on the file was a READ statement with the SET option.
7. The AREA and OFFSET attributes are supported. Allocation within an area must be controlled by a user-written procedure.
8. The OFFSET and POINTER built-in functions are supported.
9. The POSITION attribute is supported.

10. Automatic variables may be initialized. The INITIAL attribute may contain scalar expressions and asterisks with automatic variables.
11. The SET option is optional on the ALLOCATE statement if the allocated variable was declared with BASED(pointer-reference).
12. The character pair /* may be embedded in a comment.
13. It is permissible to use, as the source or target of a file I/O statement, a function reference that performs I/O on the same file and then returns to the original statement.
14. The expression in a WHILE clause or in an IF statement may be a bit string of any length. When evaluated, the expression results in a true value if any bit of the string expression is a one and in a false value if all bits in the string expression are zeros.
15. The control variable and the expressions in the TO, BY, and REPEAT options of the DO statement are not restricted to integers and pointers.

B.4 Implementation-Defined Values and Features

1. VAX-11 PL/I supports the full 256-character ASCII character set.
2. The default precisions for arithmetic data are:

   FIXED BINARY (31)
   FIXED DECIMAL (10)
   FLOAT BINARY (24)
   FLOAT DECIMAL (7)
3. The maximum record size for SEQUENTIAL files is 32767 bytes minus the length of any fixed-length control area.
4. The maximum key size is 255 bytes for character keys.
5. The default value for the LINESIZE option is as follows:
   - If the output is to a physical record-oriented device, such as a line printer or terminal, the default line size is the width of the device.
   - If the output is to a print file, the default line size is 132.
   - If the output is to a nonrecord device (magnetic tape), the default line size is 510.
6. The default value for the PAGESIZE option is as follows:
   - If the logical name SYS$LP_LINES is defined, the default page size is the numeric value of SYS$LP_LINES - 6.
   - If SYS$LP_LINES is not defined, or if its value is less than 30 or greater than 90, or if its value is not numeric, the default page size is 60.
7. The values for TAB positions are columns beginning with column 1 and every eight columns thereafter: 1, 9, 17, 25, ... 8*i+1, where i is (line size)/8.
8. The maximum length allowed for a file title is 128 characters.
9. The maximum number of digits in editing fixed-point data is 34.

10. The maximum numbers of digits for each combination of base and scale are:

    FIXED BINARY — 31
    FIXED DECIMAL — 31
    FLOAT BINARY — 113
    FLOAT DECIMAL — 34

    If the compiler option /G_FLOAT is not used, the maximum precisions are 15 and 53 for floating-point decimal and binary, respectively.

11. The default precision for integer values is 31.

12. The maximum number of arguments that can be passed to an entry point is 253.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

VAX-11 PL/I
Encyclopedic Reference
AA-H952A-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

— — — Do Not Tear - Fold Here and Tape —

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J3-5
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062

— — — Do Not Tear - Fold Here —