)

Kednos PL/I for OpenVMS Systems

Reference Manual

Order Number: AA-H952E-TM

November 2003

This manual defines Kednos PL/I for OpenVMS VAX on the OpenVMS VAX platform and Kednos PL/I for OpenVMS Alpha on the OpenVMS Alpha platform. It includes the keywords and the semantic and syntax rules of PL/I programming language statements, attributes, built-in functions, and other language elements.

Operating System and Version: For Kednos PL/I for OpenVMS VAX: OpenVMS VAX Version 5.5 or higher
For Kednos PL/I for OpenVMS Alpha: OpenVMS Alpha Version 6.2 or higher

Software Version: Kednos PL/I Version 3.8 for OpenVMS VAX
Kednos PL/I Version 4.4 for OpenVMS Alpha

Published by: Kednos Corporation, Pebble Beach, CA, www.Kednos.com

First Printing, August 1980
Revised, November 1983
Updated, April 1985
Revised, April 1987
Revised, January 1992
Revised, March 1992
Revised, November 1993
Revised, April 1995
Revised, October 1995
Revised, November 2003

Contents

EXAMPLES

FIGURES

TABLES

Preface

Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha are Kednos Corporation implementations of the PL/I programming language, General-Purpose Subset, ANSI X3.74-1981.

Intended Audience

This manual is intended for programmers using PL/I to design or implement applications on OpenVMS VAX or OpenVMS Alpha systems. A prerequisite for attaining optimal benefit from the manual is that its users understand the concepts of programming in PL/I and are familiar with the keywords and topics that will be searched for information. This manual is not suitable for use as a tutorial document.

Associated Documents

The *Kednos PL/I for OpenVMS Systems User Manual* provides information on program development with the system-specific command language, the extensive I/O capabilities provided in PL/I, and programming techniques available to PL/I programs executing under the exclusive control of the operating system.

For information on installing PL/I, see the *Kednos PL/I for OpenVMS Alpha Installation Guide.*

Conventions

Table 1 lists the conventions used in this manual.

Table 1 Conventions Used in this Manual

Conventions	Meaning
Return	In examples, the symbol Return represents a single stroke of the key on the terminal.
Ctrl/*X*	In examples, Ctrl/*X* indicates that you hold down the Ctrl key while you press another key (represented here by *X*).
monospace	This bold monospace typeface is used in interactive examples to indicate input entered by the user.
italic	This italic typeface is used to identify variable names.
. . .	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.
. . .	Horizontal ellipses indicate that additional parameters, options, or values can optionally be entered. When a comma precedes an ellipsis, it indicates that successive items must be separated by commas.

Table 1 (Cont.) Conventions Used in this Manual

Conventions	Meaning
quotation mark apostrophe	The term quotation mark is used only to refer to the double quotation mark character ("). The term apostrophe is used to refer to the single quotation mark character (′).
[OPTIONS (option, . . .)]	Except in OpenVMS file specifications, 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 can optionally 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. This convention applies to format (syntax) lines, not to code examples.
#	A # 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 I/O statements do not cover the effects of system-specific options. For details on mixed-language programming and system-specific options, see the *Kednos PL/I for OpenVMS Systems User Manual*.

Terminological Conventions

Information in this manual applies to the use of Kednos PL/I on the OpenVMS VAX and OpenVMS Alpha Operating Systems unless otherwise indicated.

The term *Kednos PL/I* refers to both Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha.

The terms "full PL/I" and "standard PL/I" refer to the ANSI standard PL/I, X3.53-1976.

1 Program Structure and Content

This chapter introduces the following elements of a PL/I program:

- The lexical elements (Section 1.1)
- The statements that make up a block and the general format and elements of a PL/I statement (Section 1.2)
- The format of a PL/I program (Section 1.3)
- The blocks that make up a program and their effects during program execution (Section 1.4)
- The PL/I data types (Section 1.5)

Future chapters discuss these topics in more detail.

1.1 Lexical Elements

This section describes the following topics:

- PL/I keywords
- Punctuation
- Identifiers
- Comments

1.1.1 Keywords

A keyword is a name that has a special meaning to PL/I when used in a specific context. In context, keywords identify statements, attributes, options, and other program elements. PL/I keywords are not reserved words, so it is possible to use them in a program in other than their keyword context.

PL/I has numerous keywords. Table A–1 describes the PL/I keywords, including brief identifications of their uses and valid abbreviations for the keywords that can be abbreviated.

1.1.2 Punctuation

PL/I recognizes punctuation marks in statements. The punctuation marks serve the following two functions:

- They specify arithmetic or other operations to be performed on expressions.
- They delimit and separate identifiers, keywords, constants, and statements.

For example:

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

In this statement, the equal sign (=), the addition operator (+), and the semicolon (;) delimit the identifiers A, B, and C, as well as define the operation to be performed. (Chapter 6 describes the effect of the various operators in expressions.)

Whenever you use a punctuation mark in a PL/I statement, you can precede or follow the character with any number of spaces (except in the case of an operator consisting of two characters, like >= or **, which must be entered without a space between the two characters). 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, all nonessential spaces are omitted; the parentheses and commas 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 1–1 lists all the punctuation marks recognized by PL/I.

Table 1–1 Punctuation Marks Recognized by PL/I

Category	Symbol	Meaning
Arithmetic operators	+	Addition or unary plus
	-	Subtraction or unary 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 (unary) and EXCLUSIVE OR (binary)
	&	Logical AND
	&:	Logical AND THEN
	|	Logical OR
	|:	Logical OR ELSE
Concatenation operator	||	String concatenation

Table 1–1 (Cont.) Punctuation Marks Recognized by PL/I

Category	Symbol	Meaning
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
	()	Encloses lists and extents; defines the order of evaluation of expressions; separates statement and option names from specific keywords; specifies a parameter list
	'	Delimits character strings and bit strings
Locator qualifier	->	Pointer resolution

The tilde (~) is equivalent to the circumflex (^), and the exclamation point (!) is equivalent to the vertical bar (|).

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 rules for entering spaces are:

— Between any identifiers, keywords, or constants

— Preceding or following punctuation marks that normally serve as delimiters, for example, tabs or commas

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.) Note that any tabs or spaces preceding NUED in the previous example will be included in the string.

1.1.3 Identifiers

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. The identifiers abc, ABC, Abc, and so on, all refer to the same object.
  - — The numeric digits 0 through 9.
  - — The underscore character (_).
  - — The dollar sign character ($).
- An identifier cannot contain any blanks, spaces, or hyphens.
- An identifier must begin with an alphabetic letter, a dollar sign ($), or an underscore (_). It cannot begin with a numeral.

Examples of valid identifiers are:

```
STATE
total
FICA_PAID_YEAR_TO_DATE
ROUND1
SS$_UNWIND
```

1.1.4 Comments

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

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

Except inside a string constant, wherever the starting characters (/*) appear in a program, the compiler ignores all text until it encounters the ending characters (*/). A comment can span several lines.

The rules for entering comments are:

- Except within a string constant, a comment can appear anywhere that a space can appear:
- A comment can contain any character except the pair */; comments cannot be nested.

The following are examples of comments:

```
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 the following:

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

To do this, precede the DECLARE statement with another /* pair, as follows:

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

The compiler will then ignore all text, including the DECLARE statement and the second /*, until it reaches the */.

1.2 Statements

A statement is the basic element of a PL/I procedure. Statements are used to do the following:

- Define and identify the structure of the program and the data that it acts upon (possibly including text from other files; see Section 10.2.13.)
- Request specific actions to be performed on data
- Control the flow of execution in a program

Table 1–2 and Table 10–1 provide summaries of PL/I statements. Detailed descriptions of these statements appear throughout this manual.

1.2.1 Statement Formats

The general format of a PL/I statement consists of an optional statement label, the body of the statement, and the required semicolon terminator.

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.

1.2.2 Statement Labels

The optional statement 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 (see Section 1.1.3) terminated by a colon. For example:

```
TARGET: A = A + B;
READ_LOOP: READ FILE (TEXT) INTO (TEMP);
```

A statement cannot have more than one label.

1.2.3 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. Examples of keyword statements are:

```
READ FILE (A) INTO (B);
GOTO LOOP;
DECLARE PRICE PICTURE '$$$99V.99';
```

In these 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 an identifer and an expression separated by an equal sign (=). For example:

```
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. For example:

```
IF A < B THEN GOTO COMPUTE;
     ELSE;
```

This IF statement shows a common use of the null statement: as the target of an ELSE clause.

1.2.4 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.

1.2.5 Summary of Statements by Function

You can group PL/I statements by function into the following categories.

Data Definition and Assignment Statements

The DECLARE statement defines variable names:

DECLARE identifier [attribute . . .];

The assignment statement gives a value to a variable or variables:

reference, 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	
LEAVE	SELECT	

Storage Allocation Statements

These statements acquire and control the use of storage in a PL/I program:

ALLOCATE
FREE

Table 1–2 gives a summary of the PL/I statements and their uses.

Table 1–2 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 or controlled 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 alternative 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 or controlled variable

Table 1–2 (Cont.) Summary of PL/I Statements

Statement	Use
GET	Obtains data from an external stream file or from a character-string expression
GOTO	Transfers control to a labeled statement
IF	Tests an expression and establishes actions to be performed based on the result of the test
LEAVE	Transfers control out of a DO group
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
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
SELECT	Tests a series of expressions and establishes the action to be performed based on the result of the test
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

1.3 Program Format

A PL/I program consists of a series of statements, which perform the following tasks:

- 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. You can organize statements into structural sequences of groups or blocks. Example 1–1 shows the structure of a PL/I program.

Example 1–1 Structure of a PL/I Program

```
SAMPLE: PROCEDURE OPTIONS (MAIN); ❶

DECLARE (X,Y,Z) FIXED, ❷
        MESSAGE CHARACTER(80),
        CALC ENTRY (FLOAT) RETURNS (FLOAT),
        TOTAL FLOAT;

         X = 0; ❸
         PUT SKIP LIST(MESSAGE);

FINISH: PROCEDURE; ❹
DECLARE TEXT (5) CHARACTER (20);

END FINISH: ❺
END SAMPLE;
```

Key to Example 1–1

❶ A PROCEDURE is the basic executable program unit.

❷ The declarations of variables in a procedure are usually, but not necessarily, placed at the beginning of the procedure.

❸ Executable statements are placed following variable declarations.

❹ Internal procedures may be placed anywhere.

❺ All procedures must terminate with END statements.

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, be on additional lines, or be written with more than one statement to a line.

Individual keywords or identifiers of a statement, however, must be confined to one line. Only a character-string constant (which must be enclosed in apostrophes) can be on more than one line.

PL/I programs are easier to read and comprehend if you follow a standard pattern in formatting. For example:

- Write source statements with no more than one statement per line.
- Use indentation to show the nesting level of blocks and DO-groups.

1.4 Blocks

PL/I is a block-structured language with each block composed of a sequence of PL/I statements. There are two types of blocks:

- 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 is a sequence of statements headed by a BEGIN statement (see Section 8.2) and terminated by an END statement (see Section 8.3). In general, you can use a begin block wherever a single PL/I statement would be valid. In some contexts, such as an ON-unit, a begin block is the only way to perform severa statements instead of one. A primary use of begin blocks is to localize variables. Because execution of a begin block causes a block activation, automatic variables declared within the begin block are local to it, and their storatge disappears when the block completes execution. Because BEGIN causes a block activation, a new stack frame will be generated in the same manner as a procedure statement. Thus, if you do not need this capability use a DO Group instead, as it is more efficient.

Scope of Names

The scope of a declaration of a name is that region of the program in which the name has meaning. A name has meaning in the following locations:

- The block in which it is declared
- Any blocks contained within the declaring block, as long as the name is not redeclared in the contained block
- Any procedure contained in the program, if the name is declared outside a procedure

Two or more declarations of the same name are not allowed in a single block unless one or more of the declarations are of structure members.

Two declarations of the same name in different blocks 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, otherwise unpredictable results will occur. Note that EXTERNAL is the default for declarations of ENTRY and FILE constants. It must be specified explicitly for variables.

The following example shows the scope of internal names:

	NAME	SCOPE
DECLARE Q STATIC FIXED;	Q	MAINP, ALPHA, BETA, and CALC
MAINP: PROCEDURE OPTIONS (MAIN);	MAINP	MAINP, ALPHA, BETA, and CALC
DECLARE (X, Y, Z) FIXED;	X, Y	MAINP, ALPHA, BETA, and CALC
	Z in MAINP	MAINP, ALPHA, and CALC
ALPHA: PROCEDURE;	ALPHA	MAINP, ALPHA, BETA, and CALC
BETA: BEGIN;	BETA	ALPHA, BETA
DECLARE Z FLOAT;	Z in BETA	BETA
GOTO ERROR;		
END BETA;		

ERROR:	ERROR	ALPHA, BETA
END ALPHA;		
CALC: PROCEDURE;	CALC	MAINP, ALPHA, and CALC
DECLARE (SUM, TOTAL) FLOAT;	SUM, TOTAL	CALC
END CALC;		
END MAINP;		

Declarations can appear outside procedures and, if contained within the same block, have meaning throughout all procedures contained in the block. However, if there are multiple blocks, declarations outside procedures must have the EXTERNAL attribute if they are to be recognized by all blocks and procedures in the program. For example:

File A.PLI

```
   DECLARE X FIXED EXTERNAL STATIC;
A: PROCEDURE OPTIONS(MAIN);
   DECLARE B ENTRY;
      .
      .
      .
   END A;
```

File B.PLI

```
B: PROCEDURE;
     .
     .
     .
   END B;
```

In this example, the variable X has meaning in both procedures. Because the two procedures are in two different files, X must be declared with the EXTERNAL attribute. If X is declared with the INTERNAL attribute, X is recognized only in the first procedure.

1.4.1 Begin Blocks

A begin block is a sequence of statements headed by a BEGIN statement (see Section 8.2) and terminated by an END statement (see Section 8.3). In general, you can use a begin block wherever a single PL/I statement would be valid. In some contexts, such as an ON-unit, a begin block is the only way to perform several statements instead of one. A primary use of begin blocks is to localize variables. Because execution of a begin block causes a block activation, automatic variables declared within the begin block are local to it, and their storage disappears when the block completes execution.

Another way to allow your program to perform several statements in place of one is to use a DO group (see Section 8.1). You should choose it when possible because it does not incur the overhead associated with block

activation. Use a begin block when there are declarations present or when you require multiple statements in an ON unit.

1.4.2 Procedure Blocks

A procedure is a sequence of statements (possibly including begin blocks and other procedures) headed by a PROCEDURE statement and terminated by an END statement. Unlike a begin block, which executes when control reaches it, a procedure executes only when it is specifically invoked. Invocation occurs in the following ways:

- Enter the DCL command RUN to invoke the main procedure of a PL/I program. This is either the procedure that has OPTIONS(MAIN) on its PROCEDURE statement or the first procedure encountered by the linker.
- Statements within a procedure can invoke other procedures. The CALL statement invokes a procedure as a subroutine. A function reference invokes a function, which is a procedure that returns a value for use in the evaluation of an expression.

A PL/I program must have at least one procedure, the main procedure. Any procedure, including the main procedure, can contain others; these are called internal procedures. A procedure that is not contained within any other is called an external procedure. The main procedure is always an external procedure.

Except for the main procedure, no procedure executes unless it is invoked by a CALL statement or a function reference. Chapter 7 discusses procedures in more detail.

1.4.3 Containment

As an example, block B is said to be contained in another block A if all of B's source text, from label (if any) to END statement inclusive, is between A's BEGIN or PROCEDURE statement and A's END statement. If block B is not contained in any other block within block A, then B is said to be immediately contained in A. For example:

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

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

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

1.4.4 Block Activation

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

You can only enter a procedure block with a CALL statement (see Section 7.4) 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 is entered when it is encountered during the normal flow of execution.

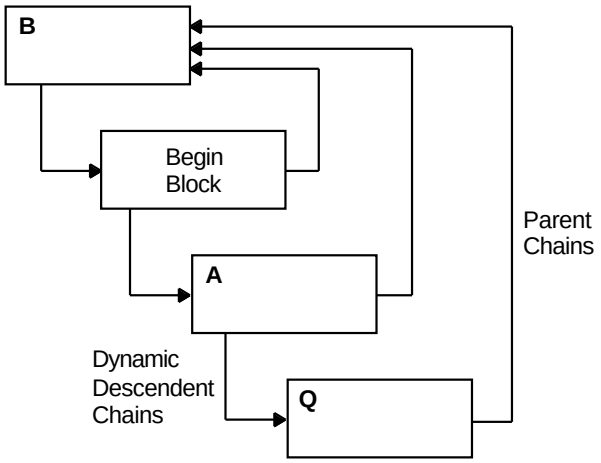
1.4.5 Relationship of Block Activations

During the execution of a program, many blocks can be simultaneously active. Two different relationships can be defined among block activations; they are the immediate dynamic descendance and the immediate parent activation. 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;
```

Figure 1–1 shows these relationships.

Figure 1–1 Relationship of Block Activations



In the immediate dynamic descendance relationship, a block activation is the immediate dynamic descendant of the block that invoked it. At a given time, the chain of immediate dynamic descendants includes all existing block activations, starting with the activation of the main procedure and terminating in the current block activation. For example, in Figure 1–1, the begin block is the immediate dynamic descendant of procedure B; the complete chain is B, begin block, A, Q. This chain is used for finding the applicable ON-unit when a condition is signaled.

The other relationship shown in Figure 1–1 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, which 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 1–1, the parent chain for the begin block, procedure A, and procedure Q leads directly back to the activation of B, because each of these blocks is immediately contained in B. This chain is used in interpreting references.

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

- If the block is an external procedure, it has no parent activation.
- 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 descendant of its immediate parent.
- If the block is an internal procedure invoked in block activation B by a reference to an entry constant (such as A in Figure 1–1) declared in block B, then the immediate parent of the new block activation of A is the activation of A in the parent chain starting at B.

- If the block is an internal procedure invoked by an entry variable, the parent activation is taken from the entry value. It was originally set when the complete entry value was generated by the assignment of an entry constant to an entry variable (Section 3.8 discusses entry data).

1.4.6 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.

1.5 Data and Variables

The statements in a PL/I program process data, generally in the form of variables that take on different values as the result of program execution. In PL/I, you must declare variables in a DECLARE statement before you can use them in other statements. Declaring a variable associates an identifier with a set of attributes and with a region of storage. Thus, when you declare a variable you must usually specify one or more data type attributes to be associated with it. (The concept of an attribute is more basic to PL/I than the concept of a data type.) Furthermore, you can specify how the variable is to be allocated by supplying a storage-class attribute in the declaration.

A few examples of PL/I attributes are BIT, CHARACTER, BINARY, DECIMAL, FILE, FLOAT, PRINT, UPDATE, and VALUE. For a complete alphabetic list of the PL/I attributes with their uses, see Section 2.2.

An identifier can refer to a single variable (called a scalar variable) or to a collection of related variables. Such a collection is called an aggregate. There are two kinds of aggregates:

- The array, in which all members have the same data type and are referenced by relative position
- The structure, in which the members can have different data types and are referenced in a hierarchical fashion

The following chapters provide information on these topics:

- Chapter 2 describes the DECLARE statement and the scope of a declaration.
- Chapter 3 describes the data types that you can specify for variables.
- Chapter 4 describes aggregates.

1.5.1 Preprocessor

PL/I supports an embedded lexical preprocessor, which recognizes a specific set of statements that are executed at compile time. These statements cause the PL/I compiler to include additional text in the source program or to change the values of constant identifiers at compile time.

Preprocessor statements are identified by a leading unquoted percent sign (%) and are terminated by an unquoted semicolon (;), except for %THEN and %IF statements. You can freely intermix preprocessor statements with the rest of the source program statements.

Table 10–1 lists the preprocessor statements. For additional information on the PL/I preprocessor, see Chapter 10.

2 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 following:

- The data type of the name, that is, whether it is a computational data item (such as a number or a string) or noncomputational program data (such as a file constant or label)
- 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 whether it is known in external blocks

A name is declared either explicitly in a DECLARE statement or implicitly by its appearance in a particular context. For example:

```
CALC: PROCEDURE;
```

This statement is an implicit declaration of the name CALC as an entry constant.

This chapter describes the DECLARE statement and data attributes.

2.1 DECLARE Statement

The DECLARE statement specifies the attributes associated with names.

The format of the DECLARE statement is:

{ DECLARE / DCL } declaration [,declaration, . . .];

declaration

One or more declarations consisting of identifiers and attributes. A declaration has the following format:

[level] declaration-item

level

Levels are used to specify the relationship of members of structures; if a level is present in the declaration, it must be written first.

declaration-item

A declaration-item has the following format:

{ identifier [(bound-pair, . . .)] / (declaration-item, . . .) / (identifier, . . .) [(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. Alternatively, 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.

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 list of declarations. For example, (2:10), lower bound is 2 and upper bound is 10. Bounds can be any valid PL/I expression.

The various formats are described individually in the following sections.

2.1.1 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, dollar signs ($), and underscores (_), but must begin with an alphabetic letter, dollar sign, or underscore.

attribute

One or more attributes of the name. Attribute keywords must be separated by spaces. They can appear in any order.

See Section 2.2 for a list of the valid attribute keywords and their meanings.

The following are examples of simple declarations:

```
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, receive the default attributes FIXED BINARY (31,0) AUTOMATIC. Note that the compiler issues a warning message whenever it gives these default attributes to a name.

2.1.2 Declarations Outside Procedures

You can declare a variable outside any procedure. Any variable so declared will be visible within all procedures contained by the module. The format for declarations outside procedures is the same as for other declarations, except that the storage-class attribute cannot be AUTOMATIC. If a storage-class is not specified or is specified as AUTOMATIC, the compiler

will issue a warning and supply the STATIC attribute. The following example shows the use of this type of declaration:

```
DECLARE A STATIC FIXED BINARY(31);
   .
   .
   .
FIRST: PROCEDURE;
    DECLARE B FIXED BINARY(31);
        .
        .
        .
END FIRST;

SECOND: PROCEDURE;
    DECLARE C FIXED BINARY(31);
        .
        .
        .
END SECOND;
```

In this example, variable A is visible in both the FIRST and SECOND procedures, but variables B and C are visible only in their containing procedures.

2.1.3 Multiple Simple Declarations

Multiple simple 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.

The following is an example of multiple declarations:

```
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.

2.1.4 Factored Simple Declarations

When two or more names have common attributes, 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 and separate them with commas. The attributes that follow the parenthetical list of names are applied to all the named identifiers.

The following are examples of factored declarations:

```
DECLARE (COUNTER, RATE, INDEX) FIXED BINARY (7) INITIAL (0);
DECLARE (INPUT_MESSAGE, OUTPUT_MESSAGE, PROMPT)
        CHARACTER (80) VARYING;
```

In these declarations, the variables COUNTER, RATE, and INDEX share the attributes FIXED BINARY (7) and are given the initial value of zero. 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 the following format:

DECLARE (declaration-item, declaration-item [,declaration-item])
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 STREAM OUTPUT 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 file.

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

where each bound pair has the following format:

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

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.

You can use the asterisk (*) 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.)

For example:

```
DECLARE SALARIES(100) FIXED DECIMAL(7,2);
```

This statement 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);
```

This 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 Section 4.1.1.

2.1.6 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 / DCL } level declaration-item [,level declaration-item . . .];

Each declaration specifies a member of the structure and must be preceded by a level number. As shown in the following example, 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 other structures. The major structure name is declared as structure level 1; minor members must be declared with level numbers greater than 1.

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST CHARACTER(80) VARYING,
            3 FIRST CHARACTER(80) VARYING,
          2 SALARY FIXED DECIMAL(7,2);
```

This statement declares a structure named PAYROLL.

Alternatively, because the last and first names have the same attributes, the same structure can be declared as follows:

```
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 Section 4.2.1.

2.2 Attributes

Attributes define and describe the characteristics of names used in a PL/I program. Each name in a PL/I program has a set of attributes associated with it. You can specify attributes in any of the following contexts:

- In a DECLARE statement for an identifier. These attributes are specified either by keyword or by syntax. For example:

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

In this declaration, the keyword attribute CHARACTER is associated with the identifier SIGNAL. The syntax length attribute of the variable is specified in parentheses following the CHARACTER keyword.
In this manual, keyword attributes are shown in format lines in uppercase letters. Attributes given by syntax are shown in lowercase letters.

- 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 chapter gives its syntax and abbreviation (if any) and describes related and conflicting attributes. See Table 2–1 for a concise alphabetic summary of PL/I attributes.

Computational Data Type Attributes

The attributes that define arithmetic and string data are:

```
CHARACTER [ (length) ] [ VARYING    ]
                       [ NONVARYING ]
BIT [ (length) ] [ ALIGNED   ]
                 [ UNALIGNED ]
{ FLOAT } { BINARY  }
{ FIXED } { DECIMAL }
 [ [PRECISION] (precision [,scale-factor]) ]
PICTURE 'picture'
```

You can specify these attributes for all elements of an array and for individual members of a structure.

Noncomputational Data Type Attributes

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

```
AREA
CONDITION
ENTRY [VARIABLE]
FILE [VARIABLE]
LABEL
```

```
OFFSET
POINTER
```

Storage-Class and Scope Attributes

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

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

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

Member Attributes

You can apply the following attributes to the major or minor members of a structure:

```
LIKE
MEMBER
REFER
STRUCTURE
TYPE
UNION
```

File Description Attributes

You can apply the following attributes to file constants and used in OPEN statements:

```
ENVIRONMENT(option, . . . )
{ RECORD [KEYED] } { INPUT          }
{ STREAM         } { OUTPUT [PRINT] }
                   { UPDATE         }
{ DIRECT     }
{ SEQUENTIAL }
```

Entry Name Attributes

You can apply the following attributes to identifiers of entry points:

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

Nondata Type Attributes

You can apply the following attributes to data declarations:

```
ALIGNED
DIMENSION
UNALIGNED
```

Table 2–1 lists the PL/I attributes. The sections following this table describe each attribute in detail.

Table 2–1 Alphabetic Summary of PL/I Attributes

Attribute	Use
ALIGNED	Requests alignment of bit-string variables in storage
ANY	Indicates that a parameter (of an external procedure not written in PL/I) can have any data type
AREA [(extent)]	Defines an area 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 } [(precision[,scale-factor])]	Defines a binary base for arithmetic data
BIT [(length)]	Defines bit-string data
BUILTIN	Defines a built-in function name
{ CHARACTER | CHAR } [(length)]	Defines character-string data
{ CONDITION | COND } (condition-name)	Defines an identifier as a condition name
{ CONTROLLED | CTL }	Defines a variable whose storage is allocated and freed in successive and fixed-sequence generations
{ DECIMAL | DEC } [(precision[,scale-factor])]	Defines a decimal base for arithmetic data
{ DEFINED | DEF } (variable-reference)	Indicates that a variable will share the storage allocated for another variable

Table 2–1 (Cont.) Alphabetic Summary of PL/I Attributes

Attribute	Use
{ DESCRIPTOR | DESC }	Requests that an argument be passed to an external non-PL/I procedure by descriptor
{ DIMENSION | DIM } (bound-pair, . . .)	Indicates that a variable is an array, and defines the number and extent of its dimensions
DIRECT	Specifies that a file will be only accessed randomly
ENTRY (descriptor, . . .)	Describes an external procedure and its parameters
{ ENVIRONMENT | ENV } (option, . . .)	Specifies system-dependent information about a file
{ 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 [(precision[,scale-factor])]	Defines a fixed-point arithmetic variable
FLOAT [(precision)]	Defines a floating-point arithmetic variable
GLOBALDEF [(psect-name)]	Defines an external variable and optionally specifies the program section in which the variable will reside
GLOBALREF	Declares an external variable which 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
KEYED	Specifies that a file can be accessed randomly by key
LABEL	Defines a label variable
LIKE structure-reference	Copies the declaration of a structure to another structure variable
LIST	Specifies that a parameter can accept a list of actual parameters, of arbitrary length
MEMBER	Specifies that an item is a member of a structure
{ NONVARYING | NONVAR }	Specifies that the length of a string is nonvarying
OFFSET [(area-reference)]	Defines an offset variable
OPTIONAL	Specifies, in the declaration of a formal parameter, that the actual parameter need not be specified in a call
OUTPUT	Specifies that a file will be used for output
{ PARAMETER | PARM }	Indicates that a variable will be assigned a value when it is used as an argument to a procedure
{ 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

Table 2–1 (Cont.) Alphabetic Summary of PL/I Attributes

Attribute	Use
{ PRECISION / PREC } [(precision[,scale-factor])]	Specifies the number of digits in an arithmetic variable and, with fixed-point 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
REFER	Defines dynamically self-defining structures
{ REFERENCE / REF }	Requests that an argument be passed to an external non-PL/I procedure by reference
RETURNS (returns-descriptor)	Specifies that an external entry is a function and describes the value returned by it
{ SEQUENTIAL / SEQL }	Specifies that a file can be accessed sequentially
STATIC	Requests static allocation of storage
STREAM	Specifies that a file will be accessed by stream I/O statements
STRUCTURE	Specifies that a variable is a structure variable
TRUNCATE	Specifies, in a declaration of a formal parameter, that the actual parameter list can be truncated at the point where this argument should occur
TYPE	Copies declarations of structures, scalars, and arrays to another variable
{ UNALIGNED / UNAL }	Specifies nonalignment for bit-string variables in storage
UNION	Indicates that a variable will share the storage allocated for another variable
UPDATE	Specifies that records in a file can be rewritten or deleted
{ VALUE / VAL }	Requests either that a global symbol be accessed by value rather than by reference, or that an argument be passed to a procedure by immediate value
VARIABLE	Defines variable entry and file data
{ VARYING / VAR }	Defines a varying-length character string

2.2.1 ALIGNED Attribute

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

ALIGNED

You can 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. 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. Specifying ALIGNED is not recommended with character strings, as all character strings are byte-aligned.

Restriction

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

2.2.2 ANY Attribute

The ANY attribute specifies that an entry's corresponding argument can be of any data type. This attribute is applicable only to the declaration of entry names denoting non-PL/I procedures. The format of the ANY attribute is:

```
ANY [ VALUE          ]
    [ CHARACTER(*)   ]
    [ REFERENCE      ]
    [ DESCRIPTOR     ]
```

Restrictions

If you specify ANY for a parameter, you cannot specify any data-type attributes for that parameter except CHARACTER(*). If ANY is used by itself, the parameter is passed by reference. If ANY is used with VALUE, the parameter is passed by immediate value. If ANY is used with CHARACTER(*), the parameter is passed by character descriptor.

Example

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

This statement identifies the system service procedure SYS$SETEF and indicates that the procedure accepts a single argument, which can be of any data type, to be passed by value. (Note that PLI$STARLET contains declarations for all system services, RTL routines, and utility routines.)

2.2.3 AREA Attribute

The AREA attribute defines an area variable. The format of the AREA attribute is:

```
AREA [(extent)]
```

extent

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

- If AREA is specified for a static variable declaration, extent must be a restricted integer expression. A restricted integer expression is one that yields only integral results and has only integral operands. Such

an expression can use only the addition (+), subtraction (–), and multiplication (*) operators.

- If AREA is specified in the declaration of a parameter or in a parameter descriptor, you can specify extent as an integer constant or as an asterisk (*).
- If AREA is specified for an automatic or based variable, you can specify extent as an integer constant or as an expression. For automatic variables, the extent expression must not contain any variables or functions declared in the same block, except for parameters.
- If no extent is specified for the area, a default of 1024 bytes is provided. Kednos recommends explicitly specifying a size, because the default varies considerably between PL/I implementations.

Restrictions

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

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

Restriction

The AUTOMATIC attribute conflicts with the following attributes (the specification of which implies that storage allocation is not to be automatic):

BASED	GLOBALREF
CONTROLLED	PARAMETER
DEFINED	READONLY
EXTERNAL	STATIC
GLOBALDEF	

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 Chapter 5.

2.2.5 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. 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 made to a based variable without an explicit pointer or offset qualifier, the reference is evaluated to obtain the pointer or offset value.

Restriction

The following attributes conflict with the BASED attribute:

AUTOMATIC	GLOBALREF
CONTROLLED	PARAMETER
DEFINED	READONLY
EXTERNAL	STATIC
GLOBALDEF	VALUE

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

See Section 5.5 for more information on the BASED variable.

2.2.6 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[,scale])]
FLOAT  [(precision)]
```

FIXED indicates a fixed-point binary value and FLOAT indicates a floating-point binary value.

For a fixed-point binary value, the precision specifies the number of bits representing an integer and must be in the range 1 through 31. For a fixed-point binary value, the scale factor represents the number of bits to the right of the binary point and must be in the range -31 through 31. The magnitude of the scale factor must be less than or equal to the specified precision.

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:

- For OpenVMS VAX systems: 1 through 113
- For OpenVMS Alpha systems: 1 through 53

The maximum floating-point binary precision is always 113 for OpenVMS VAX and 53 for OpenVMS Alpha. The default values applied to the BINARY attribute are:

Attributes Specified	Defaults Supplied
BINARY	FIXED (31,0)
BINARY FIXED	(31,0)
BINARY FLOAT	(24)

Restrictions

The BINARY attribute directly conflicts with any other data-type attribute.

2.2.7 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 you do not specify a length, the default length is 1 bit. The length must be in the range 0 through 32,767.

The rules for specifying the length are:

- If the attribute is specified for a static variable declaration or in a returns descriptor, length must be a restricted integer expression. A restricted integer expression is one that yields only integral results and has only integral operands. Such an expression can use only the addition (+), subtraction (–), and multiplication (*) operators.
- If the attribute is specified in the declaration of a parameter or in a parameter descriptor, you can specify length as a restricted integer expression or as an asterisk (*).
- If the attribute is specified for an automatic, based, controlled, or defined variable, you can specify length 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 in parentheses must 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.

Restriction

The BIT attribute directly conflicts with any other data-type attribute.

2.2.8 BUILTIN Attribute

The BUILTIN attribute indicates that the name declared 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 format of the BUILTIN attribute is:

BUILTIN

Use the BUILTIN attribute 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.

You also use the BUILTIN attribute 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.

Restriction

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

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());
```

2.2.9 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 0 through 32,767 characters.

The rules for specifying the length are:

- If the attribute is specified for a static variable declaration or in a returns descriptor, length must be a restricted integer expression.
- If the attribute is specified in the declaration of a parameter or in a parameter descriptor, you can specify length as a restricted integer expression or as an asterisk (*).
- If the attribute is specified for an automatic, based, or defined variable, you can specify length 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, NONVARYING, ALIGNED, or UNALIGNED.

Restriction

The CHARACTER attribute directly conflicts with any other data-type attribute.

2.2.10 CONDITION Attribute

You can optionally use the CONDITION attribute in a declaration to specify that the variable name is a condition name. You can specify INTERNAL or EXTERNAL scope attributes with the CONDITION attribute. The default scope is external. The format of the CONDITION attribute is:

{ CONDITION / COND } (condition-name)

condition-name

Name used for ON units to handle programmer-defined conditions.

2.2.11 CONTROLLED Attribute

The CONTROLLED attribute causes a variable's actual storage to be allocated and freed dynamically in generations, only the most recent of which is accessible to the program. The format of the CONTROLLED attribute is:

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

Restrictions

The following attributes conflict with the CONTROLLED attribute:

AUTOMATIC
BASED
DEFINED
GLOBALDEF
GLOBALREF
READONLY
STATIC
VALUE
PARAMETER

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

See Section 5.6 for more information on the CONTROLLED variable.

2.2.12 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)
```

FIXED indicates a fixed-point value, and FLOAT indicates a floating-point decimal value.

(precision[,scale-factor])

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 through 31; the scale factor, if specified, must be greater than or equal to 0 and less than or equal to the specified precision.

The precision for a floating-point decimal value must be in the range:

- For OpenVMS VAX systems: 1 through 34
- For OpenVMS Alpha systems: 1 through 15

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)

Restrictions

The DECIMAL attribute conflicts with any other data-type attribute.

2.2.13 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 another base variable. The DEFINED attribute provides a way to access the same data using different names. The format of the DEFINED attribute is:

{ DEFINED / DEF }(variable-reference)

variable-reference

A reference to a variable that has storage associated with it. The variable must not have the BASED, CONTROLLED, or DEFINED attribute. The variable and the declared variable must satisfy the rules given for defined variables in Section 5.8.

The DEFINED attribute can optionally specify a position within the referenced variable at which the definition begins. For example:

```
DECLARE ZONE CHARACTER(10)
      DEFINED(ZIP) POSITION(4);
```

Restrictions

The following attributes conflict with the DEFINED attribute:

AUTOMATIC	BASED	CONTROLLED
EXTERNAL	GLOBALDEF	GLOBALREF
INITIAL	PARAMETER	READONLY
STATIC	UNION	VALUE

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

See Section 5.8 for more information on defined variables.

2.2.14 DESCRIPTOR Attribute

The DESCRIPTOR attribute forces a parameter to be passed by descriptor to a non-PL/I procedure.

The format of the DESCRIPTOR attribute is:

{ DESCRIPTOR | DESC }

Restriction

You can use the DESCRIPTOR attribute only in parameter descriptors

See Section 7.5.1 for more information on rules for specifying parameters.

2.2.15 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:

[DIMENSION | DIM] (bound-pair, . . .)

bound-pair

One or two expressions that indicate the number of elements in a single dimension of the array. You must specify the list of bound pairs immediately following the name of the identifier in the array declaration if the optional keyword DIMENSION or DIM is omitted; otherwise, you must specify the list of bound pairs immediately following the keyword DIMENSION or DIM. See the following examples.

The maximum number of dimensions allowed is eight.

A bound pair can be specified:

- [lowerbound:]upperbound

  This format of a bound pair specifies the minimum and maximum subscripts that can be used for the dimension. The number of elements is:

  $(upperbound - lowerbound) + 1$

  If the lower bound is omitted, it is assumed to be 1.

- *

  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.

The following two declarations are exactly equivalent:

```
DECLARE A(10) FIXED BIN;
DECLARE A FIXED BIN DIMENSION(10);
```

The following two declarations are also equivalent:

```
DECLARE B(1:5,1:5) FLOAT DEC;
DECLARE B DIM(1:5,1:5) FLOAT DEC;
```

2.2.16 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 format of the DIRECT attribute is:

DIRECT

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_ID)

To to access a file both randomly and sequentially, use the SEQUENTIAL attribute instead of DIRECT.

Restriction

The DIRECT attribute conflicts with the SEQUENTIAL, STREAM, and PRINT attributes.

2.2.17 ENTRY Attribute

The ENTRY attribute declares a constant or variable whose value is an entry point and 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 entry. 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.

The following rules apply to the specification of a parameter descriptor for an array or structure:

- If the parameter is a structure, the level number must precede the attributes for each member.
- You must specify extents for a parameter using only integer constants, restricted integer expressions, or asterisks (*).
- You cannot specify storage-class attributes.

OPTIONS (VARIABLE)

An option indicating that you can invoke the specified external procedure 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 *Kednos PL/I for OpenVMS Systems User Manual*.

RETURNS (returns-descriptor)

For an entry that is invoked as a function reference, an option giving the data type attributes of the function value returned. 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.

Restrictions

You cannot declare internal entry constants 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: a varying-length character string with a maximum length of 40 and a fixed-point binary value. The RETURNS attribute indicates that COPYSTRING is invoked as a function and that it returns a character string of any length.

2.2.18 ENVIRONMENT Attribute

The ENVIRONMENT file description attribute is used in DECLARE, OPEN, and CLOSE statements to specify options that define file characteristics specific to the OpenVMS file system and options that request special processing not available in the standard PL/I language. The format of the ENVIRONMENT attribute is:

{ ENVIRONMENT / ENV }(option, . . .)

option, . . .
One or more keyword options separated by commas.

Summary of Options

The following items with asterisks (*) are options you can specify in a CLOSE statement.

APPEND
BACKUP_DATE(variable-reference)
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
INITIAL_FILL

MAXIMUM_RECORD_NUMBER(expression)
MAXIMUM_RECORD_SIZE(expression)
MULTIBLOCK_COUNT(expression)
MULTIBUFFER_COUNT(expression)
NO_SHARE
OWNER_GROUP(expression)
OWNER_ID(expression)
OWNER_MEMBER(expression)
OWNER_PROTECTION(character-expression)
PRINTER_FORMAT
READ_AHEAD
READ_CHECK
RECORD_ID_ACCESS
RETRIEVAL_POINTERS(expression)
REVISION_DATE(variable-reference)*
REWIND_ON_CLOSE*
REWIND_ON_OPEN
SCALARVARYING
SHARED_READ
SHARED_WRITE
SPOOL*
SUPERSEDE
SYSTEM_PROTECTION(character-expression)
TEMPORARY
TRUNCATE
USER_OPEN(entry-name)
WORLD_PROTECTION(character-expression)
WRITE_BEHIND
WRITE_CHECK

The previous list of options to the ENVIRONMENT attribute are described in detail in the *Kednos PL/I for OpenVMS Systems User Manual*.

You can specify all ENVIRONMENT options in OPEN statements. You can also specify all ENVIRONMENT options except those that require variable references in DECLARE statements. Certain disposition options (noted in the list) can be specified in CLOSE statements.

Some ENVIRONMENT options require you to specify a value. In a DECLARE statement, you must use a literal constant to supply the value required. In OPEN and CLOSE statements, however, you can use expressions (including but not limited to literal constants) to supply the values.

Any option that does not require a value can optionally be specified with a Boolean 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 can be true or false at run time. Boolean values must be specified as constants in DECLARE statements. You can specify Boolean values as expressions (including constants) in OPEN statements and CLOSE statements.

2.2.19 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 (declarations that contain the ENTRY attribute but not the VARIABLE attribute). For variables, the EXTERNAL attribute implies the STATIC attribute.

Restrictions

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.

2.2.20 FILE Attribute

The FILE attribute declares a file constant or file variable. The format of the FILE attribute is:

FILE

The FILE attribute is implied by any of the following file description attributes:

DIRECT	OUTPUT	SEQUENTIAL
ENVIRONMENT	PRINT	STREAM
INPUT	RECORD	UPDATE
KEYED		

See Table 9–2 for definitions of these file description attributes.

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 FILE attribute is used to declare a variable or parameter, no file description attributes may be specified. If the VARIABLE attribute is not specified, no storage-class attributes are allowed.

2.2.21 FIXED Attribute

The FIXED attribute indicates that the variable so declared is arithmetic with a fixed number of fractional digits. Such variables are called fixed-point (as opposed to floating-point) variables because the decimal point and binary point are fixed relative to the representation of the value. The format of the FIXED attribute is:

```
FIXED  [(precision[,scale-factor])]
```

precision

The precision is the number of decimal or binary digits used to represent values of the variable.

scale-factor

Scale factor indicates how much of the precision is to be used for fractional digits.

When you specify the FIXED attribute in a DECLARE statement, you can specify either the BINARY or the DECIMAL attribute to indicate a binary or decimal fixed-point variable. For example, the attributes FIXED BINARY(31,5) define a variable that takes fixed-point binary values of up to a maximum of 31 bits, 5 of which are fractional. The attributes FIXED DECIMAL(10,2) define a variable that takes fixed-point decimal values of up to 10 decimal digits, 2 of which are fractional. PL/I supplies default attributes for attributes that you do not specify (as shown in the following table).

You normally use fixed-point binary data to represent integers. The precision of a fixed-point binary variable must be in the range 1 through 31. The scale factor can be in the range -31 through 31.

You can also use fixed-point decimal data, which can represent larger absolute values. You use fixed-point 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 through 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, an integer variable is declared).

The default values given for unspecified related attributes are:

Attributes Specified	Defaults Supplied
FIXED	BINARY (31,0)
FIXED BINARY	(31,0)
FIXED DECIMAL	(10,0)

Restriction

The FIXED attribute directly conflicts with all data-type attributes except BINARY and DECIMAL.

2.2.22 FLOAT Attribute

The FLOAT attribute indicates that a variable is a floating-point arithmetic item. The format of the FLOAT attribute is:

```
FLOAT  [(precision)]
```

precision

You can specify the precision within the following ranges:

- For OpenVMS VAX systems: the range for a floating-point binary variable is 1 through 113. The range for a floating-point decimal variable is 1 through 34.
- For OpenVMS Alpha systems: the range for a floating-point binary variable is 1 through 53. The range for a floating-point decimal variable is 1 through 15.

When you specify the FLOAT attribute in a DECLARE statement, you can specify either the BINARY or the DECIMAL attribute. The default values given for unspecified related attributes are:

Attributes Specified	Defaults Supplied
FLOAT	BINARY (24)
FLOAT BINARY	(24)
FLOAT DECIMAL	(7)

Restriction

The FLOAT attribute directly conflicts with all data-type attributes except BINARY and DECIMAL.

2.2.23 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 have up to 31 characters, which can consist of the alphanumeric characters, dollar signs ($), and underscores (_). The first character cannot be 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.

Restrictions

The GLOBALDEF attribute conflicts with the GLOBALREF and INTERNAL attributes. GLOBALDEF cannot be used with ENTRY constants.

Only one procedure in a program can declare a particular external variable with the GLOBALDEF attribute.

For complete details on using the GLOBALDEF attribute to declare global external symbols, see Section 5.4.

2.2.24 GLOBALREF Attribute

The GLOBALREF attribute indicates that the declared name is a global symbol defined in an external procedure. The format of the GLOBALREF attribute is:

```
GLOBALREF
```

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.

Restriction

The GLOBALREF attribute conflicts with the INITIAL, GLOBALDEF, and INTERNAL attributes. If GLOBALREF is specified with the FILE attribute, you cannot specify any other file description attributes.

See Section 5.4 for information about using this attribute.

2.2.25 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 . . . ]) }
{ INIT    } { ((*) valid-expression)                     }
```

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 1 for a scalar variable and must not exceed the number of elements of an array variable. Each initial element must have one of the following forms:

- string-constant
- (replication-factor) string-constant
- (iteration-factor) (string-constant)
- (iteration-factor) ((replication-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.

An asterisk (*) following the iteration factor specifies that the corresponding array elements are to be skipped during the initialization.

You can use a replication factor in combination with an iteration factor in initializing a string constant. For example, the following two statements are equivalent:

```
INITIAL ((10)('ABCABC'))
INITIAL ((10)((2)'ABC'))
```

The first statement uses an iteration factor exclusively; the second statement combines an iteration factor of 10 with a replication factor of 2.

A string constant must be parenthesized if it is used with an iteration factor, because this set of parentheses prevents the iteration factor from being interpreted as a string replication factor.

```
INITIAL ((10)'ABC')
```

For example, the initial value is interpreted as a string replication factor, not an iteration factor, and cannot be used to initialize a whole array.

(*) valid-expression

A construct that initializes all elements of an array to the same value by means of the asterisk iteration factor. The expression must evaluate to a value that is valid for assignment to the initialized array. If the expression is a string constant, it must be parenthesized so that the asterisk iteration factor is not interpreted as a string replication factor. The possible expressions are:

- (string-constant)
- ((replication-factor) string-constant)
- arithmetic-constant
- scalar-reference
- (scalar-expression)
- *

An asterisk following the asterisk iteration factor results in no initializations being performed.

Examples

The following are examples of declarations that include the INITIAL attribute:

```
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());

DECLARE X(10,5) FIXED BIN(31) INITIAL ((*) -2);  /* Initializes all 50
                                                    elements to -2    */

DECLARE 1 A(10),
          2 B(10),
            3 C(10) FIXED BIN(31) INITIAL ((*) 4);  /* Initializes all
                                                       1000 elements
                                                       to 4           */

DECLARE A(10) FIXED INIT ((5) 1,(5) 2);          /* Initializes the first
                                                    5 elements to 1 and
                                                    the second 5 elements
                                                    to 2              */
```

The following declaration is not valid, because the asterisk iteration factor cannot be used to initialize part of an array; it can only be used to initialize all elements of the array to the same value:

```
DECLARE A(10) FIXED INIT ((5) 1,(*)2);           /* Invalid use of asterisk
                                                    iteration factor  */
```

Restrictions

You cannot specify the INITIAL attribute for a structure variable. You must individually initialize the members of the structure.

You cannot specify the INITIAL attribute for a variable or member of a structure that has any of the following attributes:

DEFINED
ENTRY
FILE
LABEL
PARAMETER
UNION

If the initialized variable is STATIC, only constants, restricted expressions, and references to the NULL or EMPTY built-in functions are allowed. You can use these initial values with a constant iteration factor.

Variables and functions (except for parameters) occurring in an initial element (for automatic variables) must not be declared in the same block as the variable being initialized.

2.2.26 INPUT Attribute

The INPUT file description attribute indicates that the associated file is to be an input file. The format of the INPUT attribute is:

INPUT

Specify the INPUT attribute on a DECLARE statement for a file constant or on an OPEN statement to access the file for reading.

You can specify the INPUT attribute with either the STREAM or the RECORD attribute. For a stream file, INPUT indicates that the file will be accessed with GET statements. For a record file, INPUT indicates that the file will be accessed only with 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 can be applied to files, see Table 9–2.

The INPUT attribute can be supplied by default for a file, depending on the context of its opening. See Section 9.1.3.3 for more information.

Restriction

The INPUT attribute conflicts with the OUTPUT, UPDATE, and PRINT attributes and with any data-type attribute other than FILE.

2.2.27 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 only need to use the INTERNAL attribute to explicitly declare the scope of a file constant as internal. File constants, by default, have the EXTERNAL attribute.

Restriction

The INTERNAL attribute directly conflicts with the EXTERNAL, GLOBALDEF, and GLOBALREF attributes.

2.2.28 KEYED Attribute

The KEYED file description attribute indicates that you can randomly access records in the specified file. 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 can be applied to files, see Table 9–2.

Restriction

The KEYED attribute conflicts with the STREAM attribute and with any data-type attributes other than FILE.

2.2.29 LABEL Attribute

The LABEL attribute declares a label variable; it indicates that values given to the variable will be statement labels. The format of the LABEL attribute is:

```
LABEL
```

Restriction

You cannot specify the LABEL attribute with any other data-type attribute, the INITIAL attribute, or any file description attributes.

2.2.30 LIKE Attribute

The LIKE attribute copies the member declarations contained within a major or minor structure declaration into the structure variable to which it is applied. The format of the LIKE attribute is:

```
level-number  identifier  [attributes]  LIKE  reference
```

level-number
The level number to which the declarations in the reference are copied.

identifier
The variable to which the declarations in the reference are to be copied. The identifier must be preceded by a level number.

attributes
Storage class or dimensions appropriate for the level number. You can specify a storage class and dimensions with a major structure, or you can specify dimensions with a minor structure.

reference
The name of a major or minor structure that is known in the current block.

The LIKE attribute causes the structuring and member declarations of its reference to be copied, but not the name, storage class, or dimensioning (if any) of the reference. The exception to this rule is that the UNION attribute is propagated in a LIKE declaration. While logical structuring is copied, the level numbers themselves are not copied.

You can use the LIKE attribute on a structure already containing the LIKE attribute.

2.2.31 LIST Attribute

The LIST attribute is used in the declaration of a formal parameter to indicate that the parameter can accept a list of actual parameters of arbitrary length. This list must contain at least one argument. To allow a list of zero or more arguments, you must declare the formal parameter with both the TRUNCATE attribute and the LIST attribute. The format of the LIST attribute is:

LIST

The LIST attribute is valid only on formal parameters of external procedures. It is not supported for PL/I procedures. (To simulate list parameters in PL/I, use asterisk-extent arrays.)

You can only use the LIST attribute for the last formal parameter in an argument list.

Examples

```
DCL NUMBER FIXED BINARY;
DCL LIST_PROC1 ENTRY (FIXED BINARY,FIXED BINARY LIST);
DCL LIST_PROC2 ENTRY (FIXED BINARY,FIXED BINARY LIST TRUNCATE);

CALL LIST_PROC1 (NUMBER,NUMBER);
CALL LIST_PROC1 (NUMBER,NUMBER,NUMBER);

CALL LIST_PROC2 (NUMBER);
CALL LIST_PROC2 (NUMBER,NUMBER,NUMBER,NUMBER);
```

2.2.32 MEMBER Attribute

You can optionally specify the MEMBER attribute in the declaration of a structure member (minor structure). A structure member has the MEMBER attribute implicitly. The format of the MEMBER attribute is:

MEMBER

Restriction

The MEMBER attribute cannot be used with a major structure (that is, a structure variable with level 1).

2.2.33 NONVARYING Attribute

The NONVARYING attribute keyword explicitly states that a bit-string or character-string variable has a fixed length, not a varying length. Because NONVARYING is the default for bit and character strings, it need not be specified. The format of the NONVARYING attribute is:

{ NONVARYING }
{ NONVAR }

2.2.34 OFFSET Attribute

The OFFSET attribute declares a variable that will be used to reference a based variable within an area. The format of the OFFSET attribute 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.

Examples

```
DECLARE MAP_SPACE AREA (40960),
        MAP_START OFFSET (MAP_SPACE),
        MAP_LIST(100) CHARACTER(80) BASED (MAP_START);
```

These declarations define an area named MAP_SPACE, an offset variable 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.

Restriction

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.

2.2.35 OPTIONAL Attribute

The OPTIONAL attribute indicates that an actual parameter need not be specified in a call. If the actual parameter is not specified, a placeholder for it must be specified, and PL/I will pass a longword zero as the actual parameter in that position. The format of the OPTIONAL attribute is:

OPTIONAL

Example

```
DCL E ENTRY (FIXED,FIXED OPTIONAL);
CALL E(1,2);
CALL E(1,);
```

2.2.36 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. The format of the OUTPUT attribute is:

OUTPUT

Specify the OUTPUT attribute on a DECLARE statement for a file constant or on an OPEN statement to access the file for writing. You can specify the OUTPUT attribute with either the STREAM or the RECORD attribute. For a stream file, OUTPUT indicates that the file will be accessed with PUT statements. For a record file, OUTPUT indicates that the file will be accessed with only WRITE statements.

Examples

```
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 you can apply to files and the effects of combinations of these attributes, see Chapter 9.

Restriction

The OUTPUT attribute conflicts with the INPUT and UPDATE attributes and with any data-type attributes other than FILE. The OUTPUT attribute also conflicts with ENVIRONMENT(INDEXED).

2.2.37 PARAMETER Attribute

A variable occurring in the parameter list of a PROCEDURE or ENTRY statement has the PARAMETER attribute implicitly. You can optionally use the PARAMETER keyword in the declaration of a variable name to state explicitly that it is a parameter. The format of the PARAMETER attribute is:

{ PARAMETER }
{ PARM }

Example

The following example uses the PARAMETER keyword:

```
TEST: PROC( A, B );
    DCL A CHAR(*) PARAMETER;
    DCL B FIXED BIN PARM;
        .
        .
        .
```

Refer to Section 7.5 for a discussion on parameters.

2.2.38 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 format of the PICTURE attribute is:

{ PICTURE } 'picture'
{ PIC }

picture

A string of picture characters that define the representation of the variable.

See Section 3.2.5.1 for detailed information about picture characters, syntax, and examples.

Restriction

The PICTURE attribute conflicts with all other data-type attributes.

2.2.39 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 }

Restriction

The POINTER attribute conflicts with all other data-type attributes.

2.2.40 POSITION Attribute

The POSITION attribute specifies the character or bit position in a defined variable's base at which the defined variable begins. The format of the POSITION attribute is:

{ POSITION / POS }(expression)

expression

An integer expression that specifies a position in the base. A value of 1 indicates the first character or bit.

Restriction

You can specify the POSITION attribute only in connection with DEFINED and only when the defined variable satisfies the rules for string overlay defining (see Section 5.8.2).

2.2.41 PRECISION Attribute

The PRECISION attribute specifies the maximum number of decimal or binary digits in a number. You can specify the precision of an arithmetic variable in any of the following formats, depending on the numeric base of the data item. The formats of the PRECISION attribute are:

BINARY [FIXED] [[PRECISION] (precision[,scale-factor])]
[BINARY] FLOAT [[PRECISION] (precision)]
DECIMAL [FIXED] [[PRECISION] (precision[,scale-factor])]
DECIMAL FLOAT [[PRECISION] (precision)]

precision

You can abbreviate the keyword PRECISION to PREC, or you can omit it entirely. If you use the keyword, the precision (and scale factor, if used) must immediately follow the keyword, which can be placed before or after any other attributes in the declaration. If you omit the keyword, the precision (and scale factor, if used) must follow the other attributes. For example, the following declarations are equivalent:

```
DCL A FIXED BIN(31);          DCL A FIXED BIN PRECISION(31);
DCL B FLOAT BIN(53);          DCL B PREC(53) FLOAT BIN;
DCL C FIXED DEC(5,2);         DCL C FIXED DEC PREC(5,2);
```

The precision of a floating-point data item is the number of decimal or binary digits in the mantissa of the floating-point representation.

scale-factor

The scale factor is the number of digits to the right of the decimal or binary point in fixed-point decimal or binary data. If no scale factor is specified with fixed-point data, the default is zero.

The ranges of values you can specify for the precision of each arithmetic data-type, and the defaults applied if you do not specify a precision, are summarized Section 3.2.1.

2.2.42 PRINT Attribute

The PRINT attribute is used to declare a print file. The file SYSPRINT, used as the default output by PUT statements, is a print file. The format of the PRINT attribute is:

PRINT

Print files are stream output files with special formatting characteristics. The PRINT attribute implies the OUTPUT and STREAM attributes.

Restriction

The PRINT attribute conflicts with the INPUT, RECORD, UPDATE, KEYED, SEQUENTIAL, and DIRECT attributes.

2.2.43 READONLY Attribute

You can apply the READONLY attribute to any static computational variable whose value does not change during program execution. The format for the READONLY attribute is:

READONLY

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 is given an initial value with the INITIAL attribute.

Restrictions

You can apply the READONLY attribute only to static computational variables. You must declare the variables with the EXTERNAL, STATIC, GLOBALREF, or GLOBALDEF attribute.

The value of a variable with the READONLY attribute cannot 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.

2.2.44 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 format of the RECORD attribute is:

RECORD

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.

Restriction

The RECORD attribute conflicts with the STREAM and PRINT attributes.

2.2.45 REFER Attribute

The REFER attribute defines dynamically self-defining structures. The format of the REFER attribute is:

REFER

See Section 4.2.6.3 for more information on the REFER option.

2.2.46 REFERENCE Attribute

The REFERENCE attribute forces a parameter to be passed by reference. The format of the REFERENCE attribute is:

{ REFERENCE }
{ REF   }

By default, most parameters are passed by reference in PL/I. However, the REFERENCE attribute is needed for passing an asterisk-extent array or character string by reference, because asterisk-extent parameters are passed by descriptor by default.

Example

```
DECLARE E ENTRY((*) FIXED BIN(31) REFERENCE, FIXED BIN(31));
```

This is a declaration of a non-PL/I entry point that takes an asterisk-extent parameter by reference. The first parameter of the external procedure is an arbitrarily large array of longwords, and the second parameter is the size of the array. The external procedure should have some method of determining the size of the array being passed.

Restriction

Note that you can only use the REFERENCE attribute in parameter descriptors.

2.2.47 RETURNS Attribute

The RETURNS option must be specified on the PROCEDURE or ENTRY statement if the corresponding entry point is invoked as a function. 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 RETURNS option and 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 (the 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.

You can specify the extent of a character-string value as an asterisk (*) to indicate that the string can have any length. Otherwise, extents must be specified with restricted expressions.

You cannot use the RETURNS option or the RETURNS attribute 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 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 the following:

```
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 return a fixed-point binary value to the point of invocation. The RETURN statement specifies that the value of the variable VALUE be 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 Section 6.4.

2.2.48 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.

You can apply the SEQUENTIAL attribute to files with sequential, relative, or indexed sequential file organizations.

Restriction

The SEQUENTIAL attribute conflicts with the DIRECT, STREAM, and PRINT attributes.

2.2.49 STATIC Attribute

The STATIC attribute specifies the way that PL/I is to allocate storage for a variable. The format of the STATIC attribute is:

STATIC

The STATIC attribute is implied by the EXTERNAL attribute. For more information on STATIC and on other storage-class attributes, see Chapter 5.

Restriction

The STATIC attribute directly conflicts with the BASED, CONTROLLED, DEFINED, and parameter attributes. The STATIC attribute cannot be applied to members of structures, parameters, or descriptions in an ENTRY or RETURNS attribute.

2.2.50 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 format of the STREAM attribute is:

STREAM

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.

Restriction

The STREAM attribute directly conflicts with the RECORD, KEYED, DIRECT, SEQUENTIAL, and UPDATE attributes.

2.2.51 STRUCTURE Attribute

You can optionally specify the STRUCTURE attribute in the declaration of a structure. The format of the STRUCTURE attribute is:

```
STRUCTURE
```

2.2.52 TYPE Attribute

The TYPE attribute copies the declarations contained within the type declaration to the variable to which it is applied. The format of the TYPE attribute is:

```
level-number identifier [attributes] TYPE reference
```

level-number

The level number to which the declarations in the reference are copied.

identifier

The variable to which the declarations in the reference are to be copied. The identifier must be preceded by a level number.

attributes

Storage class or dimensions appropriate for the level number. You can specify a storage class and dimensions with a major structure, or you can specify dimensions with a minor structure.

reference

The name of a type declaration that is known in the current block.

The TYPE attribute causes the declaration of its reference to be copied, but not the name, storage class, or dimensioning (if any) of the reference. The exception to this rule is that the UNION attribute is propagated in a TYPE declaration. While logical structuring is copied, the level numbers themselves are not copied.

You can use the TYPE attribute on a declaration already containing the TYPE attribute.

Restrictions

A TYPE definition cannot be:

- A pointer-qualified variable
- A subscripted variable
- An entry variable
- A variable declaration that leads to direct or indirect circular declarations

Examples of circular declarations that should not be used are:

```
DECLARE 1 S11,
        2 F1 CHARACTER(10),
        2 F2 TYPE(S11);          /* Direct circular   */
DECLARE V1 TYPE(V2);             /* Indirect circular */
DECLARE V2 TYPE(V3);
DECLARE V3 TYPE(V1);

DECLARE A1(10) TYPE(A2);         /* Indirect circular */
DECLARE A2(10) TYPE(A3);
DECLARE A3(10) TYPE(A1);

DECLARE 1 S31,
          2 F1 CHARACTER(10),
          2 F2 TYPE(S32);        /* Indirect circular */
DECLARE 1 S32,
          2 F1 CHARACTER(10),
          2 F2 TYPE(S33);
DECLARE 1 S33,
          2 F1 CHARACTER(10),
          2 F2 TYPE(S31);
```

- A structure variable with the BASED attribute and declarations with the REFER option if the TYPE variable does not have a BASED attribute. For example:

```
DECLARE N FIXED BINARY;
DECLARE 1 VARIABLE_X BASED,
          2 SIZE FIXED BINARY(15),
          2 ITEMS (N REFER (VARIABLE_X.SIZE)) CHARACTER(80);

DECLARE VARIABLE_Y TYPE (VARIABLE_X);  /* Error - VARIABLE_Y not BASED */
```

2.2.53 TRUNCATE Attribute

The TRUNCATE attribute is used in the declaration of a formal parameter to indicate that the actual parameter list can be truncated at the point where this argument should occur. The format of the TRUNCATE attribute is:

TRUNCATE

When the actual call is made, the actual parameter list can stop at the parameter before the one declared with the TRUNCATE attribute. It is possible to pass an actual parameter in a position with the TRUNCATE attribute. Note that in this case, all remaining parameters must also be specified unless they have the TRUNCATE attribute.

Example

```
DCL E ENTRY (FIXED,FIXED TRUNCATE,FIXED);
CALL E(1);
CALL E(1,2,3);
```

The following call, however, will be invalid:

```
CALL E(1,2);
```

This call is invalid because the second parameter has the TRUNCATE attribute, so the third parameter must be specified.

2.2.54 UNALIGNED Attribute

The UNALIGNED attribute is used in conjunction with the BIT attribute to specify that a bit-string variable should not be aligned on a byte boundary. Because UNALIGNED is the default for bit strings, it need not be specified. The format of the UNALIGNED attribute is:

```
{ UNALIGNED }
{ UNAL      }
```

You can use the UNALIGNED attribute in the declaration of character strings. All character strings are aligned on byte boundaries; therefore, the UNALIGNED attribute has no effect on the actual storage of a character string.

Restriction

The UNALIGNED attribute conflicts with all data-type attributes other than BIT and CHARACTER.

2.2.55 UNION Attribute

The UNION attribute, which can be used only in conjunction with a level number in a structure declaration, signifies that all immediate members of the major or minor structure so designated occupy the same storage. Immediate members are those members having a level number 1 higher than the major or minor structure with the UNION attribute. For example, if the UNION attribute were associated with level n, then all members or minor structures at level n+1 up to the next member at level n would be immediate members and would occupy the same storage. The format for the UNION attribute is:

```
level-number  identifier  [storage-class]  UNION
```

level-number

The level number of the variable with which the declarations in the reference share storage.

identifier

Names the variable with which the declarations in the reference share storage. A variable declared with the UNION attribute must be a major or minor structure.

storage-class

The storage class specified for the structure. You can specify the storage class only on level 1.

2.2.56 UPDATE Attribute

The UPDATE attribute is a file description attribute indicating that the associated file is to be used for both input and output. You can apply the UPDATE attribute to relative files, indexed files, and sequential disk files with fixed-length records. The format of the UPDATE attribute is:

UPDATE

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 are applied to files, see Section 9.1.3.3.

Restriction

The UPDATE attribute directly conflicts with the INPUT, OUTPUT, STREAM, and PRINT attributes and with any data-type attribute other than FILE.

2.2.57 VALUE Attribute

The VALUE attribute is provided for passing parameters by value rather than by reference, or it can be used to specify a global constant value. The format of the VALUE attribute is:

{ VALUE }
{ VAL }

The VALUE attribute serves two purposes:

- In a parameter descriptor in an ENTRY declaration, it specifies that the corresponding argument is to be passed using the hardware-specific convention for passing arguments by value. For this usage, VALUE must be specified in conjunction with one of the following attributes:

  ANY
  FIXED BINARY(m) where m is less than or equal to 31
  FLOAT BINARY(n) where n is less than or equal to 24
  BIT(o) ALIGNED where o is less than or equal to 32
  ENTRY
  OFFSET
  POINTER

- In conjunction with the GLOBALREF or GLOBALDEF attributes, it specifies that a global external variable has a constant value for which no storage is allocated. The compiler can use this value as an immediate value in generating instructions. With the global external variables, the format is:

  { VALUE } { GLOBALDEF[(psect-name)][INITIAL(value)] }
  { VAL } { GLOBALREF }

The VALUE attribute, when specified with the BIT attribute, implies the ALIGNED attribute.

2.2.58 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. The format of the VARIABLE attribute is:

```
VARIABLE
```

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. You can declare label constants only by using the label identifier in the program; you cannot define a label constant in a DECLARE statement.

Restriction

The VARIABLE attribute is not valid in a returns descriptor or in a parameter descriptor.

2.2.59 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     }
```

You must specify a length attribute in conjunction with VARYING by giving the maximum length allowed for the variable. The current length is stored with the value and can be determined at any time with the LENGTH built-in function. If you need to determine the maximum declared length of a varying- length character string, use the MAXLENGTH built-in function.

The value of an uninitialized CHARACTER VARYING variable is undefined.

Special rules apply to reading and writing record files into and from variables that have the VARYING attribute. See the *Kednos PL/I for OpenVMS Systems User Manual*.

Restriction

The VARYING attribute directly conflicts with any data-type attribute other than CHARACTER.

Examples

```
DECLARE STRING CHARACTER(80) VARYING;
```

A variable named STRING is declared as a varying-length character string with a maximum length of 80 characters.

```
S: PROCEDURE OPTIONS(MAIN);
DECLARE STRING CHARACTER(80) VARYING;
    STRING = 'PIE';
    PUT LIST (LENGTH(STRING));
    PUT LIST (MAXLENGTH(STRING));
    PUT LIST (SIZE(STRING));
END;
```

The value returned by the built-in function LENGTH is 3, the length of the current value of the string. The value returned by the built-in function MAXLENGTH is 80, the maximum declared length. The value returned by the built-in function SIZE is 82, the maximum declared length plus two (for the two bytes that hold the value of the current length).

3 Data Types

All programs process data items. Data items can be constants or variables. A constant data item has a value that does not change during program execution; a variable data item can represent different values.

A data item has an associated type that you can specify as an attribute or collection of attributes in a declaration. Unlike other languages that often have a distinction between data types and data attributes, a PL/I data type is entirely defined by the data attributes given to a data item identifier. The data type that you select determines the operations that you can perform on data items and how they are stored.

The rest of this chapter describes data types in more detail.

3.1 Summary of Data Types

PL/I supports the following computational data types:

- Arithmetic data types define values that can be used in arithmetic computation. They are:
  - — Fixed point (binary and decimal integers and fractions)
  - — Floating point (binary and decimal)
  - — Pictured (fixed point data stored in character form)
- Character-string data consists of a sequence of ASCII characters.
- Bit-string data consists of sequences of binary digits.

PL/I also supports the following noncomputational data types and attributes:

- Pointers and offsets represent the location in memory of data, and are used to access data and areas in system-allocated buffers.
- Label constants and variables provide a flexible means of control within a program.
- Entry constants and variables are used to invoke procedures through specified entry points.
- File constants and variables provide access to files.
- Areas are regions of storage in which based variables can be allocated and freed. Offsets represent the location of a based variable in an area.
- Programmer-defined conditions represent exceptional conditions for use with the ON statement (Section 8.10.1) and SIGNAL statement (Section 8.10.2).

You can place each of these data types in aggregate structures or arrays to form new data types. See Chapter 4 for more information.

3.1.1 Declarations

All names referenced in a PL/I program must be declared explicitly, with the exception of entry-point names, statement labels, built-in functions, and the default file constants SYSIN and SYSPRINT. 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.

3.1.2 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 default FIXED attribute implies the attributes BINARY(31,0). Thus, the variable NUMBER has the attributes FIXED BINARY(31,0).

Table 3–1 shows the default attributes implied by each computational data attribute.

Table 3–1 Implied Attributes for Computational Data

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

3.1.2.1 Attributes of Constants

Constants have attributes implied by the characters used to specify them. The following list describes the expression of constants and their attributes:

- A series of characters enclosed in apostrophes is assumed to be a string constant:
  - — If the letter B, which can be lowercase, is appended after the closing apostrophe, the constant is a bit-string constant, for example, <BIT_STRING>(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. Each digit occupies 2, 3, or 4 bits. For example, <BIT_STRING>(17777)B3 is an octal constant that is represented internally as a string of 15 bits.
  - — If the constant does not have the letter 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.
- Consider this example where a fixed-point decimal constant has the following appended characters:

  E [+ -] digit . . .

  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 because you can assign constants of any computational type to variables of any computational type and they are converted automatically to the target type.

You usually give values to binary variables by assigning decimal constants to them. For example:

```
I = 1;
```

This converts the decimal integer 1 and assigns the converted value to a fixed-point binary variable I. Consider the following example:

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

This 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;
```

This assigns the fixed-point decimal value 123.44 to a picture variable PAY_PIC. The value of PAY_PIC is a pictured value, which is 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.

3.1.2.2 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 (49);
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. The rules for operand conversion are discussed in detail in Section 6.4.2.

For information about arithmetic operators, operands, and data conversions, see Chapter 6.

3.1.3 Compatible Data Types

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

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

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 compatible data types if they have the same number of immediate members, and if corresponding members have compatible data types.

In general, you can specify string lengths, area sizes, and array bounds with expressions or with asterisks. The values used to determine whether two variables have compatible 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 containing such a variable's declaration is activated. 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 extent of the argument passed to the parameter. The extent remains the same throughout the procedure invocation.
- For based or controlled variables, extent expressions are evaluated each time the variable is referenced.

Consider the following 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.

3.2 Arithmetic Data

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

- Fixed point-for binary and decimal data with a fixed number of fractional digits

- Floating point-for calculations on very large or very small numbers with the decimal point (number of fractional digits) allowed to float
- Pictured-for fixed point decimal data that is stored internally in character form with special formatting characters

When you declare an arithmetic variable, you do not always have to define all its characteristics, or attributes.

Table 3–1 shows the implied attributes for computational data.

3.2.1 Precision and Scale of Arithmetic Data Types

The PRECISION attribute applies to decimal and binary data as follows:

- The precision of a fixed-point data item is the total number of decimal or binary digits used to represent a value.
- The precision of a floating-point data item is the number of decimal or binary digits in the mantissa of the floating-point representation.

The scale of fixed-point data is the number of digits to the right of the decimal or binary point. Floating-point variables do not have a scale factor. In this manual, the letter p is used to indicate precision, and the letter q is used to indicate the scale factor.

You can specify both precision and scale in a declaration. For example:

```
DECLARE x FIXED DECIMAL(10,3) INITIAL(1.234);
```

This example indicates that the value of x has 10 decimal digits and that 3 of those are fractional. When a value is assigned to the variable, its internal representation does not include the decimal point; the previous value for x is stored as 1234, and the decimal point is inserted when the number is output. The scale factor has the effect of multiplying the internal representation of the decimal number by a factor of 10^{-q} (where q is the absolute value of the specified scale).

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 <= p <= 31	p >= q >= -31	31
BINARY FLOAT (OpenVMS VAX systems)	1 <= p <= 113	-	24
BINARY FLOAT (OpenVMS Alpha systems)	1 <= p <= 53	-	24
DECIMAL FIXED	1 <= p <= 31	p >= q >= 0	10

Data Type Attributes	Precision	Scale Factor	Default Precision
DECIMAL FLOAT (OpenVMS VAX systems)	1 <= p <= 34	-	7
DECIMAL FLOAT (OpenVMS Alpha systems)	1 <= p <= 15	-	7

If no scale factor is specified for fixed-point data, the default is 0.

For fixed-point binary data, the scale factor must be within the range -31 through 31 and less than or equal to the specified precision. Positive scale factors for fixed binary numbers function according to the same principles as those for fixed decimal. That is, a positive scale factor is similar to multiplying the internal representation binary number by a factor of 2^{-q}.

A negative scale factor indicates that the number of fractional bits are shifted in the opposite direction. In effect, this is similar to multiplying the binary number by a factor of 2^{q}, where *q* is the absolute value of the specified scale. For example:

```
DECLARE (A,B) FIXED BINARY(31,-3),
        (C,D) FIXED BINARY(31,3);
A = 128;      /* output = 128   */
B = 7;        /* output =   0   */
C = 128;      /* output = 128.0 */
D = 7;        /* output =   7.0 */

PUT SKIP LIST (A,B,C,D);
END;
```

Internally, binary numbers undergo an implicit conversion and are represented as powers of 2. For instance, in the previous example variable A is first divided by 2^3 because it is declared with a scale factor of -3. The stored number is 16. On output, the number 16 is multiplied by 2^3 and the number is again 128. However, when variable B is first divided by 2^3, the result is 0, which is the value of the stored number. Therefore, on output, 0 is multiplied by 2^3 and the output is 0.

Integer variables declared in the previous example with a positive scale factor are output as they were input, but they are followed on the right with a decimal point and a 0.

Even though arithmetic operands can be of different arithmetic types, all operations must be performed on objects of the same type. Consequently, the compiler can convert operands to a derived type, as previously shown. Therefore, when you declare a fixed binary number with a scale factor and assign it a decimal value, the results may not be as you expect because the binary scale factor left-shifts the specified number of bits to the right of the decimal point. During conversion to a decimal representation, the difference between the resulting binary number and its decimal representation is not the equivalent of dividing or multiplying the decimal number by 10. Instead, the binary number is first converted to its internal representation and then this representation is converted to its decimal representation.

When excess fractional digits are truncated, no condition is signaled. If there is any resulting loss of precision, it may be difficult to detect because truncated fractional digits do not signal a condition.

For example:

```
A: PROCEDURE OPTIONS (MAIN);
   DECLARE A FIXED BIN (31,3),
           B DECIMAL (10,5),
           C DECIMAL (10,5);
A = .3;
B = 34.8;

C = MULTIPLY(A,B,10,5);

PUT SKIP LIST (A,B,C);
END;
```

Before the multiplication is performed, the variables are converted to fixed binary so that the operands share a common data type. However, after conversion, variable A is output as 0.2 rather than 0.3. The output from the previous example is:

```
0.2     34.80000    8.6875
```

If variable A was declared with the attributes FIXED DECIMAL (10,5), the output will be:

```
0.30000     34.80000    10.44000
```

3.2.2 Fixed-Point Binary Data

The attributes FIXED and BINARY are used to declare integer variables and fractional variables in which the number of fractional digits is fixed (that is, nonfloating-point numbers). The BINARY attribute is implied by FIXED.

For example, a fixed-point binary variable can be declared as:

```
DECLARE X FIXED BINARY(31,0);
```

The variable X is given the attributes FIXED, BINARY, and (31,0) in this declaration. The precision is 31. The scale factor is 0, so the number is an integer.

There is no representation in PL/I for a fixed-point binary constant. Instead, integer constants are represented as fixed decimal. However, fixed decimal integer constants (and variables) are converted to fixed binary when combined with fixed binary variables in expressions. For example:

```
I = I+3;
```

In this example, if I is a fixed binary variable, the integer 3 is represented as fixed decimal; however, PL/I converts it to fixed binary when evaluating the expression.

Fixed binary variables have a maximum precision of 31, and therefore fixed binary integers can have values only in the range -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.

The attributes FIXED BINARY are used to declare binary data in PL/I. The BINARY attribute is implied by FIXED. The format of a declaration of a single, fixed-point, binary variable is:

```
DECLARE identifier FIXED [BINARY] [(precision[,scale-factor])];
```

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 usually receives given values by being assigned to an expression of another computational type or another fixed-point binary variable.

3.2.2.1 Internal Representation of Fixed-Point Binary Data

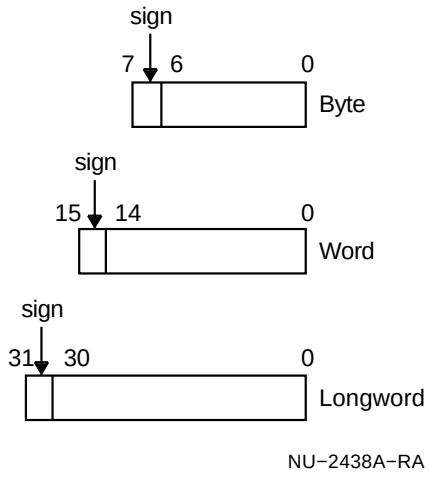
Figure 3–1 shows the internal representation of fixed-point binary data. Storage for fixed-point binary variables is always allocated in a byte, word, or longword. For any fixed-point binary value:

- If p is in the range 1 through 7, a byte is allocated.
- If p is in the range 8 through 15, a word is allocated.
- If p is in the range 16 through 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 signficant.

In all cases, the high-order bit (7, 15, or 31) represents the sign.

Figure 3–1 Internal Representation of Fixed-Point Binary Data



3.2.3 Fixed-Point Decimal Data

Fixed-point decimal data is used in calculations where exact decimal values must be maintained, for example, in financial applications. You can also use fixed-point decimal data with a scale factor of 0 whenever integer data is required.

The following sections describe fixed-point constants and variables and their use in expressions.

This discussion is divided into the following parts:

- Constants
- Variables
- Use in expressions
- Internal representation

3.2.3.1 Fixed-Point Decimal Constants

A fixed-point decimal constant can have between 1 and 31 of the decimal digits 0 through 9 with an optional decimal point or sign, or both. If there is no decimal point, PL/I assumes it to be immediately to the right of the rightmost digit. Some examples of fixed-point decimal constants are:

```
12
4.56
12345.54
-2
01.
```

The precision of a fixed-point decimal value is the total number of digits in the value. The scale factor is the number of digits to the right of the decimal point, if any. The scale factor cannot be greater than the precision.

3.2.3.2 Fixed-Point Decimal Variables

The attributes FIXED and DECIMAL are used to declare fixed-point decimal variables. The FIXED attribute is implied by DECIMAL.

If you do not specify the precision and the scale factor, the default values are 10 and 0, respectively.

The format of a declaration of a single fixed-point decimal variable is:

```
DECLARE identifier [FIXED] DECIMAL [(p[,q])];
```

Some examples of fixed-point decimal declarations are:

```
DECLARE PERCENTAGE FIXED DECIMAL (5,2);
DECLARE TONNAGE FIXED DECIMAL (9);
```

3.2.3.3 Use in Expressions

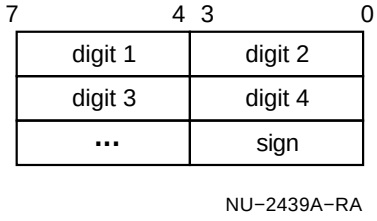
You cannot use fixed-point decimal data with a nonzero scale factor in calculations with binary integer variables. If you must use the two types of data together, use the DECIMAL built-in function to convert the binary value to a scaled decimal value before attempting an arithmetic operation. For example:

```
DECLARE I FIXED BINARY,
        SUM FIXED DECIMAL (10,2);

SUM = SUM + DECIMAL (I);
```

3.2.3.4 Internal Representation of Fixed-Point Decimal Data

Fixed-point decimal data is stored in packed decimal format. Each digit is stored in a half-byte, as shown in Figure 3–2. Bits 0 through 3 of the last half-byte contain a value indicating the sign. Normally, the hexadecimal value C indicates a positive value and the hexadecimal value D indicates a negative value.

Figure 3–2 Fixed-Point Decimal Data Representation



3.2.4 Floating-Point Data

The floating-point data types provide a way to express very large and very small numbers such as in scientific calculations. All floating-point calculations are performed on values in one of the 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.

The following sections describe floating-point constants and variables and their use in expressions.

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

3.2.4.1 Floating-Point Constants

A floating-point constant can have 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. The value to the right of the letter E is called the exponent.

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.

If you write a constant without the E and the exponent, it is considered to be fixed-point decimal. However, you can use such constants anywhere in expressions involving floating-point data.

All floating-point constants are decimal.

3.2.4.2 Floating-Point Variables

The keyword FLOAT identifies a floating-point variable in a declaration.

A floating-point value can be either binary or decimal. Because the internal representation of floating-point variables is binary, it is recommended that you use FLOAT BINARY (which is the default) to declare variables, unless you need the properties of FLOAT DECIMAL. (The difference between FLOAT BINARY and FLOAT DECIMAL appears only when a conversion to another type, such as character, for doing I/O is necessary.) You should declare all floating-point variables using the same base.

To declare a single floating-point binary variable, specify a DECLARE statement as follows:

DECLARE identifier FLOAT [BINARY] [(p)];

You can optionally specify the precision for a floating-point variable in the declaration. For example:

```
DECLARE X FLOAT BINARY(53);
```

The keyword FLOAT identifies a floating-point variable.

To declare a decimal floating-point variable, use the following format:

DECLARE identifier FLOAT DECIMAL [(p)];

For example:

```
DECLARE X FLOAT DECIMAL (30);
```

3.2.4.3 Using Floating-Point Data in Expressions

You can use both integer and scaled decimal constants 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. For example:

```
DECLARE X FLOAT BINARY (53);
X = X + 1.3;
```

Here, the constant 1.3 is converted to floating point when the expression is evaluated.

Such a conversion is normally done during compilation, but in some cases the constant is maintained in decimal until run time.

3.2.4.4 Floating-Point Data Formats

Table 3–2 summarizes the floating-point formats supported by the different implementations of PL/I.

Table 3–2 Supported Floating-Point Formats

Implementation	Supported Formats
OpenVMS VAX systems	F, D, G, and H
OpenVMS Alpha systems	F, S, D, G, and T

The S and T formats conform to the IEEE standards of floating-point formats.

Table 3–3 summarizes the approximate ranges of the floating-point formats.

Table 3–3 Ranges of Floating-Point Formats

Format	Range
F or S	$0.29 * 10^{-38}$ to $1.7 * 10^{38}$
D	Same as F but with a more precise mantissa (see Table 3–4)
G or T	$0.56 * 10^{-308}$ to $0.9 * 10^{308}$
H	$0.84 * 10^{-4932}$ to $0.59 * 10^{4932}$

Table 3–4 summarizes the ranges of precision (sign bits, exponent bits, and fractional bits of accuracy) for each type. Section 3.2.4.5 and Section 3.2.4.6 describe the internal representation of floating-point data.

Table 3–4 Ranges of Precision for Floating-Point Types

Floating-Point Type[1]	Sign Bits	Exponent Bits	Fractional Bits
F or S (single precision)	1	8	24
D (double precision)	1	8	53
G or T (double precision)	1	11	53
H (quadruple precision)	1	15	113

[1]You can perform G- and H-floating computations with software emulation on some older processors. In addition, floating-point hardware is optional on most MicroVAX systems. See the appropriate processor manual for more information.

The PL/I compiler selects the appropriate floating-point type based on the precision you specify and on a compile-time qualifier on the PLI command. The types are selected as shown in Table 3–5, where p indicates precision.

Table 3–5 Floating-Point Types Used by PL/I

Precision (DECIMAL)	Precision (BINARY)	OpenVMS VAX	OpenVMS Alpha
1 <= p <= 7	1 <= p <= 24	F	F or S[1]
8 <= p <= 15	25 <= p <= 53	D or G[2]	D, G, or T[3]
16 <= p <= 34	54 <= p <= 113	H	Warning

[1]If no qualifier is supplied, the default type is F. If /FLOAT=IEEE_FLOAT, type S is used.

[2]If no qualifier is supplied, the default type is D. If the /G_FLOAT qualifier is supplied, type G is used.

[3]If no qualifier is supplied, the default type is D. If /FLOAT=G_FLOAT, type G is used. If /FLOAT=IEEE_FLOAT, type T is used.

3.2.4.5 OpenVMS VAX Internal Representation of Floating-Point Data

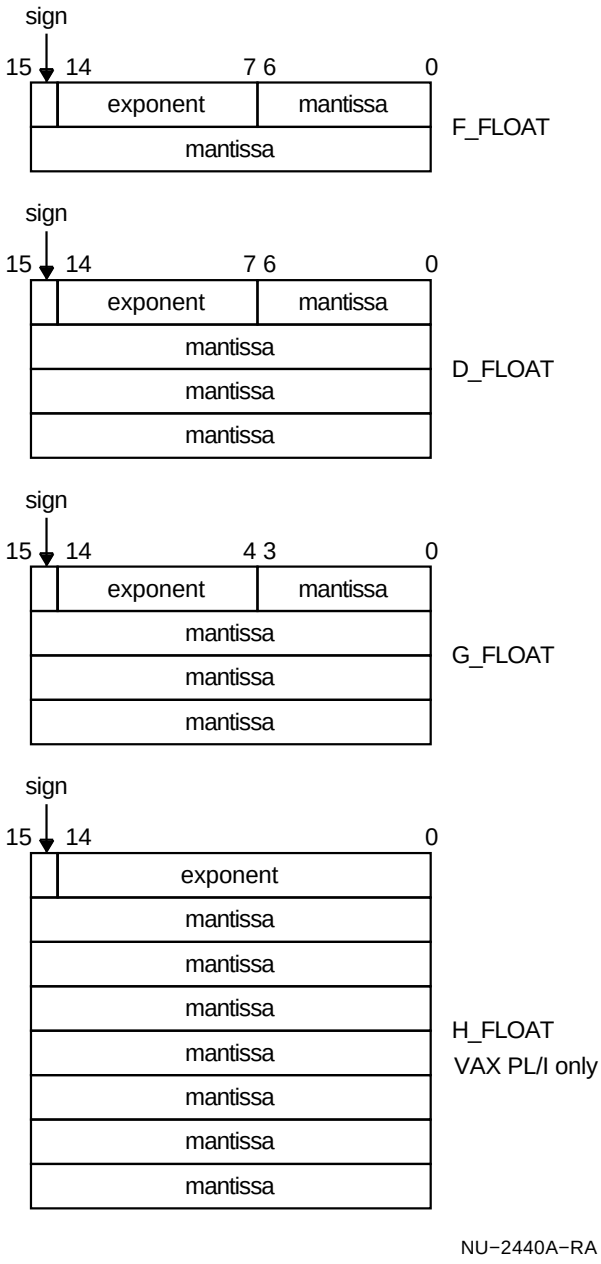
In all VAX floating-point formats, the value 0 is indicated when the sign bit and all exponent bits are set to 0. In effect, this allows for the representation of a value with a 24-bit fraction and an 8-bit exponent in single precision, even though only 23 bits in the format are allocated for the fraction.

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. The double-precision format has 56 bits available for the fraction, but only 53 bits are used by PL/I. If you specify a floating-point binary precision in the range 54 to 56, the 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 to keep the compiler from selecting 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.

Figure 3–3 shows the internal structure of floating-point data for VAX systems. For a more detailed description of VAX floating-point formats, see the *VAX Architecture Reference Manual*.

Figure 3–3 VAX Internal Representation of Floating-Point Data



3.2.4.6 OpenVMS Alpha Internal Representation of Floating-Point Data

This section describes the S and T floating-point formats supported by OpenVMS Alpha systems. OpenVMS Alpha also supports the F, D, and G floating-point formats described in Section 3.2.4.5.

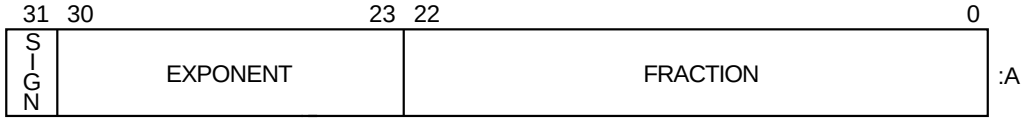
IEEE S_floating Format

The following PL/I types are represented using S_floating data, which occupies four contiguous bytes:

```
FLOAT BINARY (P) P ≤  24
FLOAT DECIMAL (P) P ≤  7
```

Bits are labeled from the right, 0 through 31, as shown in Figure 3–4.

Figure 3–4 IEEE S_floating Data Representation



NU–2996A–RA

The form of S_floating data is sign magnitude, with bit 31 the sign bit (0 for positive numbers, 1 for negative numbers), bits 30:23 an exponent, and bits 22:0 a normalized 24-bit fraction including the redundant most significant fraction bit not represented. The value of data is in the range 1.17549435E-38 (normalized) to 3.40282347E38. The precision is approximately one part in 2**23; that is, typically seven decimal digits.

When loaded into a 64-bit register, the S_floating value resides in bits 29-63.

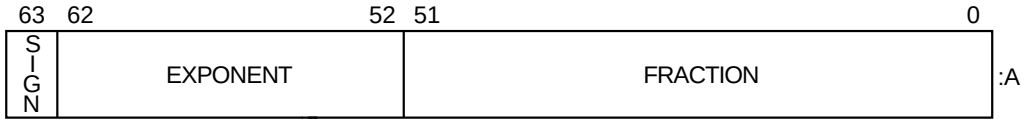
IEEE T_floating Format

The following PL/I types are represented using T_floating data, which occupies eight contiguous bytes:

```
FLOAT BINARY (P)  24 < P ≤  53
FLOAT DECIMAL (P) 7 < P ≤  15
```

Bits are labeled from the right, 0 through 63, as shown in Figure 3–5.

Figure 3–5 IEEE T_floating Data Representation



NU–2997A–RA

The form of T_floating data is sign magnitude, with bit 63 the sign bit (0 for positive numbers, 1 for negative numbers), bits 62:52 an exponent, and bits 51:0 a normalized 53-bit fraction including the redundant most significant fraction bit not represented. The value of data is in the approximate range 2.225073859E-380 (normalized) to 1.797693135E308. The precision is approximately one part in 2**52; that is, typically fifteen decimal digits.

3.2.5 Pictured Data

Use pictured data when you want to manipulate a quantity arithmetically and accept or display its value using a special format.

A picture specification (or picture) describes both the numeric attributes of a pictured variable and its input/output (I/O) format. A simple picture might look like this in a DECLARE statement:

```
DECLARE CREDIT PICTURE '$99999V.99DB';
```

This statement declares the variable CREDIT as a pictured variable. The characters within the apostrophes describe its format.

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.

This section discusses the following topics:

- The picture characters that make up a specification in the PICTURE attribute and in the P format item. It also describes picture syntax.
- The process by which a value is assigned to a pictured variable or written out with the P format item.
- The process by which a pictured value is assigned to other variables or acquired with the P format item.
- Editing by picture.
- The internal representation of pictured variables.

The formatting possible with pictured data is useful in many applications, but pictured data is less efficient than fixed-point decimal data in computations. Therefore, do not use pictured data unless you need the formatting.

3.2.5.1 Picture Characters

Table 3–6 summarizes the PL/I picture characters, their meaning, and whether they effect numeric interpretation and internal representation. The paragraphs following the table describe the picture characters and syntax. All picture characters are shown here in uppercase, but their lowercase equivalents can be used.

Table 3–6 Picture Characters

Character	Meaning	Numeric Interpretation	Internal Representation
V	Position of assumed decimal point	yes	no
9	Decimal digit, including leading zeros	yes	yes
Z	Decimal digit with leading-zero suppression	yes	yes

Table 3–6 (Cont.) Picture Characters

Character	Meaning	Numeric Interpretation	Internal Representation
*	Decimal digit with asterisk for leading zero	yes	yes
Y	Decimal digit with space for any zero	yes	yes
(n)	Iteration factor for subsequent character	yes	yes
T	Position of digit and encoded plus sign or minus sign	yes	yes
I	Position of digit and encoded plus sign if number >= 0	yes	yes
R	Position of digit and encoded minus sign if number < 0	yes	yes
$	Position(s) of (drifting) dollar sign	yes	yes
+	Position(s) of (drifting) plus sign if number >= 0	yes	yes
-	Position(s) of (drifting) minus sign if number < 0	yes	yes
S	Position(s) of (drifting) plus sign or minus sign	yes	yes
,	Position at which comma is inserted	no	yes
.	Position at which decimal point is inserted	no	yes
/	Position at which slash is inserted	no	yes
B	Position at which space is inserted	no	yes
CR	Positions at which <BIT_ STRING>(CR) is inserted if number < 0	no	yes
DB	Positions at which <BIT_ STRING>(DB) is inserted if number < 0	no	yes

Decimal Place Character (V)

The V character shows the position of the assumed decimal point, or 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 cause a decimal point to appear in the internal representation or in the output (use the period insertion character for this purpose). The following rules apply to the V character:

- Only one V character can appear in a picture.

- If a picture does not contain the V character, the V is assumed to be at the right end of the picture. That is, the pictured value has a scale factor of 0.
- 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. Values with too many or too few digits are handled as follows:
  - — 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.

The following example shows the effect of the V character:

```
DECLARE PRICE PICTURE '$$9V.99',
    BAD_PRICE PICTURE '$$9.99';
        PRICE = .98;      /* Output as $0.98  */
        BAD_PRICE = .98;  /* Output as $0.00  */
        PRICE = 98;       /* Output as $98.00 */
        BAD_PRICE = 98;   /* Output as $0.98  */
```

In this example, note that the variable PRICE, which contains the V character, represents the value properly. The variable BAD_PRICE, which contains only the period insertion character, has an assumed V character at the end of the picture, which causes the variable to misrepresent the value.

Digit Characters (9, Z, *, Y)

The characters 9, Z, and Y, and the asterisk character (*) 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. The following rules apply to these characters:

- 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. Leading zeros at positions occupied by a 9 are output.
- 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 a leading 0, it is replaced by a space. Several additional rules apply to the Z character:

— The Z character must not appear in the same picture with the asterisk character (*). It must not appear to the right of the characters 9, T, I, or R nor to the right of a drifting string.

— 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 0 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 asterisk character (*) functions identically with the Z character, except that leading zeros are replaced by asterisks instead of spaces.
- The position occupied by the Y character contains a decimal digit only if the digit is not 0. All zeros in the indicated positions, whether significant or not, are replaced by spaces.

Iteration Factor (n)

You can precede any picture character that can appear more than once in a picture by an iteration factor, which must be a positive integer constant enclosed in parentheses. For example:

```
'(4)9'
```

This picture is the same as the following one:

```
'9999'
```

Encoded-Sign Characters (T, I, R)

You can use the characters T, I, and R, which are encoded-sign characters, wherever 9 is valid. Each represents a digit that has the sign of the pictured value encoded in the same position. You can use only one encoded-sign character in a picture.

An encoded-sign character cannot be used in a picture that contains one of the following characters: S, +, -, CR, or DB (described in the following text).

The meanings of the characters are:

- The T character indicates that the position contains an encoded minus sign if the numeric value is less than 0 and an encoded plus sign if the numeric value is greater than or equal to 0.
- The I character indicates an encoded plus sign if the numeric value is greater than or equal to 0. Otherwise, the position contains an ordinary digit.
- The R character indicates an encoded minus sign if the numeric value is less than 0. Otherwise, the position contains an ordinary digit.

Table 3–7 lists the encoded-sign characters and their ASCII equivalents.

Table 3–7 ASCII Representation of Encoded-Sign Characters

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 dollar sign ($), plus sign (+), minus sign (-), and S character are drifting characters. You can use the drifting characters to indicate digits, and they also indicate a symbol to be inserted when, for example, a pictured value is written out by PUT LIST. The meanings of the characters are:

- 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 0.
- The minus sign (-) causes a minus sign to be inserted if the numeric value is less than 0.
- The S character causes a plus sign to be inserted if the numeric value is greater than or equal to 0, and a minus sign if the value is less than 0.

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 0, the leading 0 is suppressed, and the leftmost character drifts to the right; 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, because they describe at least some of the positions occupied by decimal digits. For an example of this behavior by a drifting character (the dollar sign), refer to the V decimal place character description.

The following additional rules apply to drifting characters:

- A drifting string is a series of more than one of the same type of drifting character. Only one drifting string can appear in the picture; any other drifting characters can be used only singly and therefore designate insertion characters, not digits.
- The Z and asterisk (*) 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, which are defined as follows:
  - — If the drifting string contains an insertion character, it 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 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 only when all integral and fractional digits are zeros: they are replaced by spaces in the internal representation. If any digit is not 0, all fractional digits appear as actual digits.
  - — Any insertion characters immediately to the right of a drifting string are considered part of it.

Insertion Characters (, . / B)

The insertion characters indicate that characters are inserted between digits in the pictured value. 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 ($, +, -, S) also function as insertion characters when used singly (that is, not as part of a drifting string).

The period (.) does not imply a decimal place character V (see the example in the description of the V character, described earlier).

The following rules describe insertion 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 0 suppression occurs, the insertion character is inserted only in these cases:
  - — If a significant digit appears immediately to its left

— If the V character appears immediately to its 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 0, 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 0 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 adjacent. However, 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; it will then be in the correct location and will appear whenever any fractional digit is significant. The following example shows the correct and incorrect placement of the period:

```
DECLARE NUM PICTURE 'ZZZV.ZZ',
    BAD_NUM PICTURE 'ZZZ.VZZ';
NUM=0.02;       /* Output as .02 */
BAD_NUM=0.02;   /* Output as 02  */
```

- You can use other insertion characters, such as the comma, to separate the integral and fractional portions of a number. Do not use the comma with GET LIST input, because in that context it separates different data items in the input stream.

Credit (CR) and Debit (DB) Characters

These picture characters are always specified as the character pairs CR and DB. If either pair is included, it appears if the numeric value is less than zero. In each case, the associated positions contain two spaces if the numeric value is greater than or equal to 0.

The characters are inserted with the same case as 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 combined in one picture, nor can they be used in the same picture as any other character that specifies the sign of the value (S, plus sign (+), and minus sign (-) characters). In addition, they must appear to the right of all picture characters specifying digits.

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, or ellipsis (. . .), 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:

{
9 . . . [V[9 . . .]]
V9 . . .
Z . . . [9 . . . [V[9 . . .]]]
Z . . . [V[9 . . .]]
[Z . . .]VZ . . .
* . . . [9 . . . [V[9 . . .]]]
* . . . [V[9 . . .]]
[* . . .]V* . . .
++ . . . [9 . . . [V[9 . . .]]]
++ . . . [V[9 . . .]]
- - . . . [9 . . . [V[9 . . .]]]
- - . . . [V[9 . . .]]
SS . . . [9 . . . [V[9 . . .]]]
SS . . . [V[9 . . .]]
$$ . . . [9 . . . [V[9 . . .]]]
$$ . . . [V[9 . . .]]
+[+ . . .]V+ . . .
-[- . . .]V- . . .
S[S . . .]VS . . .
$[$ . . .]V$ . . .
}

Note: The character Y, T, I, or R can appear wherever 9 is valid with the following restrictions. Only one character T, I, or R can appear in a picture. A picture cannot 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 a precision of 4 (p=4) and a scale factor of 2 (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 6-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 digit; 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 3-digit groups unless the digit preceding the slash is a suppressed 0.

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 (<BIT_STRING>($$$) and <BIT_STRING>(- -)).

'(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.

3.2.5.2 Assigning Values to Pictured Variables

Assignment of a computational value to a pictured variable is performed in the following two steps:

1 The value is converted to fixed decimal, with precision and scale as specified by the picture.

2 The resulting fixed decimal value is edited into the pictured variable.

If PL/I cannot perform one of these steps in a meaningful fashion, an error occurs. The following examples show two programming errors that are common in assignments to pictured variables.

```
CREDIT = '$12443.00';
```

This example signals the CONVERSION condition, because the character string contains a dollar sign and cannot be converted to fixed-point decimal. The value assigned to CREDIT should be either '12443.00' or 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). For example:

```
CREDIT = -12443.00;
```

If the picture of CREDIT did not contain the DB characters, this assignment 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, you can use the VALID built-in function to validate the contents of the variable.

3.2.5.3 Extracting Values from Pictured Data

When you use a pictured value in an arithmetic context (such as an assignment to an arithmetic variable), the picture is used to extract the fixed-point decimal number from the character string that internally represents the pictured value. Extraction also occurs when you input a pictured value with the GET EDIT statement and the P format item. If the contents of the pictured variable or input item do not conform to the picture, an error occurs.

For example:

```
DECLARE CREDIT PICTURE '$99999V.99DB';
```

In the picture for CREDIT, the 9 character specifies the position of a decimal digit; because the picture contains seven of these, the fixed-point decimal precision of CREDIT is 7. The V character separates the integral and fractional digits; because there are two 9 characters to the right of the V, the scale factor of CREDIT is 2. Because the V character specifies only a numeric property, a period picture character (.) is 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 0; otherwise, two spaces appear in the indicated positions. The DB character also indicates that CREDIT's value is numerically negative, so that if CREDIT is later assigned to an arithmetic variable the variable will be given a negative value.

3.2.5.4 Editing by Picture

Any computational value or expression can 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.
- 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.

3.2.5.5 The Internal Representation of 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 describing the variable's numeric value as well as special symbols. An individual picture character and its position in the picture indicate the interpretation of an associated position in the pictured value.

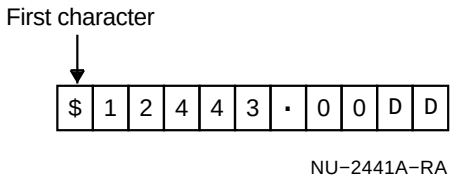
The picture characters fall into three categories:

- Characters that do not affect internal representation. The decimal place character (V) is the only one in this category.
- Characters that affect both the numeric interpretation and internal representation of the value. These characters indicate how the digits of the numeric value should be placed in the string and where to place a sign as follows:
  - — The digit characters (9, Z, *, Y)
  - — The encoded-sign characters (T, I, R)
  - — The drifting characters ($, +, -, S)
- Characters that affect only the internal representation of the value. These characters appear in the internal characters string as follows:
  - — The insertion characters (comma, period, slash, space)

— The credit (CR) and debit (DB) characters

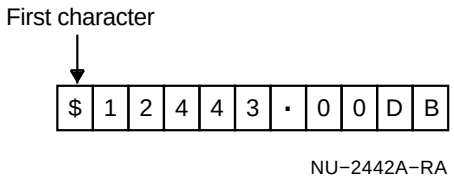
Section 3.2.5.1 describes each picture character in more detail. The assignment *CREDIT = 12443.00;* stores data internally, as shown in Figure 3–6, as a character string where delta (△) represents a space.

Figure 3–6 Internal Representation of a Pictured Variable



The assignment *CREDIT = -12443.00;* stores data internally as shown in Figure 3–7.

Figure 3–7 Internal Representation of a Pictured Variable



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)  */
```

3.3 Character-String Data

A character string is a sequence of zero or more characters. The value of a character-string variable can consist of any DEC Multinational Character to a maximum length of 32,767 characters. The first 128 characters of the DEC Multinational Character Set are the ASCII characters. See Appendix B for the entire 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).

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

- Constants

- Variables
  - — Fixed-length character strings and their internal representation
  - — Varying-length character strings and their internal representation
- Alignment of character strings

3.3.1 Character-String Constants

A character-string constant can consist of any characters in the DEC Multinational Character Set (see Appendix B). When you use character-string constants in a program, you must enclose the 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 previously shown, the final result contains only a single apostrophe.

Note that the quotation mark (") is not a legal delimiter for PL/I character constants.

3.3.1.1 Replication of String Constants

You can use a replication factor to replicate character-string and bit-string constants in any context of the program. A replication factor is an unsigned integer constant that specifies the number of times a simple string constant is replicated to produce a resulting string constant. For example:

```
(4)'season   '
```

In this example, the string is repeated four times. The character constant resulting from this specification is equivalent to:

```
'season   season   season   season   '
```

You can use a replication factor in combination with the iteration factor in INITIAL. For example, the following two statements are equivalent:

```
INITIAL ((10)('ABCABC'))
INITIAL ((10)((2)'ABC'))
```

The first example uses an iteration factor exclusively, but the second example combines an iteration factor of 10 with a replication factor of 2. Note that an extra set of parentheses is required to separate the iteration factor from the replication factor and the character string.

3.3.2 Character-String Variables

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

```
DECLARE variable-name [ CHARACTER [(n)]
                      | VARYING CHARACTER [(n)]
                      | CHARACTER [(n)] VARYING ];
```

The CHARACTER keyword identifies a character-string variable in a declaration.

The addition of the VARYING attribute indicates a varying-length character-string variable.

An optional number in parentheses specifies the length of the variable, that is, the number of bytes needed to contain its value (maximum is 32,767). This length attribute 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). If the length is not specified, PL/I uses the default length of one character, or byte. The rules for specifying the length are:

- For a static variable declaration, the length must be an integer constant.
- In the declaration of a parameter or returns descriptor, you can specify the length as an integer constant or as an asterisk (*). The resulting string is fixed length unless VARYING is also specified.
- For an automatic, based, or defined variable, you can specify the length 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.
- The maximum length in any declaration is 32,767.

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

3.3.2.1 Fixed-Length Character-String Variables

A fixed-length character string is one that does not have the VARYING attribute. For a particular allocation of a fixed-length character-string variable, all its values have the same length. When a program assigns a value to a fixed-length character-string variable, however, the value does not need to have the same length defined for the variable. Depending on the size of the value, PL/I adjusts the assignment length according to the following rules:

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

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

The final value of the variable STRING 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 character string on the right. For example:

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

Here, the final value of STRING is 'ABCD', that is, the first four characters of the value 'ABCDEF'.

3.3.2.2 Internal Representation of Fixed-Length Character Data

PL/I stores fixed-length character string data in a contiguous sequence of bytes with the leftmost character occupying the lowest memory address.

3.3.2.3 Varying-Length Character-String Variables

When you define a character-string variable, you can also specify the VARYING attribute. In a varying character-string variable, the 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, 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. 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 unless the variable is initialized.

You can use the LENGTH built-in function to determine the current length of any string, and the MAXLENGTH built-in function to determine the maximum length.

3.3.2.4 Internal Representation of Varying Character Data

A varying-length character string consists of a word specifying the string's current length, followed by a sequence of bytes in sequentially higher memory addresses.

3.3.3 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.

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.

All character strings on the VAX and Alpha hardware are aligned on byte boundaries. You should not use the ALIGNED attribute to declare character-string variables.

3.4 Bit-String Data

A bit string consists of a sequence of binary digits, or bits. It can be used as a Boolean value, which has one of two states: true (if any bit is non-zero) 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. The maximum length of any bit string is 32,767 bits. However, bit-string variables cannot be declared with the VARYING attribute.

The rest of this section discusses bit-string constants and variables, alignment of bit-string data, and the use of bit strings to represent integers.

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

- Constants
- Variables
- Alignment
- Internal representation
- Bit strings and integers

3.4.1 Bit-String Constants

To specify a bit-string constant, enclose the string in apostrophes and follow the closing apostrophe with the letter B. For example:

```
'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. You can specify a bit-string constant with a maximum of 1000 characters between the apostrophes.

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

'character-string'Bn

n

Is the number of bits to be represented by each digit in the string. n can have the value 1 through 4, and if not specified defaults to 1.

This format allows you to specify bit-string constants with bases other than 2. You can use base 4, 8, and 16, where n equals 2, 3, and 4 respectively. 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, not as integer representations of the value.

The valid characters for each type of bit-string constant are as follows:

- 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 through F can be either upper- or lowercase)

Using the B format items, you can also acquire or output (with the GET EDIT and PUT EDIT statements) bit-string data in binary, base 4, octal, or hexadecimal format. See Section 9.2.4.2 for more information on the B format item.

3.4.1.1 Replication Factor for Bit-String Constants

A replication factor is an unsigned integer constant that specifies the number of times a simple bit-string constant is replicated. A replication factor permits repetition of bit strings in any context where a simple string constant is permissible, including format items and assignment, string, and arithmetic operations. The format of a replication factor is as follows:

(r)'string'Bn

r

An unsigned integer that represents the number of times that the string is to be replicated.

string

A simple bit string constant to be replicated. The bit string is enclosed in apostrophes.

An example of replication is:

```
DECLARE (A) BIT (800);
     A = (400) '2'B2;
     PUT SKIP LIST ((A));
END;
```

In this example, A will be replicated to its maximum specified length of 800 characters.

The resulting character constant looks like this:

```
'10101010101010101010101010101010
    .
    .
    .
10101010101010101010101010'B
```

You can use the replication factor in combination with the iteration factor in INITIAL. For example, the following two statements are equivalent:

```
INITIAL ((10)('ABCABC'B4))
INITIAL ((10)((2)'ABC'B4))
```

The first statement uses an iteration factor exclusively; the second statement combines an iteration factor of 10 with a replication factor of 2. An extra set of parentheses is required to separate the iteration factor from the replication factor and the bit string.

3.4.2 Bit-String Variables

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

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

You can optionally specify the length of the variable in parentheses. The length can be from 0 to 32,767; the default length is one bit. The rules for specifying the length are as follows:

- If BIT is specified for a static variable declaration or in a returns descriptor, the length must be an integer constant.
- If BIT is specified in the declaration of a parameter or in a parameter descriptor, you can specify the length as an integer constant or as an asterisk (*).
- If BIT is specified for an automatic, based, or defined variable, you can specify the length 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.

A program can assign to a bit-string variable a value larger or smaller than the variable's defined length. In such cases, PL/I does the following:

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

You can convert bit-string variables to other data types; however, there are some precautions you must observe if you do so. Section 6.4 describes how to convert bit-string variables.

3.4.3 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 following example:

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

If the default packed alignment is in effect, 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. If natural alignment is in effect, PL/I allocates storage for an aligned bit-string variable on a longword boundary and reserves an integral number of longwords to contain the variable. See the PL/I for OpenVMS Systems User Manual for more information on the /ALIGN and /DATA command line qualifiers, which affect what type of alignment is in effect.

Unaligned bit-string variables always occupy only as many bits as are needed to contain them. They need not be on byte boundaries. You can optionally specify the UNALIGNED attribute in a declaration; UNALIGNED is the default for bit strings.

In general, operations involving unaligned bit-string variables are less efficient than those involving aligned bit-string variables. Unaligned bit-string variables are also 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 the strings to be aligned.

In most cases, you should declare bit-string variables with the ALIGNED attribute. Use unaligned bit-string variables when bit strings must be packed as tightly as possible, for example, in arrays and in structures. See Section 2.2.1 for a description of the ALIGNED attribute.

3.4.4 Internal Representation of Bit Data

In this discussion, the term most significant bit means the leftmost bit in an external representation of a 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.

The notion of significance has no meaning for bit strings unless they are used to store integers. PL/I permits the use of bit strings for this purpose, and has defined rules for conversions between bit strings and other data types.

NOTE: The use of PL/I bit-string data to store integers is not recommended, for two reasons:

- **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.**

- **As shown in the following discussions, the significance of bits results in bit strings being stored in the reverse order from actual numeric data. Consequently, conversion of bit strings to arithmetic data is expensive in terms of execution speed, except in the special case of a 1-bit string.**

You should use the UNSPEC built-in function and UNSPEC pseudovariable when you must store integers in a compact form. Otherwise, use the data types FIXED BINARY and FIXED DECIMAL for integer arithmetic.

Note: Sophisticated applications that depend on the internal representation of bit strings and other types of data may not be directly transportable from other PL/I implementations to Kednos PL/I.

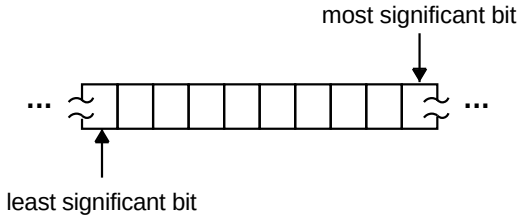
In Kednos PL/I, bit strings are stored in memory with the *leftmost bit* (as represented by PUT LIST) in the *lowest* memory location, and bits following the leftmost in successively higher memory locations. This representation of a bit string is reversed by PUT with respect to a conventional picture of memory locations, in which the *lowest* location appears on the *right* and higher locations on the left. If you are accustomed to using PL/I on computers other than OpenVMS systems and if you do not change your data to correct for this difference, the result is likely to be in error.

If you wish to use bit strings to represent integers and you would like to associate the leftmost bit (as it might be represented in a PUT LIST statement) with the Most significant bit, use the BINARY builtin function, an example of which appears in Section 11.4.43.

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 in Figure 3–8.

Figure 3–8 Unaligned Bit String Storage



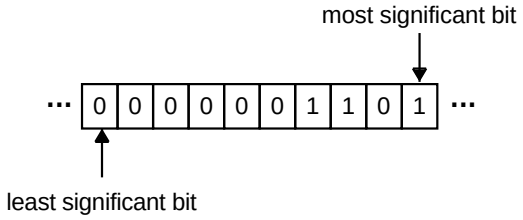
NU−2445A−RA

The following programming sequence shows 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 as shown in Figure 3–9.

Figure 3–9 Sample Unaligned Bit String Storage



NU–2446A–RA

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 be allocated is calculated as:

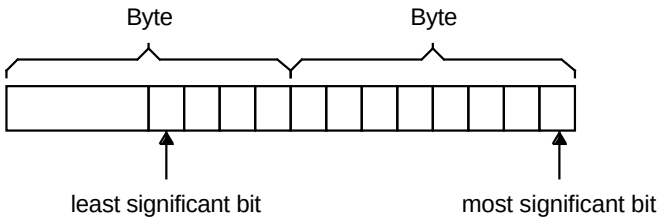
$ceil(n/8)$

Here, 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 0 each time the bit-string variable is assigned a value.

The representation is shown in Figure 3–10.

Figure 3–10 Aligned Bit String Storage



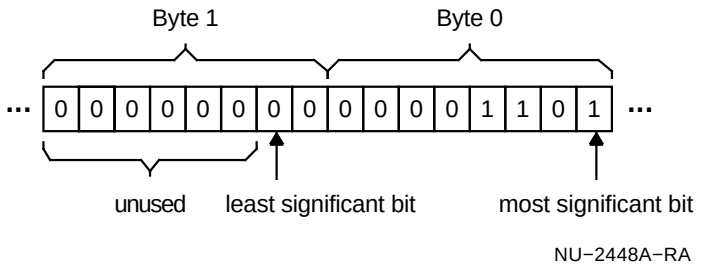
NU–2447A–RA

The following programming sequence shows 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, the value is stored as shown in Figure 3–11.

Figure 3–11 Sample Aligned Bit String Storage



3.4.5 Bit Strings and Integers

PL/I defines conversions between bit-string data and other data types, and the PL/I compiler carries out these conversions. However, the conversions defined by PL/I are not always straightforward or intuitive; the padding and truncation that take place during assignment of bit strings of different lengths result in implicit multiplication or division of the bit string's integer value. For example:

```
DECLARE BITSTR BIT (10);
        BITSTR = 1;
        PUT LIST (BITSTR);
```

The output is:

```
'0001000000'B
```

The result may seem incorrect, but it conforms to PL/I's rules for conversion to bit strings. In this case, the fixed-decimal constant 1 is converted to a FIXED BINARY(4) value, which is in turn converted to an intermediate bit string of length 4:

```
'0001'B
```

Next, this intermediate bit string is assigned to the variable BITSTR. Because BITSTR is of length 10, the intermediate bit string is padded on the right with zeros, producing the result as output by PUT LIST. If you now attempt to interpret the value of BITSTR as an integer (for example, by using BITSTR as the argument of the BINARY built-in function), the result would be 64, not 1.

Extra execution time is required to reverse the order of bits when the integer's value is computed. Using arithmetic variables to represent integers is more efficient.

Because of the unexpected results and longer execution time, avoid using bit strings to represent integers or other data types.

3.5 Pointer Data

A pointer is a variable whose value represents the address in memory of another variable or data item.

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. Pointers may also be used directly in declarations of based variables. 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.

This section discusses the following:

- Pointer variables in expressions
- Internal representation of pointer data

3.5.1 Pointer Variables in Expressions

Expressions containing pointer variables are restricted to the relational operators equal (=) and not equal (^=).

For example, to test whether a pointer is currently pointing to valid storage, you can write the following statement:

```
IF LIST_POINTER ^= NULL() THEN
   DO;
```

The NULL built-in function always returns a null pointer value.

You can use pointer variables in simple assignment statements that assign a pointer value to a pointer variable. For example:

```
LIST_POINTER_1 = LIST_POINTER_2;

LIST_END = NULL();
```

You can also use a pointer variable as the source or target in an assignment statement involving an offset variable or offset value.

3.5.2 Internal Representation of Pointer Data

A pointer occupies a longword (32 bits) of storage and represents a virtual memory address.

3.6 Offset Data

You declare an offset variable with the OFFSET attribute, optionally followed by an area variable reference. The value of the offset variable will be interpreted as an offset within the specified area, unless the POINTER function is used to explicitly specify another area. You must omit the area reference if the OFFSET attribute is specified within a returns descriptor, parameter declaration, or parameter descriptor. For example:

```
DECLARE MAP_SPACE AREA (40960),
        MAP_START OFFSET (MAP_SPACE),
        MAP_LIST(100) CHARACTER(80) BASED (MAP_START);
```

These declarations define an area named MAP_SPACE; an offset variable, MAP_START, that will contain offset values within that area; and a based variable whose storage is located by the value of MAP_START.

Offset variables are given values by assignment from existing offset values or from conversion of pointer values. The OFFSET built-in function (described in Section 11.4.59) converts a pointer value to an offset value. PL/I also automatically converts a pointer value to an offset value, or an offset value to a pointer value, 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 assignment 2, any area references are ignored in the assignment; therefore, the offset value and variable can refer to different areas. In assignments 3 and 4, the offset variable must have been declared with an area reference.

Expressions containing offset variables are restricted to the relational operators = and ^=, for testing the equality or inequality of two values.

3.7 Label Data

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 and 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 this 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. You cannot explicitly declare label constants.

This section discusses the following:

- Label array constants
- Label values
- Label variables
- Internal representation or variable label data

3.7.1 Label Array Constants

Any label constant except the label of a PROCEDURE or FORMAT statement can have a single subscript. Subscripts must be specified with integer constants; a subscript must appear in parentheses following the label name. An example for VAX and Alpha is:

```
PART(1):
   .
   .
   .
PART(2):
   .
   .
   .
PART(*):
```

When labels are written this way, the unscripted label name represents the implicit declaration of a label array constant. In this example, the array is named PART and is treated as if it were declared within the block containing the subscripted labels. In VAX, a default label can be created by using the asterisk (*) in place of a label constant. If a default label is used, it must be the last label in the list. If the variable subscript is out of range and the default label is present, the default label will be executed.

Elements of the array can be referenced in GOTO statements that specify a subscript. For example:

```
GOTO PART(I);
```

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, you cannot use the same subscript value in two different subscripted references with the same name. For example:

```
PART(1):
```

This label array constant 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 this example, the value of ELEMENT cannot cause control to pass 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.

3.7.2 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 with the current block.

The GOTO statement with a 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 the GOTO is nonlocal. 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.

3.7.3 Label Variables

When an identifier is explicitly declared with the LABEL attribute, it acquires the VARIABLE attribute by default. You can use such a variable 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.

You can also give values to label variables by passing label values as arguments or by returning a label value as the value of a function (although the latter method can 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 following:

- 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 the following:

- A DECLARE statement
- A statement beginning an ON-unit or THEN, ELSE, WHEN, or OTHERWISE clauses

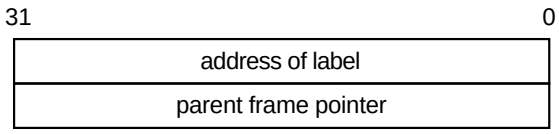
Labels on PROCEDURE, ENTRY, and FORMAT statements are not considered statement labels and cannot be used as the targets of GOTO statements.

An identifier occurring as a label in a block cannot be declared in that block (except as a structure member), and cannot occur in a parameter list of that block.

3.7.4 Internal Representation of Variable Label Data

Figure 3–12 shows the internal representation of variable label data.

Figure 3–12 Variable Label Data Representation



NU–2449A–RA

3.8 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.

This section discusses the following:

- Entry constants
- Entry values
- Entry variables
- Internal representation of variable entry data

3.8.1 Entry Constants

You declare entry constants by using labels on PROCEDURE or ENTRY statements.

You declare internal entry constants by using labels on PROCEDURE or ENTRY statements whose procedure blocks are nested in another block. You can use an internal entry constant anywhere within its scope to invoke its procedure block.

You declare external entry constants either by using labels on PROCEDURE or ENTRY statements that belong to external procedures, or by explicitly declaring the constant names with the ENTRY attribute. You can use an external entry constant 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, you declare external entry constants 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.

3.8.2 Entry Values

Whenever a reference to an entry constant is interpreted, the result is an entry value. An entry value is the entry point of a procedure, and it serves to activate 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.

No conversions are defined between entry data and other data types. You can assign an entry variable 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.

PL/I supports the passing of external procedures, but not internal procedures, as entry value parameters. To pass an internal procedure, use an entry parameter.

3.8.3 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 to an entry variable either another entry variable or an entry constant.

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 the EXTERNAL nor the INTERNAL attribute is specified with the entry variable, the default is INTERNAL.

You can use the entry variable 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;
```

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);
```

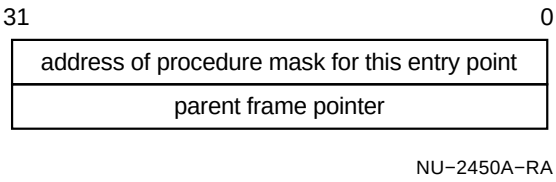
This assignment statement references the third element of the array EXTRACT. When the statement is executed, this array element must contain a valid entry value.

Note: 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.

3.8.4 Internal Representation of Variable Entry Data

Figure 3–13 shows the internal representation of variable entry data.

Figure 3–13 Variable Entry Data Representation



3.9 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.

No conversions are defined between file data and other data types. You can assign a file variable only the value of a file constant or the value of another file variable. The only operations that are valid for file data are comparisons for equality (=) and inequality (^=).

This section discusses the following:

- File constants
- Files values

- File variables

3.9.1 File Constants

You declare file constants by using the FILE attribute without the VARIABLE attribute. All file constants are external by default. To define an internal file constant, you must specify the INTERNAL attribute. For example:

```
DECLARE INFILE FILE;
```

This declaration declares a file constant named INFILE whose attributes include EXTERNAL by default.

```
DECLARE INFILE FILE INTERNAL;
```

This declaration specifies that the file constant named INFILE is internal to the block in which it is declared.

If you declare a file constant as EXTERNAL, you must use identical attributes, including ENVIRONMENT attributes, in all blocks that declare the constant. Otherwise, PL/I uses the last set of attributes encountered during compilation and ignores the others.

3.9.2 File Values

Whenever a reference to a file constant is interpreted, the result is a file value. A file value is a pointer to the file control block for the file with which the constant is associated.

PL/I supports the passing of external files, but not internal files, as file value parameters. To pass an internal file, use a file parameter.

3.9.3 File Variables

File variables are variables (including parameters) that take file values. If the VARIABLE attribute is specified with the FILE attribute in a DECLARE statement, the declared identifier is a file variable. You can assign to a file variable either another file variable or a file constant.

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.

The scope of a file variable name can be either internal or external. If neither the EXTERNAL nor the INTERNAL attribute is specified with the file variable, the default is external.

If you declare a file variable implicitly or explicitly as EXTERNAL, you must use identical attributes, including ENVIRONMENT attributes, in all blocks that declare the variable. Otherwise, PL/I uses the last set of attributes encountered during compilation and ignores the others.

You can use the file variable to represent different files during the execution of the PL/I program. For example:

```
DECLARE F FILE VARIABLE,
        (A,B) FILE;
        E = A;
        CALL READFILE(E);
```

The file constant A is assigned to the file variable E. The CALL statement results in the invocation of the entry point READFILE with file A as its parameter.

You can also declare arrays of file variables. The following example shows an array of external file variables:

```
DECLARE FILELIST(10) FILE VARIABLE,
        MYFILE FILE VARIABLE;

        MYFILE = FILELIST(3);
```

This assignment statement references the third element of the array FILELIST. When the statement is executed, this array element must contain a valid file value.

3.10 Area Data

An area is a region of storage in which based variables can be allocated and freed. You define an area by declaring 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 throughout assignment or transmission.
- You can control the allocation of storage for related variables by placing them in the same area, which improves the locality of reference. Also, you can use one operation to recover the storage for all allocations within an area by freeing or initializing the area itself.
- You can use a structure containing an area to represent a disk file that is mapped into a process's virtual address space.

All areas must be declared with the AREA attribute before they can be referenced in a BASED attribute or an ALLOCATE statement with the IN option. For example:

```
DECLARE MYAREA AREA;
DECLARE PTR OFFSET(MYAREA);
DECLARE MYDATA FIXED BIN(31) BASED(PTR);
```

The variable MYAREA is given the AREA attribute. Then it is used as the target in another declaration, which defines a pointer offset based on MYAREA. To allocate storage for MYDATA in area MYAREA, use the IN option of the ALLOCATE statement as follows:

```
ALLOCATE MYDATA IN(MYAREA) SET (PTR);
```

When these statements are executed, the ALLOCATE statement allocates storage for a variable MYDATA in the area MYAREA and sets PTR to the offset in the area of the allocated storage.

This section discusses the following:

- Area variables in expressions
- Reading and writing areas
- Internal representation of area data

3.10.1 Area Variables in Expressions

Expressions containing area variables are restricted to the relational operators equal (=) and not equal (^=) and to comparison to the empty () built-in function (BIF).

For example, to test whether an area is empty, that is, to determine whether it currently has storage allocated in it, you can write the following statement:

```
IF MYAREA = EMPTY() THEN
   DO;
```

The EMPTY() built-in function always returns an empty area value.

You can use area variables in simple assignment statements that assign one area variable to another. For example:

```
AREA_1 = AREA_2;
AREA_2 = EMPTY();
```

3.10.2 Reading and Writing Areas

An area can be the source or target of data transmission in READ and WRITE record I/O statements. If the area is written by itself (not as a member of a structure), only the current allocated portion is transmitted unless the SCALARVARYING option of the ENVIRONMENT attribute was specified when the file was opened.

3.10.3 Internal Representation of Area Data

An area occupies the number of bytes, specified in the extent, of storage. Since this storage includes overhead used by PL/I for bookkeeping, slightly less than the full amount (specified in the extent) is available for program allocations.

3.11 Condition Data

PL/I provides a CONDITION attribute for declaring programmer-defined conditions. These conditions may only be signaled by the SIGNAL statement.

Condition data occupies a longword (32 bits) of storage.

No conversions are defined between condition data and other data types. The only operations that are valid for condition data are comparisons for equality (=) and inequality (^=).

Unlike some other noncomputational data type (such as ENTRY and FILE), the CONDITION data type may only be used as a constant. You cannot declare condition variables. For example, the following results in a compile-time error:

```
DECLARE (C1, C2, C3) CONDITION;
DECLARE C CONDITION VARIABLE;
```

The compiler will reject the declaration of C in the previous example.

4 Aggregates

An aggregate is a data structure, either an array or structure composed of items as follows:

- An array is an aggregate in which all items, called elements, have the same data type. An individual element of an array is referred to by an integer subscript that designates the element's position, or order, in the array. Elements can be scalar data items or aggregrates.
- A structure is an aggregate in which individual items, called members, can have different data types. Individual members are referred to by qualified references. Members can be scalar data items or aggregrates.

4.1 Arrays

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 elements that the array contains and the organization of those elements. These attributes of an array are called its dimensions.

4.1.1 Array Declarations

To declare an array, specify its dimensions in a DECLARE statement using one of the following syntaxes:

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

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

To declare two or more array variables that have the same dimensions, bounds, and attributes, use the following syntax:

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

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 one of the following:

- Two expressions separated by a colon giving the lower and upper bounds for that dimension
- A single expression giving the upper bound only (the lower bound is then 1 by default)

- An asterisk (*), used in the declaration of array parameters, indicating that the parameter can be matched to array arguments with varying numbers of elements in that dimension

Bound pairs in series 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 optional keyword DIMENSION or the list of declarations. The following rules apply to specifying the dimensions of an array and the bounds of a dimension:

- An array can have up to eight dimensions.
- 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. A restricted integer expression is one that yields only integral results and has only integral operands, which can be evaluated at translation time. Such an expression can use only the addition (+), subtraction (-), and multiplication (*) operators.
  - — If the array has the AUTOMATIC, BASED, CONTROLLED, 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.
- The value of the lower bound you specify must be less than the value of the upper bound.

Table 4–1 shows several forms of bound pairs as used in declarations. Note that all the examples in Table 4–1 would be identical in effect if the optional keyword DIMENSION were added.

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.

Table 4–1 Specifying Array Dimensions

Bound Pairs	Examples
ARRAY_NAME (bound) A single value specifies: • That the array has a single dimension. • That the number of elements in the dimension is the bound (that is, 6). • That the value specified is the high bound, which is the largest numbered element. By default, the low bound is 1.	`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. • That the first value specified is the low bound and the second value specified is the high bound.	`DECLARE TEMPERATURES (-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) ;`

Table 4–1 (Cont.) Specifying Array Dimensions

Bound Pairs	Examples
ARRAY_NAME (low-bound1:high-bound1,low-bound2,high-bound2, . . .)	
A list of ranges specifies: • That the array is multidimensional. Each range of values represents a dimension of 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 ;`

The declaration of an array specifies its dimensions, the bounds of each dimension, and the attributes of the elements.

One bound pair is specified for each dimension of the array to define the number of elements in that dimension. The total number of elements in the array, called its extent, is the product of the number of elements in all the dimensions of the array. If omitted, the lower bound is 1 by default.

You can use an asterisk (*) as the bound pair when you declare arrays 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.) For example:

```
DECLARE SALARIES (100) FIXED DECIMAL (7,2);
```

This statement 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 following statement declares a two-dimensional array of 64 integers:

```
DECLARE GAME_BOARD (8,8) FIXED BINARY (7);
```

The following statement declares a one-dimensional array of 12 character strings, each having a length of 2:

```
DECLARE PM_HOURS(13:24) CHARACTER(2);
```

The elements of the previous array is numbered 13 through 24 instead of 1 through 12.

You can replace the identifier in a statement with a list of declarations, which declares several arrays with the same attributes. For example:

```
DECLARE (SALARIES,PAYMENTS)(100) FIXED DECIMAL(7,2);
```

This statement declares SALARIES and another array, PAYMENTS, with the same dimensions and other attributes.

4.1.2 References to Individual Elements

You refer to an individual element in the array with subscripts. Because 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 those attributes.

You must enclose subscripts 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 you can specify for that dimension.

The lower and upper bounds that you declare for a dimension determine the range of subscripts that you can specify for that dimension. The number of elements in any dimension of any array is:

$$(upperbound) - (lowerbound) + 1$$

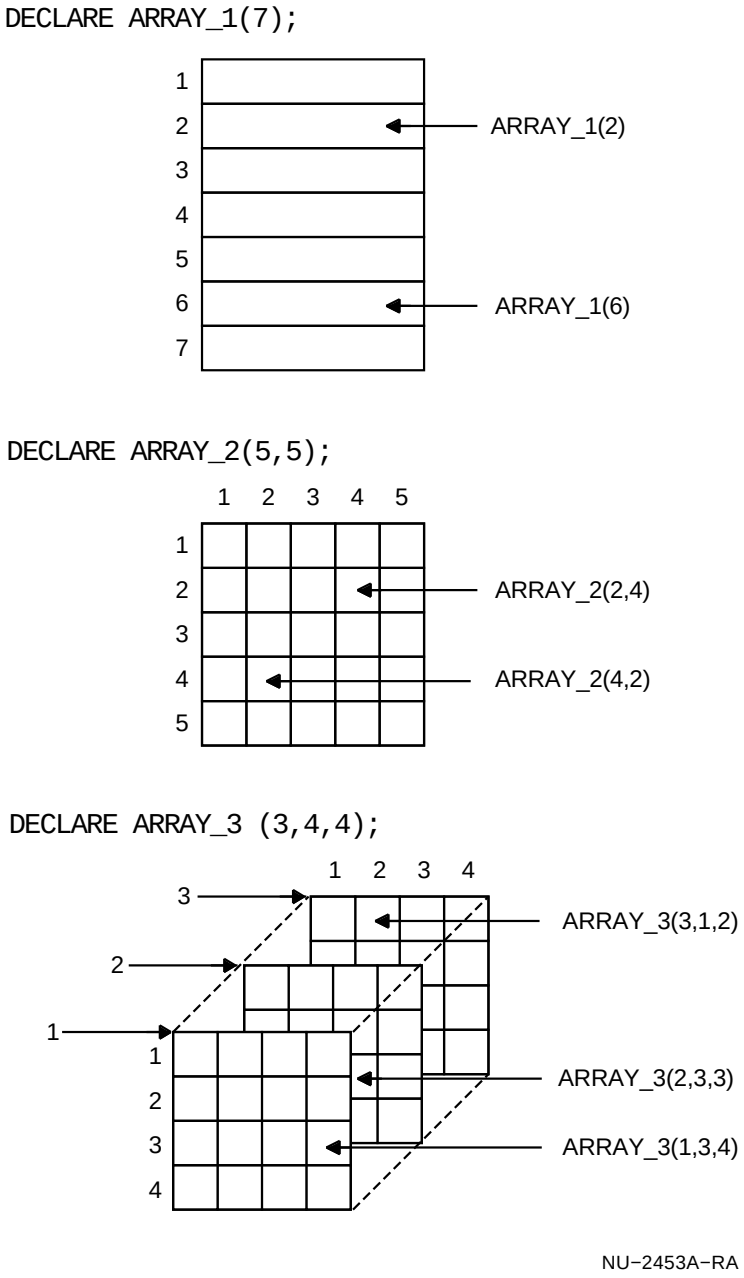
For multidimensional arrays, the subscript values represent an element's position with respect to each dimension in the array. Figure 4–1 shows subscripts for elements of one-, two-, and three-dimensional arrays. In subscripted references for multidimensional arrays, the number of subscripts must match the number of dimensions of the array and must be separated by commas.

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.

Figure 4–1 Specifying Elements of an Array



4.1.3 Initializing Arrays

Specify the INITIAL attribute for an array to initialize its values in the declaration. For example:

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

Each element of the array MONTHS is assigned a value according to the order of the character-string constants in the initial list: 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.

To assign identical initial values to some or all elements of an array, you can use an iteration factor with the INITIAL attribute. For example:

```
DECLARE TEST_AVGS (30,4) FIXED DECIMAL (5,2)
    STATIC INITIAL ((120) 50);
```

This statement declares the array TEST_AVGS with 120 elements, each of which is given an initial value of 50.

You can also use the asterisk (*) iteration factor to initialize all the elements of an array to the same value. For example:

```
DECLARE TEST_AVGS (30,4) FIXED DECIMAL (5,2)
    STATIC INITIAL ((*) 50);
```

This statement also declares the array TEST_AVGS with 120 elements, each of which is given an initial value of 50.

Although Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha both support the initialization of automatic arrays with the INITIAL attribute, for the following reasons this is not always the most efficient way (in terms of program compilation and execution) to initialize array elements:

- When you initialize elements in an array that has the AUTOMATIC, BASED, or CONTROLLED attribute, the compiler does not check that all elements are initialized until run time. Thus, you do not receive any compile-time checking of initialization, even if you used constants to specify the array bounds and iteration factors.
- Your programs will run more efficiently if you initialize automatic arrays with assignment statements rather than the INITIAL attribute.

If the array is not modified by your program, you can increase program efficiency by declaring the array with the STATIC and READONLY attributes and using the INITIAL attribute to initialize its elements. In this case, the compiler checks that you have initialized all the elements and that they are valid.

Iteration Factors

When more than one successive element of an array is to be assigned the same value with the INITIAL attribute, 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 the specified value is to be used 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 element is not to be assigned an initial value.

You can use any of these forms for arrays that have the AUTOMATIC attribute. For arrays with the STATIC attribute, you can use only constants and the NULL built-in function.

For example, the following declaration of the array SCORES initializes all elements of the array to 1:

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

The next 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 following declaration initializes the first 49 elements to 1; the next 2 elements are not initialized; and the next 49 elements are initialized to -1:

```
DECLARE SCORES (100) FIXED STATIC INITIAL ((49)1,(2)*,(49)-1);
```

The declaration in the next example initializes all 10 elements of an array of character strings to the 26-character value in apostrophes. The string constant is enclosed in parentheses; this is required to differentiate between iteration factors and replication factors.

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

4.1.4 Assigning Values to Array Variables

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

array-variable = expression;

This is valid 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. You can use the asterisk in an assignment:

array-variable (*, . . .) = expression;

You can use a single asterisk regardless of how many dimensions an array has, or you can use an asterisk for each dimension.

Note that the arithmetic operators, such as the plus sign (+) and the minus sign (-), cannot have arrays as operands. An assignment of the following form is invalid:

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

The following assignment is valid where the specified array variables have identical data-type attributes and dimensions:

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

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 they occupy must not overlap, unless the arrays are identical.

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

4.1.5 Order of Assignment and Output for Multidimensional Arrays

When a multidimensional array is initialized, or when it is assigned values 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 previously shown.

4.1.5.1 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 input field delimited by blanks, tabs, or commas 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.

4.1.6 Passing Arrays as Arguments

You can pass an array variable 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 as follows:

- If you specify the bounds with integer constants, they must match exactly the bounds of the corresponding argument.
- You can specify all bounds as asterisks (*). Then, 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 can 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. Arrays are always passed by reference.

4.1.7 Built-In Functions Providing Array Dimension Information

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 procedure that follows 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 one-dimensional array as a parameter and initializes the elements of the array with integral values beginning with 1.

4.2 Structures

A structure is a data aggregate consisting of one or more members. The members can be scalar data items or aggregrates. Different members can have different data types. Structures are useful when you want to group related data items having different data types.

4.2.1 Structure Declarations and Attributes

The declaration of a structure defines its organization and the names of members at each level in the structure. The major structure name is declared as structure level 1; minor members must be declared with level numbers greater than 1. For example:

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST CHARACTER(80) VARYING,
            3 FIRST CHARACTER(80) VARYING,
          2 SALARY FIXED DECIMAL(7,2);
```

This statement declares a structure named PAYROLL. You can access the last name with a qualified reference:

```
PAYROLL.NAME.LAST = 'ROOSEVELT';
```

Alternatively, because the last and first names have the same attributes, you can declare the same structure as follows:

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 (LAST,FIRST) CHARACTER(80) VARYING,
          2 SALARY FIXED DECIMAL(7,2);
```

The following additional rules apply to the specification of level numbers:

- Level numbers must be specified with 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 have any integer value, 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.

Attributes for Structure Variables

Within a structure, you can only declare members at the lowest level of each substructure with data-type attributes. Additional rules for specifying attributes for the various components of a structure are as follows:

- Only the following attributes are valid for the major structure:

AUTOMATIC	GLOBALREF
BASED	INTERNAL
CONTROLLED	READONLY

DEFINED	STATIC
EXTERNAL	STRUCTURE
GLOBALDEF	UNION
TYPE	

- You can dimension the major structure, a minor structure, or any member of the structure: that is, there can be arrays of structures and structures whose members are arrays.
- Member names cannot have any of the attributes a major structure can have except for INTERNAL and UNION attributes. You can use the UNION attribute on any member with a level number.
- If a structure has the STATIC attribute, the extents of all members (lengths for character- and bit-string variables, dimensions for array variables, and area extents) must be specified with optionally signed decimal integer constants.

4.2.2 Using The UNION Attribute On Structure Declarations

A union is a variation of a structure in which all immediate members occupy the same storage. The UNION attribute (which must be associated with a level number in a structure declaration) declares a union. All immediate members of the union-that is, all members having a logical level number one higher-occupy the same storage. A reference to one member of a union refers to storage occupied by all members of the union. Therefore, a union provides a convenient way to look at a large entity (such as a character string or a bit mask) as a series of smaller entities (such as component character strings or individual flag bits).

A variable declared with the UNION attribute must be a major or minor structure. All members of a union must have a constant size (see Chapter 2 for format and details).

The UNION attribute is not part of the PL/I General-Purpose Subset; it is provided in Kednos implementations of PL/I to give users convenient access to data as it is internally represented. Potential applications of unions might depend on the internal representation of data, and would therefore not be transportable between OpenVMS VAX and OpenVMS Alpha systems. The following example shows unions:

```
DECLARE 1 CUSTOMER_INFO,
            .
            .
            .
          2 PHONE_DATA UNION,
            3 PHONE_NUMBER CHARACTER (13),
            3 COMPONENTS,
              4 LEFT_PAREN CHARACTER (1),
              4 AREA_CODE CHARACTER (3),
              4 RIGHT_PAREN CHARACTER (1),
              4 EXCHANGE CHARACTER (3),
              4 HYPHEN CHARACTER (1),
              4 SPECIFIC_NUMBER CHARACTER (4),
          2 ADDRESS_DATA,
            .
            .
            .
```

The UNION attribute associated with the declaration of PHONE_DATA signifies that PHONE_DATA's immediate members (PHONE_NUMBER and COMPONENTS) occupy the same storage. Any modification of PHONE_NUMBER also modifies one or more members of COMPONENTS; conversely, modification of a member of COMPONENTS also modifies PHONE_NUMBER. Note, however, that the UNION attribute does not apply to the members of COMPONENTS because they are not immediate members of PHONE_DATA. The members of COMPONENTS occupy separate storage in the normal fashion for structure members.

Unions provide capabilities similar to those provided by defined variables. However, the rules governing defined variables are more restrictive than those governing unions. The following example (for VAX only) demonstrates a use of a union that would not be possible with a defined variable:

```
DECLARE 1 X UNION,
          2 FLOAT_NUM FLOAT BINARY (24),
          2 BREAKDOWN,
            3 FRAC_1 BIT (7),
            3 EXPONENT BIT (8),
            3 SIGN BIT (1),
            3 FRAC_2 BIT (16);
```

The union X has two immediate members, FLOAT_NUM (a floating-point variable) and BREAKDOWN. The members of BREAKDOWN are bit-string variables that overlay the storage occupied by FLOAT_NUM and provide access to the individual components of an VAX floating-point value. Assignment to FLOAT_NUM modifies the members of BREAKDOWN and vice versa. For example:

```
EXPONENT = '0'B;
SIGN = '1'B;

FLOAT_NUM = FLOAT_NUM + 1;
```

The first two assignment statements set the exponent and sign fields of FLOAT_NUM to the reserved operand combination; the expression FLOAT_NUM + 1 causes a reserved operand exception to occur.

Note that, unlike the character-string example that precedes it, this example depends on the VAX internal representation of data.

4.2.3 Initializing Structures

You can initialize a structure 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.

In a union, the same data can only be initialized once.

4.2.4 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.

4.2.5 Passing Structure Variables as Arguments

You can pass a structure variable 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.

4.2.6 Member Attributes

PL/I supports three member attributes, so named because they apply specifically to the declaration of structure members rather than to the structure as a whole. The member attributes are as follows:

- The TYPE attribute
- The LIKE attribute
- The REFER option

Each is discussed in detail in the following sections.

4.2.6.1 Using the TYPE Attribute

The TYPE attribute copies a scalar, array, or member declaration in a major or minor structure into another scalar, array, or structure variable respectively. The TYPE attribute copies the attributes to the target variable. For structures, the TYPE attribute also copies the logical structuring and member declarations from the major or minor structure to the target variable. TYPE does not copy any storage class or INITIAL attributes or dimensioning (except for dimensioning that is applied to arrays and members) from scalars, arrays, or structures.

Note that the TYPE attribute is a superset of the LIKE attribute. The TYPE attribute is identical to the LIKE attribute when it is used to copy a member declaration in a major or minor structure declaration into another structure variable.

An identifier names the variable to which the declarations for the reference are copied. The reference is the name of a scalar, an array, or a major or minor structure known to the current block. For structures, the identifier must be preceded by a level number. Any attributes that can be used with a structure variable at that level can be used with the identifier. For example, a major structure can specify a storage class and dimensions, and a minor structure can specify dimensions. The following example shows the TYPE attribute:

```
DECLARE NO_OF_SINGLE_ROOMS FIXED BINARY(31);
DECLARE NO_OF_DOUBLE_ROOMS TYPE (NO_OF_SINGLE_ROOMS);
```

In the previous example, the declaration of NO_OF_DOUBLE_ROOMS uses the TYPE attribute to create a declaration that duplicates the attributes of NO_OF_SINGLE_ROOMS. The declaration of NO_OF_ DOUBLE_ROOMS is equivalent to the following:

```
DECLARE NO_OF_DOUBLE_ROOMS FIXED BINARY(31);
```

In the next example, the declaration uses the TYPE attribute to create the declaration that duplicates the attributes of BED_SERIAL_NOS:

```
DECLARE BED_SERIAL_NOS((NO_OF_SINGLE_ROOMS + NO_OF_DOUBLE_ROOMS),2)
                                                    CHARACTER(12);
DECLARE TABLE_SERIAL_NOS TYPE(BED_SERIAL_NOS);
```

The declaration of TABLE_SERIAL_NOS in the previous example is equivalent to the following:

```
DECLARE TABLE_SERIAL_NOS((NO_OF_SINGLE_ROOMS + NO_OF_DOUBLE_ROOMS),2)
                                                    CHARACTER(12);
```

In the following example, the declaration of NEW_RESER uses the TYPE attribute to create a set of member declarations that duplicate those in RES_DATA:

```
DECLARE 1 RES_DATA BASED (RPTR),
          2 DATE CHARACTER(8),
          2 HOTEL_CODE CHARACTER(3),
          2 PARTY_NAME,
            3 LAST CHARACTER(20),
            3 FIRST CHARACTER(10),
          2 STAY FIXED BIN(7),
        1 NEW_RESER TYPE(RES_DATA),
            .
            .
            .
    GET LIST (NEW_RESER.DATE,NEW_RESER.HOTEL_CODE);
       .
       .
       .
    RES_DATA = NEW_RESER;
```

The declaration of NEW_RESER in the previous example is equivalent to the following:

```
DECLARE 1 NEW_RESER,
          2 DATE CHARACTER(8),
          2 HOTEL_CODE CHARACTER(3),
          2 PARTY_NAME,
            3 LAST CHARACTER(20),
            3 FIRST CHARACTER(10),
          2 STAY FIXED BINARY(7);
```

In the previous example, the members of NEW_RESER are assigned data after that data is validated; the entire contents of NEW_RESER are assigned to RES_DATA. This assignment is possible because the two structures are identical as a result of using the TYPE attribute.

You can use the TYPE attribute to copy a minor structure to a major structure and vice versa; neither the level numbers nor the logical levels must match. For example:

```
DECLARE 1 PARTY_NAME,
          2 LAST CHAR(20),
          2 FIRST CHAR(10);

DECLARE 1 SPOUSE_NAME TYPE(PARTY_NAME);
```

Given the declarations in the preceding example, this declaration is equivalent to the following:

```
DECLARE 1 SPOUSE_NAME,
          2 LAST CHAR(20),
          2 FIRST CHAR(10);
```

You can also apply dimensions or, for a major structure, storage-class attributes to a structure variable declared with the TYPE attribute, as follows:

```
DECLARE 1 KID_NAMES (10) TYPE(PARTY_NAME);
```

Or, you can use:

```
DECLARE 1 DAILY_DATA,
          2 DATE CHAR(8),
          2 TODAYS_RESERS (NO_OF_RES) TYPE(RES_DATA);
             .
             .
             .
```

4.2.6.2 Using the LIKE Attribute

The LIKE attribute copies the member declarations in a major or minor structure declaration into another structure variable. It copies the logical structuring and member declarations from the major or minor structure to the target variable, but does not copy any storage-class attributes or dimensioning (except for dimensioning that is applied to members).

An identifier names the variable to which the declarations in the reference are copied. The reference is the name of a major or minor structure known to the current block. The identifier must be preceded by a level number. Any attributes that can be used with a structure variable at that level can be used with the identifier; for example, a major structure can specify a storage class and dimensions, and a minor structure can specify dimensions.

The following example shows the LIKE attribute:

```
DECLARE 1 RES_DATA BASED (RPTR),
          2 DATE CHARACTER(8),
          2 HOTEL_CODE CHARACTER(3),
          2 PARTY_NAME,
            3 LAST CHARACTER(20),
            3 FIRST CHARACTER(10),
          2 STAY FIXED BIN(7),
        1 NEW_RESER LIKE RES_DATA,
            .
            .
            .
    GET LIST (NEW_RESER.DATE,NEW_RESER.HOTEL_CODE);
       .
       .
       .
    RES_DATA = NEW_RESER;
```

In the previous example, the declaration of NEW_RESER uses the LIKE attribute to create a set of member declarations that duplicate those in RES_DATA. The declaration of NEW_RESER is equivalent to the following:

```
DECLARE 1 NEW_RESER,
          2 DATE CHARACTER(8),
          2 HOTEL_CODE CHARACTER(3),
          2 PARTY_NAME,
            3 LAST CHARACTER(20),
            3 FIRST CHARACTER(10),
          2 STAY FIXED BINARY(7);
```

In the previous example, the members of NEW_RESER are assigned data after that data is validated, the entire contents of NEW_RESER are assigned to RES_DATA. This assignment is possible because the two structures are identical as a result of using of the LIKE attribute.

You can use the LIKE attribute to copy a minor structure to a major structure and vice versa; neither the level numbers nor the logical levels must match. For example:

```
DECLARE 1 SPOUSE_NAME LIKE PARTY_NAME;
```

Given the declarations in the preceding example, this declaration is equivalent to the following:

```
DECLARE 1 SPOUSE_NAME,
          2 LAST CHAR(20),
          2 FIRST CHAR(10);
```

You can also apply dimensions or, for a major structure, storage-class attributes to a structure variable declared with the LIKE attribute:

```
DECLARE 1 KID_NAMES (10) LIKE PARTY_NAME;
```

or

```
DECLARE 1 DAILY_DATA,
          2 DATE CHAR(8),
          2 TODAYS_RESERS (NO_OF_RES) LIKE RES_DATA;
             .
             .
             .
```

4.2.6.3 Using the REFER Option

Use the REFER option to create self-defining based structures. In a based structure, the value of one member is used to determine the size of the storage space allocated for another member of the same structure. You can use the REFER option in a DECLARE statement to specify array bounds, the length of a string, or the size of an area. The format of the REFER option is as follows:

refer-element REFER (refer-object-reference)

refer-element

An expression that represents the value assigned to the refer object when the structure is allocated. The refer element must satisfy the following conditions:

- It must be an expression that produces a FIXED BINARY(31) value or a value that can be converted to FIXED BINARY (31).
- It cannot reference storage in the structure containing the refer element.

refer-object-reference

A reference to a scalar variable. The refer object reference must satisfy the following conditions:

- It cannot be a subscripted variable reference.
- It cannot be locator qualified.
- It must reference a refer object that is a previous member of the structure containing the REFER option.

The refer object is a scalar variable contained by the structure. The refer object must satisfy the following conditions:

- It must be a previous member of the structure containing the REFER option, which references the refer object.
- It must be scalar; it cannot be dimensioned or a dimensioned array.

- It must have a computational data type.

An example of a structure declaration containing the REFER option is as follows:

```
DECLARE 1 STRUCTURE S BASED(P),
          2 I FIXED BINARY(31),
          2 A CHARACTER(20 REFER(I));
```

For the compiler to allocate storage for a based structure, the structure must have a known size. In the example, the initial length for A is taken from the refer element, 20. However, the REFER option permits the size of the structure to change at run time as the value of the refer object (I) changes. After allocation, the length of A is determined by I.

You can have multiple REFER options within a structure.

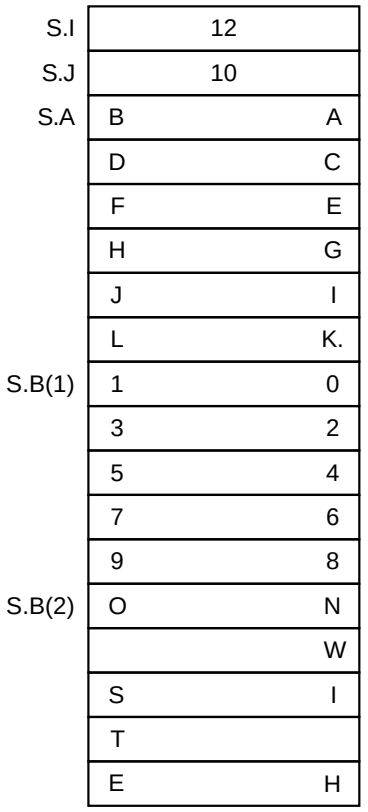
The following example and figures show storage mapping with the REFER option.

```
DECLARE 1 S BASED (POINTER),
          2 I FIXED BINARY(15),
          2 J FIXED BINARY(15),
          2 A CHARACTER ((X*2+2) REFER(I)),
          2 B(2) CHARACTER (Y REFER(J));

X = 5;
Y = 10;

ALLOCATE S;
S.A = 'ABCDEFGHIJKL';
S.B(1) = '0123456789';
S.B(2) = 'NOW IS THE';
    .
    .
    .
END;
```

When this structure is allocated, the refer elements (X*2+2) and Y are evaluated and used to determine the length of the associated string. The evaluated refer element value (X*2+2) is assigned to the refer object I and Y is assigned to J. Thereafter, the sizes of strings A and B are determined by the value of the refer objects I and J.

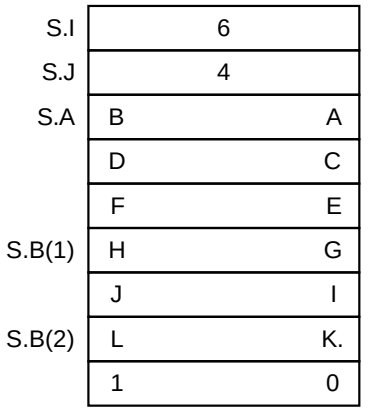
Storage for the previous structure is shown in Figure 4–2.

Figure 4–2 Storage of Structure with REFER Option



If the refer object I is assigned the value 6 and the refer object J is assigned the value 4, the resulting storage is remapped as shown in Figure 4–3.

Figure 4–3 Remapped Storage of Structure with REFER Option



NU–2455A–RA

Note: PL/I does not restrict the use of the REFER option within structure declarations: therefore, exercise caution in its use.

If you change a value that causes the size of one or more structure members to decrease, then some storage at the end of the allocated storage will become inaccessible for future reference.

If the scalar variable (the refer object) does not satisfy the following criteria, the results are undefined:

- It must not be assigned a value that is less than 0 or greater than the refer element value used for structure allocation.
- It must have the value used for allocation, if the structure is freed.

The following rules apply to structures containing the REFER option:

- A structure containing the REFER option cannot be the target of a LIKE reference.
- When a based structure is allocated, the order in which the refer elements are selected for evaluation is undefined.
- When a based structure is allocated, the order in which the refer objects are selected for initialization is undefined.

4.2.7 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 the name of a member or minor structure, you must ensure only that the reference uniquely identifies it. You can qualify the variable name by preceding it with the name or names of higher-level (lower-numbered) variables in the structure; names in this format, called a qualified reference, must be separated by periods (.).

The following sample structure definition shows 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.
- You can qualify the name of any minor structure or member in a structure 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 previous sample, completely qualified, are as follows:

  STATE.NAME
  STATE.POPULATION
  STATE.CAPITAL.NAME
  STATE.CAPITAL.POPULATION
  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 (so long as there are no other variables with these names):

  CAPITAL
  SYMBOLS
  FLOWER
  BIRD

- You can omit intermediate qualification names 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.

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.

4.3 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:

```
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 following example:

```
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 these references are equivalent:

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

4.3.1 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, AVERAGE_TEMPS is an array of 12 elements. Because AVERAGE_TEMPS inherits the dimension of STATE, any of AVERAGE_TEMPS's elements must be referred to by two subscripts:

- The first subscript references an element in STATE.
- The second subscript references an element in 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.

4.3.2 Connected and Unconnected Arrays

A connected array is one 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, NEWSPAPERS is a connected array.

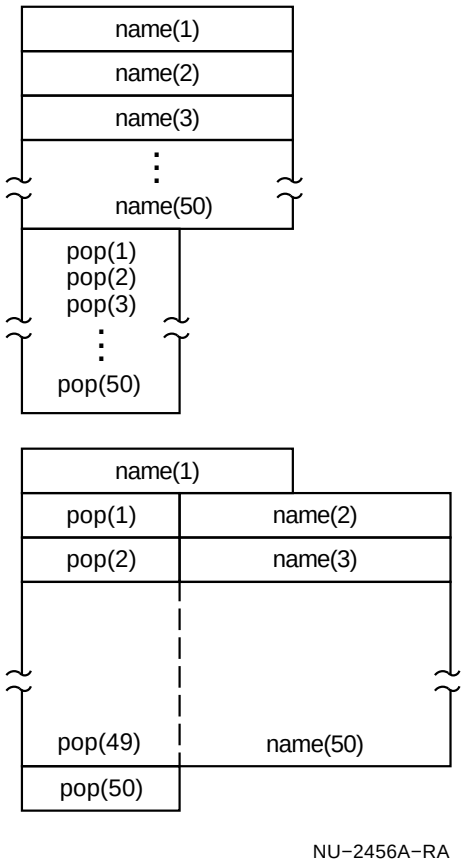
A connected array is valid as the target of an assignment statement, as long as the source expression is a similarly dimensioned array or a single scalar value. The top diagram in Figure 4–4 shows the storage of a connected array.

In an unconnected array, the elements do not occupy consecutive storage locations. The bottom diagram in Figure 4–4 shows the storage of an unconnected array. An unconnected array is not valid in an assignment statement or as the source or target of a record I/O statement. 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 this 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 4–4 Connected and Unconnected Arrays



4.4 Internal Representation of Aggregate Data

Structures can be unaligned or naturally aligned. When a structure is unaligned, each of its members (except for unaligned bit string members) is aligned on a byte boundary. Unaligned bit-string members are bit aligned. In an array of unaligned structures (which contain members other than unaligned bit strings), each structure is aligned on a byte boundary. In an array of unaligned structures that contain only unaligned bit strings, the array elements are bit aligned.

When a structure is naturally aligned, each of its members is aligned as described in Table 4–2. In an array of naturally aligned structures, each structure is aligned on the boundary that is the maximum alignment of its members.

The alignment you select is determined by the compile-time /NOALIGN or /ALIGN switch. The /NOALIGN switch (the default) produces unaligned structures. The /ALIGN switch produces aligned structures as described in Table 4–2.

Table 4–2 Natural Alignment for Structure Members

Data Type	Precision	Alignment
FIXED BINARY(p)	p <= 7	byte
FIXED BINARY(p)	7 < p <= 15	word
FIXED BINARY(p)	p > 15	longword
FIXED DECIMAL(p,q)		word
FLOAT BINARY(p)	p <= 24	longword
FLOAT BINARY(p)	24 < p <= 53	quadword
FLOAT BINARY(p)	p > 53	octaword
FLOAT DECIMAL(p)	p <= 7	longword
FLOAT DECIMAL(p)	7 < p <= 15	quadword
FLOAT DECIMAL(p)	p > 15	octaword
CHAR (UNALIGNED)		byte
CHAR ALIGNED BYTE		byte
CHAR VARYING		word
BIT (UNALIGNED)		bit
BIT ALIGNED		longword
POINTER		longword
LABEL		quadword
ENTRY		quadword
FILE		longword
STRUCTURE		maximum of members
PICTURE		byte
OFFSET		longword

<fnLOCALDATA READONLYCONSTDATA;v>

5 Storage Classes

The storage class to which a variable belongs determines whether PL/I allocates its storage at compile time or dynamically at run time. This chapter describes the following classes of variables:

- Automatic variables, which are allocated storage on activation of the declaring procedure
- Static variables, which are allocated storage at program activation, and which exist for the duration of the program execution
- Internal variables, which can be referenced only by the declaring procedure and its dynamic descendants
- External variables, which can be known to blocks outside the block in which they are declared
- Based variables, which are allocated storage dynamically under program control at run time, and which are accessed by means of a pointer
- Controlled variables, which are allocated dynamically under program control at run time, and which are accessed sequentially, as on a stack
- Defined variables, which are not allocated storage, but instead share with the variable upon which it is defined

Section 5.7 describes the mechanisms for dynamically allocating storage. Section 5.9 describes how variables can share physical storage locations.

Note: Both Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha compilers place an upper limit of 536,870,911 ($2^{29} - 1$) bytes as the maximum size of any data object. The OpenVMS operating system may impose stricter limits depending on the storage-class parameters of the operating system and the parameters associated with your user name. For detailed information on limits, consult the system manager of your operating system.

5.1 Automatic Variables

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 it 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, you can specify the length of an automatic character-string variable as follows:

```
DECLARE STRING_LENGTH FIXED;
  .
  .
  .
COPY: BEGIN;
DECLARE TEXT CHARACTER(STRING_LENGTH);
```

When this begin block is activated, the length of TEXT is evaluated. The variable is allocated storage depending on the value of STRING_LENGTH, which must have a valid value.

5.2 Static Variables

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 you declare it with any of the attributes STATIC, EXTERNAL, GLOBALDEF, or GLOBALREF. In declaring static arrays and strings, you must use restricted expressions. (Note that the EXTERNAL scope attribute implies static storage for variables.)

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 0 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 with 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.

5.3 Internal Variables

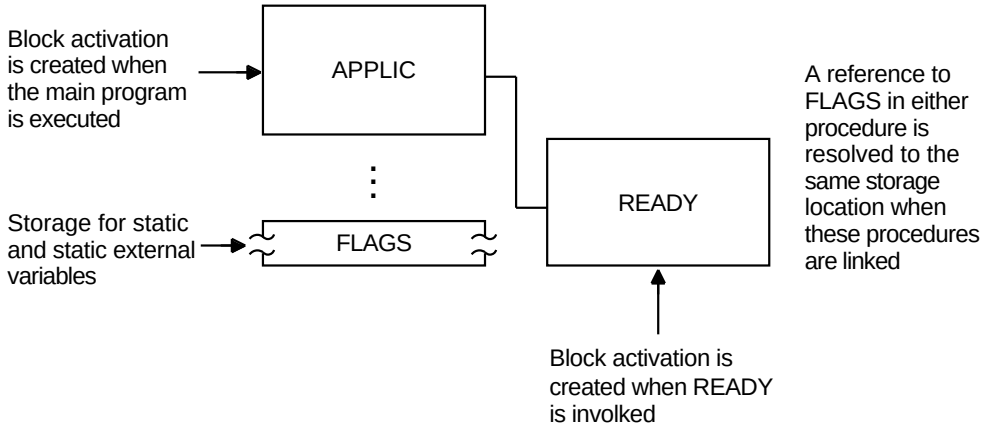
An internal variable is known only within the block in which it is defined and within all contained blocks. By default, PL/I gives all variables the INTERNAL attribute with the exception of data with the FILE and CONDITION attributes.

5.4 External Variables

An external variable provides a way for external procedures to share common data. All declarations that refer to an external variable must also declare it with the EXTERNAL attribute (or with an attribute that implies EXTERNAL) and with identical data type attributes. You can abbreviate the EXTERNAL keyword to EXT. The following example and Figure 5–1 shows how procedures can use external variables:

```
APPLIC: PROCEDURE OPTIONS (MAIN);
DECLARE FLAGS BIT (64) ALIGNED EXTERNAL;
   .
   .
   .
   CALL READY;

READY: PROCEDURE;
DECLARE FLAGS BIT (64) ALIGNED EXTERNAL;
```

Figure 5–1 External Variables



The OpenVMS Linker allows more control over the definition and allocation of external variables than does PL/I. 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 a constant whose value is used immediately in instructions generated by the compiler.

The EXTERNAL attribute is implied by the FILE, GLOBALDEF, GLOBALREF, and CONDITION attributes, and also 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.

The following rules apply to the use of external names:

- The EXTERNAL attribute directly conflicts with the AUTOMATIC, BASED, DEFINED, and INTERNAL 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.
- If you declare a file constant or file variable explicitly or implicitly as EXTERNAL, you must use identical attributes, including ENVIRONMENT attributes, in all blocks that declare the file.

5.5 Based Variables

A based variable is a variable that describes a data type associated with storage that will be accessed through a pointer or offset value. PL/I does not allocate any storage for a based variable. Instead, you must explicitly allocate storage.

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 data storage location. See Section 2.2.5 for information about the BASED attribute. The following example shows a declaration of a based variable:

```
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 allocate any storage for it. 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 by means of the ADDR built-in function. This effectively associates the description of BUFFER with the actual data value of LINE.

You can associate a based variable with a storage location by using the ADDR built-in function, as in the preceding example; by using the ALLOCATE statement; by using a locator-qualified reference to the based variable; by using the SET option of the READ statement; or by explicit allocation within an area.

The following sections cover these topics:

- Data types used with based variables: pointers, areas, and offsets
- Allocation in areas

- Mechanisms for referring to based variables and for obtaining pointer values to them
- Based variables and dynamic storage allocation
- The ADDR built-in function
- Data type matching for based variables
- Examples of based variables in use, including allocation in areas

5.5.1 Data Types Used with Based Variables

The data types most commonly associated with based variables are pointers, areas, and offsets.

A pointer is a variable whose value represents the location in memory of another variable or data item. Pointers are used to access based variables and buffers allocated by the system as a result of the SET option of the READ and ALLOCATE statements.

Areas are regions of storage in which based variables can be allocated and freed. The use of areas can simplify and speed operations involving large or numerous based variables.

An offset is a value indicating the location of a based variable relative to the beginning of an area.

5.5.2 Allocation in Areas

PL/I supports storage management in areas (see Section 3.10 for a description of area data). If you use the ALLOCATE statement with an area (either implied or explicitly specified), you can cause the allocation of storage to be performed in that area, instead of in the general memory pool for based and controlled storage.

Storage management in areas has a number of uses, including the following:

- To allow an area to be moved to different addresses without invalidating its data.
- To allocate storage that can be freed all at once with low overhead, by allocating variables in an area and then emptying the areas with the EMPTY built-in function rather than freeing the generations one at a time.
- To allocate storage that can be rolled back, by allocating variables in an area and making periodic assignment of the area to a backup area.

  If it is found that some operations need to be rolled back, the backup area can then be copied back into the current area (or they can be swapped). Note that when areas are assigned to each other, all offsets into the old area are valid for the new area as well.
- To use areas to overlay shared memory sections, which can be mapped into different address ranges in different processes.

Note that PL/I will signal the AREA condition if allocation or freeing operations are attempted on such an area from multiple processors simultaneously. (This mechanism is intended only as a debugging aid. If you are going to use shared areas, you need to provide the synchronization for the shared data, typically by using the OpenVMS lock manager or run-time library (RTL) routines such as LIB$BBSSI and LIB$BBCCI that provide multiprocessor interlocking capability.)

You allocate storage in areas with the IN and SET options on the ALLOCATE statement, the AREA and OFFSET data attributes, and the EMPTY built-in function.

The IN option on the ALLOCATE statement takes a reference to an area. If the IN option is not specified, the area is implied from the SET option if the SET option specifies an offset variable with a base area. The SET option itself can be implied from the base variable. Whether an area is specified explicitly or implied, the allocation is performed in that area instead of in the available memory pool for based and controlled storage. If an error is detected in the process, the AREA condition is raised.

PL/I provides full support for allocation in areas as specified in the ANSI full PL/I language standard. In addition, it provides extensions to full PL/I, which enhance the usefulness of areas or provide for improved compatibility with other implementations of PL/I. These extensions are as follows:

- Area control information is stored in the form of offsets so that an area can be moved to different addresses and still be correct (see Section 3.6 for a description of offset data). As a result, you can assign areas as members of structures as long as their extents are identical. Note, however, that if such an assignment is performed, the entire area will be copied rather than just the extent. The position independence of the area control information also allows an area to be written to secondary storage and retrieved by another program at a different address. You must ensure that any data allocated in the area is position independent, by restricting locator values stored in the area to offsets specifying that area as a base.
- You can assign areas of differing sizes to each other directly (that is, not as members of structures). The AREA condition will be raised if the target area is not large enough to hold the extent of the source area.
- If you transmit an area by itself (as opposed to a member of an aggregate) with a WRITE or REWRITE statement, only the current extent of the area is transmitted (as in the case of varying-length character strings). You can transmit the entire area by using the SCALARVARYING ENVIRONMENT option.
- The base area in an ALLOCATE statement need not match the base area of the offset. However, if they do match, then one area must contain the other.

- A normal return from an AREA condition due to a full area during an allocation attempt will result in another allocation attempt. An infinite loop will occur if the problem is not corrected, because the area reference is not reevaluated before the retry is attempted after a normal return. The ON-unit must correct the condition by deallocating storage in the area, or by using the EMPTY built-in function.
- The control information for an area is stored inside the area. The control information occupies at least 24 to 31 bytes for header information, plus space for linking unused portions of the area. The number of links needed to link unused portions of the area depends on how fragmented the area is. As a result, there are slightly fewer bytes available for the allocation of user variables in an area than the number of bytes the area is declared with.
- Area variables are not initially empty. They must be explicitly initialized. For example:

```
A = EMPTY();
```

The initialization can be in an INITIAL clause of the declaration; for example:

```
DECLARE A AREA(100) STATIC INITIAL(EMPTY());
```

In both examples, the EMPTY built-in function returns an empty area value.

Note that the last three items in this list are features that differ in some other implementations of PL/I.

For examples showing allocation in areas, see Section 5.5.7.

5.5.3 Referring 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. This qualifying pointer or offset reference can be implicit, if it is specified with the BASED attribute, or explicit, if the based variable reference is prefixed with a locator qualifier. A complete based variable reference (with the locator qualifier) has the following form:

qualifying-reference -> based-reference

Whether explicit or implicit, the qualifying reference must be 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. 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.

You can use both implicit and explicit qualifications with the same based variable; the explicit qualifier overrides the implicit one. For example:

```
DECLARE X FIXED BIN BASED(P),
    P POINTER,
    (A,B) FIXED BIN;
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.

In PL/I, you can obtain a valid pointer value 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
- 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 the application of ADDR to a parameter or an automatic variable is valid only as long as the parameter's procedure invocation exists, even though the storage pointed to may exist longer.

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

5.5.4 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 or offset variable to the location of the storage in memory. The storage allocated can also be assigned values if the variable is declared with the INITIAL attribute. For example:

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

ALLOCATE LIST SET (LIST_PTR_A); ❷
ALLOCATE LIST SET (LIST_PTR_B); ❸

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

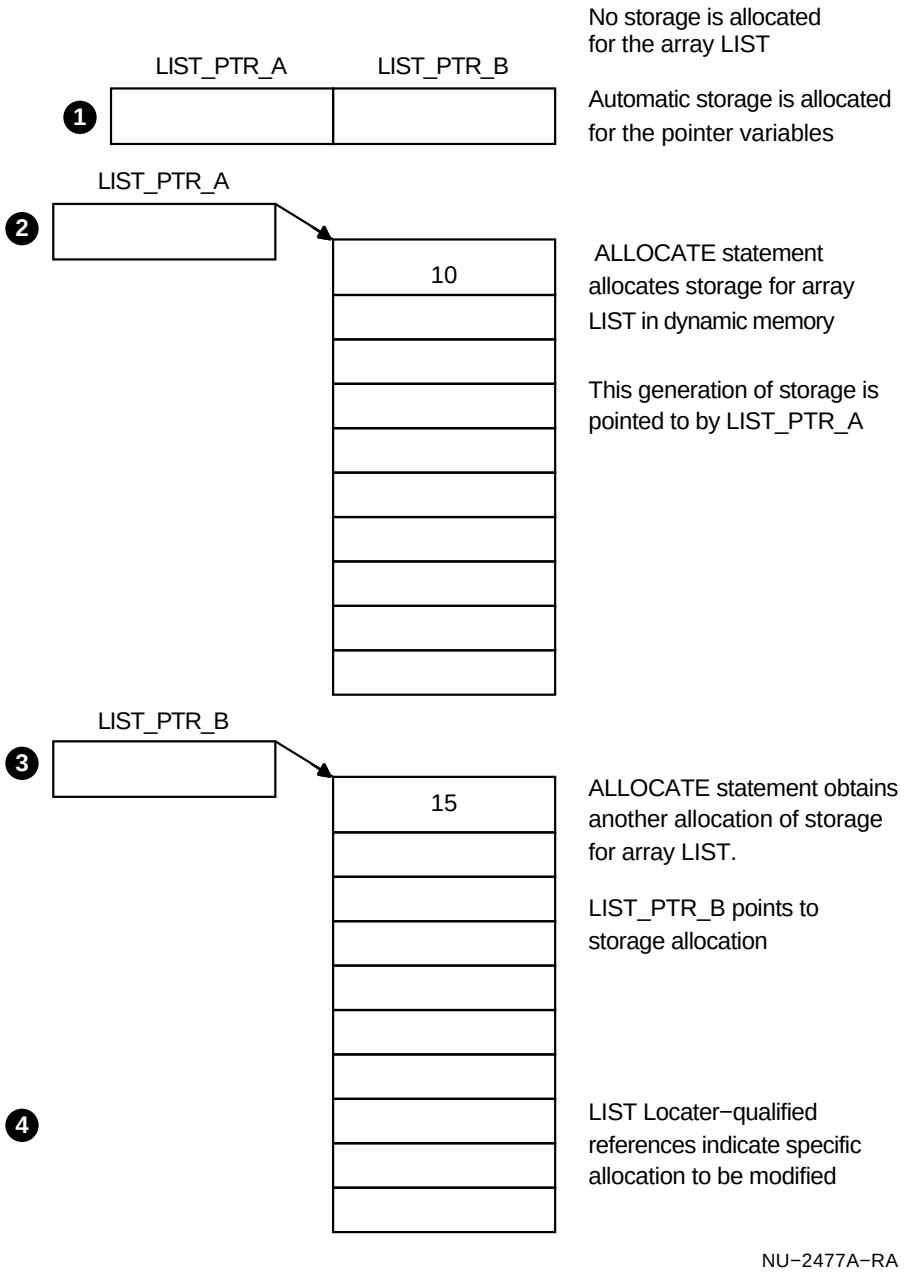
The numbered items in this example are shown in Figure 5–2.

As you can see 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.

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 5–2 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 I/O 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:

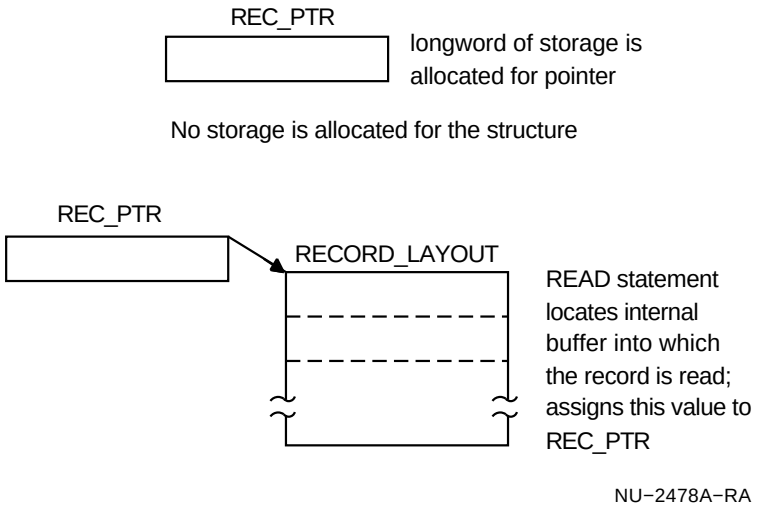
```
DECLARE REC_PTR POINTER,
        NEW_BALANCE FIXED DECIMAL (6,2),
        INFILE FILE RECORD INPUT SEQUENTIAL;
DECLARE 1 RECORD_LAYOUT BASED (REC_PTR),
          2 NAME CHARACTER (15),
          2 AMOUNT PICTURE '999V99',
          2 BALANCE FIXED DECIMAL (6,2);
   .
   .
   .
READ FILE (INFILE) SET (REC_PTR) ;
   .
   .
   .
REC_PTR->BALANCE = NEW_BALANCE;
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 read into a system buffer and the pointer REC_PTR is set to its location.

When you use the SET option with the READ statement, a subsequent 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 5–3 shows this example.

Figure 5–3 Using the READ Statement with a Based Variable



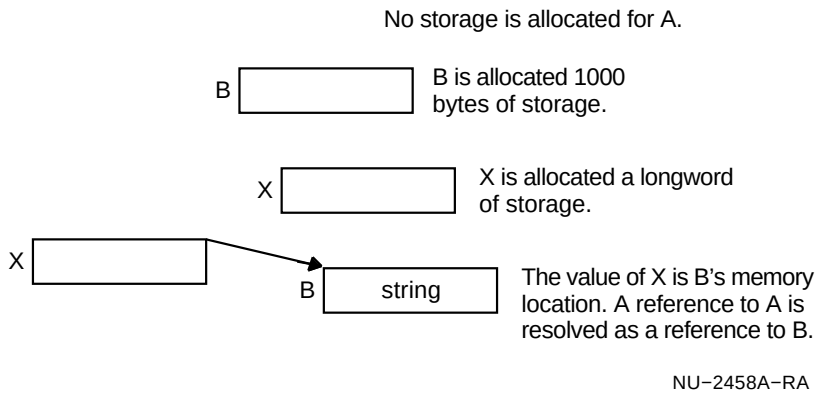
5.5.5 Using the ADDR Built-in Function

The ADDR built-in function returns the storage location of a variable. You can use it 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 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 5–4 shows this example.

Figure 5–4 Using the ADDR Built-In Function



5.5.6 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. However, it is not required that the data types be identical. The following sections discuss type-matching criteria in more detail.

5.5.6.1 Matching by Overlay Defining

Matching by overlay defining is in effect if the based variable reference and the variable for which the storage was originally allocated are both suitable for character-string or bit-string overlay defining. The only further restriction is that the size n (in characters or bits) of the based variable reference 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.

The first program in Section 5.5.7 contains an example of this type of matching. The structure members PAY_RECORD.GROSS_PAY (a character string) and HEALTH_RECORD.EXAM DATE (a picture) are not identical data types. However, both are stored as a character string of length 9; therefore, they meet the criteria for string overlay defining and for data-type matching.

5.5.6.2 Matching by Left-to-Right Equivalence

Matching by left-to-right equivalence 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 l-to-r 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.)

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.

Note that the UNION attribute allows this type of declaration to be written more easily.

5.5.6.3 Nonmatching Based Variable References

In PL/I, 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 be less than or equal to the size of the original variable in bits. However, use of such nonmatching references requires knowledge of the internal representation of data. You should not expect the resulting code to be transportable between OpenVMS systems or to other vendors' hardware. 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 VAX representation of a 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.

5.5.7 Examples of Based Variables

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, which are identified by a 1-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, assume that the file PERSONNEL.DAT contains these records:

```
PMary A. Ford        125000.55
EMary A. Ford        22July 80
t12345678901234567890pppppp.pp
```

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: t12345678901234567890pppppp.pp
All processing complete.
```

Notice these other features of the program:

- The references to based variables have a locator qualifier (P->) for clarity. However, because all are declared with P as their pointer reference, the locator qualifier can be 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.

The UNION attribute can be used to declare a single record with a variant portion in place of PAY_RECORD and HEALTH_RECORD. For example:

```
1 RECORD BASED(P),
  2 RECORD_TYPE CHARACTER(1),
  2 NAME CHARACTER(20),
  2 VARIANS UNION,
    3 GROSS_PAY PICTURE '999999V.99',
    3 EXAM_DATE CHARACTER(9)
```

Note that the UNION attribute is not available in many other PL/I implemenations.

5.6 Controlled Variables

A controlled variable is a variable whose actual storage is allocated and freed dynamically in generations, of which only the most recent is accessible to the program. Controlled variables are declared with the CONTROLLED attribute. A controlled variable can be a scalar, array, area, or major structure variable possessing any of the attributes that do not conflict with the CONTROLLED attribute. See Section 2.2.11 for information about the CONTROLLED attribute.

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

A controlled variable has no storage assigned to it until an ALLOCATE statement allocates storage for it. Each storage assignment is a generation of the variable. Subsequent ALLOCATE statements allocate subsequent generations. At any time in the program's execution, a reference to a controlled variable is a reference to the most recent generation of that

variable, that is, the generation created by the most recent ALLOCATE statement.

The FREE statement frees the most recent generation of a controlled variable. If an attempt is made to free a controlled variable for which no generation exists (or to refer to such a variable), PL/I signals the ERROR condition. The following example shows the use of controlled variables:

```
CONT: PROCEDURE OPTIONS (MAIN);

DECLARE STR CHARACTER (10) CONTROLLED;

    ALLOCATE STR;
    STR = 'First';
    ALLOCATE STR;
    STR = 'Second';
    ALLOCATE STR;
    STR = 'Third';
    PUT SKIP LIST (STR);
    FREE STR;
    PUT SKIP LIST (STR);
    FREE STR;
    PUT SKIP LIST (STR);
    FREE STR;

END;
```

The output of this program is as follows:

```
Third
Second
First
```

5.6.1 Using the ALLOCATION Built-In Function

Because only the most recent generation of a controlled variable is available to a program, controlled variables provide an easy way to implement a stack. The ALLOCATE statement is equivalent to a push operation, and the FREE statement is equivalent to a pop operation. The ALLOCATION built-in function returns the number of generations of a variable, so you can use it to find out if the stack is empty. For example:

```
DECLARE NEXT_MOVE CHARACTER(5) CONTROLLED,
    DIRECTIONS(4) CHARACTER(5) INITIAL(
    'North','East','South','West'),
    D FIXED BINARY (7);
       .
       .
       .
    ALLOCATE NEXT_MOVE;         /* Part of a loop that reports */
    NEXT_MOVE = DIRECTIONS(D);  /* moves in reverse order      */
       .
       .
       .
    DO WHILE                    /* Print moves in correct order */
        (ALLOCATION(NEXT_MOVE) ^= 0);
        PUT SKIP LIST ('Go ', NEXT_MOVE);
        FREE NEXT_MOVE;
    END;
```

See Section 11.4.7 for more information about the ALLOCATION built-in function.

5.6.2 Using the ADDR Built-In Function

You can use a controlled variable as the argument of the ADDR built-in function. If a generation exists, ADDR returns a pointer to it. If no generation of the variable exists, ADDR returns the null pointer. Thus, you can use ADDR to preserve a pointer to a generation of a controlled variable that later becomes hidden under further generations, as in the following example:

```
DECLARE STOPS CHARACTER (20) VARYING CONTROLLED,
    MIDPOINT CHARACTER (20) VARYING BASED (P),
    P POINTER;
       .
       .
       .
    ALLOCATE STOPS;
    STOPS = CURRENT_LOC;
    IF I = 5 THEN P = ADDR(STOPS);
       .
       .
       .
    PUT SKIP LIST (
        'End reached!  Halfway point was', MIDPOINT);
```

At a certain point during the execution of this program, the ADDR built-in function captures the address of the current generation of STOPS and assigns it to P. After more generations of STOPS have been allocated, MIDPOINT (which is based on P) has the value of that same intermediate generation of STOPS.

Note that the value of P (and therefore of MIDPOINT) is valid only so long as the intermediate generation of STOPS to which P points is allocated. As soon as that generation is freed, the value of P becomes invalid, and it must not be used in a pointer-qualified reference until it is reassigned.

A controlled variable cannot be used in a pointer-qualified reference. In the previous example, a reference like the following would be illegal:

```
P->STOPS
```

5.7 Dynamically Allocated Variables

This section describes the mechanisms for dynamically allocating storage.

5.7.1 ALLOCATE Statement

The ALLOCATE statement obtains storage for a based or controlled variable and sets (with based variables) a locator variable equal to the address of the allocated storage. The format of the ALLOCATE statement is as follows:

{ ALLOCATE / ALLOC } allocate-item, . . . ;

allocate-item

The syntax of the allocate item is:

variable-reference [SET(locator-reference)] [IN(area-reference)]

variable-reference

A based or controlled 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 or CONTROLLED attribute.

SET(locator-reference)

The specification of a pointer or offset variable (for based variables) that is assigned the value of the location of the allocated storage. If the SET option is omitted, the based variable must be declared with BASED(locator-reference); the variable designated by that locator reference is assigned the location of the allocated storage.

You cannot use the SET option to allocate controlled variables.

IN(area-reference)

The specification of an area reference (for based variables) in which the storage is to be allocated. If the IN option is omitted, the SET option (or implied SET option if the locator variable is an offset) must be an offset declared with OFFSET(area-reference).

You cannot use the IN option to allocate controlled variables.

Examples

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

ALLOCATE STATE;
```

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 the amount of storage needed 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 evaluates the current value of the expression to determine the amount of storage to be allocated. For example:

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

Here, the value of BUFLEN is evaluated 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.

The ALLOCATE statement is also used to allocate storage for controlled variables. A controlled variable is one whose actual storage is allocated and freed dynamically in generations, only the most recent of which is

accessible to the program. Unlike based variables, a controlled variable cannot be used in a pointer-qualified reference.

If the variable being allocated has been declared with initial values, these values are assigned to the variable after allocation.

5.7.2 FREE Statement

The FREE statement releases the storage that was allocated for a based or controlled variable. The format of the FREE statement is as follows:

FREE free-item[,free-item . . .];

free-item

The syntax of the free-item is:

variable-reference [IN(area-reference)]

variable-reference

A reference to the based or controlled variable whose storage is to be released.

If you do not explicitly free the storage acquired by the 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. You can only free a variable once for each allocation.

IN(area-reference)

The specification of an area reference (for based variables) in which the storage is to be freed. If the IN option is omitted, the variable reference must be either implicitly or explicitly based on an offset variable with a base area.

You cannot use the IN option in conjunction with controlled variables.

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.

5.7.3 Other Mechanisms for Dynamic Storage Allocation

PL/I has a variety of dynamic storage management mechanisms available besides those for based and controlled variables. You can also use the following mechanisms:

- Explicitly specified calls to LIB$GET_VM and LIB$FREE_VM
- Explicitly specified calls to LIB$GET_VM and LIB$FREE_VM using *zones*
- Explicitly specified calls to OpenVMS system memory management services

These storage control mechanisms are generally similar in the amount of overhead that they require both in execution time and in storage space, although certain mechanisms have characteristics that make them useful in specific circumstances.

In general, the standard PL/I language manipulation of dynamic memory provides reasonable performance with some built-in checking.

5.8 Defined Variables

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 another variable called the base variable. The DEFINED attribute provides a way to access the same data using different names (see Section 2.2.13 for a description of the DEFINED attribute).

In a declaration of a defined variable, the DEFINED keyword, which you can abbreviate to DEF, is followed by a variable reference (which must not have the BASED or DEFINED attribute), and optionally by the position in the variable at which the defined variable begins. If you specify the position, you use the POSITION attribute followed by an expression in parentheses. The expression is an integer expression that specifies a position in the base; a value of 1 indicates the first character or bit. You can use the POSITION attribute only when the defined variable satisfies the rules for string overlay defining, which is described later in this section.

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 defined variables that meet the criteria for string overlay defining, you can use the POSITION attribute to specify the position in the base variable at which the definition begins. For example:

```
DECLARE ZIP CHARACTER(20),
        ZONE CHARACTER(10) DEFINED(ZIP) POSITION(4);
```

This statement declares the variable ZONE and maps it to characters 4 through 13 of the variable ZIP.

The extent of a defined variable is determined at the time of block activation, but the base reference (and the position, if the POSITION attribute is also specified) is interpreted each time the defined variable is referenced. For example:

```
DECLARE I FIXED,
    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 cannot 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 both be suitable for character-string overlay defining.
- They must both be suitable for bit-string overlay defining.

5.8.1 String Overlay Defining

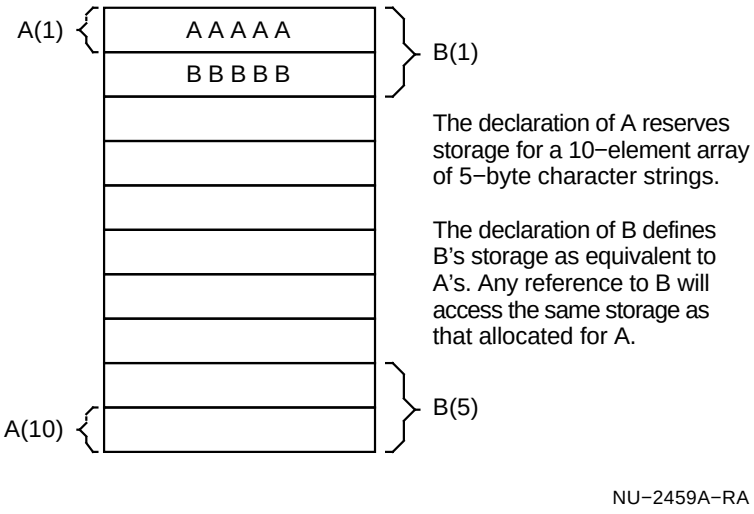
If the defined variable is specified with the POSITION attribute, then both the defined variable and the base reference must be suitable for bit- or character-string overlay defining.

In brief, a variable is suitable for overlay defining if it consists entirely of characters or bits, and those characters or bits are packed into adjacent storage without gaps. Such a variable can be treated as a string or interpreted as different types of aggregates. For example:

```
DECLARE A (10) CHARACTER (5);
DECLARE B (5) CHARACTER (10) DEFINED (A);

A (1) = 'AAAAA';
A (2) = 'BBBBB';
PUT LIST (B(1));
```

Figure 5–5 shows a 50-byte region of storage treated either as a 10-element array (A) of 5-character strings or as a 5-element array (B) of 10-character strings.

Figure 5–5 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 position be its value at the moment of reference; otherwise, let position equal 1.

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 position. The reference must lie entirely within the base reference; that is, position and m must satisify the following formula:
$1 <= position <= position + m <= n + 1$

n is the total number of characters or bits in the base reference.

5.8.2 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:

- V has the CHARACTER attribute, but not ALIGNED or VARYING.
- V has the PICTURE attribute.
- V is a structure, and each of V's members and submembers that is not a structure satisfies one of the first two criteria.

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:

- V has the BIT attribute but not ALIGNED.
- V is a structure, and each of V's members or submembers that is not a structure satisfies the first criterion.

5.9 Storage Sharing

Variables that have any of the attributes BASED, DEFINED, UNION, or PARAMETER can 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 then 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 Section 5.5.

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, suitable for a technique 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 Section 5.8

Unions provide capabilities similar to those of defined variables, but the rules governing unions are less restrictive. A union is a variation of a structure in which all immediate members occupy the same storage.

The UNION attribute, which is used only in conjunction with a level number in a structure declaration, signifies that all immediate members of the major or minor designated structure occupy the same storage. Immediate members are those members having a level number one higher than the major or minor structure with the union attribute. For more details, see Section 4.2.2

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 returning the value specified in its RETURN statement. For more information, see Section 7.5.

6 Expressions and Data Type Conversions

An expression is a representation of a value or of the computation of a value, and an assignment gives the value contained in an expression to a variable. Together, expressions and assignments form the mechanism for performing computation.

This chapter describes the following topics:

- The assignment statement
- Operators and operands, the elements of an expression
- The manner in which expression evaluation takes place
- Conversion of operands and expressions
- Conversion of the data types of operands during expression evaluation and assignment

6.1 Assignment Statement

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

target, . . . = expression;

target
A reference to a variable to be assigned the expression's value. If there are two or more targets, they are separated by commas. A target can be:

- A reference to a scalar variable or scalar array element
- A reference to 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 the targets and the expression in any order. Thus, a program should not depend on the evaluation of the targets before the expression.

PL/I performs the following steps for assignment. Note that the only certain things about the order of steps performed are that step 1 precedes step 3 and that step 4 is performed last.

1 The expression is evaluated, producing a value to be assigned to the target. An expression can consist of many subexpressions and operations, each of which must be evaluated.

2 Each target is evaluated. If a target contains a pseudovariable, any expressions in the argument list are evaluated.

3 If the data type of the result does not match the data type of a target variable, the resulting value is converted to the data type of the target, if possible. The compiler issues a WARNING message to alert you to the implicit conversion.

4 The value of the expression is assigned to the targets.

Some general rules regarding the types of data you can specify in assignment statements are listed in Table 6–1. For the complete rules for data conversion in assignments, see Section 6.4.

Table 6–1 Data Types for Assignment Statement

Data Type	Rules
Area	Only the current extent of an area is moved from the source area to a target. If the target area is not large enough to hold the extent, the AREA condition is raised. Note that the assignment is performed in such a way that all offsets in the source area are valid in the target area after the assignment. Areas cannot be assigned as members of structures.
Arithmetic	PL/I converts an arithmetic expression to the type of its 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 a target of an assignment statement are invalid.
Bit	When a 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.

Table 6–1 (Cont.) Data Types for Assignment Statement

Data Type	Rules
Character	When a 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 a 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	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	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	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 a 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.

The following are examples of assignment statements:

```
A = 1;
A = B + A;
SUM = A + 3;
STRING = 'word';
```

6.2 Operators and Operands

An operator is a symbol that requests a unique operation. Operands are the expressions on which operations are performed. Built-in functions can also be considered operators, as well as their arguments considered operands.

Operators

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 logical NOT (^) prefix operator performs a logical NOT operation on a bit-string operand; the bit value is complemented.

The following are examples of expressions containing prefix operators:

```
A = +55;
B = -88;
BITC = ^BITB;
```

An infix operator appears between two operands, and indicates the operation to be performed on them. PL/I has infix operators for arithmetic, logical, and relational (comparison) operations, and for string concatenations. Following are some examples of expressions containing infix operators:

```
RESULT = A / B;
IF NAME = FIRST_NAME || LAST_NAME THEN GOTO NAMEOK;
```

An expression can contain both prefix and infix operators. For example:

```
A = -55 * +88;
```

You can apply prefix and infix operators to expressions by using parentheses for grouping.

For a table giving the categories of operators and the operator symbols, see Chapter 1.

Operands

Because all operators must yield scalar values, operands cannot 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 before the operation to a single type, called the derived data type. Section 6.4.2 describes this process.
- Logical operators must have bit-string operands.
- Relational operators must have two operands of the same type. (Note, however, that comparisons are allowed between offsets and pointers.)
- The operators greater than (>), less than (<), not greater than (^>), not less than (^<), greater than or equal to (>=), and less than or equal to (<=) are valid only with computational operands.
- The concatenation operator must have two bit-string operands or two character-string operands.

6.2.1 Arithmetic Operators

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

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

In addition, there are two prefix operators: unary plus (+) and 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 any arithmetic operator, operands must be arithmetic; that is, they must be constants, variables, or other expressions with the data type attribute BINARY, DECIMAL, or PICTURE. Operands of different arithmetic types are converted to a common type before the operation is performed.

Arithmetic operators have a predefined precedence that governs the order in which operations are performed. All expressions can be enclosed in parentheses to override the rules of precedence. Table 6–2 lists the precedence of operators.

6.2.2 Logical Operators

The logical operators perform logical operations on one or two operands. The operands of the logical operators must be bit-string expressions, except that 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 1, and they can 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.

Except for AND THEN and OR ELSE, 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 five infix operators and one prefix operator:

Prefix Operator	Operation
^ (circumflex)	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).

Infix Operator	Operation
& (ampersand)	Logical AND. In a logical AND operation, two operands are compared. If corresponding bits are 1, the result is 1; otherwise, the result is 0.
| (vertical bar) or ! (exclamation point)	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.)
&: (ampersand and colon)	Logical AND THEN. The operation is like AND except that the second operand is evaluated only if the first operand is true, and except that AND THEN does not do bit-by-bit operations on bit-string operands.
^ (circumflex)	Logical EXCLUSIVE OR. Two operands are compared, and the result is 1 if one of the corresponding bits is 1 and the other is 0.
|: (vertical bar and colon)	Logical OR ELSE. The operation is like OR except that the second operand is evaluated only if the first operand is false, and except that OR ELSE does not do a bit-by-bit operation on bit-string operands.

You can define additional operations on bit strings with the BOOL built-in function.

Logical expressions will 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 can be terminated. For example:

```
A & B & C & D & E
```

In this expression, evaluation will 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 logical NOT expression yields <BIT_STRING>(1)B; when this value is assigned to BITC, a BIT(4) variable, the value is padded with zeros and becomes <BIT_STRING>(1000)B.

6.2.2.1 NOT

The logical NOT operator in PL/I is the circumflex character (^), used as a prefix operator. 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.

The NOT operator can test the falsity of an expression in an IF statement. For example:

```
IF ^(MORE_DATA) THEN  . . .
```

6.2.2.2 AND

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 resulting bit is 0.

The result of a logical AND operation is a bit-string value. All relational expressions result in bit strings of length 1; they can therefore be 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 the greater length 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 <BIT_STRING>(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  . . .
```

6.2.2.3 OR

The vertical bar 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 this 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 can 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 <BIT_STRING>(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  . . .
```

You can use the exclamation point (!) in place of the vertical bar, for compatibility with other PL/I implementations.

6.2.2.4 EXCLUSIVE OR

The EXCLUSIVE OR operator (infix or dyadic ^) causes a bit-by-bit comparison of two bit-string operands. If the two operands are not of equal length, the shorter is padded with 0s until it is the same length as the other, and this length is also the length of the result. If either of two corresponding bits is 1 and the other is 0, the result is 1. If both are 1, or if both are 0, the result is 0.

All relational expressions result in bit strings of length 1, and they can therefore be used as operands in an EXCLUSIVE OR operation.

The result of the EXCLUSIVE OR operation is a bit-string value. For example:

```
DECLARE (BITA, BITB, BITC) BIT (4);
BITA = '0011'B;
BITB = '1011'B;
BITC = BITA ^ BITB;
```

The resulting value of BITC is '1000'B.

The EXCLUSIVE OR operator can be used to test whether one and only one of the expressions in an IF statement is true. For example:

```
IF (A > 0) ^ (B > 0) THEN  . . .
```

6.2.2.5 AND THEN

The ampersand-colon token (&:) is the AND THEN operator in PL/I. The AND THEN operator causes the first operand to be evaluated; if it is false, the result returned is '0'B. The second operand will never be evaluated if the first operand is false. If and only if the first operand is true, the second operand is evaluated. If both are true, the result returned is true ('1'B); otherwise, the result is false ('0'B).

The AND THEN operator performs a Boolean truth evaluation, not a bit-by-bit operation, even when the two operands are bit strings. For example, '00001'B &: '10000'B yields '1'B (not '00000'B, which would be the result of an AND operation on these two bit strings). The reason is that each operand is a non-zero bit value, and therefore each evalutes to '1'B.

The AND THEN operator yields the same result as the AND operator (&) when expressions are tested in an IF statement (as in the last example in the "AND Operator" entry). The difference is that the AND operator can have its operands evaluated in either order.

The AND THEN operator is useful in compound test expressions in which the second test should occur only if the first test was successful. For example:

```
IF (P ^= NULL()) &: (P->X ^= 4) THEN  . . .
```

This statement causes P->X to be evaluated only if P is not a null pointer. If the AND operator were used instead of AND THEN, this expression could cause an access violation (invalid pointer reference).

6.2.2.6 OR ELSE

The vertical bar and colon characters (|:) together are the OR ELSE operator in PL/I. The OR ELSE operator causes the first operand to be evaluated. If it is true, the result returned is '1'B. If and only if the first operand is false, the second operand is evaluated. If either or both operands are true, the result returned is '1'B; otherwise, the result is '0'B.

The OR ELSE operator performs a Boolean truth evaluation, not a bit-by-bit operation, even when the two operands are bit strings. For example:

```
'00001'B |: '10000'B
```

This yields:

```
'1'B
```

It does not yield '10001'B, which would be the result of an OR operation on these two bit strings. The reason is that each operand is a nonzero bit value, and therefore each evalutes to '1'B.

The OR ELSE operator yields the same result as the OR operator (|) when expressions are tested in an IF statement (as in the last example in the "OR Operator" entry). The difference is that the OR operator can have its operands evaluated in any order.

The OR ELSE operator is useful in compound test expressions in which the second test should occur only if the first test failed. For example:

```
IF (A=0) |: (B/A > 1) THEN  . . .
```

This results in the second expression (B/A > 1) being evaluated only if the first expression is false. Thus, the OR ELSE operator prevents an attempt to divide by zero.

6.2.3 Relational Operators

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 1). If the comparison is true, the resulting value is <BIT_STRING>(1)B; if the comparison is false, the resulting value is <BIT_STRING>(0)B. The relational operators are all infix operators. The following table describes all the relational operators:

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

Note that PL/I recognizes the tilde symbol (~) as synonymous with the circumflex (^).

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 scalar.
- Both operands must be arithmetic, or they must have the same data type.

6.2.3.1 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.

6.2.3.2 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 equal.

6.2.3.3 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 characters are the first 128 characters of the DEC Multinational Character Set, which is in Appendix B.

Note the following characteristics of the collating sequence:

- Uppercase letters are less than any lowercase letters.
- Numeric characters are less than any letters.

If the operands do not have 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. Null character strings are equal.

6.2.3.4 Comparing Noncomputational Data

Only the following operators are valid, or meaningful, for comparisons of any of the noncomputational data types except areas (condition, 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.

Condition Data

Two condition values are equal if they identify the same condition values.

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.

Offset Data

Two offset values are equal if they identify the same storage location or if they are both null.

Pointer Data

Two pointer values are equal if they identify the same storage location or if they are both null.

6.2.4 Concatenation Operator

The concatenation operator produces a single string from two strings specified as operands. The concatenation operator is two vertical bars (| |).

The operands must both be character strings or both be bit strings. (If not, the appropriate conversion is performed, and you get a warning message about the conversion. 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((3)'001'B||'07'B3);

END CONCAT;
```

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

```
ABCDEF
'001110'B
'001001001000111'B
```

Note that the exclamation point can be used in place of the vertical bar, for compatibility with other PL/I implementations.

6.3 Precedence of Operators and Expression Evaluation

The precedence, or priority, of operators defines the order in which expressions are evaluated when they contain more than one operator. Table 6–2 gives the priority of PL/I operators. Low numbers indicate high priority. For example, the exponentiation operator (**) has the highest priority (1), so it is performed first, and the OR ELSE operator (| :) has the lowest priority (9), so it is performed last.

Table 6–2 Precedence of Operators

Operator	Priority	Left/Right Associative	Order of Evaluation
()	0	N/A	deepest first
**	1	right	left to right
+ (prefix)	1	N/A	N/A
- (prefix)	1	N/A	N/A
^ (prefix)	1	N/A	N/A
*	2	left	left to right
/	2	left	left to right
+ (infix)	3	left	left to right
- (infix)	3	left	left to right
||	4	left	left to right
>	5	left	left to right
<	5	left	left to right
^>	5	left	left to right
^<	5	left	left to right
=	5	left	left to right
^=	5	left	left to right
<=	5	left	left to right

Table 6–2 (Cont.) Precedence of Operators

Operator	Priority	Left/Right Associative	Order of Evaluation
>=	5	left	left to right
&	6	left	left to right
|	7	left	left to right
^ (infix)	7	left	left to right
&:	8	left	left to right across entire expression
|:	9	left	left to right across entire expression

Expressions are evaluated from left to right, with the following qualifications:

- Some PL/I operators take precedence over others used in the same expression. Operations with higher precedence 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 precedence.
- 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.

  Consider the expression:

```
A |: B &: C
```

  It should be parenthesized according to the rules of associativity as:

```
(A |: (B &: C))
```

  However, the semantics of the OR ELSE (\|:) and AND THEN (&:) operators dictate that the operands be evaluated in the order A, B, C as necessary. This means that the B &: C will be evaluated before A which might not be the intent of the programmer and might not conform to the semantic rules of the OR ELSE (\|:) operator.

  When PL/I determines which to evaluate first, (1) the deeply nested B &: C (consistent with the order of evaluation for parentheses), or (2) the A \|: (consistent with the order of evaluation of \|:), (1) the deeply nested B &: C is chosen.

  Whenever an expression that contains any combination of two or more &: or \|: operators, the order of operand evaluation will be performed in the order dictated by the associativity of the operators. This may or may not be the desired behavior. To work around this, you will have to construct similar program sequences using IF THEN constructs in conjunction with either the AND (&) or OR (\|) operators to achieve the desired behavior.
- Exponential operations of the form $A**B**C$ are evaluated from right to left.

- The run-time evaluation of a logical expression can be terminated as soon as its result is known. For instance:

```
A & USER_FUNCTION(ALPHA,BETA)
```

  Evaluation of this expression can be terminated without the USER_FUNCTION reference being evaluated if the evaluation of A results in a *false* Boolean value. However, the evaluation of A might not occur first, because the order of evaluations is not guaranteed in AND operations. To ensure that the first operand is evaluated first, use &:, which is the AND THEN operator, instead of &.

- The compiler may evaluate the subexpressions of an expression in any order that produces an algebraically correct result. For example:

```
A + B + FUNC(I) + C
```

  The subexpression A+B might not be evaluated and the result stored before FUNC(I) is evaluated. Therefore, if FUNC(I) alters A, B, or C, results may not be as expected.

- If a function referenced in an expression executes a nonlocal GOTO statement, the expression is not evaluated further.

6.4 Data Type Conversion of Operands and Expressions

Conversion is the changing of a data item from one data type to another. Data conversion in PL/I takes place in many contexts, not all of them obvious ones. Program results that seem improper may in fact be caused by data conversion at some point in the program's execution. This section discusses the following topics:

- When PL/I converts data.
- How arithmetic operands of different types are converted to a single derived type during expression evaluation.
- How nonarithmetic operands of different types are converted to the same type.
- How you can control conversions precisely by using conversion built-in functions designed for that purpose.
- Contexts in which PL/I automatically converts data from one type to another-for example, in input and output by the GET and PUT statements.
- Assignments to arithmetic variables
  - — From any arithmetic data type to any other arithmetic data type
  - — From pictured to any arithmetic type
  - — From bit-string to any arithmetic data type
  - — From character-string to any arithmetic data type
- Assignments to bit-string variables
  - — From any arithmetic data type to bit-string

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

6.4.1 Contexts in which PL/I Converts Data

PL/I can perform data conversions in the following contexts:

- 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:

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

If F is a variable with the attributes FIXED DECIMAL (5,2), then the statement assigns the numeric value 133.45 to F, as expected, although the compiler issues a WARNING message about the implicit conversion, stating that the constant '133.45' has been converted to a FIXED DECIMAL target. The warning does not prevent you from linking and running the program. However, note the following example:

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

This statement results not only in a compiler WARNING message, but if you go on to link and run the program, you receive a CONVERSION condition, which will normally be fatal unless it is handled with an ON CONVERSION ON-unit.

Table 6–3 illustrates the contexts in which PL/I performs conversions. The table also lists the built-in conversion functions, such as BINARY and CHARACTER, which you can use when you want to explicitly indicate a conversion and to specify such characteristics as the precision or string length of the converted result.

Table 6–3 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 x||y x & y x | y x&:y x|:y x ^ y x > y x < y x = y x^=y	In any expression, if operands do not have the required data type, they are converted to a common data type before the operation. For most operators, the data types of all operands must be identical. A warning message is issued in the case of a concatenation conversion.
BINARY (expression) BIT (expression) CHARACTER (expression) DECIMAL (expression) DECODE (expression) ENCODE (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.

Table 6–3 (Cont.) Contexts in Which PL/I Converts Data

Context	Conversion Performed
PAGESIZE (expression) LINESIZE (expression) SKIP (expression) LINE (expression) COLUMN (expression) format items A, B, E, F, and X TAB (expression)	Values specified for various options to PL/I statements must be converted to integer values.
DO control-variable . . .	Values are converted to the attributes of the control variable.
parameter	Actual parameters are converted to the type of the formal parameter, if necessary.
INITIAL attribute	Initial values are converted to the type of the variable being initialized.

6.4.2 Derived Data Types for Arithmetic Operations

Even though arithmetic operands can be of different arithmetic types, all operations will be 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 type is BINARY. Otherwise, the derived type is DECIMAL.
- If any operand has the attribute FLOAT, the derived type is FLOAT. Otherwise, the derived type is FIXED.

Table 6–4 gives the derived data type for two arithmetic operands of different types. (Note that the types derived from FIXED DECIMAL in Table 6–4 are also derived when one operand is pictured.)

Table 6–4 Derived Data Types

Type of Operand 1	Type of Operand 2	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

Table 6–5 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 6–5 Converted Precision as a Function of Target and Source Attributes

Target Data Type	Binary Fixed Source[1]	Decimal Fixed Source[1]	Binary Float Source[1]	Decimal Float Source[1]
Binary Fixed	p	min(ceil(p*3.32)+1,31)	N/A	N/A
	q	min(ceil(q*3.32),31)	N/A	N/A
Decimal Fixed	min(ceil(p/3.32)+1,31)	p	N/A	N/A
	max(0,min(ceil(q*3.32),31))	q	N/A	N/A
Binary Float	OpenVMS VAX: min(p,113)	min(ceil(p*3.32),113)	p	min(ceil(p*3.32), 113)
	OpenVMS Alpha: min(p,53)	min(ceil(p*3.32),53)	p	min(ceil(p*3.32), 53)
Decimal Float	OpenVMS VAX: min(ceil(p/3.32),34)	min(p,34)	min(ceil(p/3.32),p34)	
	OpenVMS Alpha: min(ceil(p/3.32),15)	min(p,15)	min(ceil(p/3.32),p15)	

[1]The constant 3.32 is an approximation of log2(10), the number of bits required to represent a decimal digit.

All arithmetic operations except exponentiation are performed in the derived type of the two operands. Exponential operations are performed in a data type that is based on the derived type of the operands. All operations, including exponentiation, have results of the same type as that in which they are performed.

The result of an arithmetic operation can be assigned to a target variable of any computational type. The result is converted to the target type, following the rules in Section 6.4.5.

6.4.3 Conversion of Operands in Nonarithmetic Operations

As operations must be performed on operands of the same type, the following conversions are performed when operands do not match in nonarithmetic operations:

- PICTURE is converted to CHARACTER.
- DECIMAL is converted to CHARACTER.
- FIXED BINARY is converted to BIT.
- If either operand is CHARACTER, after other conversions have been performed, the noncharacter operand is converted to CHARACTER.

A warning message is issued about a conversion in a concatenation expression, except for picture to character.

6.4.4 Built-In Conversion Functions

The built-in conversion functions can take arguments that are either arithmetic or string expressions. They are 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 in 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, with a scale factor of 0.
- The derived type of a character string is fixed-point decimal; its converted precision is 31, with a scale factor of 0.

PL/I uses these derived attributes to determine the precision of values returned by the conversion functions if no precision is specified in the functions' argument lists. Note that the value of a string argument must also be convertible to the result type; for instance, '1.333' is convertible to arithmetic, but 'XYZ' is not.

Table 6–6 indicates which built-in functions you should use for each conversion between an arithmetic and a nonarithmetic type. In addition, you can use the BINARY, DECIMAL, FIXED, and FLOAT built-in conversion functions to control conversions between two arithmetic types.

Table 6–6 Built-In Functions for Conversions Between Arithmetic and Nonarithmetic Types

Conversion	Function
Arithmetic to bit	BIT
Arithmetic to character	CHARACTER
Arithmetic or character to fixed-point arithmetic	FIXED
Bit to arithmetic	BINARY
Bit to character	CHARACTER
Character to bit	BIT
Character to decimal	DECIMAL
Character to float	FLOAT
Arithmetic or character to binary	BINARY
Character to fixed binary	DECODE
Decimal integer to character	ENCODE
Integer (non-negative) to character	BYTE

6.4.5 Implicit Conversion During Assignment

During assignment, PL/I automatically converts the derived data type of an expression to the data type of a target, if necessary. In assignments, conversions are defined between the noncomputational types POINTER and OFFSET, and between any two computational types. However, a conversion during assignment results in an error if PL/I cannot perform it in a meaningful way. For example, you can assign the string '123.4' to a fixed decimal variable; you cannot, however, assign the string 'ABCD' to the same variable. Similarly, an assignment of an arithmetic type to a fixed variable results in the FIXEDOVERFLOW condition if integral digits are lost.

Although PL/I performs conversions in assignment statements, such conversions may represent programming errors and are in violation of the PL/I G subset standard. Therefore, the compiler issues a warning message that an implicit conversion is taking place. These messages do not terminate the compilation and may not indicate errors; they simply alert you to the fact that your program converts one data type to another in a way that may cause a problem when the program is run. You can prevent such warning messages in two ways:

- Use the built-in conversion functions to convert data types explicitly. This method is recommended. Section 6.4.4 summarizes the functions.
- Use the /NOWARNINGS qualifier to the PLI command to suppress diagnostic warning messages. (The compiler will continue to print messages of greater severity.) However, you run the risk of missing important diagnostic information.

For example:

```
DECLARE (A,B) FIXED DECIMAL (5,2);
    A = '123.45';              /* Warning message */
    B = FIXED('123.45',5,2);   /* No warning      */
```

Both assignment statements assign the same value to their targets; however, the first statement causes a warning message from the compiler, while the second statement does not.

6.4.6 Assignment to Arithmetic Variables

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

6.4.6.1 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 can be larger than the target's declared precision.

- If the target is a variable of type FIXED DECIMAL(p,q) or FIXED BINARY(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 floating-point type, then the OVERFLOW condition is signaled, and the value of the target is undefined. On OpenVMS VAX systems only, if the absolute source value is too small to be represented, the value zero is assigned to the target, and if UNDERFLOW is enabled it is signaled. Alpha hardware does not support UNDERFLOW and does not signal the underflow condition.

6.4.6.1.1 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*

6.4.6.1.2 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.

6.4.6.1.3 Conversions from FIXED BINARY to Other Data Types

Conversions from FIXED BINARY to other data types follow the rules outlined below. Notice that these rules assume both precision and scale.

Precisions of the source and target are (p,q) and (p1,q1), respectively. The precision of the result is (p2,q2).

Target	Result
FIXED DECIMAL(p1,q1)	p2=1+CEIL(p1/3.32) and q2=CEIL(q1/3.32).

Target	Result
FIXED BINARY(p1,q1)	Precision and scale of the source are maintained during conversion; therefore, padding or truncation can occur. If nonzero bits are lost on the left, the result is undefined.
FLOAT DECIMAL(p1)	p2=CEIL(p1/3.32). The exponent indicates any fractional value.
FLOAT BINARY(p1)	p2=p1. The mantissa indicates any fractional value.
PICTURE	The target implies FIXED DECIMAL and is converted accordingly.
CHARACTER	The binary precision (p,q) is converted to a FIXED DECIMAL with precision (p1,q1), where p1=1+CEIL(p/3.32) and q1=CEIL(q/3.32). Then the rules for conversion from FIXED DECIMAL to CHARACTER are in effect.
BIT	The binary precision (p,q) is converted to an intermediate bit string where the size or precision is MIN(31,p-q). Then the intermediate bit string is converted to BIT(n). If (p-q) is negative or zero, the result is a null bit string.

If the scale factor is negative, substitute the FLOOR value for CEIL in the above calculations which contain q's.

6.4.6.2 Pictured to Arithmetic Conversions

In 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 value is assigned to the target, following the PL/I rules for arithmetic to arithmetic conversion.

6.4.6.3 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,0), 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; therefore, you should avoid this conversion 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;
END CONVTB;
```

Note that because the program CONVTB performs implicit conversions, the compiler issues WARNING messages. (Linking and running are accomplished successfully because the conversions are valid.)

The program CONVTB produces the following output:

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

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 <BIT_STRING>(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.

6.4.6.4 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; for example, 342.122E-12 and <BIT_STRING>(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, a plus (+) or minus (-) sign, a decimal point (.), and the letter E. If the character string contains any invalid characters, the CONVERSION condition is signaled. See the following examples.

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 PL/I 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';
      /* CONVERSION condition */

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

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

6.4.7 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 does 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.

6.4.7.1 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,s)	min(31,r-s)
FLOAT BINARY(r)	min(31,r)
FIXED DECIMAL(r,s)	min(31,ceil((r-s)*3.32))

Source	Precision p
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,0). If $n >= 2^p$, 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.

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;
```

Note that because the program CONVB performs implicit conversions, the compiler issues WARNING messages. (Linking and running are accomplished successfully because the conversions are valid.)

The program CONVB produces the following 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
```

The values 35 and -35 produce the same bit string because the sign is lost in the conversion. In the first assignment, 35, which is FIXED DECIMAL(2,0), is converted to FIXED BINARY(7,0) and then to a 7-bit string (<BIT_STRING>(0100011)B). Three additional bits are appended to this intermediate bit string when it is assigned to NEW_STRING. Notice that the low-order bit of 8001 is lost when the constant is assigned to a BIT(10) variable.

6.4.7.2 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.

6.4.7.3 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 CONVERSION 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';
      /* CONVERSION condition */
```

6.4.8 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 does 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 rules for how PL/I arrives at the intermediate string for conversion of each data type are described below. Examples illustrate the intermediate value as well as the resulting value.

6.4.8.1 Arithmetic to Character-String Conversions

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

6.4.8.1.1 Conversion from Fixed-Point Binary or Fixed-Point Decimal

If the data item source value is of type FIXED BINARY(p1,q1), PL/I first converts it to type FIXED DECIMAL(p2,q2), where:

$$p2 = min(ceil(p1/3.32) + 1, 31)$$

$$q2 = max(0, min(ceil(q1/3.32), 31))$$

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 0 character.

Alternatively, 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 following 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 following 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 following 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 can be zero; 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' */
```

6.4.8.1.2 Conversion from Floating-Point Binary or Floating-Point Decimal

If the data item source value is of type FLOAT BINARY(p1), it is converted to FLOAT DECIMAL(p2), where:

For OpenVMS VAX systems:

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

For OpenVMS Alpha systems:

$p2 = min(ceil(p1/3.32), 15)$

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 2-digit exponent.

Note: If the value is a G-floating-point number, three characters are allocated to the exponent, and the length of the string is p+7. For OpenVMS VAX systems only, if the value is an H-floating-point number, four characters are allocated to the exponent, and the length of the string is p+8.

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 can 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 following 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 are surrounded with apostrophes to make the spaces distinguishable.) 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.

6.4.8.2 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.

6.4.8.3 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' */
```

6.4.9 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 the picture specification that specify decimal digits. The target 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 PL/I rules for the source data type. This value is then edited to a character string s, as specified by the picture specification, and the value s is assigned to the pictured target.

When the value v is being edited to the string s, the CONVERSION 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 signaled if the value v has more integral digits than are specified by the picture specification of the target.

6.4.10 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:

```
pointer-variable = pointer-value;
offset-variable = offset-value;
pointer-variable = offset-variable;
offset-variable = pointer-value;
```

In the third and fourth assignments above, the offset variable must have been declared with an area reference.

7 Procedures

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. There are three kinds of procedures:

- A MAIN procedure is the procedure where program execution begins.
- A subroutine procedure is invoked with a CALL statement and returns values to the invoking procedure by means of a parameter list.
- A function procedure is invoked by a function reference and returns a scalar value to the invoking procedure. It can also return values through a parameter list.

Subroutines and function procedures can be passed data from the invoking procedure by means of an argument list.

This chapter discusses the following topics:

- The PROCEDURE statement which defines the beginning of a procedure block
- Built-in and user-written functions
- The ENTRY statement and how to specify entry points
- The CALL statement which transfers control to an entry point
- Parameters and argument passing
- Calling external and internal procedures
- Terminating procedures
- Passing Arguments to non-PL/I procedures

7.1 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 can denote the beginning of either an internal or an external subroutine or function.

The format of the PROCEDURE statement is as follows:

```
entry-name: { PROCEDURE } [ (parameter, . . . ) ]
            { PROC      }
      [ OPTIONS (option, . . . ) ]
      [ RECURSIVE    ]
      [ NONRECURSIVE ]
```

[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 contained in any other block, and EXTERNAL if the procedure is not contained in any other block.

parameter, . . .
One or more parameters (separated by commas) that the procedure expects when it is activated. 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.

OPTIONS (option, . . .)
An option that specifies one or more options, separated by commas:

IDENT(string)

An option specifying a character-string constant giving the module ident for the listing and the object files. Only the first 31 characters of the string are recognized.

Each module should contain only one procedure with the IDENT option. Should there be more than one, the last specified indent will be the one used.

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.

A program must have one procedure with OPTIONS(MAIN) in order for condition handling to work properly.

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.

RECURSIVE or NONRECURSIVE
An option that indicates (for program documentation) that the procedure will or will not be invoked recursively. In standard PL/I, the RECURSIVE option must be specified for a procedure to be invoked recursively. However, any procedure can be invoked recursively, and the RECURSIVE and NONRECURSIVE options are ignored by the compiler.

RETURNS (returns-descriptor)
An option specifying that the procedure is invoked by a function reference, as well as specifying the attributes of the function value returned. One of the possible attributes is TYPE. The syntax of the TYPE attribute is:

[(TYPE (reference)];

RETURNS must be specified for functions. It is invalid for procedures that are invoked by CALL statements.

For valid return descriptors, see the RETURN statement section of Section 7.7.

7.2 Functions and Function References

A function is a procedure that returns a value and that receives control when its name is referenced in an expression. There are two types of functions:

- PL/I built-in functions
- User-written functions

A user-written function must have the following elements:

- The RETURNS option on the PROCEDURE statement
- A value on the RETURN statement; the value must be of a data type that is valid for conversion to the one specified in the RETURNS option

For example:

```
ADDER: PROCEDURE (X,Y) RETURNS (FLOAT);
DECLARE (X,Y) FLOAT;
        RETURN (X+Y);
        END;
```

The function ADDER has two parameters, X and Y. They are floating-point binary variables declared within the function. When the 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. The function ADDER can be referenced as follows:

```
TOTAL = ADDER(5,6);
```

The arguments in the reference to ADDER are converted to FLOAT.

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 follows:

```
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 example:

```
DECLARE P POINTER,
        NULL BUILTIN;
   .
   .
   .
P = NULL;
```

This example assigns a null pointer value to P. Without the declaration of NULL as a built-in function, the assignment statement would have been as follows:

```
P = NULL();
```

7.3 ENTRY Statement

The ENTRY statement defines an alternate entry point to a procedure. Its format is as follows:

```
entry-name: ENTRY [ (parameter, . . . ) ]
                  [ RECURSIVE    ]
                  [ NONRECURSIVE ]
                [ RETURNS (returns-descriptor) ];
```

entry-name

A 1- to 31-character label 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 is 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 to one, with arguments specified for the procedure when it is invoked via the ENTRY statement.

RECURSIVE or NONRECURSIVE

An option that indicates (for program documentation) that the procedure will or will not be invoked recursively. In standard PL/I, the RECURSIVE option must be specified for a procedure to be invoked recursively. However, any procedure can be invoked recursively, and the RECURSIVE and NONRECURSIVE options are ignored by the compiler.

RETURNS (returns-descriptor)

For an entry that is invoked as a function reference, an option giving the data type attributes of the function value returned.

One of the possible attributes is TYPE. The syntax of the TYPE attribute is:

[(TYPE (reference)];

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.

You should avoid unnecessary use of ENTRY statements, because their effect is detrimental to the overall optimization of the program and they make debugging more complicated.

7.3.1 Specifying Entry Points

The entry points of a procedure are the points at which it can be invoked. The PROCEDURE statement specifies one entry point. You can specify additional entry points with ENTRY statements within the procedure block. ENTRY statements are allowed anywhere except as specified in the restrictions described in Section 7.3.

The labels used on PROCEDURE and ENTRY statements declare those names as entry constants. The scope of the declarations is internal if the PROCEDURE and ENTRY statements appear in internal procedures, and external if they appear in external procedures.

You declare an entry name in the block containing the procedure to which the entry point belongs. For example:

```
P: PROCEDURE;
   DECLARE E: ENTRY;

Q: PROCEDURE;
   DECLARE E FIXED BINARY;
END Q;
END P;
```

The entry names E and Q are declared in procedure P. Within procedure Q, E is declared as a fixed-point binary variable. This does not conflict with the declaration of E as an entry in procedure P.

You can invoke an entry point 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 Example 7–3. 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 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.

If PITCH is to be used as a function, the DECLARE statement must also include a RETURNS attribute to describe the attributes of the returned value, as in the following example:

```
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.

7.3.2 Multiple Entry Points

A procedure can be entered at more than one point. However, 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 follow:

- 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 through 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. (In addition, it must be invoked as a "subroutine"-that is, with the CALL statement.) 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.
- An ENTRY statement is not executable. If control reaches it sequentially, control immediately continues to the next statement.

The following example shows a procedure with two alternative 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 invoked at QUEUES, the procedure must be passed two parameters. If invoked at either of the alternative entries ADD_ELEMENT or REMOVE_ELEMENT, the procedure must be passed only one parameter.

When this procedure is entered at either alternative entry point, the entire block beginning at QUEUES is activated, but execution begins with the first executable statement following the entry point.

7.4 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 as follows:

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

entry-name
The name of an external or internal procedure that does not have the RETURNS attribute. 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.

You must enclose arguments in parentheses. Multiple arguments must be separated by commas.

You can use the CALL statement to call an internal or external procedure. The following example illustrates a main procedure, CALLER, and a call to an internal procedure, 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;
```

7.5 Parameters and Arguments

A PL/I procedure can invoke other procedures, as well as pass values to and receive them from the invoked procedure. Values are passed 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.

You can specify arguments 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 can return values only through 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.
- A function can return values through its parameter list like a subroutine 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.

Example 7–1 illustrates the relationship between arguments (specified in a CALL statement) or function reference and parameters (specified in a PROCEDURE or ENTRY statement).

Example 7–1 Parameters and Arguments

```
CALLER:PROCEDURE;
  DECLARE COMPUTER EXTERNAL ENTRY  ❶
          (FIXED BINARY (7), CHARACTER (80) VARYING);
  CALL COMPUTER (5,'ABC'); ❷
END CALLER;
COMPUTER:PROCEDURE (X,Y); ❸
   DECLARE X FIXED BINARY (7); ❹
   DECLARE Y CHARACTER (80) VARYING;
END COMPUTER;
```

❶ The ENTRY attribute in a DECLARE statement provides a parameter descriptor for each parameter of the called procedure. A parameter descriptor is a set of data type attributes.

❷ In a CALL statement or a function reference, arguments appear in parentheses following the name of the procedure. Arguments can be variables, expressions, aggregates, or (as in this example) constants.

The data type of each argument is matched with the corresponding parameter descriptor in the declaration of the entry.

❸ The PROCEDURE statement for the called procedure specifies the parameters of the procedure. These parameters correspond, in the order specified, to the arguments specified in the CALL statement.

❹ Each parameter specified in the PROCEDURE statement must be declared within the procedure.

A parameter is a variable that occurs in the parameter list of a PROCEDURE or ENTRY statement. When the procedure is invoked, each argument variable in the list is associated with a parameter. Within the called procedure, any reference to the parameter is equivalent to a reference to the associated argument variable.

If the invoked procedure 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 procedure.

An argument variable associated with a parameter is passed in a dummy argument if the argument is specified as a constant. References to that parameter in the invoked procedure do not modify any storage in the invoking procedure. For example:

```
test: procedure options(main);

    declare a fixed bin(31);

    a = 100;
    call subroutine(a,100);
    put skip list (a,100);

end test;

subroutine: procedure(x,y);
    declare (x,y) fixed bin(31);

    x = 101;
    y = 101;

end subroutine;
```

The result of this example would be:

```
    101   100
```

7.5.1 Rules for Specifying Parameters

The general rules listed below for specifying parameters are followed by specific rules that pertain only to certain data types.

- You must declare a parameter explicitly in a DECLARE statement (to give it a data type) within the invoked procedure. This declaration must not be part of a structure.
- You cannot declare a parameter with any of these attributes:

AUTOMATIC	EXTERNAL	READONLY
BASED	GLOBALDEF	STATIC
CONTROLLED	GLOBALREF	

DEFINED INITIAL

- 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 through an entry variable must be specified by parameter descriptors in the ENTRY attribute of the variable's declaration. You cannot declare an internal entry (and its parameters) in the containing procedure.
- 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 or if the parameter is a constant, expression, or is enclosed in parentheses.

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 declare the bounds of a dimension for an array parameter 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 can be passed to this procedure.

All the data type attributes of the array argument and parameter must match.

Arrays are always passed by reference. They cannot be passed by dummy argument.

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. The level numbers do not have to be identical, but the levels must be logically equivalent. You can specify array bounds and string lengths with asterisks or with optionally signed integer constants. 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 can have any length. The string is fixed length unless VARYING is also included in the declaration.

Entry, File, and Labels Parameters

An entry, file, or label can be passed as an argument. The actual parameter is a variable.

7.5.2 Argument Passing

This section describes how PL/I passes an argument to procedures written in PL/I.

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 in the argument list against the number of parameters 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 ENTRY declaration list matches the number of arguments in the procedure invocation.

You specify arguments for a subroutine or function by enclosing the arguments in parentheses following the procedure or entry-point name. For example, a procedure call can be written 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 to the parameters X, Y, and Z.

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 variable is the actual argument for this procedure invocation. It can be one of the following:

- A reference to the written argument
- A dummy argument

The data type of the actual argument is the same as that 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 they match, 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 they do not match, the compiler creates a dummy argument and assigns to it the value of the written argument.

Dummy Arguments

A dummy argument is a unique temporary variable allocated by the compiler, which 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 the dummy argument following the rules of data type conversion before the call (this is described later). 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 cannot be a reference to an unconnected array. A dummy argument is never created for an array, structure, or area.

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 an internal procedure, the attributes of the argument must match those specified in the declaration of the parameter.
- For an external procedure or a procedure invoked through an ENTRY variable, the attributes specified in the ENTRY attribute parameter descriptor must match those 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 in Section 5.4.3.

7.6 Calling External and Internal Procedures

An external procedure is one whose text is not contained in any other block. The source text of an external procedure can be compiled separately from that of a calling procedure. The differences between internal and external procedures are as follows:

- Before an external procedure can be invoked (except through 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 types of the parameters that the procedure requires, if any, as well as a RETURNS attribute for a function procedure.

  You cannot explicitly declare internal procedures. The procedure name is implicitly declared by its occurrence in the PROCEDURE or ENTRY statement.

- External procedures can reference the same variable only if it is declared (implicitly or explicitly) with the EXTERNAL attribute in all of them.

  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 through an entry variable.

Example 7–2 illustrates the use of an internal procedure:

Example 7–2 Invoking an Internal Procedure

```
MAINP: PROCEDURE OPTIONS (MAIN);
   COMPUTE: PROCEDURE;
      ADD_NUMBERS: PROCEDURE;
      END ADD_NUMBERS;
   END COMPUTE;
   PRINT_REPORT: PROCEDURE;
   END PRINT_REPORT;
END MAINP;
```

In Example 7–2, 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.

Example 7–3 illustrates the use of an external procedure:

Example 7–3 Invoking an External Procedure

```
WINDUP: PROCEDURE;
DECLARE PITCH EXTERNAL ENTRY (CHARACTER(15) VARYING,
                              FIXED BINARY(7) );
CALL PITCH (PLAYER_NAME,NUMBER_OF_OUTS);
```

The procedure WINDUP declares the procedure PITCH with the EXTERNAL and ENTRY attributes. The text of PITCH is in another source program that is separately compiled. When the object module that contains WINDUP is linked, the linker must be able to locate the object module that contains PITCH. You can accomplish this by including both object modules on the linker command line, or by placing PITCH in an object module library and including the library on the linker command line.

When a 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 the example above. Such a declaration 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 them must define an entry point named PITCH as the label either of a PROCEDURE statement or an ENTRY statement. If the linker cannot locate an external entry point, it issues a warning message.

The parameter descriptors define the data types of the parameters for the entry point PITCH. Arguments of these types must be supplied when PITCH is invoked.

If PITCH is to invoke a function, the DECLARE statement must also include a RETURNS attribute describing the attributes of the returned value, as follows:

```
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.

A PL/I program can invoke an external procedure that is not written in PL/I. A common instance is the use of a system service by a PL/I program to obtain some system function not available directly through PL/I. Or, a PL/I program can invoke an external procedure written in another language that provides an application-specific function. Such instances are possible because of the OpenVMS Procedure Calling and Condition Handling Standard, which includes a set of conventions for passing arguments among procedures.

7.7 Terminating Procedures

The execution of subroutines and functions can be terminated with the following statements:

- 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. See Section 8.3 for more information about the END statement.

- 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. See Section 8.6 for more information about the GOTO statement.

- STOP statement

  A STOP statement ends the entire program execution. It does not pass a return value. See Section 8.8 for more information about the STOP statement.

- 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. The rest of this section describes the RETURN statement.

RETURN Statement

The RETURN statement terminates execution of the current procedure. The format of the RETURN statement is as follows:

```
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 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, control returns from the containing procedure to the calling 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.

7.8 Passing Arguments to Non-PL/I Procedures

There are three ways that a PL/I procedure can pass an argument to a non-PL/I procedure:

- By immediate value. The actual value of the argument is passed.
- By reference. The address in storage of the argument is passed.
- By descriptor. The address in storage of a data structure describing the argument is passed.

The following sections describe the requirements for each of these argument-passing mechanisms.

7.8.1 Passing Arguments by Immediate Value

To pass an argument by immediate value, use the VALUE attribute in a parameter description. The following declaration of the external entry VHF illustrates a declaration for an external routine that receives its parameter by immediate value.

```
DECLARE VHF ENTRY (FIXED BINARY(31) VALUE);
```

You can also define PL/I procedures that receive arguments by immediate value. To do this, you must specify the VALUE attribute in the declaration of the parameter. For example, the corresponding definition of the procedure VHF would be as follows:

```
VHF: PROCEDURE (LENGTH);
.
.
DECLARE LENGTH FIXED BINARY(31) VALUE;
.
```

Arguments that can be passed by immediate value are limited to the following data types:

- FIXED BINARY(m), where m <= 31
- FLOAT BINARY(n), where n <= 24
- BIT(o) ALIGNED, where o <= 32
- ENTRY
- OFFSET
- POINTER

PL/I supports the passing of external procedures, but not internal procedures, as value parameters. To pass an internal procedure, use an entry parameter.

When you specify the VALUE attribute in a parameter descriptor, you can specify the ANY attribute instead of declaring any data type attributes. For example, the declaration of SYS$SETEF can appear as follows:

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

At the time of the procedure's invocation, PL/I converts the written argument as needed to create a longword dummy argument.

You can use the VALUE built-in function to force an argument to be passed by immediate value to a non-PL/I procedure, regardless of the declaration of the formal parameter (see Section 11.4.97).

7.8.2 Passing Arguments by Reference

By default, PL/I passes all arguments by reference except character strings and arrays with nonconstant extents. The parameter descriptor for an argument to be passed by reference need specify only the data type of the parameter.

On OpenVMS systems for example, the Read Event Flags (SYS$READEF) system service requires that its first argument be passed by immediate value and its second by reference. You could declare this procedure as follows:

```
DECLARE SYS$READEF ENTRY (FIXED BINARY(31) VALUE,
                          BIT (32) ALIGNED);
```

When the procedure is invoked, the second argument must be a variable declared as BIT(32) ALIGNED. PL/I passes the argument by reference.

An argument of any data type can be passed by reference. Bit-string variables, however, must have the ALIGNED attribute.

The data types in the parameter descriptors of all output arguments must match the data types of the written arguments. For convenience, you can specify ANY in the parameter descriptor. To describe an argument to be passed by reference, you can specify the ANY attribute without the VALUE attribute. When you specify ANY for an argument to be passed by reference, you cannot specify data type attributes. Note that if you specify the VALUE attribute in conjunction with the ANY attribute, PL/I passes the argument by immediate value.

The ANY attribute is especially useful when you must specify a data structure as an argument. You need not declare the structure within the parameter descriptor, only the ANY attribute.

When an argument is passed by reference, PL/I passes the address of the actual argument. This address can be interpreted as a pointer value; you can explicitly specify a pointer value as an argument for data to be passed by reference; for example:

```
DECLARE SYS$READEF (ANY VALUE, POINTER VALUE),
        FLAGS BIT(32) ALIGNED;

CALL SYS$READEF (4, ADDR(FLAGS));
```

At this procedure invocation, PL/I places the pointer value returned by the ADDR built-in function directly in the argument list.

7.8.3 Passing Arguments by Descriptor

A descriptor is a structure that describes the data type, extents, and address of a data item. When passing an argument by descriptor, PL/I creates the descriptor and places its address in the argument list for the called procedure.

PL/I passes arguments by descriptor when a parameter descriptor specifies the following:

- A character string with an asterisk length or an array with asterisk extents
- An unaligned bit string or an array or structure consisting entirely of unaligned bit strings
- A structure containing any strings or arrays with asterisk extents

- ANY without VALUE, and the corresponding written argument is specified with the DESCRIPTOR built-in function

For example, PL/I passes by descriptor the arguments associated with the following parameter descriptors:

```
DECLARE UNSTRING ENTRY (CHARACTER(*)),
        TESTBITS ENTRY (BIT(3)),
        MODEST ENTRY (1,
                      2 CHARACTER(*),
                      2,
                        3 BIT(3),
                        3 BIT(3));
```

When you declare a non-PL/I procedure that requires a character-string descriptor for an argument, specify the parameter descriptor as CHARACTER(*). For example, the Set Process Name (SYS$SETPRN) system service requires the address of a character-string descriptor as an argument. You can declare this service as follows:

```
DECLARE SYS$SETPRN ENTRY (CHARACTER(*));
```

When a parameter is declared as CHARACTER(*), its written argument can be one of the following:

- A character-string constant or expression.
- A fixed-length character-string variable.
- A varying character-string variable or a variable declared as CHARACTER(*)VARYING.

For any of those arguments, PL/I constructs a character-string descriptor and passes its address.

To force an argument to be passed by descriptor, use the DESCRIPTOR built-in function. For example:

```
DECLARE P ENTRY (ANY);
DECLARE (X,Y) FIXED DECIMAL (7,2);

CALL P(DESCRIPTOR (X));
CALL P(Y);
```

Here, X is passed by descriptor as specified by the DESCRIPTOR built-in function. Y is passed by reference (see Section 11.4.28).

8 Program Control

The statements described in this chapter direct the run-time flow of execution from statement to statement. They are the DO, BEGIN, END, IF, SELECT, GOTO, LEAVE, STOP, null, ON, SIGNAL, and REVERT statements.

The remainder of the chapter is devoted to handling conditions that could arise during the execution of your program.

8.1 DO Groups and Statements

A DO-group is a sequence of PL/I statements delimited by a DO statement and its corresponding END clause. 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 to be executed in this conditional DO-group are complete, execution continues with the next executable statement following the END statement.

Normally, all the statements contextually nexted one level below the DO in the group are executed. However, control can be transferred out of a DO-group in the following ways:

- By execution of a GOTO statement that transfers control to a label outside the DO-group. The GOTO statement can be present in the DO-group itself, in a procedure invoked directly or indirectly from within the DO-group, or in an ON-unit activated while the DO-group is active.
- By execution of a LEAVE statement that transfers control outside the containing DO-group or to the next executable statement following the END statement that terminates the DO-group.
- By execution of a RETURN or STOP statement that terminates the current procedure or program.

You can nest DO-groups to a maximum level of 64.

The DO statement begins a sequence of statements to be executed in a group; the group ends with the nonexecutable statement END. DO-groups have several formats. These formats are described individually under the following subheadings:

- Simple DO
- DO WHILE
- DO UNTIL
- Controlled DO
- DO REPEAT

8.1.1 Simple DO

A simple DO statement is a noniterative DO. The statements nested directly between the DO statement and its corresponding END statement are executed once. PL/I treates these nested statements as if they are one statement. After all statements in the group are executed, control passes to the next executable statement in the program.

The format of a simple DO statement is:

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

Examples

```
IF A < B THEN DO;
   PUT LIST ('More data needed');
   GET LIST (VALUE);
   A = A + VALUE;
   END;
```

The simple DO statement is commonly used as the action of the THEN clause of an IF statement, as shown above, or of an ELSE option.

8.1.2 DO WHILE

A DO WHILE statement causes a group of statements to be repeatedly executed as long as a particular condition is satisfied. When the condition is not true, the group is not executed.

The format of the DO WHILE statement is:

```
DO WHILE (test-expression);
.
.
.
END;
```

test-expression

Any expression that yields a scalar 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. (Comparison operations yield a value with the type BIT(1).)

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 over the DO-group to the next executable statement following the END statement that terminates the group.

Examples

```
DO WHILE (A < B);
   .
   .
   .
END;
```

This DO-group is executed as long as the value of the variable A is less than the value of the variable B.

```
DO WHILE (LIST->NEXT ^= NULL());
   .
   .
   .
END;
```

This DO-group is executed until a forward pointer in a linked list has a null value.

```
DECLARE EOF BIT(1) INITIAL('0'B);
   .
   .
   .
ON ENDFILE(INFILE) EOF = '1'B;
READ FILE(INFILE) INTO(INREC);
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 <BIT_STRING>(1)B until the ENDFILE ON-unit sets the value of EOF to <BIT_STRING>(1)B.

8.1.3 DO UNTIL

A DO UNTIL statement causes a group of statements to be repeatedly executed until a particular condition is satisfied. That is, while the condition is false, the group is repeated.

The format of the DO UNTIL statement is:

```
DO UNTIL (test-expression);
.
```

```
.
.
END;
```

test-expression

Any expression that yields a scalar 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. (Comparison operations yield a value having the type BIT(1).)

This expression is evaluated after each execution of the DO-group. It must have a false value for the DO-group to be repeated. Otherwise, control passes to the next executable statement following the END statement that terminates the DO-group. The test expression must be enclosed in parentheses.

Note: Both the WHILE and UNTIL options check the status of test expressions, but they differ in that the WHILE option tests the value of the test expression at the beginning of the DO-group, and UNTIL tests the value of the test expression at the end of the DO-group. Therefore, a DO-group with the UNTIL option and no WHILE option will always be executed at least once, but a DO-group with the WHILE option may never be executed.

Examples

```
DO UNTIL (A=0);
 .
 .
 .
END;
```

This DO-group is executed at least once and continues as long as the value of A is not equal to zero.

```
DO UNTIL (K<ALPHA);
 .
 .
 .
END;
```

This DO-group is executed at least once and continues as long as the value of the variable K is greater than or equal to the value of the variable ALPHA.

8.1.4 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 repeatedly 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) ]
    [ UNTIL (test-expression) ]
    ;
.
.
.
END;
```

control-variable
A reference to a variable whose current value, as compared to the end value specified in the TO option, determines whether the DO-group is executed. If none of the options are specified, the DO-group is executed a single time regardless of the value of the control variable. 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. This expression is evaluated exactly once when the statement is executed for the first time. Thus if the end value changes as the loop progresses, only this initial value is used.

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. This expression is evaluated exactly once when the statement is executed for the first time. Thus if the modify value changes as the loop progresses, only this initial value is used. 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 condition must be true at the beginning of each DO-group iteration for the DO-group to be executed. The specified test expression must yield a scalar 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.

UNTIL (test-expression)
An option specifying a condition that further controls the execution of the DO-group. This expression is evaluated at the end of each interation of the DO-group, before the BY clause is applied to the control variable. The condition must be false at the end of a DO-group execution for the next DO-group iteration to be executed. The specified test expression must yield a scalar 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.

Note: If the TO, WHILE, and UNTIL options are omitted, the controlled DO 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.

Example

This DO-group will prompt the user for integer input values, and add each input value to the current sum. When the sum is greater than 100, the DO-group will exit.

```
DECLARE (NEXT_VALUE,SUM) FIXED BIN;

    SUM = 0;
    DO UNTIL ( SUM > 100 );
 GET LIST (NEXT_VALUE) OPTIONS (PROMPT ('Next value to add? '));
 SUM = SUM + NEXT_VALUE;
    END;

    PUT SKIP LIST ('The total sum is ',SUM);
```

The controlled DO-group is executed by the following steps:

1 The following measures 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 BY options (if specified) are evaluated, and their values are stored. These expressions can contain references to the object referenced by the control variable. If they do, 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 than the end value.

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 WHILE option is present, its test expression 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 can be terminated during this step by the execution of a STOP or RETURN statement or by the execution of a GOTO or LEAVE statement that transfers control out of the DO-group.

The body of the DO-group can also contain statements that change the values of the control variable, modify value, end value, or test expression. Changing the modify value or the end value in the body of the loop will not affect the number of times the loop is iterated. However, changing the value of the control variable or the test expression can affect the number of iterations if the control variable was not made into a temporary by step 1.

6 If an UNTIL option is present, its test expression is evaluated. If it produces a true value, the execution of the DO-group terminates.

7 Unless none of the options are specified, the value of the control variable is modified as follows:

control variable = control variable + modify value;

8 Execution continues at step 3 unless none of the options are specified, in which case control passes to the next executable statement in the program.

Examples

```
DO I = 2 TO 100 BY 2;
```

This DO-group is executed 50 times, with values for I of 2, 4, 6, and so on.

```
DO I = LBOUND(ARRAY,1) TO HBOUND(ARRAY,1);
```

This DO-group is executed as many times as there are elements in the array variable ARRAY, using the subscript values of the array's elements for the values of I.

```
DO I = 1 BY 1 WHILE (X < Y);
```

This DO-group continues to be executed with successively higher values for I while the value of X is less than the value of Y.

```
DO I = 1 BY 1 WHILE (X < Y) UNTIL (X = 12);
```

This DO-group resembles the DO-group in the preceding example, except that the DO-group continues to be executed while the value of X is less than the value of Y or until the value of X is equal to 12.

A controlled DO statement that does not specify a TO or BY option results in a single iteration of the following DO-group. For example:

```
DO X = 1 WHILE (A);
```

Even if A is true, this DO-group executes a single time only. If A is false, it is not executed at all. Because there is no expression to change the value of X, the DO-group will not be executed again.

```
DO X = 1;
```

This DO-group executes a single time only, regardless of the value of X.

8.1.5 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 can also be included. If it is included, the WHILE expression is evaluated before each iteration (including the first), but after the control variable has been assigned. The format of the DO REPEAT statement is:

```
DO variable = start-value REPEAT expression
        [WHILE (test-expression)] [UNTIL (test-expression)];
.
.
.
        END;
```

variable
A reference to a control variable. The control variable can be any scalar variable.

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 iteration after the first. 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 DO REPEAT group continues while the condition is true. The specified test expression must yield a scalar 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 to the next executable statement following the END statement that terminates the DO-group.

See Section 8.1.4 for a discussion of this option when used with the controlled DO statement.

UNTIL (test-expression)

An option specifying a condition that further controls the termination of the DO REPEAT group. The DO REPEAT group continues until the condition is true. The specified test expression must yield a scalar value. If any bit in 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 after the first execution of the DO-group; the test expression must have a true value in order for the DO-group to be executed a second time. Otherwise, control passes to the next executable statement following the END statement that terminates the DO-group.

Note: If the WHILE and UNTIL options are 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 by the following steps:

1 The following measures 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 a WHILE option is present, its test expression 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 or LEAVE statement that transfers control outside the DO-group. Statements in the group can also modify the values of the control control variable, REPEAT expression, and test expression.

5 If an UNTIL option is present, its test expression is evaluated. If it produces a true value, the execution of the DO-group terminates. If the test expression is not present, execution continues.

6 The REPEAT expression is evaluated and its value is assigned to the control variable.

7 Execution continues at step 3.

Examples

```
DO LETTER='A' REPEAT (BYTE(I));
```

This example will repeat the group with an initial LETTER value of <BIT_STRING>(A) and with subsequent values assigned by the built-in function BYTE(I). The control variable I can 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 following 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 control variable LIST_HEAD. The DO-group is then 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.

8.2 BEGIN Statement

The BEGIN statement denotes the start of a begin block.

The format of the BEGIN statement is:

BEGIN;

A begin block is a sequence of statements headed by a BEGIN statement and terminated by an END statement. 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. Variables declared as internal within a begin block are not allocated storage until the block is activated. When the block terminates, storage for internal automatic variables is released. A begin block terminates in the following situations:

- When its corresponding END statement is encountered. Control continues with the next executable statement in the program.
- When it executes a nonlocal GOTO to transfer control to a previous block.
- When it executes a RETURN statement.

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 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. Furthermore, there are some cases (such as ON-units) in which DO-groups cannot be used. Otherwise, DO-groups are often more efficient, because they do not have the overhead associated with block activation. In general, you should use a DO-group instead of a begin block unless there are declarations present or you require multiple statements in an ON-unit.

A begin block can designate a series of statements to be executed 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;
```

See Section 1.4 for more details on blocks.

8.3 END Statement

The END statement marks the end of the block or group headed by the most recent BEGIN, DO, SELECT, or PROCEDURE statement.

The format of the END statement is:

END [label-reference];

label-reference

A reference to the unsubscripted label on the PROCEDURE, BEGIN, SELECT, 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, SELECT, or PROCEDURE statement that is not already matched with an END statement. If the label reference is omitted, the most recent textual, non-terminated PROC, BEGIN, SELECT, or DO statement is matched by default.

Note that a procedure declared with the RETURNS option must execute a RETURN statement before it encounters the END statement marking the end of the procedure.

When the END statement is encountered, one of the following actions is performed, depending on the type of block or group that it terminates:

- When an END statement denotes the end of a procedure, the 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 returns to the statement following the CALL statement or function invocation that invoked the procedure.

- When an END statement denotes the end of a begin block, the block is terminated. Storage allocated for the block is released, and all automatic variables are made inaccessible. Control passes to the next executable 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 executable statement following the END statement. If the DO-group is headed by a simple DO, that is, one that causes the DO-group to be executed only once, control passes to the next executable statement. Otherwise, control returns to the head of the DO-group.
- When an END statement denotes the end of a SELECT-group, the SELECT-group is terminated and control passes to the next executable statement following the end statement.

8.4 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 the nonexecutable statements: DECLARE, END, ENTRY, FORMAT, or PROCEDURE
- 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 the 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).

8.4.1 Nested IF Statements

The action specified in a THEN or an ELSE clause can 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 this 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 can require a null statement (a semicolon) 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.

8.5 SELECT Statement

The SELECT statement tests a series of expressions and performs a specified action depending on the result of the test. The statement has two forms: in the first form, the expressions in a WHEN clause are tested for truth; in the second form, the expressions in a WHEN clause are compared to see if any have the same value as another specified expression called the *select expression*. Any of the expressions can be, but need not be, constants. An optional OTHERWISE clause is available to name an action to be performed if none of the preceding expressions have satisfied the condition specified.

The two forms of the SELECT statement and the OTHERWISE clause are described in more detail below.

The general form of the SELECT statement is:

```
SELECT [(select-expression)];
    [WHEN [ANY | ALL] (case-expression, . . . ) [action] ; ] . . .
    [{OTHERWISE | OTHER} [action] ; ]
END;
```

select-expression
An expression that can be evaluated to any type of value.

case-expression, . . .
One or more expressions to be tested, evaluating to bit-string values, or, if a select expression is used, with values that will be compared to the select expression's value.

action
Any statement (including a null statement, another SELECT statement, a DO-group, or a BEGIN-END block) except a DECLARE, END, ENTRY, FORMAT, or PROCEDURE statement.

8.5.1 The Two Forms of the SELECT Statement

Depending on whether you use a select expression or not, SELECT has two different forms, which are explained in detail below.

SELECT Without a Select Expression

The first form of the SELECT statement omits the select expression. In this form, the expressions in a WHEN clause are evaluated, and a specified action is performed if the result of *any* test is true (or, if ALL is specified, the results of all tests are true); an expression is *true* if it evaluates to a bit string containing any bit with the value of '1'B. In the usual case, the test for truth results in a bit string containing one bit: '1'B for true or '0'B for false.

When the keyword ANY (the default) appears in the WHEN clause, then if any one of the expressions evaluates to true the corresponding action is performed. No further expressions in that WHEN clause or in subsequent WHEN clauses are evaluated (and thus the expressions need not have unique values), and no subsequent actions are performed.

The WHEN clauses are checked in the order listed. However, the expressions within one WHEN clause might be evaluated in any order, and not all these expressions are necessarily evaluated. As soon as any expression is found true, subsequent expressions are not evaluated.

If the keyword ALL appears in the WHEN clause, the action is performed only if all expressions in that WHEN clause evaluate to true. Once one action is performed, no subsequent WHEN clauses are evaluated and no subsequent actions are performed. If any expression in the WHEN clause does not result in a true value, no further expressions in that clause are evaluated and the action is not performed.

Following is an example of the first form of SELECT:

```
SELECT;
    WHEN ANY (A=10,A=20,A=30) B=B+1;
    WHEN (A=50) B=B+2;
    WHEN (A=60) B=B+3;
    WHEN (A=70) B=B+4;
    WHEN (A=80) B=B+5;
    WHEN (A=90) B=B+6;
    WHEN ALL (A>90,A<500) B=B+10;
    OTHERWISE B=B+C;
END;
```

The SELECT statement defines the action to be taken if the variable A has any of the values specified in the WHEN clauses (or, in the case of the WHEN ALL clause, if A is both greater than 90 and less than 500). If none of the WHEN clauses is true, the action specified in the OTHERWISE clause (B=B+C) is performed.

SELECT With a Select Expression

The second form of the SELECT statement has a select expression after the keyword SELECT. This form of the SELECT statement evaluates expressions in the WHEN clauses and then compares their values to the value of the select expression (instead of testing the expressions for truth or falsity, as in the first form of SELECT). It performs a specified action if any expression has the same value as the select expression (or, if ALL is used, all expressions have the same value as the select expression). In this form of the SELECT statement, as in the previous form, the expressions in a WHEN clause might be evaluated in any order, and not all the expressions are necessarily evaluated.

Following is an example of the second form of SELECT:

```
SELECT(A);
    WHEN  (50) C=C+1;
    WHEN ANY (60,61,62,B+C) C=C+2;
    WHEN ALL (70,D) C=C+3;
    OTHERWISE C=C+D;
END;
```

The SELECT statement defines the action to be taken if the select expression (A in the example) evaluates to any or all of the values of the expressions following a WHEN clause. The first action (the assignment statement C=C+1) will be performed if A has a current value of 50. In that case, none of the subsequent WHEN clauses will be evaluated. The second WHEN clause includes the ANY keyword, and so the second action will be performed if A evaluates to or equals 60 or 61 or 62 or the sum of B and C. If neither the first nor the second action is performed, the third WHEN clause's expressions are tested. The third WHEN clause includes the ALL keyword, so the third action will be performed only if A equals both 70 and D. If none of the WHEN clauses causes an action to be performed, then the action in the OTHERWISE clause (the assignment statement C=C+D) will be performed.

8.5.2 OTHERWISE Clause

If none of the WHEN clauses causes the corresponding action to be performed, the action specified in the optional OTHERWISE clause is performed; but if the OTHERWISE clause is omitted, an ERROR condition is signalled. OTHERWISE can be followed by a semicolon (a null statement) to cause execution to continue and avoid an ERROR condition when you do not want to specify an action after OTHERWISE. For example:

```
OTHERWISE;
```

After an action is performed following a WHEN or OTHERWISE clause, control passes to the next executable statement following the END statement that terminates the SELECT statement, unless normal flow is altered within the action.

8.5.3 Nested SELECT Statements

Note that the action specified in a WHEN or OTHERWISE clause can be another SELECT statement, resulting in nested SELECT statements, as in the following example:

```
SELECT;
   WHEN (condition A)
        SELECT;
           WHEN (condition A1) statement 1;
           WHEN (condition A2) statement 2;
           END;
   WHEN (condition B)
        SELECT;
           WHEN (condition B1) statement 3;
           WHEN (condition B2) statement 4;
           OTHERWISE statement 5;
           END;
   OTHERWISE statement 6;
END;
```

In this example, statement 1 is executed when both condition A and condition A1 are true. Statement 2 is executed when both condition A and condition A2 are true and A1 is false. If A is true but neither A1 nor A2 is true, an ERROR condition is reported because no OTHERWISE clause exists within this SELECT statement.

If condition A is false, condition B is checked. If B is true but B1 and B2 are both false, statement 5 (in the corresponding OTHERWISE clause) is executed. If conditions A and B are both false, statement 6 (in the outermost OTHERWISE clause) is executed.

If you want to avoid the possibility that execution could be stopped by an ERROR condition, which occurs in this example if condition A is true and A1 and A2 are false, you can put in an OTHERWISE clause with a null statement (a semicolon) as its action, which would cause control to pass to the first executable statement following the end of the outermost SELECT statement.

An END statement must terminate each SELECT statement.

8.6 GOTO Statement

The GOTO statement causes control to be transferred to a labeled statement in the current or any outer procedure.

The format of the GOTO statement is:

{ GOTO / GO TO } label-reference [OTHERWISE];

label-reference

A label constant or an expression that, when evaluated, yields a label value. A label value denotes a statement in the program.

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;
X(1) = NEXT;

GOTO X(1);
```

In the case of a label variable, the resulting label value must designate an existing block activation. (Similarly, a label constant must designate an existing block activation.) If the designated block activation is the current block activation, the GOTO statement causes a local GOTO. No special processing occurs.

OTHERWISE

This option can be used only when the label-reference is a subscripted label with a variable subscript. If present in any other case, it will be reported as an error.

If the variable subscript is out of range and the OTHERWISE option is present, the statement following the GOTO will be executed next. If the OTHERWISE option is not specified and the subscript of the last label is not an asterisk (*), the subscript is reported out of range at run-time and the process will be terminated.

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 both 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.
- A special case of a nonlocal GOTO occurs if a GOTO is executed in an ON-unit. The condition handling mechanism signals SS$_UNWIND before control transfers to the label. This allows programs to clean up intervening blocks before proceding.

Examples

The following example shows the use of the GOTO statement:

```
RESTART:;
  .
  .
  .
  BEGIN;
    ON ERROR GOTO RESTART;
     .
     .
     .
  END;
```

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.

The following restrictions apply to the use of labels and label data:

- No statement can have more than one label. However, an executable statement can be preceded by any number of labeled null statements, which have the same effect as would multiple labels.

- Operations on label values are restricted to the operators = and ^=, for testing equality or inequality. Two values are equal if they refer to the same statement in the same block activation.
- Any statement in a PL/I program can be labeled except the following:
  - A DECLARE statement
  - A statement beginning an ON-unit, THEN clause, ELSE clause, WHEN clause, or OTHERWISE clause
- Labels on PROCEDURE, ENTRY, and FORMAT statements are not considered statement labels and cannot be used as the targets of GOTO statements.
- An identifier occurring as a label in a block cannot be declared in that block (except as a structure member) or occur in the block's parameter list.
- Any reference to a label value after its block activation terminates is an error with unpredictable results.

For more information on labels, see Section 3.7.

8.7 LEAVE Statement

The LEAVE statement causes control to be transferred out of the immediately containing DO-group or out of the containing DO-group whose label is specified with the statement.

The format of the LEAVE statement is:

```
LEAVE [label-reference];
```

label-reference

A reference to a label on a DO statement that heads a containing DO-group. The label reference can be a label constant or a subscripted label constant for which the subscript is specified with an integer constant. The label reference cannot be a label variable, nor can it be a subscripted label constant for which the subscript is specified with a variable.

On execution, a LEAVE statement with no label reference causes control to be transferred to the first statement following the END statement that terminates the immediately containing DO-group. If the LEAVE statement has a label, control is passed to the first executable statement following the END statement for the corresponding label indicated in the LEAVE statement. Thus, the LEAVE statement provides an alternative means of terminating execution of a DO-group. In the case of a LEAVE statement with a label reference, several nested DO-groups can be terminated as control transfers outside the referenced DO-group.

Restrictions

The following restrictions apply to the use of the LEAVE statement:

- A LEAVE statement must be contained within a DO-group.
- A LEAVE statement must be in the same block as the DO statement to which it refers.

- If a LEAVE statement has a label reference, it must refer to a label on a DO statement that heads a DO-group that contains the LEAVE statement. The LEAVE statement must be in the same block as the labeled DO statement.
- The label reference specified with a LEAVE statement must be a label constant or a subscripted label constant with an integer constant subscript.

Examples

The following example shows a LEAVE statement without a label reference:

```
DO I = 1 TO 100;
   .
   .
   .
    IF COMMAND = 'QUIT' THEN LEAVE;
   .
   .
   .
END;
PUT LIST ('Job finished');
```

In this example, the LEAVE statement transfers control directly to the PUT statement if the condition in the IF statement is satisfied.

The next example shows a LEAVE statement with a label reference:

```
LOOP1: DO WHILE (MORE);
   .
   .
   .
    LOOP2: DO I = 1 TO 12;
   .
   .
   .
        IF QUAN(I) > 150 THEN LEAVE LOOP1;
    END;    /* Loop 2 */
   .
   .
   .
END;        /* Loop 1 */
```

In this example, the LEAVE statement transfers control to the first statement beyond the END statement that terminates LOOP1.

The following examples show some invalid uses of the LEAVE statement:

```
LEAVE;                    /* LEAVE statement must be in */
                          /* DO-group */

DO;
    BEGIN;
        LEAVE;            /* LEAVE statement must be in */
    END;                  /* same block as DO statement */
END;

ON ENDFILE(SYSIN) LEAVE;  /* ON-unit is separate block */
```

```
DECLARE LABVAR LABEL VARIABLE;
LABVAR = LOOP;
LOOP: DO I = 1 TO 10;
    LEAVE LABVAR;          /* Label reference cannot be a variable */
END;

LAB(1): DO;
    LAB(2): DO;
        I = 1;
        LEAVE LAB(I);      /* Subscript must be a constant */
    END;
END;
```

8.8 STOP Statement

The STOP statement terminates execution of a program, regardless of the current block activation.

The format of the STOP statement is:

STOP;

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.

8.9 Null Statement

The null statement performs no action. Its format is:

;

The null statement usually serves as the target statement of a THEN or ELSE clause in an IF statement, as the target of a WHEN or OTHERWISE clause in a SELECT statement, or as an action in an ON-unit. The following examples illustrate these uses.

```
IF A < B THEN GOTO COMPUTE;
     ELSE ;
```

In this example, no action takes place if A is greater than or equal to B; execution continues at the statement following ELSE ;. A construction of this type may be necessary when IF statements are nested (see Section 8.4.1).

```
SELECT;
    WHEN (condition A,B,C) GOTO FILE_READ;
    WHEN (condition D,E) GOTO UPDATE;
    OTHERWISE;
END;
```

In this example, control is passed to the next executable statement after END if conditions A, B, C, D, and E are not true.

```
ON ENDPAGE(SYSPRINT);
```

In this example, no action takes place upon execution of the ON-unit; the I/O operation that caused the ENDPAGE condition continues.

The null statement can also be used to declare two labels for the same executable statement, as in the following example:

```
LABEL1: ;
LABEL2: statement  . . .
```

8.10 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.

An ON condition is any one of several named conditions whose occurrence during the execution of a program interrupts the program. When a condition occurs or is signaled, a statement or sequence of statements, called an ON-unit, is executed. The SYSTEM option can be specified in the ON statement, causing the default system condition handling to be executed.

The following list of condition handling topics are discussed in subsequent sections.

- The ON statement
- The SIGNAL statement
- The REVERT statement
- 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

8.10.1 ON Statement

The ON statement defines the action to be taken when a specific condition or conditions are signaled during the execution of a program. The ON statement is an executable statement. It must be executed before the statement that signals the specified condition.

The format of the ON statement is:

ON condition-name, . . . [SNAP] { on-unit / SYSTEM; }

condition-name, . . .

The name or names of the specific conditions for which an ON-unit or the SYSTEM option is specified. There is a keyword name associated with each condition. Successive keyword names must be separated by commas. The conditions are summarized in Table 8–1; each condition is described in an individual entry in this manual.

SNAP

This option invokes the debugger and causes a traceback of all active routines to be displayed when the condition is raised. If you use the SNAP option, you should specify the /DEBUG qualifier on both the PLI command and the LINK command in order to have all the debugger symbol table information accessible.

If you want to run a program containing the SNAP option in a batch job and cause the program to resume execution after any display of traceback information, you can define DBG$INIT to point to a debug initialization file that contains the following line:

```
WHILE PC ^= 0 DO(GO)
```

on-unit

The action to be taken when the specified condition or conditions are signaled. An ON-unit can be:

- Any single, unlabeled statement except DECLARE, DO, END, ENTRY, FORMAT, IF, ON, PROCEDURE, RETURN, or SELECT.
- An unlabeled begin block.
- A null statement (a semicolon alone), which causes program execution to continue as if the condition had been handled.

Only the most recent ON-unit established for a given condition can be active. If two successive ON statements are executed for the same condition, the second ON statement nullifies the first.

If no ON-unit is established for a particular condition, the condition ERROR is signaled. If no ON-unit is established for ERROR condition, the default system ON-unit is executed. See Section 8.10.5.

SYSTEM

This option invokes the default system condition handling for the specified condition, overriding any existing ON-unit for the condition. See Section 8.10.6.

8.10.2 SIGNAL Statement

The SIGNAL statement causes a specified condition to be signaled, which causes the system to search for and execute an ON-unit to handle the condition. See Section 8.10.8.

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 in Table 8–1. Each of these conditions is described in its own section.

Most conditions occur as a result of a hardware trap or fault, or as a result of signaling by PL/I run-time procedures. You can use the SIGNAL statement within a program as a general-purpose communication technique. In particular, the VAXCONDITION and CONDITION conditions let you signal unique user-defined condition values.

8.10.3 REVERT Statement

The REVERT statement cancels an ON-unit established for a specified condition or conditions in the current block only.

The format of the REVERT statement is as follows:

REVERT condition-name, . . . ;

condition-name, . . .

The keyword name or names associated with the condition or conditions for which the ON-unit is to be reverted. Successive names must be separated by commas. The valid condition names are the same as for the ON statement.

If no ON-unit is established for a specified condition for the current block, the REVERT statement has no effect.

The REVERT statement does not cancel all ON-units that may be active at the same time it is executed (see Section 8.10.8), only a handler in the current block. If you want to temporarily block activation of all user-written ON-units for a condition, define an ON-unit with the SYSTEM option for the given condition.

An ON-unit can be re-established after execution of a REVERT statement by subsequently executing an ON statement.

The REVERT statement has no effect on an ON-unit established for the ANYCONDITION condition unless the statement explicitly references the ANYCONDITION condition. For example:

```
REVERT ANYCONDITION;
```

Therefore, a statement such as the following has no effect on an ON-unit established for the ANYCONDITION condition:

```
REVERT ZERODIVIDE;
```

The next example shows this more clearly:

```
PROGRAM: PROCEDURE OPTIONS(MAIN);
   DECLARE A FIXED;
   DECLARE B FIXED INITIAL (5);
   DECLARE C FIXED INITIAL (0);

   ON ANYCONDITION BEGIN;
      PUT SKIP LIST ('Handled condition = ',ONCODE());
      END;

   REVERT ZERODIVIDE;
```

```
   A = B / C;    /* Signal divzero */

   ON ZERODIVIDE BEGIN;
      PUT SKIP LIST ('Handled ZERODIVIDE');
      END;

   A = B / C;    /* Signal divzero */

   REVERT ZERODIVIDE;

   A = B / C;    /* Signal divzero */

   ON ZERODIVIDE BEGIN;
      PUT SKIP LIST ('Handled ZERODIVIDE');
      END;

   REVERT ANYCONDITION;

   A = B / C;    /* Signal divzero */

   REVERT ZERODIVIDE;

   A = B / C;    /* Signal divzero */

END PROGRAM;
```

When you run the program, the following conditions result:

```
Handled condition =       1156
Handled ZERODIVIDE
Handled condition =       1156
Handled ZERODIVIDE
PL/I ERROR condition.
```

8.10.4 Summary of ON Conditions

Most, but not all, ON conditions are associated with errors. The types of conditions for which you can establish ON-units are grouped in the following categories.

- Conditions that occur during I/O operations:
  - — ENDFILE, to take action when the end-of-file occurs while a file is being read
  - — ENDPAGE, to take action when the last line on a page is printed
  - — KEY, to take action when an error occurs when a record is accessed 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:
  - — FIXEDOVERFLOW, to respond when integer or fixed-point values become too large to be expressed
  - — OVERFLOW, to respond when floating-point values become too large to be expressed
  - — UNDERFLOW (OpenVMS VAX systems only), 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
- Other conditions:
  - — AREA, to respond when an error has been detected during performance of an operation on an area (various subconditions can be determined through use of the ONCODE built-in function)
  - — CONDITION, to respond to programmer-defined conditions
  - — CONVERSION, to respond to data conversion errors from CHARACTER to any arithmetic data type or bit string
  - — STORAGE, to respond when an error has been detected during allocation of a controlled variable or a based variable other than in an area
  - — STRINGRANGE, to respond to substring references that are beyond the length of the string.
  - — SUBSCRIPTRANGE, to respond to array references with out-of-bound subscripts.
- General classes of exceptional conditions:
  - — ANYCONDITION, to respond to all conditions for which no specific ON-unit is established in the current block
  - — ERROR, to respond to language-specific and run-time-specific errors
  - — FINISH, to respond when a STOP statement is executed
  - — VAXCONDITION, to respond to condition values that are specific to the operating system or to be used as user-defined conditions created by SIGNAL VAXCONDITION(n).

Table 8–1 summarizes ON conditions. Each condition is described individually in the sections that follow.

Table 8–1 Summary of ON Conditions

Condition Name	Function
ANYCONDITION	Handles any condition not specifically handled by another ON-unit
AREA	Handles a condition that occurs during an operation on an area
CONDITION	Handles programmer-defined conditions
CONVERSION	Handles data conversion errors
ENDFILE	Handles end-of-file 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

Table 8–1 (Cont.) Summary of ON Conditions

Condition Name	Function
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 during keyed access to a specified file
OVERFLOW	Handles floating-point overflow exception conditions
STORAGE	Handles a condition that occurs during allocation of a controlled variable or a based variable other than in an area
STRINGRANGE	Handles out-of-bound substring references
SUBSCRIPTRANGE	Handles out-of-bound array references
UNDEFINEDFILE	Handles any errors in opening a specified file
UNDERFLOW (VAX only)	Handles floating-point underflow exception conditions
VAXCONDITION	Handles operating system or programmer-specified condition values
ZERODIVIDE	Handles divide-by-zero exception conditions

8.10.4.1 ANYCONDITION Condition Name

The ANYCONDITION keyword can be specified in an ON, REVERT, or SIGNAL 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 OpenVMS operating system environment. For detailed information on OpenVMS condition handling, see the *Kednos PL/I for OpenVMS Systems User Manual*.

8.10.4.2 AREA Condition Name

The AREA condition is raised when various operations fail in relation to areas. For example, it is raised if the extent of an area is not large enough to contain the variable or variables allocated to it, or if the area is incorrectly formatted.

For more information see the *Kednos PL/I for OpenVMS Systems User Manual*.

8.10.4.3 CONDITION Condition Name

The CONDITION condition name is used for ON-units to handle programmer-defined conditions. The value returned by the ONCODE built-in function is PLI$_CONDITION. There is no way to distinguish between multiple programmer-defined conditions if they are specified in the same ON statement.

The format of the CONDITION condition name is:

CONDITION (cond-name)

cond-name

A name declared with the CONDITION attribute.

8.10.4.4 CONVERSION Condition Name

The CONVERSION condition name can be specified in an ON, SIGNAL, or REVERT statement to designate a CONVERSION condition or ON-unit.

PL/I signals the CONVERSION condition when the source character data in a conversion to bit-string or arithmetic data contains characters that are not valid in the specified context. In particular, the CONVERSION condition is raised when a character string is being converted and one of the following conditions is true:

- The target of the conversion is an arithmetic type, and the source string does not contain a valid, optionally signed arithmetic constant.
- The target of the conversion is a picture, and the source string does not conform to the picture specification.
- The target of the conversion is a bit string, and a character other than 0 or 1 appears in the source string.

The CONVERSION condition can be raised either by a non-I/O conversion, such as an explicit conversion using a built-in function or an implicit conversion generated by the compiler, or by an I/O conversion in a GET statement. For example, A = BIT('1014') would cause the CONVERSION condition to be raised, because 4 is not a valid binary digit. Likewise, a GET statement with an arithmetic target would also cause the CONVERSION condition to be raised if the characters '12K45' appeared in the input field, because 'K' is not a valid numeric character.

You can use the ONSOURCE and ONCHAR built-in functions and pseudovariables inside an ON CONVERSION ON-unit. The ONSOURCE built-in function returns the source string that caused the CONVERSION condition to be raised. The ONCHAR built-in function returns the specific character that caused the conversion to fail. You can use the ONSOURCE pseudovariable to change the value of the conversion. Likewise, you can use the ONCHAR pseudovariable to modify only the single character in error.

If the CONVERSION condition was raised during a conversion required by the GET statement, the ONFILE built-in function returns the name of the file constant inside the CONVERSION ON-unit. If the CONVERSION condition was not raised during a conversion required by the GET statement, the ONFILE built-in function returns a null string.

ON-Unit Completion

A normal return from a CONVERSION condition will cause the conversion to be reattempted if the ONSOURCE or ONCHAR pseudovariables have had values assigned to them. If the ONSOURCE value has not been modified, the ERROR condition is raised instead.

For example:

```
/*
* Sample program that displays a 'quick-fix' CONVERSION
* ON-unit. At the end of this program, TARGET1 contains
* the value 14015, and TARGET2 contains the value '11100'B.
* Note that SOURCE1 and SOURCE2 are not modified.
*/
MAIN: PROCEDURE OPTIONS(MAIN);

    DCL SOURCE1 CHARACTER(5) VARYING INITIAL('14$15');
    DCL SOURCE2 CHARACTER(5) VARYING INITIAL('11q00');

    DCL TARGET1 FIXED BINARY(31);
    DCL TARGET2 BIT(5) ALIGNED;

    /*
    * Sample 'quick-fix' CONVERSION ON-unit that replaces
    * erroneous lowercase q's with 1's, and all other
    * erroneous characters with 0's.
    */
    ON CONVERSION BEGIN;

    PUT SKIP EDIT('"',ONSOURCE(),'" "',ONCHAR(),'"')((5)A);

    IF ONCHAR() = 'q'
    THEN
        ONCHAR() = '1';
    ELSE
        ONCHAR() = '0';

    END; /* ON */

    /*
    * Note that the CONVERSION condition is raised for all
    * 3 of the following statements.
    */
    TARGET1 = SOURCE1;
    TARGET1 = SOURCE1;
    TARGET2 = SOURCE2;

    PUT SKIP(2) EDIT(SOURCE1,SOURCE2)(A,X,A);
    PUT SKIP EDIT(TARGET1,TARGET2)(F(8),X,B(5));

    END MAIN;
```

The output from this program is:

```
"14$15" "$"
"14$15" "$"
"11q00" "q"

14$15 11q00
   14015 11100
```

The target of the conversion is undefined when the CONVERSION condition is raised.

ON-Unit Completion

The retry attempted on a normal return is for the single field that was in error. Attempts to assign a string containing, for example, a comma list of values will not be used for successive data items in a GET statement.

The actual value modified by the ONSOURCE and ONCHAR pseudovariables is a temporary value that is discarded once the conversion is complete, or the control flow cannot return to the point of the error. This means that invalid data stored in a character string variable will cause the CONVERSION condition to be raised each time the value is converted, not just the first time the conversion is attempted, regardless of modifications to the ONSOURCE and ONCHAR pseudovariables inside the CONVERSION ON-unit.

8.10.4.5 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 whose 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) EOF = '1'B;
EOF = '0'B;
OPEN FILE (RECEIPTS) RECORD SEQUENTIAL;
READ FILE (RECEIPTS) INTO (RECORD);
DO WHILE (^EOF);
      .
      .
      .
   READ FILE (RECEIPTS) INTO (RECORD);
   END;
```

In this example, the ON statement establishes the default action to be taken when the last record in the input file has been processed: the flag EOF is set to '1'B.

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.

8.10.4.6 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. The maximum number of lines allowed on a single page is 32767. If not specified, PL/I determines a default page size using the formula LIB$LP_LINES minus 6.

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 the page size plus 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 large amount of output is written to a terminal, you can use the following ON-unit, that contains only the null statement:

```
ON ENDPAGE(SYSPRINT);
```

This is optional because PL/I ignores the ENDPAGE condition on SYSPRINT by default. You cannot catch the ENDPAGE(SYSPRINT) condition.

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 1 and the program continues execution of the PUT statement. 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. However, if a LINE option on a PUT statement specifies a line number that is less than that of the current line, a new page is output and the current line is set to 1.

Default PL/I Action

If the ENDPAGE condition is signaled during file processing, PL/I starts output on a new page and continues processing. An exception is made for SYSPRINT which is to take no action. 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.

8.10.4.7 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.

8.10.4.8 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 an interruption by an external CTRL key function or as a result of a call to the system procedure: SYS$EXIT or SYS$FORCEX (Force Exit)
- When the SIGNAL FINISH statement signals the condition

ON-Unit Completion

If a FINISH ON-unit that executes as a result of a SIGNAL FINISH statement does not execute a nonlocal GOTO statement, control returns to the statement following SIGNAL FINISH. If the FINISH ON-unit executes as a result of any of the other three causes listed above, the program terminates.

8.10.4.9 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 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 through 127 is assigned to a fixed-point binary variable with a precision of 7 bits and scale equal to zero. 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

Two hardware exceptions exist 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 specifically to either of these errors, use the VAXCONDITION condition name.

Example

To respond to a FIXEDOVERFLOW condition caused by either decimal or integer overflow, write an ON-unit as follows:

```
ON FIXEDOVERFLOW BEGIN;
   IF ONCODE() = SS$_DECOVF THEN DO;   /* Decimal overflow handling */
   END;

   IF ONCODE() - SS$_INTOVF THEN DO;   /* Fixed binary overflow handling */
   END;
END;  /* ON */
```

To respond to a decimal overflow only, write an ON-unit like the following:

```
ON VAXCONDITION (SS$_DECOVF) BEGIN;  /* Decimal overflow handling */
END;  /* ON */
```

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.

8.10.4.10 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 follow:

- The record indicated by the specified key cannot be found.
- The key specification requires 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. This error is shown in the Example section.

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 RMS condition value associated with the error.
- The ONKEY built-in function returns the key value that caused the condition to be signaled.

Example

The following example shows the key of a relative file exceeding the maximum record number specifed.

```
%INCLUDE $RMSDEF;

KEYTEST: PROCEDURE OPTIONS(MAIN);

DECLARE
  RECBUF CHAR(80),
  MYFILE FILE;

ON KEY(MYFILE) BEGIN;

  PUT SKIP LIST('Key condition raised');
  IF ONCODE()=RMS$_MRN THEN
    PUT SKIP LIST('You have exceeded the maximum record.');

  STOP;

END;

  OPEN FILE(MYFILE) TITLE('MYFILE.DAT') OUTPUT KEYED
                    ENVIRONMENT(FIXED_LENGTH_RECORDS,
                                MAXIMUM_RECORD_SIZE(80),
                                MAXIMUM_RECORD_NUMBER(20));

  RECBUF = 'This record will not ever make it into the file';

  WRITE FILE(MYFILE) FROM(RECBUF) KEYFROM(100);

END;
```

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.

8.10.4.11 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. That is, the precision is maximized and the exponent is minimized. The maximum precisions allowed are:

- OpenVMS VAX systems: binary floating-point value is 113; decimal floating-point value is 34.
- OpenVMS Alpha systems: binary floating-point value is 53; decimal floating-point value is 15.

PL/I signals the OVERFLOW condition when the result of an arithmetic operation on a floating-point value exceeds the maximum exponent size allowed by the hardware.

The value resulting from an operation that causes this condition is undefined.

ON-Unit Completion

Control returns to the point of the interruption.

8.10.4.12 STORAGE Condition Name

The STORAGE condition is raised when an error has been detected during allocation of a controlled variable or a based variable other than to an area. The ONCODE value is the error returned by LIB$GET_VM. The most common cause is the exhaustion of virtual memory; another cause might be an erroneous attempt to allocate a negative amount of storage.

8.10.4.13 STRINGRANGE Condition Name

The STRINGRANGE condition is raised when a substring reference is beyond the length of the string. The error is detected either by compiled code or by a run-time library routine.

STRINGRANGE can be abbreviated STRG.

Any one of several subconditions can cause the STRINGRANGE condition to be raised. You can use the ONCODE built-in function to determine which one. Following are the possible values of the ONCODE built-in function for the STRINGRANGE condition:

ONCODE value	Raised by
PLI$_STRRANGE	SIGNAL STRINGRANGE
PLI$_SUBSTR2	Out-of-range SUBSTR 2nd argument
PLI$_SUBSTR3	Out-of-range SUBSTR 3rd argument
PLI$_BIFSTAPOS	Out-of-range starting position for an INDEX, SEARCH, or VERIFY built-in function

Note that STRINGRANGE is always enabled in RTL code (which is currently used for more complex cases of INDEX, SEARCH, and VERIFY),

but in-line checking is only performed if /CHECK=BOUNDS is used to compile the code in which the condition would be raised.

An example of the use of the STRINGRANGE condition and the ONCODE built-in function follows.

```
%INCLUDE $PLIDEF;
ON STRINGRANGE BEGIN;
           /*
            * The THEN clause below will be executed for all
            * SUBSTR starting-position range errors. All other
            * STRINGRANGE errors will be resignaled. Note that
            * SUBSTR is processed in-line, so the code must be
            * compiled with /CHECK=BOUNDS for this ON-unit to
            * be effective.
            */
           IF ONCODE() = PLI$_SUBSTR2
           THEN
               .
               .
               .
           ELSE
               CALL RESIGNAL();
           END;
```

8.10.4.14 SUBSCRIPTRANGE Condition Name

The SUBSCRIPTRANGE condition is raised in response to out-of-bounds subscripts in references to arrays. The value returned by the ONCODE built-in function for the SUBSCRIPTRANGE condition is PLI$_SUBRANGE or PLI$_SUBRANGEn, where n is the number of the subscript, in the range 1 through 8.

8.10.4.15 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 / UNDF } (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. Following are some examples of errors that cause the UNDEFINEDFILE condition:

- 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 system-detected file 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 variable defined as FILE that was 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 I/O 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.

8.10.4.16 UNDERFLOW Condition Name (Kednos PL/I for OpenVMS VAX only)

On Kednos PL/I for OpenVMS VAX you can specify the UNDERFLOW condition name (which can be abbreviated UFL) in an ON, REVERT, or SIGNAL statement to designate a floating-point underflow condition or ON-unit. The Alpha hardware does not support UNDERFLOW detection; therefore, the underflow condition is not raised on OpenVMS Alpha systems.

Kednos PL/I for OpenVMS VAX 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 hardware.

ON-Unit Completion

On completion of the ON-unit, control is returned to the point of the interrupt. Continued execution is unpredictable.

This condition is signaled by Kednos PL/I for OpenVMS VAX only in procedures in which the UNDERFLOW option is enabled. The option is enabled when you specify UNDERFLOW in the procedure options. Kednos PL/I for OpenVMS Alpha ignores the UNDERFLOW option.

The value resulting from an operation that causes the UNDERFLOW condition is undefined. (The value would be set to zero only if UNDERFLOW were not specified in the procedure options.)

8.10.4.17 VAXCONDITION Condition Name

The VAXCONDITION condition name can be specified in an ON, RESIGNAL, REVERT, or SIGNAL statement. The VAXCONDITION condition name provides a way to signal and handle operating-system or programmer-specified 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 operating system routines. For details on using the VAXCONDITION condition name and the meanings of system- and user-defined values that you can specify, see the *Kednos PL/I for OpenVMS Systems User Manual*.

8.10.4.18 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.

8.10.5 Default PL/I ON-Unit

PL/I defines a default ON-unit for the procedure that is designated as the main procedure. This is why there must be exactly one procedure with OPTIONS(MAIN) specified in any executable image. This default ON-unit performs the following actions depending on the condition signaled. Note that the severity of the signal is determined by the low three bits of the condition code.

- 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. Note that ENDPAGE is ignored by default for SYSPRINT (see Section 8.10.4.6 for more information).
- 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 if it 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 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 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.

8.10.6 Establishment of ON-Units

An ON-unit is established for a specific ON condition or conditions following the execution of an ON statement that specifies the condition name(s). 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 descendant blocks, unless one of the following occurs:

- Another ON statement is specified for the same condition in a dynamically descendant block. The ON-unit established within the descendant block remains in effect as long as the descendant 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. (See Section 8.10.3).
- 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 reverted.
- For an ON-unit is established inside the ON-unit:

```
ON OVERFLOW BEGIN
   .
   .
   .
   ON OVERFLOW BEGIN
   .
   .
   .
   END;
END;
```

8.10.7 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	SELECT
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.

8.10.8 Search Path 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. This occurs unless you have used the SYSTEM option in an ON statement for the condition; the SYSTEM option causes the system default action to be executed regardless of the existence of any ON-unit.) PL/I first searches the current block, that is, the block in which the condition occurred. 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, default PL/I condition handling is performed.

8.10.9 Completion of ON-Units

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.

- When any ON-unit (except for ERROR or FINISH) completes normally, control returns 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.

9 Input and Output

PL/I provides two distinct types of I/O processing, each of which handles input and output data in a different manner, and each of which has a unique set of I/O statements. These types of I/O are:

- Stream (the GET and PUT statements)
- Record (the READ, WRITE, DELETE, and REWRITE statements)

When a file is read or written with stream I/O, the data is treated as if it formed 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 with record I/O, however, a single record is processed upon the execution of an I/O statement.

This chapter describes I/O concepts that apply to both stream and record I/O.

9.1 Opening and Closing Files

This section discusses the following:

- File declarations
- File variables
- Opening a file
- File description attributes and options
- File access modes
- Closing a file

9.1.1 File Declarations

A file declaration specifies an identifier, the FILE attribute, and one or more file description attributes that describe the type of I/O operation that will be used to process the file.

A file is denoted in an I/O statement by the FILE option as follows:

FILE(file-reference)

file-reference
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.

9.1.2 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 previous example, the OPEN statement opens the file INFILE.

A file variable can also be given a value by receiving a file constant or variable passed as an argument, or by receiving a file constant or variable 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.

9.1.3 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.

The OPEN statement explicitly opens one or more PL/I files with a specified set of attributes that describe the file and the method for accessing it. The format of the OPEN statement is as follows:

OPEN FILE(file-reference) [file-description-attribute . . .]
[,FILE(file-reference) [file-description-attribute . . .]] . . . ;

FILE(file-reference)

A reference to the file to be opened. If the file is already open, the OPEN statement has no effect. Therefore, if you want to change any attributes of an open file, you should first close it, and then reopen it with the new attributes.

file-description-attribute

The attributes and options 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 while the file is open.

The attributes you can specify with the OPEN statement are as follows:

DIRECT	PRINT
ENVIRONMENT(option, . . .)	RECORD
INPUT	SEQUENTIAL
KEYED	STREAM
OUTPUT	UPDATE

The attributes are described in Chapter 2.

The OPEN options are described in Section 9.1.3.1.

Examples

```
DECLARE INFILE FILE,
        STATE_FILE FILE KEYED;

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

The DECLARE and OPEN statements for INFILE do not specify any file description attributes; PL/I applies the default attributes STREAM and INPUT. If any statement other than GET is used to process this file, the ERROR condition is signaled.

The file STATE_FILE is declared with the KEYED attribute. With 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 can be used to operate on records in the file. The KEYED attribute implies the SEQUENTIAL attribute; thus, records in the file can be accessed sequentially or by key.

The second OPEN statement specifies the INPUT and SEQUENTIAL attributes. During this opening, the file can be accessed by sequential and keyed READ statements; REWRITE, DELETE, and WRITE statements cannot be used.

```
DECLARE COPYFILE FILE OUTPUT;
OPEN FILE(COPYFILE) TITLE('COPYFILE.DAT');
```

The file specified by the file constant COPYFILE is opened for output. Each time this program is run, it creates a new version of the file COPYFILE.DAT.

9.1.3.1 OPEN Statement Options

The options that you can use in the OPEN statement are:

LINESIZE Option

The LINESIZE option specifies the maximum number of characters that can be output on 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 to 32767, giving the number of characters per line. If the expression is outside this range, a run-time error occurs.

The value specified in the LINESIZE option is used as the output line length for all subsequent output operations on the stream file, and it overrides 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 determined by the device.
- If the output is to the default file, SYSPRINT, the default line size is 80.
- 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.

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 through 32767, giving the number of lines per page. If the expression is outside this range, a run-time error occurs.

The value specified in the PAGESIZE option is used as the output page length for all subsequent output operations on the print file, and overrides the system default page size. The default page size is the following:

- If the logical name SYS$LP_LINES is defined, the default page size is the numeric value of SYS$LP_LINES minus 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.

The PAGESIZE option is valid only for print files.

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 is specified only on the OPEN statement for a file. Its format is as follows:

```
TITLE(expression)
```

expression
A character-string expression which represents an external file specification for the file.

For details on how the file specification is determined see the Section 9.1.3.4.

9.1.3.2 Effects of Opening a File

Opening a file in PL/I has the following effects:

- Any permanent attributes specified in a DECLARE statement of a file constant 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 in the next section.
- 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. This process is decribed in Section 9.1.3.4.
- If the file already exists, it is located and its attributes are checked for compatibility with the attributes specified or implied by the OPEN statement.
- 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 the file's organization.
- If the file is opened successfully, 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 the *Kednos PL/I for OpenVMS Systems User Manual*.

9.1.3.3 Establishing the File's Attributes

The description attributes specified when a file is opened are merged with the file's permanent attributes. Duplicate specification of an attribute is allowed only for an attribute that does not specify a value.

An incomplete set of attributes is augmented with implied attributes. Table 9–1 summarizes the attributes that can be added to an incomplete set.

Table 9–1 File Description Attributes Implied when a File is Opened

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 following steps to complete the set:

1 If neither STREAM nor RECORD is present or implied, 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 or implied, 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.

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.

9.1.3.4 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 can be a logical name, or a portion of it can 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 10 translations.

2 After translation, PL/I applies any default values 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, which is the name of the file constant which declared the file.

The rules for logical name translation and for the application of system defaults are described in detail in the *Kednos PL/I for OpenVMS Systems User Manual*.

The maximum length of the expanded file specification is 128.

9.1.3.5 Accessing an Existing File

A file opening 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.

9.1.3.6 Creating a File

A file opening creates a new file if the following are all true:

- The OUTPUT attribute is specified.
- The TITLE option, after name translation and the application of system defaults, specifies a mass-storage device (such as a disk or a tape).
- The ENVIRONMENT(APPEND) option is not specified.

You can specify the organization and record format of a new file with 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 a maximum record size of 480 bytes and a maximum record number of -2147883647.

- If the PRINT attribute is present, PL/I creates a sequential file with variable-length records, maximum record length of 508, and a 2 byte fixed-control field used to store carriage-control information.
- If neither KEYED nor PRINT is specified, PL/I creates a sequential file with variable-length records and a maximum record size of 510 bytes.

When a file is opened with the RECORD and OUTPUT attributes, only WRITE statements can be used to access the file. If the file has the KEYED attribute as well, the WRITE statements must include the KEYFROM option.

9.1.3.7 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 Section 9.3.5. For a definition of file-positioning information for stream files, see Section 9.2.1.

9.1.4 File Description Attributes and Options

The file description attributes are summarized in Table 9–2. These attributes can be specified on DECLARE and OPEN statements.

Table 9–2 Summary of File Description Attributes

Attribute	Description
DIRECT	Records in the file will be accessed randomly.
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 with record I/O statements.
SEQUENTIAL	Records in the file will be accessed sequentially.
STREAM	The file will be accessed with stream I/O statements.
UPDATE	The file will be accessed for both reading and writing, and records can be rewritten and deleted.

For detailed descriptions of these attributes, see Chapter 2.

9.1.5 Closing a File

The CLOSE statement dissociates PL/I files from the physical files with which they were associated when opened. The format of the CLOSE statement is as follows:

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

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

FILE(file-reference)
A file to be closed. If the file is already closed, the CLOSE statement has no effect.

ENVIRONMENT(option, . . .)
One or more of the following ENVIRONMENT options, separated by commas:

BATCH

DELETE

REVISION_DATE

REWIND_ON_CLOSE

SPOOL

TRUNCATE

No other ENVIRONMENT options are valid. All ENVIRONMENT options are described in detail in the *Kednos PL/I for OpenVMS Systems User Manual*.

Examples

This CLOSE statement closes the file constant INFILE:

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

This CLOSE statement closes two files specified in a comma list, each with a different ENVIRONMENT option:

```
CLOSE FILE(A) ENV(DELETE), FILE(B) ENV(REVISION_DATE(X));
```

Another example of a CLOSE statement is:

```
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. The first OPEN statement that specifies this file is given the DIRECT and UPDATE attributes and opened for updating; the file can be accessed only by key.

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

9.2 Stream I/O

Stream I/O is one of the two general kinds of I/O performed by PL/I. Stream input and output is performed by the statements GET and PUT, respectively. Both statements can perform either list-directed or edit-directed operations.

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. In contrast, record I/O only processes one record of a file in each READ or WRITE statement.

Table 9–3 summarizes the file description attributes and access modes for stream files.

Table 9–3 Attributes and Access Modes for Stream Files

Attributes Specified	Attributes Implied	Valid Devices and File Organizations	Usage
PRINT	STREAM OUTPUT	Any output device and any 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 can fill part or all of one or more lines. Data conversion and alignment within lines can use the default processing provided by the PUT LIST form of the PUT statement or can be explicitly controlled by format specifications in the PUT EDIT form of the PUT statement. The output fields can be aligned to specific tab positions. The PAGESIZE and LINESIZE options can be specified to control the formatting of lines on pages. The ENDPAGE condition is signaled when the end-of-page is reached.
STREAM INPUT		Any input device and any file except indexed	Individual data items are read by GET statements. A single GET statement can process all or part of one or more lines or records. The format of an input field can be determined by the default processing provided by the GET LIST form of the GET statement or can be explicitly controlled by format specifications in the GET EDIT form of the GET statement.
STREAM OUTPUT		Any output device and any 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 can be read back in with GET LIST statements when the file is opened with the STREAM and INPUT attributes.

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 cannot 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 match 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, see the *Kednos PL/I for OpenVMS Systems User Manual*.

Successive PUT statements write their output to the same line or record until the line size is reached or until 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.

The next sections describe the following aspects of stream I/O:

- Processing and Positioning of Stream Files
- Input by the GET statements.
- Output by the PUT statements.
- Format items
- Processing and Positioning of Character Strings
- Terminal I/O

9.2.1 Processing and Positioning of Stream Files

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 Section 9.1 for LINESIZE option). The line size is used to determine when to skip to the next line. 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 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 do the following:

- 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 (the 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. 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.
- Use the control format items to advance to a specified line or character position. The control format items are COLUMN (move to a specified character position), SKIP (the 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. You can evaluate the current page number for a specified print file with the PAGENO built-in function. You can also set it 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 Section 9.1). 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 8-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 Section 9.2.6.)

9.2.2 Input by the 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 can be a file declared with the STREAM attribute or the default file SYSIN, usually associated with the user's default input device. See Section 9.2.6 for more information.

This section describes the syntax, options, and execution of GET statements.

9.2.2.1 Syntax Summary of the GET Statement

The GET statement has several forms; they are:

```
GET EDIT (input-target*, . . . ) (format-specification, . . . )
  [ FILE(file-reference)*                  ]
  [       [SKIP[(expression)]]*            ]
  [       [OPTIONS(option, . . . )]*       ];
  [ STRING(expression)*                    ]

GET LIST (input-target*, . . . )
  [ FILE(file-reference)*                  ]
  [       [SKIP[(expression)]]*            ]
  [       [OPTIONS(option, . . . )]*       ];
  [ STRING(expression)*                    ]

GET [FILE(file-reference)]* SKIP [(expression)] ;
```

*Options**
NO_ECHO
NO_FILTER
PROMPT(expression)
PURGE_TYPE_AHEAD

**Syntax elements common to two or more forms*

input-target

The names of one or more variables to be assigned values from the input stream. Multiple input targets must be separated by commas.

An input target has one of the following forms:

reference

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.

(input-target, . . . DO reference=expression [TO expression]
[BY expression] [WHILE(expression)] [UNTIL(expression)])

Another form is:

(input-target, . . . DO reference=expression
[REPEAT expression] [WHILE(expression)] [UNTIL(expression)])

The input target can be any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding the 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 Section 9.2.4.13.

format-specification
A list of format items to control the conversion of data items in the input list. You can use 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 Section 9.2.4 for format items.

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.

SKIP [(expression)]
An option that advances the input file a specified number of lines before processing the input list. This option can be used only with the implied or explicit FILE option. If the expression is specified, it indicates the number of lines to be advanced; 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, regardless of its position in the statement.

OPTIONS (option, . . .)
An option that specifies one or more of the following options. This option can be used only with the default or explicit FILE option; it cannot be used with the STRING option. Multiple options must be separated by commas.

CANCEL_CONTROL_O
NO_ECHO
NO_FILTER
PROMPT (string-expression)

PURGE_TYPE_AHEAD

The options are described fully in the *Kednos PL/I for OpenVMS Systems User Manual*.

STRING(expression)

An option specifying that the input stream is a character-string expression. The STRING option cannot be used with the FILE, OPTIONS, or SKIP option.

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 or apostrophes) that would be required if the character string were in an external file.

9.2.2.2 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 can 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 as follows:

```
GET EDIT (input-target, . . . ) (format-specification, . . . )
┌ FILE(file-reference)                  ┐
│       [SKIP[(expression)]]            │
│       [OPTIONS(option, . . . )]       │;
└ [STRING(expression)]                  ┘
```

The syntax is described in more detail in Section 9.2.2.1.

Examples

```
GET EDIT (FIRST,MID_INITIAL,LAST)
 (A(12),A(1),A(20));
```

This statement reads the next 12 characters from the default stream input file (SYSIN) and assigns the string to FIRST. It then reads the next character into MID_INITIAL, and then the next 20 characters into LAST.

```
GET EDIT (SOCIAL_SECURITY) (A(12))
 FILE (SOCIAL) SKIP (12);
```

This statement opens the stream file SOCIAL if the file was closed, 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. This reads the value of N from the input stream using the format item F(4). The process

then skips to the next line (record). It reads N elements into the array A, using the format item F(10,5) for each element.

```
GET EDIT (NAME.FIRST,NAME.LAST)
 (A(10),X(3),A(20))
 STRING('Philip    A. Rothberg            ');
```

This statement assigns <BIT_STRING>(Philip) to the structure member NAME.FIRST, skips the middle initial, period, and space, and assigns <BIT_STRING>(Rothberg) to NAME.LAST.

9.2.2.3 GET LIST

The GET LIST statement acquires character-string data from an input stream, which can be a stream file or a character-string expression. The stream file can be a declared file or the default file SYSIN. The acquired character strings are assigned to input targets named in the GET LIST statement, after 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 GET LIST does not require that data be aligned in specific columns, it is useful for acquiring input from a terminal.

The form of the GET LIST statement is as follows:

```
GET LIST (input-target, . . . )
[ FILE(file-reference)                ]
|       [SKIP[(expression)]]          |;
|       [OPTIONS(option, . . . )]     |
[ STRING(expression)                  ]
```

The syntax is described in more detail in Section 9.2.2.1.

Specifying 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 can 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 * 2 apostrophes; for instance, the word *isn't* would be specified as either:

```
isn't or '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, a comma, or a carriage return.

- 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 null character string ('') is assigned, with appropriate conversion, to the corresponding 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 Section 9.2.6 for examples. 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 CONVERSION 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',33,150.60,5.87,15000.50,4E-6,
```

then the program GETS writes the following output to OUTFILE.DAT:

```
Thomas R. Dooley 33 150.60  5.87  $15000.50  4.0000000E-06
```

In the input file (INFILE.DAT), the string <BIT_STRING>(Thomas R. Dooley) is surrounded by apostrophes so that the spaces between words will 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 Section 9.2.6.

9.2.2.4 GET SKIP

The GET SKIP statement positions the input file at the start of a new line. The format of this GET statement is as follows:

GET [FILE(file-reference)] SKIP [(expression)] ;

The syntax is described in more detail in Section 9.2.2.1.

9.2.2.5 Execution of the GET Statement

When a GET statement is executed, the first action is to evaluate the FILE option, if there is one. For example:

```
GET FILE(INFILE) LIST(A);
```

If INFILE references an open file, PL/I checks that the file has the INPUT and STREAM attributes.

If INFILE has not been opened, PL/I implicitly opens the file with the attributes INPUT and STREAM.

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 Section 9.2.6).

If the input stream is a file, the next action is to execute the SKIP option, if there is one. 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:

```
GET FILE(INFILE) SKIP(2);
```

This statement 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. Because 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. 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 continues at the next record.

In list-directed (GET LIST) statements, an input field is acquired by examining the input stream for the next character that is not a space character. The following actions are taken depending on the character found:

- If the next nonspace character is an apostrophe, the input field is assumed to contain a bit- or character-string constant, in the same format as that used to write a string constant in a program. The constant is acquired and can 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 PL/I conversion rules (see Section 6.4).
- 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 null character string (`''`) is assigned to the input target with appropriate type conversion. If the input file was opened with

ENVIRONMENT(IGNORE_LINE_MARKS), the carriage return or end-of-record is ignored.

- If 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, 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.

9.2.3 Output by the PUT Statement

The PUT statement transfers data from the program to the output stream. The output stream can be either a stream file or a character-string variable. The output file can be a declared file or the default file SYSPRINT.

This entry describes the syntax, options, and execution of PUT statements.

9.2.3.1 Syntax Summary of the PUT Statement

The PUT statement has several forms; they are:

```
PUT EDIT (output-source*, . . . ) (format-specification, . . . )
[ FILE(file-reference)*
      [PAGE]* [LINE(expression)]*
      [SKIP[(expression)]]*
      [OPTIONS(option)]*
  STRING(reference)* ];

PUT [FILE (file-reference)*] LINE (expression);

PUT LIST (output-source, . . . )*
[ FILE(file-reference)*
      [PAGE]* [LINE(expression)]*
      [SKIP[(expression)]]*
      [OPTIONS(option)]*
  STRING(reference)* ];

PUT [FILE(file-reference)*] PAGE;

PUT [FILE(file-reference)]* SKIP [(expression)] ;
```

**Syntax elements common to two or more forms*

output-source
A construct that specifies one or more expressions to be placed in the output stream. Multiple output sources must be separated by commas.

An output source has the following forms:

expression

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.

(output-source, . . . DO reference=expression
[TO expression][BY expression][WHILE(expression)][UNTIL(expression)])

Another form is:

(output-source, . . . DO reference=expression
[REPEAT expression][WHILE (expression)][UNTIL(expression)])

The output source can be 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 Section 9.2.4.13.

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 Section 9.2.4.13 and Section 9.2.4).

FILE(file-reference)
An option that specifies that the output stream be 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. SYSPRINT 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 can 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 Section 9.1.3). The PAGESIZE option can be used only with print files.

LINE (expression)

An option that advances the output file to a specified line. You can use the LINE option only with implied or explicit 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. You can use the SKIP option 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 1.

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, underscoring, and so forth. For further information on pages in stream files, see Section 9.2.6.

OPTIONS (CANCEL_CONTROL_O)

A statement option that can be included only with the implied or explicit FILE option. The option is described fully in the *Kednos PL/I for OpenVMS Systems User Manual*.

STRING(reference)

An option that specifies that the output stream be the referenced character-string variable. The STRING option cannot be used in the same statement with FILE, OPTIONS, PAGE, LINE, or SKIP.

9.2.3.2 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 as follows:

```
PUT EDIT (output-source, . . . ) (format-specification, . . . )
        [ FILE(file-reference)                    ]
        [     [PAGE] [LINE(expression)]           ]
        [      [SKIP[(expression)]]               ]
        [       [OPTIONS(option, . . . )]         ] ;
        [                                         ]
        [ STRING(reference)                       ]
```

The syntax is described in more detail in Section 9.2.3.1.

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
```

9.2.3.3 PUT LINE

The PUT LINE statement advances a print file to a specified line. Its format is as follows:

```
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.

The syntax is described in more detail in Section 9.2.3.1.

9.2.3.4 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 placed at the start of the next tab stop, where a tab stop is in column 1, 9, 17, and so on. Otherwise, the strings are separated by spaces.

With PUT LIST, the conversion of the output sources and formatting of the output data are automatic and follow the PL/I rules for conversion to character strings.

The form of the PUT LIST statement is as follows:

```
PUT LIST (output-source, . . . ) [ FILE(file-reference)
                                   [PAGE] [LINE(expression)]
                                   [SKIP[(expression)]]
                                   [OPTIONS(option, . . . )]

                                   STRING(reference)          ] ;
```

The syntax is described in more detail in Section 9.2.3.1.

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;
```

Assume that 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.

9.2.3.5 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;
```

The syntax is described in more detail in Section 9.2.3.1.

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).

9.2.3.6 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 as follows:

```
PUT [FILE(file-reference)] SKIP [(expression)];
```

The syntax is described in more detail in Section 9.2.3.1.

9.2.3.7 Execution of 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 and is not already open, the referenced file is either opened or created with the STREAM and OUTPUT attributes. The file is opened if it has the APPEND attribute; otherwise, it is created.

If neither the FILE option nor the 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:

```
PUT FILE(OUT) PAGE LINE(20);
```

This statement 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.

However, if 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. Because 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 PL/I rules for converting a computational value to a character string (see Section 6.4).

- If the PUT statement is edit directed, the output source is converted as specified by a corresponding format item. For details, see Section 9.2.4.
- If the output stream is a character-string variable or file with the attributes STREAM and 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 <BIT_STRING>(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 and 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 can be signaled if the output stream is a print file. For more information on print files, see Section 9.2.6.
- 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.
- If the output stream is a character-string variable, the output process is identical to that for 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.

9.2.4 Format Items

In PL/I, formatted input and output 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:

- *Data format items*: A, B, E, F, and P

- *Remote format item*: R
- *Control format items*: COLUMN, LINE, PAGE, SKIP, TAB, and X

The data format items refer to a field of characters in the stream. Each data format item specifies the field width 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 should they be followed by the radix factor 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 CONVERSION 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.

The rest of this section describes each format item in detail. The following subsections describe format-specification lists and the FORMAT statement.

9.2.4.1 A Format Item

The A format item (both uppercase and lowercase) describes the representation of a character string in the input or output stream. The form of the A format item is as follows:

```
A [(w)]
```

w

A nonnegative integer or an integer expression 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.

Input with GET EDIT

The value w must be included when the A format item is used with GET EDIT. If w has a positive value, a character-string value comprising the next w characters in the input stream is acquired and assigned to the input variable.

The acquired character string is converted, if necessary, to the data type of the input target, following the PL/I data conversion rules. Apostrophes should enclose the stream data only if 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 equals the length of the converted output source. If w is zero, the A format item and the associated output source are skipped.

Output strings are not surrounded automatically by apostrophes. The converted output source is truncated or appended with trailing spaces, according to the value of w. The conversion of a computational data item to a character string is performed following the PL/I data conversion rules (see Section 6.4).

The next tables 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 following table is a field of characters beginning at the current position in the stream and continuing to the right. The # character is used to signify a space. 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)	*##SHRUBBERY# . . .*	CHAR(10)	*##SHRUBBER*
A(6)	*##SHRUBBERY# . . .*	CHAR(10)	*##SHRU####*
A(6)	*##SHRUBBERY# . . .*	CHAR(10) VAR	*##SHRU*
A(10)	*##1.2345#### . . .*	DECIMAL(4,1)	*001.2*
A(5)	*##1.2345#### . . .*	DECIMAL(4,2)	*01.20*
A(6)	*##1.2345#### . . .*	DECIMAL(4,2)	*01.23*

Output Examples

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

Output Source Value	Format Item	Output Value
'STRING'	A(10)	*STRING####*
'STRING'	A	*STRING*
1.2345	A(2)	*##*
1.2345	A	*##1.2345*
-1.2345	A(4)	*#-1.*
-1.2345	A	*#-1.2345*
''	A(10)	*##########*
''	A	[no output]
0	A(3)	*###*
0	A	*###0*
-12345	A(6)	*##-123*
-12345	A	*##-12345*

9.2.4.2 B Format Items

The B format items B, B1, B2, B3, and B4 describe representations of bit strings in an input or output stream. Note that the B can be typed lowercase. The form of the B format items is as follows:

B[m] (w)

m

The integer 1, 2, 3, or 4, specifying the radix factor, that is, 2^m. 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 (binary) in the stream. When the radix factor is 2, the bit string is represented by the characters 0, 1, 2, and 3 (base 4). When the radix factor is 3, the bit string is represented by the characters 0, 1, 2, 3, 4, 5, 6, and 7 (octal). When the radix factor is 4, the bit string is represented by the characters 0 through 9 and A through F (hexadecimal).

w

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

The interpretation of the B format items on input and output is described below.

Input with GET EDIT

The value w must be included when the B format items are used with GET EDIT. 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 PL/I conversion rules (for details, see Section 6.4).

The string of characters in the stream can be preceded or followed by spaces, which are ignored. All characters in the input field (except any leading and trailing spaces) must be those implied by the radix factor; otherwise, a CONVERSION condition is signaled. Consequently, input strings should not be enclosed in apostrophes and should not include the suffix Bm.

Output with PUT EDIT

The output source is converted, if necessary, to a bit string, following the PL/I rules for converting data to bit strings (see Section 6.4). 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.

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 has 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;
```

This program produces the following 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, in effect multiplying the value of the string by 2. For B3, two extra bits were appended to make the string 33 bits long and thus divisible into an exact number of 3-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 the following example:

```
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 produces the following output:

```
Decimal:  5
Binary: 00000000000000000000101
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 following table is a field of characters beginning at the current position in the stream and continuing to the right. The # character is used to signify a space. 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 following 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*

Output Source Value	Format Item	Output Value
4095	B2	*333333*
4095	B3	*7777*
4095	B4	*FFF*

9.2.4.3 COLUMN Format item

The COLUMN format item sets a stream file to a specific character position within a line. In other words, 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; for information on how to refer to a relative position, see Section 9.2.4.12.

The form of the COLUMN format item is:

{ COLUMN } (w)
{ COL }

w

A nonnegative integer or expression that identifies the wth position from the beginning of the current line. The value of the converted expression must be zero or positive. If the value of the converted expression is zero, a value of 1 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.

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 1 is assumed. See Section 9.1.

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 following 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'
```

9.2.4.4 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 or expression that specifies the total width in characters of the field in the stream.

d

An optional nonnegative integer or expression that specifies the number of fractional digits in the stream representation and whose value is interpreted depending on GET EDIT and PUT EDIT statements as described below.

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 plus (+) or minus (-) 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 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 is 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 acceptable forms, an 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 as follows:

```
 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 is ordinarily a 2-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 7 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. (On OpenVMS VAX systems for values of type H-float, the value of w should be 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, an ERROR condition is signaled.

Examples

The next tables 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 this 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)	*-1.23343000000000E-10*

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. The # character is used to signify a space.

Output Source Value	Format Item	Output Value
-12234	E(11)	*-1.2234E+04*
-12234	E(11,2)	*##-1.22E+04*
12234	E(11)	*#1.2234E+04*
12234	E(11,2)	*###1.22E+04*
-12.234	E(11,1)	*###-1.2E+01*
-1.23456E3	E(12)	*-1.23456E+03*
-1.23456E3	E(12,2)	*###-1.23E+03*

9.2.4.5 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 or expression that specifies the total width in characters of the field in the stream.

d

An optional nonnegative integer or expression that specifies the number of fractional digits in the stream representation and whose value is interpreted depending on GET EDIT and PUT EDIT statements as described below.

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 converts it to an input target of any computational type. 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 plus (+) or minus (-) symbol.

The following are valid representations:

124333
-123333
-123.

A CONVERSION condition is signaled if the field is not blank and does not contain a valid representation; otherwise, the fixed-point decimal number is extracted from the field and is assigned to the input target, with any necessary conversions. A decimal point included in the number 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 value 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, which might include invalid syntax, are acquired.

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 zero appears to the left of the decimal point.

To account for negative values with fractional digits, the specified width integer should be 2 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, an 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 this 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 this table is either a constant or the value of a variable that is written out with the associated format item. The # character is used to specify a space.

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*

Output Source Value	Format Item	Output Value
'1000000000000000'B	F(5)	*32768*
'100000'B3	F(5)	*32768*
'ABCEDF'B4	F(10)	*##11259615*

9.2.4.6 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, or an expression, that specifies a line on the current page, where line 1 is the first line. The maximum value for a print file's line number is 32767. If a program generates a value in excess of 32767, a run-time error occurs.

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 Section 9.1)
- 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, the file is positioned at the beginning of a new page, and the page number is incremented by 1 (for instance, when w is zero).

9.2.4.7 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 (see Section 2.2.38. The field width is the total number of characters, exclusive of V, in the picture.

The interpretation of the P format item, for input and output, is given below.

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. If necessary, the value is converted to the type of the input target, following the usual rules (see Section 6.4). If the string is not valid, the CONVERSION condition is signaled.

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 PL/I conversion rules. 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 capable of being edited by the picture specified in the P format item. Otherwise, the ERROR condition is signaled.

Examples

The following tables 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 this table is a field of characters beginning at the current position in the stream and continuing to the right. The # character is used to specify a space. 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)	*[CONVERSION]*
P'SSSSV.SSSSS'	*+1001.2345 . . .*	DECIMAL(8,5)	*[CONVERSION]*

The last two cases signal the CONVERSION 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.

Note 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 this 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*

Output Source Value	Format Item	Output Value
-1.23456E3	P ′+ZZZ9V.99′	*#1234.56*

9.2.4.8 PAGE Format Item

The PAGE format item is used with print files to begin a new page. See Section 9.2.6 for more information on print files.

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.

9.2.4.9 R Format Item

The R (remote) 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.

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 or PUT EDIT statements, or both. 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. A label is required and is specified in the GET EDIT or PUT EDIT statement that contains an R format item 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.

Although the FORMAT statement can contain another R format item, the following restrictions apply:

- 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;
```

Assume the file RFRM.IN contains the following data:

```
 25005.50
Thomasina
A.
Delacroix
6 July 1976
 2 15003.65
```

The following output, with spacing as shown, will be written to the print file RFRM.OUT:

```
Delacroix, Thomasina A.:        Hired 6 July 1976 at    $15003.65
             2   years prior experience
Present salary:    $25005.50
```

9.2.4.10 SKIP Format Item

The SKIP format item sets a stream file to a new position relative to the current line. It is used with input and output files.

The form of the SKIP format item is:

SKIP [(w)]

w

An integer, or an expression, giving the number of lines to be skipped; the expression must not convert to a negative integer and must be greater than zero, except for print files. If w is omitted, a value of 1 is assumed.

If w is 1 or is omitted, the file is positioned at the beginning of the next line. If w is greater than 1, 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.

9.2.4.11 TAB Format Item

The TAB format item sets a print file to a specified tab stop. It is used only for output to print files. Within a line, tab stops always occur every eight columns, starting at column 1. The form of the TAB format item is:

```
TAB [(w)]
```

w

An integer, or an expression, that identifies the wth tab stop from the current position; w must not be negative. If w equals zero, no operation is performed. If w is omitted, a value of 1 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 is one. Otherwise, cc is the column in the current line at which the next output character would appear. For example, if seven characters have already been written on a line, then the cc is column 8; this is where the next output would occur. 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 Section 9.1 for LINESIZE option.
- 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 Section 9.1 for 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 <BIT_STRING>(COL1). The exclamation point appears in column 1 of the next line because there are 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 <BIT_STRING>(abcdefg), after which the current column is 17, a tab stop. Because 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.

9.2.4.12 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, or an expression, that specifies a number of consecutive character positions in the stream; w must not yield a negative integer value. If w yields zero, no operation is performed. If the w is omitted, its value is assumed to be 1.

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 Section 9.1 for 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 and receiving an answer to the prompt <BIT_STRING>(Line>). Assume that the interaction is as follows:

```
Line> beautiful losers Return
```

The following output will be written to OUTFILE.DAT:

```
beautiful  -FIRST WORD TYPED
```

The X format item has correctly inserted two spaces between <BIT_STRING>(beautiful) and <BIT_STRING>(-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. Assume that the interaction is:

```
Line> ABCDEFGHIJKLMNOPQRSTUVWXYZ Return
```

The following output will be written to OUTFILE2.DAT:

```
KLMNOPQRSTUVWXYZ
```

The first 10 letters (A to J) have been ignored on input.

9.2.4.13 Format Specifications

In the GET EDIT, PUT EDIT, and FORMAT statements, format items are used singly or in combination to create format specifications. The syntax of a format specification is as follows:

{ format-item
iteration-factor format-item
iteration-factor(format-specification, . . .) }

The iteration factor is an integer or an integer expression that repeats the following format item or the following list of format specifications. Expressions must be enclosed in parentheses. If an integer iteration factor precedes a single format item that is not in parentheses, the iteration factor and the format item must be separated by a space. For example:

```
PUT EDIT (A) (F(5,2));
```

This statement specifies a 5-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) (2F(5,2));
```

Or:

```
PUT EDIT (A,B) (2 (F(5,2)));
```

An iteration factor can also repeat an entire list of format specifications:

```
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, this specification looks like this:

```
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)
```

If an expression is used as the iteration factor, it must be enclosed in parentheses, but does not require spaces. For example:

```
PUT EDIT (A) ((Z*4)F(5,2));
```

In general, data listed in the GET EDIT or PUT EDIT statement is matched to the expanded list of data format items, from left to right, until the end of the input-target or output-source list is reached. Matching occurs only between I/O data and data format items; control format items are executed only if they are encountered while the matching is in progress.

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.

- 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 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 both 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.

All edit-directed input statements include the following syntax:

EDIT (input-target, . . .) (format-specification, . . .)

All edit-directed output statements include the following syntax:

EDIT (output-source, . . .) (format-specification, . . .)

format-specification
One of the following:

- A single control or data format item.
- A construct containing an iteration factor followed by one or more format items.
- An R format item, which specifies the label of a FORMAT statement. In effect, the entire format-specification list in the FORMAT statement is acquired and inserted at the position of the R format item.

Except for picture (P) and remote (R) format items, arguments to format items can be integer expressions.

input-target
One of the following:

- A variable reference, which can be to a scalar or aggregate variable of any computational data type
- A construct with the following syntax:

  (input-target, . . . DO reference=expression[TO expression] [BY expression] [WHILE(expression)][UNTIL(expression)])

- A construct with the following syntax:

  (input-target, . . . DO reference=expression[REPEAT expression] [WHILE (expression)][UNTIL(expression)])

output-source
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 or structure or contains a DO specification, it is expanded to form a list of individual data items; an array is expanded in row-major order, a structure is expanded in the order of its declaration, and items preceding a DO specification are expanded according to the DO specification.

Within a single target or source, items at the innermost 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, shown first, are with input targets that are scalar variable references. The next cases shown are with aggregate (array and structure) references. The last cases shown are with DO specifications.

Scalar Variables

The following examples have input targets that are scalar variables:

```
GET EDIT (A,B,C,D) (A(12),F(5,2),F(6,2),A(14));
```

This statement acquires four values from the input stream: a 12-character string, a 5-digit fixed-point decimal number, a 6-digit fixed-point decimal number, and a 14-character string, and assigns these values, with any necessary conversions, to the target variables A, B, C, and D, respectively.

```
GET EDIT (A,B,C,D) (A(12));
```

This statement acquires four 12-character strings and assigns 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));
```

This statement acquires a 12-character string, two fixed-point decimal numbers, and a 14-character string, in that order, and assigns them to A, B, C, and D. (You can use embedded spaces in format lists, as elsewhere, for clarity; the space between 2 and F(5,2) is required.)

```
GET EDIT (A,B,C,D,E) ( 2( A(12),A(14) ), A(20) );
```

This statement acquires, in order, a 12-character string, a 14-character string, another 12-character string, another 14-character string, and a 20-character string, and assigns 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) );
```

This statement performs the same operation as in the previous example, but acquires the 20-character string from the next line.

Aggregates

The following examples have input targets that are references to array and structure variables:

```
GET EDIT (A) ( 2( A(12),A(14) ), A(20) );
```

A is an array of five elements or a structure with five scalar members. This statement expands A to a list of individual data items. Then it acquires, in order, a 12-character string, a 14-character string, another 12-character string, another 14-character string, and a 20-character string, and assigns 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) );
```

Both A and B are aggregates with five elements or members. For A, this statement performs the same operation as in the previous example, and then repeats the operation for B, using the same format list each time.

Because 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));
```

NAME is a structure declared as:

```
DECLARE 1 NAME
          2 FIRST CHARACTER(20) VARYING,
          2 LAST  CHARACTER(80) VARYING;
```

This statement skips to the next line and acquires a 20-character string. It assigns the string to NAME.FIRST. This statement skips to the next line and acquires an 80-character string. This statement assigns that string to NAME.LAST.

```
GET EDIT (A,B) ( 2( A(12),A(14) ), SKIP, A(20) );
```

Both A and B are 4-element arrays. From the current line, this statement executes A(12), A(14), A(12), and A(14), in that order, and assigns the results to A(1) through A(4). This statement then skips to the next line and executes A(20), A(12), A(14), and A(12), in that order, and assigns the results to B(1) through B(4); the list of data items is now exhausted, so this statement does not execute SKIP a second time.

DO Specifications

The following examples have 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) );
```

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. This statement executes the format items A(12), A(14), A(12), and A(14), in that order, and assigns the resulting strings to elements B(10), B(8), B(6), and B(4), respectively. This statement then executes A(20) and assigns the result to B(1).

```
GET EDIT ( ( (A(I,J) DO J=1 TO 10) DO I=1 TO 20) )
 (F(5),F(6));
```

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 5-digit integers, and those with even-numbered columns, 6-digit integers. Because the DO specifications specify row-major order, the same operation is performed by the next example.

```
GET EDIT (A) (F(5),F(6));
```

Because row-major order is the default, nested DO specifications are used to change the order in which values are assigned.

The example has the same effect as the following DO-group:

```
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 identical effect is not generally true for all stream I/O statements. For instance:

GET SKIP EDIT(input-target, . . .) (format-specification, . . .);

This statement 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, before any other input processing is performed.

```
GET EDIT  ( ( CURRENT->PERSON.NAME
   DO CURRENT = FIRST
   REPEAT CURRENT->PERSON.NEXT
   WHILE (CURRENT ^= NULL) ) )
       (A(80));
```

CURRENT and FIRST are pointers, and PERSON is a based structure declared as:

```
DECLARE /* Based structure for list elements: */
        1 PERSON BASED,
          2 NEXT POINTER,   /* Pointer to next element: */
          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 string is assigned to FIRST->PERSON.NAME. On subsequent iterations of the DO specification, the pointer to the next element, 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 often used in this type of application. You should provide a WHILE or UNTIL clause in this or any DO REPEAT construct to be sure that the operation has a defined termination. However, the WHILE or UNTIL clause is not required.

9.2.5 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 either at 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 is 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 is used for positioning.

9.2.6 Terminal I/O

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. The stream input and output statements, GET and PUT, use the default PL/I files SYSIN (the terminal) and SYSPRINT, respectively, when no file reference is included in the statement.

This file 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 *Kednos PL/I for OpenVMS Systems User Manual*.

9.2.6.1 Simple Input from a Terminal

Simple input from a terminal is accomplished with the GET LIST statement, which in its simple form has the following format:

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 pauses until enough values are typed by the user to satisfy the input-target list.

The user must separate the values with the Return key, spaces, or commas. The user must press the Return key to send the typed line to the program.

In the context of simple terminal input, the input targets are usually simple variable references. For example:

```
GET LIST (SALARY,CONTRIBUTION(42),PAYROLL.DEDUCTION);
```

This statement gets 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 the following:

```
15500,500,1200 Return

15500 Return
500 Return
1200 Return

15500 500 Return
1200 Return
```

If you press Return in response to an input request from GET LIST, the null character string '' is assigned to the input target. If you press Return 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 causes an error for B formats.)

For full details on input targets, see the GET LIST statement (Section 9.2.2).

9.2.6.2 Simple Output to a Terminal

You can send data to a terminal with the PUT LIST statement. A simple form of PUT LIST is as:

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:

```
PUT LIST (A,B,C);
```

This statement 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. For example, on RISC ULTRIX, the ENDPAGE condition is signaled when the terminal's page size is exceeded.

9.2.6.3 Print File

A print file is a stream output file that is intended for output on a terminal, line printer, or other output device. You can declare any stream output file to be a print file by using the PRINT attribute. The default stream output file, SYSPRINT, is a print file.

The following list describes the special features of print files, as opposed to ordinary stream output files:

- 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 8-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.
- An internal record is kept 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. An internal record is kept of the number of lines per page. You can specify a page size when the print file is created (see Section 9.1 for 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 1. 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 *OpenVMS Record Management Services Reference Manual*.)
- Print files usually cannot be read properly with GET LIST or GET EDIT.

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;
```

PL/I does not display 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:');
```

The cursor remains on the same line after the prompt is displayed, so the input can be entered on the same line. The completed line might be as:

```
Enter badge number, social security number: 7,116-40-0482 Return
```

The GET statement also has a PROMPT statement option that displays a prompt on the user's terminal. See the *Kednos PL/I for OpenVMS Systems User Manual.*

```
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;
```

```
PUT SKIP LIST('Enter string, integer, float>');
GET LIST(STRING,I,A);
DO WHILE(^EOF); /* stop when CTRL/Z is typed */

      PUT SKIP LIST(STRING,I,A);
      PUT SKIP LIST('Enter string, integer, float>');
      GET 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 user answers the prompt with a Ctrl/Z, which signals the ENDFILE condition for SYSIN; the program then terminates. The following is a sample dialog with the program:

```
$ RUN TIN Return

Enter string, integer, float> JONES,27,3.75 Return
JONES              27   3.7500000E+00
Enter string, integer, float> JONES 27 3.75 Return
JONES              27   3.7500000E+00
Enter string, integer, float> JONES Return
27 Return
3.75 Return
JONES              27   3.7500000E+00
Enter string, integer, float> DOOLEY Return
 Return
4E-6 Return
DOOLEY              0   4.0000000E-06
Enter string, integer, float>  Ctrl/Z
$
```

Notice that input fields are separated by commas, spaces, or the RETURN key. Notice also that entering a blank line after <BIT_STRING>(DOOLEY) causes the program to set the value of I to zero.

Other Topics

The following topics are of interest in terminal I/O applications:

- For using GET STRING and certain built-in functions for string handling, see the GET statement (Section 9.2.2).
- For using GET EDIT and PUT EDIT to control the format of input or output data, see the GET and PUT statements (Section 9.2.2 and Section 9.2.3.1).
- For using PUT SKIP, PUT LINE, and PUT PAGE to create formatted displays, see the PUT statement (Section 9.2.3.1).
- For using the OPTIONS keyword with GET and PUT to override default operations, see the *Kednos PL/I for OpenVMS Systems User Manual*.

9.3 Record I/O

Record I/O 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.) Table 9–4 summarizes the file description attributes that apply to record I/O.

Table 9–4 Attributes and Access Modes for OpenVMS Record Files

Attributes Specified	Attributes Implied	Valid Devices and File Organizations	Usage
SEQUENTIAL OUTPUT	RECORD	Any output device or file except indexed	Records can be added to the end of the file with 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 with READ statements. Each statement reads a single record.
SEQUENTIAL UPDATE	RECORD	Relative, indexed, stream	READ statements read a file's records in order. PL/I maintains the current record, which is the record just read. This record can be replaced in a REWRITE statement. In a relative or indexed sequential file, the current record can also be deleted with a DELETE statement. Each statement processes a single record.
DIRECT OUTPUT[1]	KEYED RECORD	Relative, stream	WRITE statements insert records into the file at positions specified by keys. Each statement inserts a single record.
DIRECT INPUT	KEYED RECORD	Relative, indexed, stream	READ statements specify records to be read randomly by key. Each statement reads a single record.
DIRECT UPDATE	KEYED RECORD	Relative, indexed, stream	READ, WRITE, and REWRITE statements specify records randomly by key. In a relative or indexed file, records can also be deleted by key.
KEYED SEQUENTIAL OUTPUT[1]	RECORD	Relative, stream	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, stream	READ statements access records in the file randomly by key or sequentially.
KEYED SEQUENTIAL UPDATE	RECORD	Relative, indexed, stream	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]You cannot create indexed sequential files directly from a PL/I OPEN statement.

9.3.1 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.

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(file-reference)

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 INPUT. The implied attributes are merged with the attributes specified in the file's declaration.

INTO (variable-reference)

An option that specifies that the contents of the record are to be assigned to the specified variable. The variable must be an addressable variable. The INTO and SET options are mutually exclusive.

- 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 larger than the variable, it is truncated and the ERROR condition is signaled.
- If the variable has the AREA attribute and the file does not have the attribute ENVIRONMENT(SCALARVARYING), the entire record is treated as an area value and assigned to the variable. If the extent of the area in the record is larger than the variable, the AREA condition is signaled and the target area is unmodified.
- For any other type of variable, the record is 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 that the specified pointer variable should be assigned the value of the location of the buffer in storage. The SET and INTO options are mutually exclusive.

This buffer remains allocated until the next operation on the file but no longer. Therefore, do not use either the pointer value or the buffer 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, you can rewrite the record 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 KEY and KEYTO options are mutually exclusive.

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 value 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. The KEYTO and KEY options are mutually exclusive.

KEYTO is specified only for a file that has both the KEYED and SEQUENTIAL attributes.

OPTIONS (option, . . .)

An option that specifies one or more of the following READ statement options, separated by commas:

FIXED_CONTROL_TO (variable-reference)

INDEX_NUMBER (expression)

LOCK_ON_READ

LOCK_ON_WRITE

MANUAL_UNLOCKING

MATCH_GREATER

MATCH_GREATER_EQUAL

MATCH_NEXT

MATCH_NEXT_EQUAL

NOLOCK

NONEXISTENT_RECORD

READ_REGARDLESS

RECORD_ID (variable-reference)

RECORD_ID_TO (variable-reference)

TIMEOUT_PERIOD (variable-reference)

WAIT_FOR_RECORD

All these options except INDEX_NUMBER remain in effect for the current statement only.

These options are described fully in the *Kednos PL/I for OpenVMS Systems User Manual*.

9.3.1.1 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 operation, 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, the end-of-file.

If any error other than an incorrect record size occurs, the current record becomes undefined and the next record is the same as it was before the read operation was attempted.

Examples

```
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 program COPY reads a file with variable-length records into a character string with the VARYING attribute and writes the records to a new output file.

It 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 be taken when the end-of-file occurs: it sets the bit ENDED to <BIT_STRING>(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.

```
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 ENVIRONMENT(INDEXED);
    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',
         STATE.SYMBOLS,STATE.SYMBOLS.FLOWER);
```

This example shows the use of a keyed READ statement to access a record in an indexed sequential file. 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.

```
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,
              EMPLOYEE.NAME.FIRST,EMPLOYEE.NAME.LAST,
              EMPLOYEE.NAME.MIDDLE_INIT);
         READ FILE(EMP_FILE) SET(EP) KEYTO(NUMBER);
         END;
      CLOSE FILE(EMP_FILE);

END;
```

This program accesses a relative file sequentially with READ statements and obtains the key value of each record, that is, the relative record number. 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. The 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.

9.3.2 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 (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.

FROM (variable-reference)

A reference to the variable containing data for the output record. The variable must be addressable.

If the variable has the VARYING or the AREA attribute and the file does not have the attribute ENVIRONMENT(SCALARVARYING), the WRITE statement writes only the current value of the varying string or the area 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 the 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 copies the key value specified into the correct key field position (or positions, if segmented keys are used). PL/I also sets the key number to the primary index.

  The value of the specified expression is converted to the data type of the key. 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 following WRITE statement options, separated by commas:

FIXED_CONTROL_FROM (variable-reference)

RECORD_ID_TO (variable-reference)

These options are described fully in the *Kednos PL/I for OpenVMS Systems User Manual*.

9.3.2.1 File Positioning Following a WRITE Statement

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 the end-of-file.

Examples

```
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));

      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;
```

This program 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.

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.

```
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 (EMPLOYEE.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;
```

This procedure 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.

The file is opened with the DIRECT and UPDATE attributes, because records to be written will be chosen by 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 designate 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.

9.3.3 DELETE Statement

The DELETE statement deletes a record from a file. This record can be the current record, the record specified by the KEY option, or the record specified by the RECORD_ID option. The file must have the UPDATE attribute.

The format of the DELETE statement is:

```
DELETE FILE(file-reference) [KEY (expression)] [OPTIONS(option, . . . )];
```

FILE(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 will be located by the key specified in the expression. The file must have the KEYED attribute.

The nature of the key depends on the file's organization, as follows:

- If it is a relative file, the key is a fixed binary value indicating the relative record number of the record to be deleted.
- If it is an indexed sequential file, the key specifies a value 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 following DELETE statement options:

FAST_DELETE

INDEX_NUMBER (expression)

MATCH_GREATER

MATCH_GREATER_EQUAL

MATCH_NEXT

MATCH_NEXT_EQUAL

RECORD_ID (expression)

Multiple options must be separated by commas.

All these options except INDEX_NUMBER remain in effect for the current statement only.

These options are described fully in the *Kednos PL/I for OpenVMS Systems User Manual.*

9.3.3.1 File Positioning Following a DELETE Statement

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') ENVIRONMENT(INDEXED);
      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.

9.3.4 REWRITE Statement

The REWRITE statement replaces a record in a file. The record is 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(file-reference)

The file that 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.

FROM (variable-reference)

An option giving the variable that contains the data needed to rewrite the record. The variable must be an addressable variable.

If the FROM option is not specified, there must be a currently allocated buffer from an immediately preceding READ statement that specifies 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. Note that if the file organization is sequential, the record being rewritten must be the same length as the one read.

If the variable has the VARYING or the AREA attribute and the file does not have the attribute ENVIRONMENT(SCALARVARYING), the REWRITE statement writes only the current value of the varying string or area 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 value 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 following REWRITE statement options. Multiple options must be separated by commas.

FIXED_CONTROL_FROM (variable-reference)

INDEX_NUMBER (expression)

MATCH_GREATER

MATCH_GREATER_EQUAL

MATCH_NEXT

MATCH_NEXT_EQUAL

RECORD_ID (expression)

RECORD_ID_TO (variable-reference)

All these options except INDEX_NUMBER remain in effect for the current statement only.

These options are described fully in the *Kednos PL/I for OpenVMS Systems User Manual*.

9.3.4.1 File Positioning Following a REWRITE Statement

The next record position is set to denote the record immediately following the record that was rewritten or, if there is no following record, the 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).

9.3.5 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 the 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, the end-of-file.

After a sequential read operation, 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 described in the following chart. (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.)

Statement	Current Record	Next Record
READ	X	X+1
WRITE	X	X+1
REWRITE	X	X+1
DELETE	undefined	X+1

10 Preprocessor

The PL/I preprocessor permits you to alter a source program at compile time. Preprocessor statements can be mixed with nonpreprocessor statements in the source program, but preprocessor statements are executed only at compile time. Any resulting source program changes are then used for further compilation.

The preprocessor is embedded in the compiler, and so is also called the *embedded preprocessor*.

During compilation, the preprocessor performs two types of preprocessing:

- It interprets preprocessor statements, including preprocessor expression evaluation.
- It replaces the value of preprocessor variables and procedures.

Preprocessor statements allow you to include text from alternative sources (INCLUDE libraries, directories, and the VAX Common Data Dictionary), control the course of compilation (%DO, %GOTO, %IF, and %PROCEDURE), issue user-generated diagnostic messages, and selectively control listings and formats. The preprocessor statements are summarized in Table 10–1.

This chapter describes the preprocessor statements and functions.

10.1 Preprocessor Compilation Control

At compile time, preprocessor variables, procedures, and variable expressions are evaluated in the order that they appear in the source text, and the new values are substituted in the source program in the same order. Thus, the course of compilation becomes conditional, and the resulting executable program can have a variety of features. Note that preprocessor variables must be declared and activated before replacement occurs.

For example:

```
EXAMPLE: PROCEDURE OPTIONS(MAIN);

%DECLARE HOUR FIXED;
%HOUR = SUBSTR(TIME(),1,2);

%IF HOUR > 7 & HOUR < 18
%THEN
%FATAL 'Please compile this outside of prime time';

%DECLARE T CHARACTER;
%ACTIVATE T NORESCAN;

%T='''Compiled on '||DATE()||''''';

DECLARE INIT_MESSAGE CHARACTER(60) VARYING INITIAL(T);
```

```
%IF VARIANT() = '' | VARIANT() = 'NORMAL'
%THEN
  %INFORM 'NORMAL';
%ELSE %DO;
  %IF VARIANT() = 'SPECIAL'
    %THEN
      %INFORM 'SPECIAL';
    %ELSE
      %IF VARIANT() = 'NONE'
      %THEN %;
      %ELSE
         %DO;
         %T='''unknown variant '||variant()||'''';
         %WARN t;
         INIT_MESSAGE=INIT_MESSAGE||' with '||T;
         %END;
%END;

PUT SKIP LIST (INIT_MESSAGE);

END EXAMPLE;
```

This example illustrates several aspects of the preprocessor. First, the programmer specified that this program must be compiled outside of prime time. Second, the VARIANT preprocessor built-in function is used to determine which variant was specified on the command line using the /VARIANT switch. Third, user-generated preprocessor messages remind the programmer which value was given to VARIANT.

Notice the number of apostrophes around the string constant assigned to T. Apostrophes are sufficient if the value of T is used only in a preprocessor user-generated diagnostic message; the value of T is concatenated with nonpreprocessor text and assigned to INIT_MESSAGE. During preprocessing, apostrophes are stripped off string constants. In order to ensure that the run-time program also has apostrophes around the string, additional apostrophes are needed.

10.2 Preprocessor Statements

All preprocessor statements are preceded by a percent sign (%) and terminated by a semicolon (;). All text that appears within these delimiters is considered part of the preprocessor statement and is executed at compile time. For example:

```
%DECLARE HOUR FIXED;   /* declaration of a preprocessor
                                        single variable */

%DECLARE (A,B) CHARACTER; /* a factored preprocessor
                                           declaration  */

%HOUR = SUBSTR(TIME(),1,2);  /* preprocessor assignment
                               statement using two built-in
                                             functions  */

%STATE: PROCEDURE (X) RETURNS (BIT); /* preprocessor
                                              procedure */
```

Notice that a percent sign is required only at the beginning of the statement. The percent sign alerts the compiler that until the line is terminated with a semicolon, all subsequent text is preprocessor text. Therefore, no other percent signs are required on the line. However, when

you include Common Data Dictionary record definitions, you may need to include the usual PL/I punctuation.

Labels (preceded by a percent sign) are permitted on preprocessor statements and required on preprocessor procedures. As with other labels, preprocessor labels are used as the target of program control statements.

A preprocessor label must be an unsubscripted label constant. The format for a preprocessor label is:

%label: preprocessor-statement;

Table 10–1 summarizes the preprocessor statements. Each statement is then described in detail in an individual subsection.

Table 10–1 Summary of PL/I Preprocessor Statements

Statement	Use
%Assignment	Evaluates a preprocessor expression and gives its value to a preprocessor identifier
%;	Null statement, specifies no preprocessor operation
%ACTIVATE	Makes the value of declared preprocessor variables eligible for replacement
%DEACTIVATE	Makes the value of declared preprocessor variables ineligible for replacement
%DECLARE	Defines the preprocessor variable names and identifiers to be used in a PL/I program and specifies the data attributes associated with them
%DICTIONARY	Specifies data definitions to be included from the VAX Common Data Dictionary (CDD)
%DO	Denotes the beginning of a group of preprocessor statements to be executed as a unit
%END	Denotes the end of a block or group of statements that started with a %PROCEDURE or a %DO statement
%ERROR	Generates a user-defined error diagnostic message
%FATAL	Generates a user-defined fatal diagnostic message
%GOTO	Transfers control to a labeled preprocessor statement
%IF	Tests a preprocessor expression and establishes action to be performed based on the results
%INCLUDE	Copies the text of an external file into the source file at compile time
%INFORM	Generates a user-defined informational diagnostic message
%[NO]LIST	Same as %[NO]LIST_ALL
%[NO]LIST_ALL	Does or does not include CDD records, INCLUDE files, machine code, and source statements in the listing from that point on
%[NO]LIST_DICTIONARY	Does or does not include CDD records in the listing from that point on

Table 10–1 (Cont.) Summary of PL/I Preprocessor Statements

Statement	Use
%[NO]LIST_INCLUDE	Does or does not include INCLUDE files in the listing from that point on
%[NO]LIST_MACHINE	Does or does not include machine code in the listing from that point on
%[NO]LIST_SOURCE	Does or does not include source code in the listing from that point on
%PAGE	Provides listing pagination without form feeds in the source text
%PROCEDURE	Begins a preprocessor procedure
%REPLACE	Assigns a constant value to an identifier at compile time
%RETURN	Returns a value from execution of a preprocessor procedure to the point of invocation
%SBTTL	Allows specification of a listing subtitle line
%TITLE	Allows specification of a listing title line
%WARN	Generates a user-defined warning diagnostic message

10.2.1 %Assignment Statement

The preprocessor assignment statement gives a value to a specified preprocessor variable.

The format of the assignment statement is:

```
%target = expression;
```

target
The name of the preprocessor variable to be assigned a value. It must be an unsubscripted reference to a preprocessor variable.

expression
Any valid preprocessor expression.

For arithmetic operations, only decimal integer arithmetic of precision (10,0) is performed. Each operand and all results are converted, if necessary, to a fixed decimal value of precision (10,0). Fractional digits are truncated.

10.2.2 %Null

The %Null statement performs no action.

The format of the %Null statement is:

```
%;
```

The most common use for the %Null statement is as the target statement of a %THEN or %ELSE clause in an %IF statement. For example:

```
%IF
   ERROR() > 0;
%THEN
   %GOTO FIXIT;
%ELSE
   %;
```

10.2.3 %ACTIVATE

The %ACTIVATE statement makes preprocessor variable and procedure identifiers eligible for replacement. If the compiler encounters the named identifier after executing a %ACTIVATE statement, it initiates variable replacement.

The format of the %ACTIVATE statement is:

% { ACTIVATE / ACT } element [RESCAN / NORESCAN] , . . . ;

element

The name of a previously declared preprocessor identifier and/or a list of identifiers, where the identifiers are separated by commas and the list is enclosed in parentheses.

RESCAN or NORESCAN

Specifies that the preprocessor is to continue or discontinue checking the text for secondary value replacement.

The RESCAN option specifies that preprocessor scanning continue until all possible identifier replacements are completed. RESCAN is the default option.

The NORESCAN option specifies that replacement be done only once; the resulting text is not rescanned for possible further replacement.

An identifier is activated by either a %ACTIVATE statement or a %DECLARE statement. When an activated identifier is encountered by the compiler, in unquoted nonpreprocessor statements, the variable name or procedure reference is replaced by its value. Replacement continues throughout the rest of the source program unless replacement is stopped with the %DEACTIVATE statement.

You can activate several variables with a single statement. For example:

```
%DECLARE (A,B,C) FIXED;
%ACTIVATE (A,B), C NORESCAN;
```

Because RESCAN is the default action, this statement activates A and B with the RESCAN option. C is activated, but is not rescanned.

If an identifier that is not a preprocessor variable or procedure is the target of a %ACTIVATE statement, a warning message is issued and the identifier is implicitly declared as a preprocessor variable with the FIXED attribute. Thereafter, the identifier variable is eligible for replacement when activated.

For example:

```
DECLARE (A,B,C) FIXED;
%DECLARE (A,B) FIXED;
%ACTIVATE (A,B);

%A = 1;
%B = (A + A);

C = A + B;
PUT SKIP LIST (C);    /* C = 3 */
```

In this example, the activated preprocessor variables A and B are assigned values by the preprocessor. Notice that variables A and B are also declared as nonpreprocessor variables; this establishes them as variables within the nonpreprocessor program.

In the following example, the variable B is deactivated by the %DEACTIVATE statement.

```
   .
   .
   .
%DEACTIVATE B;
B = 900;
C = A + B;
PUT SKIP LIST (C);    /* C = 901 */
END;
```

Because the preprocessor variable B is deactivated, the preprocessor assignment statement %B = (A + A) is not in effect and the value of B is taken from the run-time assignment of B = 900. However, the value of A remains 1.

10.2.4 %DEACTIVATE

The %DEACTIVATE statement makes preprocessor variable and procedure identifiers ineligible for replacement. After a variable or procedure has been deactivated, it will not be replaced during preprocessing. Replacement of a deactivated variable or procedure occurs again only after it is reactivated with the %ACTIVATE statement.

The format of the %DEACTIVATE statement is:

% { DEACTIVATE / DEACT } element, . . . ;

element

The name of a preprocessor identifier, or a list of identifiers that is enclosed in parentheses. Deactivated elements must have been previously declared preprocessor variables.

For example:

```
TESTF:PROCEDURE OPTIONS (MAIN);
      DECLARE Y FIXED DECIMAL;
      Y = 10;                  /* initial value: Y = 10 */
      %DECLARE Y FIXED;
      %Y = 3;                  /* replacement value: Y = 3 */
      PUT SKIP LIST(Y);        /* output: Y = 3 */
      %DEACTIVATE Y;
      PUT SKIP LIST(Y);        /* output: Y = 10 */
      END;
```

In this example, %Y, when activated, replaces all the occurrences of the variable Y by the value assigned to %Y, until %Y is deactivated by the %DEACTIVATE statement. The identifier %Y is implicitly activated when it is declared as a preprocessor identifier.

It is possible to deactivate several variables with a single statement. For example:

```
%DEACTIVATE (A,B,C,D,E,F);
```

10.2.5 %DECLARE

The %DECLARE statement establishes an identifier as a preprocessor variable, specifies the data type of the variable, and activates the identifier for replacement. %DECLARE can occur anywhere in a PL/I source program.

The format of the %DECLARE statement is:

$$\% \left\{ \begin{matrix} \text{DECLARE} \\ \text{DCL} \end{matrix} \right\} \text{element} \left[\begin{matrix} \text{FIXED} \\ \text{CHARACTER} \\ \text{BIT} \end{matrix} \right] , \ldots ;$$

element

The name of a preprocessor identifier or a list of identifiers, which are separated by commas and enclosed in parentheses. You can give elements the attribute BIT, FIXED, or CHARACTER, but you cannot specify precision or length. The compiler supplies the variables with the following implied attributes:

Specified Attribute	Implied Attributes
BIT	(31) INITIAL ((31)'0'B)
FIXED	DECIMAL (10,0) INITIAL (0)
CHARACTER	VARYING (32500) INITIAL (' ')

If no data type is specified, FIXED is assumed.

When a variable is declared in a preprocessor statement, it is activated for replacement and rescanning. The scope of a preprocessor variable is all of the text following the declaration of the variable, unless the variable is declared inside a preprocessor procedure. Using %DECLARE inside of a preprocessor procedure has the effect of declaring a local variable.

For example:

```
%DECLARE HOUR FIXED;
```

In this example, HOUR is declared as a preprocessor variable identifier with the FIXED attribute. The compiler supplies the default values that make this declaration the equivalent of the following:

```
DECLARE HOUR FIXED DECIMAL (10,0) INITIAL (0);
```

Note: In preprocessor declarations, the attribute FIXED implies FIXED DECIMAL. In nonpreprocessor declarations, FIXED implies FIXED BINARY.

Factored declarations are permitted and follow the same usage rules as nonpreprocessor declarations. For example:

```
%DECLARE (A,B) CHARACTER, C BIT;
```

Both A and B are declared with the CHARACTER attribute. The compiler supplies default values that make this declaration the equivalent of the following:

```
%DECLARE (A,B) CHARACTER VARYING(32500) INITIAL(''),
         C BIT(31)INITIAL((31)'0'B);
```

10.2.6 %DICTIONARY

The %DICTIONARY statement causes VAX Common Data Dictionary (CDD) data definitions to be incorporated into the current PL/I source file during compilation. The statement can occur anywhere in a PL/I source file. For information on using PL/I with CDD, see Appendix E in the *Kednos PL/I for OpenVMS Systems User Manual*.

The format of the %DICTIONARY statement is:

%DICTIONARY cdd-path;

cdd-path

Any preprocessor expression. It is evaluated and converted to a CHARACTER string if necessary. The resulting character string is interpreted as the full or relative path name of a CDD object. The resultant path name must conform to all rules for forming VAX CDD path names. See the *Common Data Dictionary Utilities Manual* for details.

For example, assume that you have a record with the following path name:

```
CDD$TOP.SALES.JONES.SALARY
```

You can then specify a relative path name as follows:

```
%DICTIONARY 'SALARY';
```

Or you can specify an absolute path name as follows:

```
%DICTIONARY '_CDD$TOP.SALES.JONES.SALARY';
```

The compiler extracts the record definition from the CDD and inserts the PL/I structure declaration corresponding to the record description in the PL/I program.

If the %DICTIONARY statement is not embedded in a PL/I language statement, then the resulting structure is declared with the logical level 1 and the BASED storage attribute is furnished. The logical member levels are incremented from 2. For example:

```
DECLARE PRICE FIXED BINARY(31);
%DICTIONARY 'ACCOUNTS';
```

This would result in a declaration of the following form:

```
DECLARE PRICE FIXED BINARY(31);
DECLARE 1 ACCOUNTS BASED,
          2 NUMBER,
            3 LEDGER CHARACTER(3),
            3 SUBACCOUNT CHARACTER(5),
          2 DATE CHARACTER(12),
             .
             .
             .
```

Notice that in the above example, ACCOUNTS is a relative dictionary path name.

If the %DICTIONARY statement is embedded in a PL/I language statement, as in a structure declaration, then the resulting structure is declared with no logical level and no storage attribute. Logical member numbers are supplied and incremented from 100. For example:

```
DECLARE 1 COMMON_INTERFACES STATIC EXTERNAL,
          %DICTIONARY 'ACCOUNTS';   ,
          %DICTIONARY 'ADDRESSES';  ;
```

Notice the syntax in the above %DICTIONARY example. The %DICTIONARY statement is terminated with the preprocessor terminator semicolon before the normal PL/I line punctuation. The normal PL/I punctuation must also be included so that the final structure declaration will contain proper PL/I punctuation. The previous declaration would result in a declaration of the following form:

```
DECLARE 1 COMMON_INTERFACES STATIC EXTERNAL,
          100 ACCOUNTS,
              101 NUMBER,
                  102 LEDGER CHARACTER(3),
                  102 SUBACCOUNT CHARACTER (5),
              101 DATE CHARACTER(12),
                 .
                 .
                 .
          100 ADDRESSES,
             .
             .
             .
```

The CDD supports data types that are not native to PL/I. If a data definition contains an unsupported data type, PL/I makes the unsupported data type accessible by declaring it as data type BIT_FIELD or data type BYTE_FIELD. PL/I does not attempt to approximate a data type that is not supported by PL/I. For example, an F_FLOATING_COMPLEX number is declared BYTE_FIELD(8), not (2)FLOAT(24).

Note, however, that use of these two data types is limited. Data declared with the BIT_FIELD or BYTE_FIELD data type can be manipulated only with the PL/I built-in functions ADDR, INT, POSINT, SIZE, and UNSPEC. A variable declared with either of these data types can be passed as a parameter provided the parameter is declared as ANY. Thus, references to data declared as BIT_FIELD or BYTE_FIELD are limited to contexts in which the interpretation of a data type is not applied to the reference.

PL/I ignores CDD features that are not supported by PL/I, but issues error messages when the features conflict with PL/I.

When you extract a record definition from the CDD, you can choose to include this translated record in the program listing by using the /LIST/SHOW=DICTIONARY qualifiers in the PLI command line. Even if you choose not to list the extracted record, the names, data types, and offsets of the CDD record definition are displayed in the program listing allocation map.

CDD data definitions can contain explanatory text in the CDDL DESCRIPTION IS clause. This text is included in the PL/I listing comments, if /LIST/SHOW=DICTIONARY is specified. For example, you could use CDDL comments to indicate the data type of each structure and member. The punctuation for CDDL comments is the same as for other PL/I programs: the slash-asterisk (/*) and the asterisk-slash (*/).

10.2.7 %DO

The %DO statement begins a preprocessor DO-group, a sequence of statements terminating with the %END statement. The preprocessor DO-group must be a simple DO-group and is noniterative, but it can be usefully combined with an %IF statement.

The format of the %DO statement is:

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

You can include both preprocessor and nonpreprocessor text in a preprocessor DO-group. For example:

```
%DECLARE T CHARACTER;          /* declare T */
%ACTIVATE T NORESCAN;          /* activate T for replacement */
   .
   .
   .
%IF VARIANT() = 'NONE';
   %THEN
   %DO;
   %T = '''unknown variant'''; /* assign string to T */
   %WARN T;                    /* output unknown variant
                                  warning at compile time */
   INIT_MESSAGE = INIT_MESSAGE||' with '||T; /* assign
                               value of T to nonpreprocessor
                                    variable */
   %END;
```

This preprocessor DO-group performs several steps. First, a string constant is assigned to T. Then the value of T is used in a preprocessor user-generated diagnostic message. This message is issued at compile time to warn the programmer that the program is compiled with an unknown variant. Finally, the value of T is concatenated with a nonpreprocessor string constant. INIT_MESSAGE, including the value of T, is part of the run-time image.

10.2.8 %END

The %END statement terminates a preprocessor procedure or DO-group.

The format of the %END statement is:

%END;

Preprocessing then continues with the next executable preprocessor statement.

10.2.9 %ERROR

The %ERROR statement provides a diagnostic error message during program compilation.

The format of the %ERROR statement is:

%ERROR preprocessor-expression;

preprocessor-expression
A maximum of 64 characters giving the text of the error message to be displayed. Messages of more than 64 characters are truncated.

Returned Message

The message displayed by %ERROR is:

```
%PLIG-E-USERDIAG, preprocessor-expression
```

Compilation errors that result in the display of the %ERROR statement increment the informational diagnostic count displayed in the compilation summary, and inhibit production of an object file.

10.2.10 %FATAL

The %FATAL statement provides a diagnostic fatal message during program compilation.

The format of the %FATAL statement is:

%FATAL preprocessor-expression;

preprocessor-expression
The text of the fatal message you want displayed. The text is a character string with a maximum length of 64 characters. It is truncated if necessary.

Returned Message

The message displayed by %FATAL is:

```
%PLIG-F-USERDIAG, preprocessor-expression
```

Compilation errors that result in a fatal error terminate compilation after the message is displayed.

10.2.11 %GOTO

The %GOTO statement causes the preprocessor to interrupt its sequential processing of source text and continue processing at the point specified in the %GOTO statement. A %GOTO is useful for avoiding large segments of text in the source program.

The format of the %GOTO statement is:

%GOTO label-reference;

label-reference
A label of a preprocessor statement. The label reference determines the point to which the compiler processing will be transferred.

Nonlocal %GOTOs are not permitted. In other words, if a %GOTO is used within a preprocessor procedure, control must not be passed out of the containing procedure. Also, a %GOTO must not transfer control into a preprocessor procedure.

The following example illustrates transfers (forward and backward) and the use of %GOTO:

```
%TEXT:PROCEDURE RETURNS(CHARACTER);
          .
          .
          .
%CHANG_TEXT: DO;
    .
    .
    .
%IF WARN() = 5
    %THEN
         %GOTO CHANG_TEXT;
```

```
    %ELSE
         %GOTO INSERT_TEXT;
              .
              .
              .
%INSERT_TEXT: DO;
    .
    .
    .
%END;
```

Depending on the status of the %IF statement in this example, program compilation takes one of two courses. Control is transferred either forward to the statement labeled INSERT_TEXT or backward to the statement labeled CHANG_TEXT. The compiled program will then include one of the two blocks, but not both. Notice also in this example that the preprocessor built-in function WARN is used to determine preprocessor action, which makes the program self-diagnostic.

If program text is not compiled because the %GOTO statement transferred control over it, the compiler still checks the basic syntax of all statements. Therefore, comment delimiters and parentheses must balance, apostrophes must be paired correctly, and all statements must end with a semicolon.

10.2.12 %IF

The %IF statement controls the flow of program compilation according to the scalar bit value of a preprocessor expression. The %IF statement tests the preprocessor expression and performs the 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 preprocessor expression that yields a scalar bit value. If any bit of the value is 1, then the expression is true; otherwise, the expression is false.

action

A single, unlabeled preprocessor statement, %DO-group, %GOTO statement, or a preprocessor null statement. The specified action must not be an %END statement.

The %IF statement evaluates the preprocessor test expression. If the expression is true, the action specified following the keyword %THEN is compiled. Otherwise, the action, if any, following the %ELSE keyword is compiled. In either case, compilation resumes at the first executable statement following the termination of the %IF statement, unless a %GOTO in one of the action clauses causes compilation to resume elsewhere.

If an action is not compiled because the alternative action was compiled instead, the compiler still checks the basic syntax of all statements. Therefore, comment delimiters and parentheses must balance, apostrophes must be paired correctly, and all statements must end with a semicolon.

10.2.13 %INCLUDE

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 part of a procedure.

The format of the %INCLUDE statement is:

```
%INCLUDE { 'file-spec'                      };
         { module-name                      }
         { 'library-name(module-name)'      }
```

file-spec
A file specification enclosed in apostrophes interpreted as the complete file specification or path, including the device, directory, and file name. For the specification is subject to logical name translation and the application of default values by the OpenVMS operating system.

module-name
The 1- to 31-character name of a text module in a library of included files and/or other text modules.

library-name
The library containing the specified text module. Enclose the library and module in apostrophes. If you do not specify the library in the %INCLUDE statement, and if the text module is not in PLI$STARLET or in the text library pointed to by PLI$LIBRARY, you must specify the name of the library containing the module in the PLI compilation command.

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 in the command that compiles this program, or the logical name PLI$LIBRARY can point to the library that contains it.

```
%INCLUDE 'PROJECT_LIBRARY(MY_MODULE)';
```

This statement includes module MY_MODULE from the text library PROJECT_LIBRARY.TLB in the default directory.

Restrictions

%INCLUDE statements can be nested up to a maximum of four levels.

10.2.14 %INFORM

The %INFORM statement specifies a user-written diagnostic informational message to be displayed during program compilation.

The format of the %INFORM statement is:

%INFORM preprocessor-expression;

preprocessor-expression
The text of the informational message displayed. The text is a character string of up to 64 characters. The string is truncated if necessary.

Returned Message

The format of the message to be displayed by %INFORM is:

```
%PLIG-I-USERDIAG, preprocessor-expression
```

The %INFORM statement increments the informational diagnostic count displayed in the compilation summary.

10.2.15 %LIST_xxx

The %LIST_xxx statement (where xxx is either ALL, DICTIONARY, INCLUDE, MACHINE, or SOURCE) enables the selective listing display of INCLUDE file contents, extracted Common Data Dictionary (CDD) record descriptions, machine code, and source code. The %LIST_xxx statement has a number of forms, each of which enables listing control for specific portions of the source text.

The format of the %LIST_xxx statements are:

{ %LIST_ALL;
  %LIST_DICTIONARY;
  %LIST_INCLUDE;
  %LIST_MACHINE;
  %LIST_SOURCE; }

You must compile the program with the appropriate value specified for the /SHOW qualifier before the above statements can be effective.

The %LIST_xxx form of each statement enables the appearance of the specified information starting with the listing line following the %LIST_ xxx statement. If you previously specified %NOLIST_xxx, the %LIST_xxx statement has the effect of reenabling the display.

The text displayed with each form of the %LIST_xxx statement is summarized as follows:

- %LIST_ALL displays all of the following information. You can shorten this statement to %LIST.
- %LIST_DICTIONARY displays the PL/I translation of an included Common Data Dictionary record.

- %LIST_INCLUDE displays the contents of INCLUDE files and modules in the program listing.
- %LIST_MACHINE displays the machine code generated during compilation. For PL/I for RISC ULTRIX, see the *-S* option in the *pli(1)* manual page.
- %LIST_SOURCE displays source program statements in the program listing.

To disable a %LIST_xxx statement, specify %NOLIST_xxx at the appropriate line in the source text.

%LIST_xxx statements cannot be nested.

10.2.16 %NOLIST_xxx

The %NOLIST_xxx statement (where xxx is either ALL, DICTIONARY, INCLUDE, MACHINE, or SOURCE) disables the selective listing display of INCLUDE file contents, extracted Common Data Dictionary (CDD) record descriptions, machine code, and source code. The %NOLIST_xxx statement has a number of forms; each disables listing control for specific portions of the source text.

The format of the %NOLIST_xxx statements are:

```
{ %NOLIST_ALL;        }
{ %NOLIST_DICTIONARY; }
{ %NOLIST_INCLUDE;    }
{ %NOLIST_MACHINE;    }
{ %NOLIST_SOURCE;     }
```

You must compile the program with the /SHOW qualifier before any of these statements can be effective.

The %NOLIST_xxx form of each statement disables the appearance of the specified information starting with the listing line following the %NOLIST_xxx statement. If you previously specified %LIST_xxx, the %NOLIST_xxx statement has the effect of disabling the display.

The following summarizes the text suppressed with each form of the %NOLIST_xxx statement:

- %NOLIST_ALL does not display any of the following information. You can shorten this statement to %NOLIST.
- %NOLIST_DICTIONARY does not display the PL/I translation of an included Common Data Dictionary record.
- %NOLIST_INCLUDE does not display the contents of INCLUDE files and modules in the program listing.
- %NOLIST_MACHINE does not display the machine code generated during compilation.
- %NOLIST_SOURCE does not display source program statements in the program listing.

To cancel the effect of any of the %NOLIST_xxx statements, specify %LIST_xxx (see Section 10.2.15) at the appropriate line in the source text.

10.2.17 %PAGE

The %PAGE statement provides listing pagination without inserting form-feed characters into the source text.

The format of the %PAGE statement is:

```
%PAGE;
```

The first source record following the record that contains the %PAGE statement is printed on the first line of the next page of the source listing.

10.2.18 %PROCEDURE

A preprocessor procedure is a sequence of preprocessor statements headed by a %PROCEDURE statement and terminated by a %END statement. A preprocessor procedure executes only at compile time. Invocation is similar to a function reference and occurs in two ways:

- Preprocessor statements can invoke preprocessor procedures. In addition, preprocessor statements from within preprocessor procedures can invoke other preprocessor procedures.
- Statements from the source program can invoke preprocessor procedures.

The format of the %PROCEDURE statement is:

```
%label:PROCEDURE [(parameter-identifier, . . . )]
[STATEMENT]
RETURNS ( { CHARACTER } );
          { FIXED     }
          { BIT       }
    .
    .
    .

[%]RETURN (preprocessor-expression);
    .
    .
    .

[%]END;
```

label

An unsubscripted label constant. A preprocessor procedure is invoked by the appearance of the label name on the %PROCEDURE statement and terminated by the corresponding %END statement. The label name must be active if invoked from a nonpreprocessor statement.

Preprocessor label names can be activated and deactivated, but cannot be specified in a %DECLARE statement.

parameter-identifier

The name of a preprocessor identifier. Each identifier is a parameter of the procedure.

STATEMENT

A preprocessor procedure option. The STATEMENT option permits the use of a keyword argument list followed by an optional positional argument list in the preprocessor procedure invocation. The STATEMENT option returns strings that can be used as PL/I statements at run time.

RETURNS

A preprocessor procedure attribute. The RETURNS attribute defines the data type to be returned to the point of invocation in the source code. If you specify a data type that is inconsistent with the returned value, a conversion error may result.

preprocessor-expression

Value to be returned to the invoking source code. The preprocessor expression must be specified. The preprocessor expression is converted to the data type specified in the RETURNS option and is returned to the point of invocation. Therefore, the expression must be capable of being converted to CHARACTER(32500), FIXED(10), or BIT(31).

The %PROCEDURE statement defines the beginning of a preprocessor procedure block and specifies the parameters, if any, of the procedure. Because the preprocessor procedure is always invoked as a function, the %PROCEDURE statement must also specify (via the RETURNS option) the data type attributes of the value that is returned to the point of invocation.

For example:

```
%A_VAR = A_PROC();
```

In this statement, the preprocessor procedure A_PROC is invoked and evaluated, and the result is returned and assigned to the preprocessor variable A_VAR.

As with other PL/I procedures, a parenthesized parameter list specifies the parameters that the preprocessor procedure expects when it is invoked. Each preprocessor parameter specifies the name of a variable declared in the preprocessor procedure. The preprocessor parameters must correspond one-to-one with arguments specified for the preprocessor procedure when it is invoked, except when the STATEMENT option is used.

The value to be returned to the invoking source code is converted to the data type specified in the RETURNS option. The return value replaces the preprocessor procedure reference in the invoking source code. Preprocessor procedures cannot return values through their parameter list. The return value must be capable of being converted to one of the data types CHARACTER, FIXED, or BIT. The maximum precision of the value returned by the %RETURNS statement is BIT(31), CHARACTER(32500), or FIXED(10).

Preprocessor procedures cannot be nested. The scope of a preprocessor procedure is the procedure itself; that is, variables, labels, and any %GOTO statements used inside of the procedure must be local.

A preprocessor procedure is invoked by the appearance of its entry-name and list of arguments. If the reference occurs in a nonpreprocessor statement, the entry name must be active before the preprocessor procedure is invoked. If the entry name is activated with the RESCAN option, the value of the preprocessor procedure is rescanned for further possible preprocessor variable replacement and procedure invocation. You can invoke preprocessor procedures recursively.

When a preprocessor procedure (with or without the STATEMENT option) is invoked from a preprocessor statement, each argument is treated as an expression and the result of executing the preprocessor procedure is returned to the statement containing the invocation.

When a preprocessor procedure is invoked from nonpreprocessor source text, the arguments are interpreted as character strings and are delimited by the appearance of a comma or a right parenthesis occuring outside of balanced parentheses. For example, the positional argument list (Q(E,D), XYZ) has two arguments; the strings 'Q(E,D)' and 'XYZ'.

Examples

```
%A1: PROCEDURE RETURNS(FIXED);
     DECLARE (A,B,C) FIXED;
        A = 2;
        B = 10;
        C = A + B;
        RETURN(C);
     END;
```

This example declares the preprocessor procedure A1 and specifies that the procedure return a fixed decimal result after the preprocessor statements within the procedure have been executed.

The procedure returns the value 12 to the point of invocation. Note that the leading percent signs, normally associated with preprocessor statements, are not required within a preprocessor procedure.

```
PPFIB: PROCEDURE OPTIONS (MAIN);
       DECLARE Y CHAR(14) INITIAL('Fibonacci Test'); ❶
       %DECLARE Y FIXED; ❷
       %F: PROCEDURE(X) RETURNS (FIXED); ❸
           DECLARE X FIXED;
           IF (X <= 1)
              THEN RETURN(1);
              ELSE RETURN(F(X-1)+F(X-2));
           END;          /* End preprocessor procedure */
       %Y = F(10); ❹
       PUT SKIP LIST(Y);
       %Y = F(11); ❺
       PUT SKIP LIST(Y);
       %Y = F(12); ❻
       PUT SKIP LIST(Y);
       %DEACTIVATE Y; ❼
       PUT SKIP LIST(Y); ❽
       END;              /* End run-time procedure */
```

This example uses a preprocessor procedure to return a Fibonacci number. The recursive preprocessor procedure labeled %F is invoked to return a single value, a Fibonacci number, to the point of invocation. The following notes correspond to the example:

❶ The run-time variable Y is declared with the CHARACTER attribute and initialized to Fibonacci Test.

❷ The preprocessor variable Y is declared with the FIXED attribute, which implies FIXED DECIMAL (10,0). This declaration automatically activates the preprocessor variable Y.

❸ The preprocessor procedure F is defined. The percent sign for the END statement is optional in a preprocessor procedure.

Note that this procedure is recursive.

❹ The preprocessor procedure is called, passed the value 10, and the 10th number in the Fibonacci series is calculated. The resulting value is assigned to the preprocessor variable Y.

Because the preprocessor variable Y is active by default, the compiler replaces the occurrence of Y in the PUT statement with the new preprocessor Y value.

❺ Step 4 is repeated for the value 11.

❻ Step 4 is repeated for the value 12.

❼ The preprocessor variable Y is deactivated. No more scanning or replacement occurs. The preprocessor variable Y retains its final replacement value, 233.

❽ The run-time value of Y (Fibonacci Test) is output.

The output from this program is:

```
89
144
233
Fibonacci Test
```

Using the STATEMENT Option

All preprocessor procedures (with or without the STATEMENT option) return a value to the invoking source code; that is, they are function procedures. Through the use of the STATEMENT option, the argument list to a preprocessor procedure can be a keyword argument list. Keyword argument lists are unique to preprocessor procedures and provide a powerful tool for manipulating PL/I.

A keyword argument list ends with a semicolon rather than the right parenthesis. In this way, the STATEMENT option permits you to use a preprocessor procedure as if it were a statement. Consequently, preprocessor procedures using the STATEMENT option permit you to extend the PL/I language by simulating features that may not otherwise be available.

Preprocessor procedures can have one of two distinctly different types of argument lists: positional or keyword. Positional argument lists (ending with a right parenthesis) use parameters sequentially, as in a parenthesized list. You can use positional argument lists in any preprocessor procedure. Keyword argument lists (ending with a semicolon) use parameters in any order, as long as each keyword matches the name of a parameter. This permits the option of specifying the order in which parameters are passed. You can use keyword argument lists only when the preprocessor procedure contains the STATEMENT option and is invoked from a nonpreprocessor statement.

When a preprocessor procedure is invoked from a nonpreprocessor statement, the STATEMENT option permits the use of a keyword argument list that follows the optional positional argument list in a preprocessor procedure invocation.

When you use keyword arguments in nonpreprocessor statements, the keywords can be used in any order. The following reference examples would produce a variety of results with positional arguments, because values would be used sequentially. Keyword arguments produce consistent results because keyword parameters are matched with keyword arguments.

```
%B: PROCEDURE (ALPHA, BETA, GAMMA) STATEMENT . . .  ;
    DECLARE (ALPHA, BETA, GAMMA) FIXED;
       .
       .
       .
    END;

B(1,2,3);
B ALPHA(1) GAMMA(3) BETA(2);
B(1) GAMMA(3) BETA(2);
B (,2,3) ALPHA(1);
```

The next example shows a more common use of the STATEMENT option; to generate PL/I source statements that define a unique run time feature. The preprocessor procedure APPEND returns a string, which is incorporated into the source program at compile time. At run time, the returned string is used as a PL/I function.

This preprocessor procedure permits a varying string to accumulate text up to its maximum size without danger of undetected truncation. Normally, strings that exceed their maximum size are truncated. The text returned by the preprocessor procedure provides the run-time program with a way to handle truncation. If the string would be truncated, a message is printed and the FINISH condition is signaled.

```
%APPEND: PROCEDURE (string,to) STATEMENT RETURNS(CHARACTER); ❶

    %DECLARE (string,to) CHARACTER; ❷
    %RETURN (
        'DO;'|| ❸
        'IF LENGTH('||string||')+LENGTH('||to||') > SIZE('||to||')-2'||
        ' THEN DO;'||
           'PUT SKIP LIST (''Buffer overflowed appending to '||to||''');'||
           'SIGNAL FINISH;'|| ❹
           'END;'||
        'ELSE '||to||' = '||to||'||'||string||';'||
        'END;'
             );
     %END;
```

The following notes are keyed to this example:

❶ The preprocessor procedure APPEND is defined with the parameters 'string' and 'to' and the STATEMENT option.

❷ 'String' and 'to' are declared as parameters within the preprocessor procedure.

❸ The %RETURN statement returns the value contained by the parentheses. This text then becomes part of the PL/I nonpreprocessor source program.

Notice the punctuation within the character string returned by %RETURN. At compile time, single quotes are stripped when the text is incorporated into the run-time PL/I program. In addition, the semicolon that delimits the invocation is not retained when the replacement takes place. All customary PL/I punctuation must be included in the character string.

❹ If the current varying string and the additional string together are greater than the maximum length of the varying string, an informational message is printed and the FINISH condition is signaled.

The following invocations of the preprocessor procedure APPEND are all equivalent:

```
APPEND STRING('New String') TO (My_string);
APPEND TO(My_string) STRING('New String');
APPEND('New String') TO(My_string);
```

Notice that if you have a preprocessor procedure (A) with a label that is the same as the name of a keyword argument in another preprocessor procedure (B) with the STATEMENT option, then when B is invoked the keyword argument is treated as a call to procedure A, and not as a keyword parameter in B.

10.2.19 %REPLACE Statement

The preprocessor %REPLACE statement specifies that an identifier is a constant of a given value. It can 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. PL/I keywords are not valid identifiers in a %REPLACE statement. The identifier must not be the name of a declared preprocessor or program variable. PL/I permits multiple %REPLACE statements and %REPLACE statements that redefine the %REPLACE identifier.

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.

10.2.20 %RETURN Statement

The %RETURN statement terminates execution of the current preprocessor procedure.

The format of the %RETURN statement is:

[%]RETURN (preprocessor-expression);

preprocessor-expression
Value to be returned to the invoking procedure. The preprocessor expression must be specified. The preprocessor expression is converted to the data type specified in the RETURNS option, and the value of the expression is returned to the point of invocation. Therefore, the expression must be capable of being converted to CHARACTER (32500) VARYING, FIXED (10), or BIT (31).

The value returned by a preprocessor procedure cannot contain preprocessor statements.

When the value of the evaluated preprocessor expression is passed back to the point of invocation, control returns to the evaluation of the statement that contained the reference to the preprocessor procedure.

Within a preprocessor procedure, the leading percent sign (%) is optional.

Multiple %RETURN statements are permitted in preprocessor procedures. See Section 10.2.18 for examples of %RETURN.

10.2.21 %SBTTL

The %SBTTL statement allows specification of an arbitrary compile-time string for the listing subtitle line. PL/I uses the procedure IDENT, or 01 if no IDENT was specified. If %SBTTL is used, the specified subtitle appears to the right of IDENT or 01. Subtitles do not appear on the first page of the listing file.

The format of the %SBTTL statement is:

%SBTTL preprocessor-expression

preprocessor-expression
A character string with a maximum length of 30 characters. It is truncated if necessary.

10.2.22 %TITLE

The %TITLE statement allows specification of an arbitrary compile-time string for the listing title line. If %TITLE is used, the specified title appears to the right of the customary title. (If no TITLE option is specified, PL/I uses the name of the first level-1 procedure in the source program as the title.)

The format of the %TITLE statement is:

%TITLE preprocessor-expression

preprocessor-expression
A character string with a maximum length of 30 characters. It will be truncated if necessary.

10.2.23 %WARN

The %WARN statement provides a diagnostic warning message during program compilation.

The format of the %WARN statement is:

%WARN preprocessor-expression;

preprocessor-expression
The text of the warning message to be displayed. The text is a character string with a maximum length of 60 characters. It is truncated if necessary.

Returned Message

The message displayed by %WARN is:

```
%PLIG-W-USERDIAG, preprocessor-expression
```

The %WARN statement increments the warning diagnostic count displayed in the compilation summary.

10.3 User-Generated Diagnostic Messages

The PL/I embedded preprocessor provides four statements that permit user-generated diagnostic capability: %INFORM, %WARN, %ERROR, and %FATAL. Preprocessor diagnostic messages are compile-time messages, but you define the circumstances that invoke the message and the text displayed.

The action of each statement is to generate a diagnostic message of the appropriate severity level, with the preprocessor expression as the text of the message.

Examples

The first example shows how %INFORM can be used to return the value of VARIANT.

```
%IF VARIANT() = '' | VARIANT() = 'NORMAL'
%THEN
      %INFORM 'NORMAL';
```

In this example, the %INFORM diagnostic message is used to let the programmer know that compilation is continuing according to a *normal* plan.

If the value of VARIANT is not specified at compile time or if the value is 'NORMAL', then the following informational message is issued:

```
%PLIG-I-USERDIAG, NORMAL
```

In the following example, an unknown variant is included at compile time. The %WARN statement issues a compile-time warning diagnostic message and saves the message so that when the program is run, the appropriate text is output by the program.

```
DECLARE INIT_MESSAGE CHAR(40) VARYING INITIAL (T);
   .
   .
   .
%IF VARIANT() = 'NONE';
%THEN %;
%ELSE
      %DO;
      %T = '''unknown variant''';
      %WARN T;
      INIT_MESSAGE = 'Compiled with '||T;
      %END;

PUT SKIP LIST (INIT_MESSAGE);
```

The preprocessor built-in functions INFORM, WARN, and ERROR return the number of user-generated diagnostics issued at any specified point during compilation. Therefore, you can use user-generated diagnostics to control the course of compilation. For example:

```
%IF WARN() > 5
%THEN
     %GOTO change_text;
%ELSE;
```

This example specifies that compilation take a different course if there are more than five warning messages at that point in program compilation. If there are fewer than five warnings, then compilation proceeds along the current path.

```
%IF ERROR() >= 1
%THEN
     %FATAL 'Ending Compilation';
```

This example stops compilation if there is an error that would inhibit the production of an object file.

User-generated diagnostic messages increment the count displayed in the diagnostic summary.

10.4 Preprocessor Built-In Functions

A number of PL/I built-in functions are available for use at compile time. Since preprocessor built-in functions work the same way as run-time PL/I built-in functions, refer to Chapter 11 for detailed descriptions of the functions.

The built-in functions are summarized in Table 10–2 according to the following functional categories:

- Arithmetic built-in functions provide information about the properties of arithmetic values, or perform common arithmetic calculations.
- String-handling built-in functions process character-string and bit-string values.
- Conversion built-in functions convert data from one data type to another.
- Timekeeping built-in functions return the system date and time of day.
- Miscellaneous built-in functions are specifically preprocessor built-in functions.

Table 10–2 Summary of PL/I Preprocessor Built-In Functions

Category	Function Reference	Value Returned
Arithmetic	ABS(x)	Absolute value of 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 modulo y
	SIGN(x)	-1, 0, or 1 to indicate the sign of x
String-Handling	COPY(s,c)	c copies of specified string s
	INDEX(s,c[,p])	Position of the character string c within the string s, starting at position p

Table 10–2 (Cont.) Summary of PL/I Preprocessor Built-In Functions

Category	Function Reference	Value Returned
	LENGTH(s)	Number of characters or bits in the string s
	LTRIM(s,[b])	Removes blanks from left of string s; or if b is supplied, removes string b from left of string s
	REVERSE(s)	Reverse of the source character string or bit string
	RTRIM(s,[e])	Removes blanks from right of string s; or if e is supplied, removes string e from right of string s
	SEARCH(s,c[,p])	Position of the first character in s, starting at position p, that is found in c
	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
	TRIM(s[,e,f])	String s with all characters in e removed from the left and all characters in f removed from the right
	VERIFY(s,c[,p])	Position of the first character in s, starting at position p, which is not found in c
Conversion	BYTE(x)	ASCII character represented by the integer x
	DECODE(c,r)	Fixed binary value of the character string c converted to a base r number
	ENCODE(i,r)	Character string representing the base r number that is equivalent to the fixed binary expression i
	RANK(c)	Integer representation of the ASCII character c
Timekeeping	DATE( )	System date of compilation in the form YYMMDD
	DATETIME( )	System date and time of compilation in the form CCYYMMDDHHMMSSXX
	TIME( )	System time of day of compilation in the form HHMMSSXX
Miscellaneous	ERROR( )	Count of user-generated diagnostic error message
	INFORM( )	Count of user-generated diagnostic informational message
	LINE( )	Line number in source program that contains the end of a specified preprocessor statement
	VARIANT( )	String result representing the value of the /VARIANT command qualifier
	WARN( )	Count of user-generated diagnostic warning message

11 Built-In Functions, Subroutines, and Pseudovariables

PL/I provides a set of predefined functions, subroutines, and variables called built-in functions, built-in subroutines, and pseudovariables respectively.

- Built-in functions are procedures to use wherever an expression is valid.
- Built-in subroutines are routines which offer additional capabilities.
- Pseudovariables can be used in certain assignment statements in place of ordinary variables.

This chapter describes the built-in functions, subroutines, and pseudovariables that you can use in your PL/I programs.

11.1 Built-In Function Arguments

Built-in functions are similar to operators, and their arguments are similar to operands. Built-in function arguments, if arithmetic, are converted to their derived type before the function reference is evaluated. 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 result is the greater of the precisions of the two arguments.

For instance, all the mathematical functions 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. For example:

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

Here 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 then converted to FLOAT BINARY(8), and the ATAN operation is performed.

Note the following restrictions on built-in function arguments:

- 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, such as NULL(), unless the function's name has been declared with the BUILTIN attribute.

11.2 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. Refer to Section 8.10.4 for more information about condition handling.

11.3 Summary of Built-In Functions

The built-in functions are summarized in PL/I, according to the following categories:

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

Section 11.4 provides detailed descriptions of the functions listed in PL/I.

Table 11–1 Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
Arithmetic	ABS(x)	Absolute value of x
	ADD(x,y,p[,q])	Value of x+y, with precision p and scale factor q
	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

Table 11–1 (Cont.) Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
	FLOOR(x)	Largest integer that is less than or equal to x
	MAX(x,y)	Larger of the values x and y
	MIN(x,y)	Smaller of the values x and y
	MOD(x,y)	Value of x modulo y
	MULTIPLY(x,y,p[,q])	Value of x*y, with precision p and scale factor q
	ROUND(x,k)	Value of x rounded to k digits
	SIGN(x)	-1, 0, or 1 to indicate the sign of x
	SUBTRACT(x,y,p[,q])	Value of x-y, with precision p and scale factor q
	TRUNC(x)	Integer portion of x
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 sine is x and whose cosine is y)
	ATANH(x)	Hyperbolic arc tangent 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
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
	EVERY(s)	Boolean value indicating whether every bit in bit string s is '1'B
	HIGH(c)	String of length c of repeated occurrences of the highest character in the collating sequence

Table 11–1 (Cont.) Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
	INDEX(s,c[,p])	Position of the character string c within the string s, starting at position p
	LENGTH(s)	Number of characters or bits in the string s
	LOW(c)	String of length c of repeated occurrences of the lowest character in the collating sequence
	LTRIM(s,[b])	Removes white space (form feeds, carriage returns, tabs, vertical tabs, line feeds, and spaces) from left of string s; or if b is supplied, removes string b from left of string s
	MAXLENGTH(s)	Maximum length of varying string s
	REVERSE(s)	Reverse of the source character string or bit string
	RTRIM(s,[e])	Removes white space (form feeds, carriage returns, tabs, vertical tabs, line feeds, and spaces) from right of string s; or if e is supplied, removes string e from right of string s
	SEARCH(s,c[,p])	Position of the first character in s, starting at position p, that is found in c
	SOME(s)	Boolean value indicating whether at least one bit in bit string s is '1'B
	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
	TRIM(s[,e,f])	String s with all characters in e removed from the left, and all characters in f removed from the right
	VERIFY(s,c[,p])	Position of the first character in s, starting at position p, which is not found in c
Conversion	BINARY(x[,p[,q]])	Binary value of x with precision p and scale factor q
	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
	DECODE(c,r)	Fixed binary value of the character string c converted to a base r number
	ENCODE(i,r)	Character string representing the base r number that is equivalent to the fixed binary expression i
	FIXED(x,p[,q])	Fixed arithmetic value of x
	FLOAT(x,p)	Floating arithmetic value of x
	INT(x[,p[,l]])	Signed integer value of variable x, located at position p with length l
	POSINT(x[,p[,l]])	Unsigned integer value of variable x, located at position p with length l
	RANK(c)	Integer representation of the ASCII character c
	UNSPEC(x[,p[,l]])	Internal coded form of x, located at position p with length l

Table 11–1 (Cont.) Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
Condition-Handling	ONARGSLIST( )	Pointer to argument lists of exception condition
	ONCHAR( )	Character that caused the CONVERSION condition to be raised
	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
	ONSOURCE( )	Field containing the ONCHAR character when the CONVERSION condition was raised
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
	PROD(x)	Arithmetic product of all the elements in x
	SUM(x)	Arithmetic sum of all the elements in x
Storage	ADDR(x)	Pointer identifying the storage referenced by x
	ADDREL(p,o)	Pointer which is the sum of pointer p and offset o
	ALLOCATION(x)	Number of existing generations for controlled variable x
	BYTESIZE(x)	Number of bytes allocated to variable x; same as SIZE function
	EMPTY( )	An empty area value
	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
	SIZE(x)	Number of bytes allocated to variable x
Timekeeping	DATE( )	System date in the form YYMMDD
	DATETIME( )	System date and time in the form CCYYMMDDHHMMSSXX
	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
Preprocessor	ABS(x)	Absolute value of x
	BYTE(x)	ASCII character represented by integer x
	COPY(s,c)	c copies of specified string s
	DATE( )	Compilation date in the form YYMMDD
	DATETIME( )	System date and time in the form CCYYMMDDHHMMSSXX
	DECODE(c,r)	Fixed binary value of the character string c converted to a base r number
	ENCODE(i,r)	Character string representing the base r number that is equivalent to the fixed binary expression i
	ERROR( )	Count of user-generated diagnostic error messages

Table 11–1 (Cont.) Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
	INDEX(s,c[,p])	Position of the character string c within the string s, starting at position p
	INFORM( )	Count of user-generated diagnostic informational messages
	LENGTH(s)	Number of characters or bits in the string s
	LINE( )	Line number in source program that contains the end of the specified preprocessor statement
	LTRIM(s,[b])	Removes white space (form feeds, carriage returns, tabs, vertical tabs, line feeds, and spaces) from left of string s; or if b is supplied, removes string b from left of string s
	MAX(x,y)	Larger of the values x and y
	MIN(x,y)	Smaller of the values x and y
	MOD(x,y)	Value of x modulo y
	RANK(c)	Integer representation of the ASCII character c
	REVERSE(s)	Reverse of the source character string or bit string
	RTRIM(s,[e])	Removes white space (form feeds, carriage returns, tabs, vertical tabs, line feeds, and spaces) from right of string s; or if e is supplied, removes string e from right of string s
	SEARCH(s,c[,p])	Position of the first character in s, starting at position p, that is found in c
	SIGN(x)	-1,0, or 1 to indicate the sign of x
	SUBSTR(s,i[,j])	Part of string s beginning at i for j characters
	TIME( )	Compilation time of the day in the form HHMMSSXX
	TRANSLATE(s,c[,d])	String s with substitutions defined in c and d
	TRIM(s[,e,f])	String s with all characters in e removed from the left and all characters in f removed from the right
	VARIANT( )	String result representing the value of /VARIANT of the PLI command qualifier (for OpenVMS) or *-variant* command option (for RISC ULTRIX)
	VERIFY(s,c[,p])	Position of the first character in s, starting at position p, which is not found in c
	WARN( )	Count of user-generated diagnostic warning messages
Argument-passing	ACTUALCOUNT( )	Number of parameters the current procedure was called with
	DESCRIPTOR(x)	Forces its argument to be passed by descriptor to a non-PL/I procedure
	PRESENT(p)	Boolean value indicating whether parameter p was specified in a call
	REFERENCE(x)	Forces its argument to be passed by reference to a non-PL/I procedure
	VALID(p)	Boolean value, indicating whether the pictured variable p has a value consistent with its picture specification
	VALUE(x)	Forces its argument to be passed by value to a non-PL/I procedure

11.4 Descriptions of Built-In Functions

This section presents the built-in functions in alphabetical order.

11.4.1 ABS

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: to ensure that an expression has a positive value before it is used as an argument to the square root (SQRT) built-in function.

11.4.2 ACOS

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:

$0 <= w <= \pi$

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:

ACOS(x)

11.4.3 ACTUALCOUNT

The ACTUALCOUNT built-in function allows you to determine how many parameters the current procedure was called with. The function returns a FIXED BINARY(31) result.

The format of the function is:

ACTUALCOUNT();

11.4.4 ADD

The ADD built-in function returns the sum of two arithmetic expressions x and y, with a specified precision p and an optionally specified scale factor q. The format of the function is:

ADD(x,y,p[,q])

p

An unsigned integer constant greater than zero and less than or equal to the maximum precision of the result type, which is:

- For OpenVMS Alpha systems: 31 for fixed-point data, 15 for floating-point decimal data, and 53 for floating-point binary data
- >For OpenVMS VAX systems: 31 for fixed-point data, 34 for floating-point decimal data, and 113 for floating-point binary data

q

An integer constant less than or equal to the specified precision. The scale factor can be optionally signed when used in fixed-point binary addition. The scale factor for fixed-point binary must be in the range -31 to p. The scale factor for fixed-point decimal data must be in the range 0 to p. If you omit q, the default value is zero. You should not use a scale factor for floating-point arithmetic.

Expressions x and y are converted to their derived type before the addition is performed.

For example:

```
ADDBIF: PROCEDURE OPTIONS (MAIN);

DECLARE X FIXED DECIMAL (8,3),
        Y FIXED DECIMAL (8,3),
        Z FIXED DECIMAL (9,3);

X=9500.374;
Y=2278.897;
Z = ADD (X,Y,9,3);

PUT SKIP LIST ('TOTAL =',Z);

END;
```

This program prints the following:

```
TOTAL = 11779.271
```

11.4.5 ADDR

The ADDR built-in function returns a pointer to storage denoted by a specified variable. The variable reference must 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.)

11.4.6 ADDREL

The ADDREL built-in function adds an offset value to a pointer and returns a pointer which is the sum of the two arguments. The format of the function is:

ADDREL(pointer,offset)

pointer
A reference to a pointer variable whose current value represents the location of a based variable.

offset
Any integer expression.

11.4.7 ALLOCATION

The ALLOCATION built-in function returns a fixed-point binary integer that is the number of existing generations of a specified controlled variable. If no generations of the specified variable exist, the function returns zero. The format of the function:

{ ALLOCATION / ALLOCN } (reference)

reference
The name of a controlled variable.

Examples

```
DECLARE INPUT CHARACTER(10) CONTROLLED,
   A CHARACTER(3) VARYING;
        .
        .
        .
    DO UNTIL (INPUT = 'QUIT');
       ALLOCATE INPUT;
       GET LIST(INPUT);
          .
          .
          .
       END;
    A = ALLOCATION(INPUT);
    PUT SKIP LIST('Generations = ;A);
```

This example uses the ALLOCATION built-in function to return the number of generations of the controlled variable INPUT. The example illustrates how input in an interactive program can be stored on a stack for future use.

```
ALLO: PROCEDURE OPTIONS (MAIN);
DECLARE STR CHARACTER (10) CONTROLLED;
        ALLOCATE STR;
        STR='FIRST';
        ALLOCATE STR;
        STR='SECOND';
        ALLOCATE STR;
        STR='THIRD';
```

```
DO WHILE (ALLOCATION(STR)^=0);
        PUT SKIP LIST (STR);
        FREE STR;
  END;
END;
```

This example shows how the ALLOCATION built-in function can be used to count generations of controlled variables and therefore control the loop. Strings are freed while generations still exist, but when all generations have been freed, the value of ALLOCATION is zero and the process ends, thus avoiding a fatal run-time error.

11.4.8 ASIN

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:

$-\pi/2 <= w <= \pi/2$

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)

11.4.9 ATAN

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 both have nonzero values after they have been converted 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 arc tangent(s), such that:

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

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

If x is present, the returned value v equals arc tangent(s/r), such that if $s >= 0$, then $0 <= v <= \pi$, 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.

11.4.10 ATAND

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 both have nonzero values after their conversion to floating point.

The format of the function is:

ATAND(y[,x])

Returned Value

The floating-point value returned (which represents an angle in degrees) equals:

$ATAN(y, x) * 180/\pi$

11.4.11 ATANH

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 1.

The format of the function is:

ATANH(x)

11.4.12 BINARY

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

The format of the function is:

{ BINARY / BIN } (x[,p[,q]])

p

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:

- For OpenVMS Alpha: 63 if fixed-point binary and 53 if floating-point binary
- For OpenVMS VAX: 31 if fixed-point binary and 113 if floating-point binary

The precision p must be specified if x is a fixed-point value with fractional digits.

q
The scale factor q, if specified, must be an integer constant less than or equal to the specified precision and in the range -31 to 31.

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 PL/I rules for conversion.

The returned value is the value v, with precision p, and scale factor q. If p is omitted (integer and floating-point arguments only), the precision of the returned value is the converted precision of x. FIXEDOVERFLOW, OVERFLOW, or UNDERFLOW is signaled if appropriate.

11.4.13 BIT

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 PL/I rules for conversion to bit strings.

The format of the function is:

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

11.4.14 BOOL

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.

The format of the function 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 in the following truth table:

String-1 Bit	String-2 Bit	Result of Boolean Operation
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

Thus, an AND operation, for instance, would be specified by the operation-string '0001'B.

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

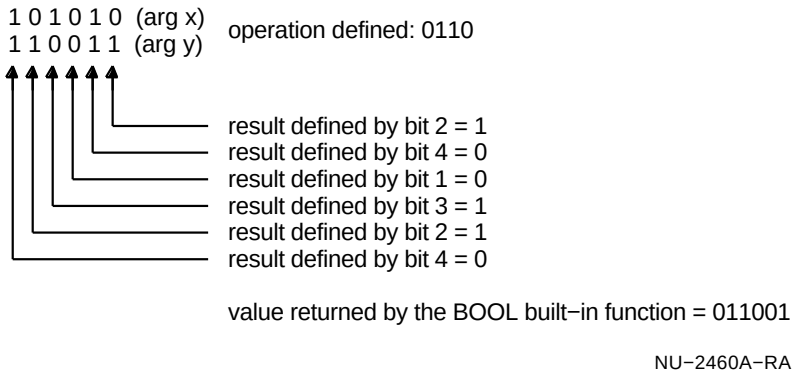
Examples

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

The operation string is '0110'B, which defines an EXCLUSIVE OR operation. The operation is performed as follows on the corresponding bits in the strings X and Y: The leftmost bit in X is 1 and the leftmost bit in Y is 1. The truth table above specifies that when the two corresponding bits in the two strings are both 1, then bit 4 of the operation string will be the result; in this case, bit 4 of the operation string '0110'B is 0. Thus, 0 is the first bit of the value to be returned. The second bit of X is 0 and of Y is 1. The truth table specifies that when the bit in the first string is 0 and in the second string is 1, the result will be bit 2 of the operation string. Here, bit 2 of the operation string '0110'B is 1, and so 1 is the second bit of the value to be returned. The operation continues in this manner with each two corresponding bits in the strings. The value returned is <BIT_STRING>(011001)B.

Figure 11–1 illustrates this example.

Figure 11–1 Example of the BOOL Built-In Function



11.4.15 BYTE

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 modulo 256. The format of the function is:

BYTE(x)

Examples

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

11.4.16 BYTESIZE

This function is the same as the SIZE function. See Section 11.4.81.

11.4.17 CEIL

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:

$precision = min(31, p - q + 1)$

$scalefactor = 0$

Here, 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 */
```

11.4.18 CHARACTER

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 PL/I rules for conversion to character strings. The format of the function is:

{ CHARACTER / CHAR } (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 <BIT_STRING>(12.34567) is actually preceded by two spaces; this is the case with any nonnegative number converted to a character string. 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 as ' 12'.

11.4.19 COLLATE

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

COLLATE()

11.4.20 COPY

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

Value of Count	String Returned
1	The string argument
n	Concatenated copies of the string argument

Examples

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

This function reference returns the character-string value <BIT_STRING>(121212).

11.4.21 COS

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)

11.4.22 COSD

The COSD built-in function returns a floating-point value that is 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)

11.4.23 COSH

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)

11.4.24 DATE

The DATE built-in function returns a 6-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()

The date returned is the run-time date. However, if DATE is used as a preprocessor built-in function, the date returned is the compile-time date.

11.4.25 DATETIME

The DATETIME built-in function returns a 16-character string in the form ccyymmddhhmmssxx, where:

cc	Is the current century (00-99)
yy	Is the current year (00-99)
mm	Is the current month (01-12)
dd	Is the current day of the month (01-31)
hh	Is the current hour (00-23)
mm	Is the minutes (00-59)
ss	Is the seconds (00-59)
xx	Is the hundredths of seconds (00-99)

The format of the function is:

```
DATETIME()
```

The date and time returned is the run-time date and time. However, if DATETIME is used as a preprocessor built-in function, the date and time returned is the compile-time date and time.

Note that the DATETIME function is identical to the century concatenated with DATE( ) and TIME( ).

11.4.26 DECIMAL

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, 34 for floating-point). If q is specified, x must be a fixed-point expression and p must also be specified; if q is omitted or has a negative value, 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 PL/I rules for conversion. 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. FIXEDOVERFLOW, UNDERFLOW, or OVERFLOW is signaled if appropriate.

11.4.27 DECODE

The DECODE built-in function converts a character string representing a number to a fixed binary number. It takes two arguments: a character string and an integer expression specifying the radix of the string expression. It converts the string to an unsigned, base r integer, where r is the specified radix. The function returns a FIXED BINARY(31,0) number representing the base ten equivalent of the string.

The format of the function is:

DECODE(character-expression,radix-expression)

character-expression

A character-string constant or variable whose component characters can be any of the digits from '0' through '9', from 'a' through 'f', and from 'A' through 'F'. The digits must be within the range of digits valid for the base specified in the radix-expression.

radix-expression

An expression evaluating to any integer from 2 through 16.

Examples

```
DECLARE (X,Y) FIXED BINARY;
X = DECODE('1010',2);
Y = DECODE('f0',16);
```

The fixed binary variables X and Y are given the values 10 and 240, respectively.

11.4.28 DESCRIPTOR

The DESCRIPTOR built-in function forces its argument to be passed by descriptor to a non-PL/I procedure. A reference to the built-in function must occur only as an argument in such a context and has no other use. The format of the function is:

{ DESCRIPTOR / DESC }(expression)

expression

The argument to be passed by descriptor. Its data type must be computational but cannot be pictured. It can be an array variable.

11.4.29 DIMENSION

The DIMENSION built-in function returns a fixed-point binary integer that is the number of elements in an array dimension. Its format is:

{ DIMENSION / DIM }(reference[,dimension])

reference

A reference to an array variable.

dimension

An integer constant specifying the dimension of the array for which the extent is to be determined. If the dimension is not specified, the dimension parameter defaults to 1. Thus, DIMENSION(A) is equivalent to DIMENSION(A,1).

Examples

```
INIT: PROCEDURE (ARRAY);
DECLARE ARRAY(*) FIXED,
        I FIXED;

        DO I = 1 TO DIM(ARRAY);
            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. (Because the array is one-dimensional, the optional second parameter is omitted and defaults to 1.)

11.4.30 DIVIDE

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 an optionally specified scale factor q. The scale factor q must be an integer following these rules:

- If either x or y is fixed binary, q must be in the range -31 through 31.
- If both x and y are fixed decimal, q must not be negative.
- If either x or y is floating point, q must be zero.
- If q is omitted, it is assumed to be zero.

The expressions x and y are converted to their derived types before the division is performed. If y is zero after this conversion, the ZERODIVIDE condition is signaled. The quotient has the derived type of the two arguments.

The format of the function is:

DIVIDE(x,y,p[,q])

11.4.31 EMPTY

The EMPTY built-in function returns an empty area value for use in initializing areas. Its format is:

EMPTY()

The EMPTY built-in function is useful in initializing the contents of an area. It is normally much faster than the FREE statement is in freeing all the variables in an area (freeing all the area's storage). Note that an area value must be assigned to an area before the area is used.

The following is an example of its use in a declaration:

```
DECLARE A AREA(1024) STATIC INITIAL(EMPTY());
```

11.4.32 ENCODE

The ENCODE built-in function converts a decimal integer to a character string. It converts the decimal integer (stored as a FIXED BINARY(31,0) number) to a base r number, where r is the radix you specify, and returns the resulting number as a character string. The function takes two arguments: a decimal integer and a radix; the radix is an integer in the range 2 through 16.

The format of the function is:

ENCODE(integer-expression,radix-expression)

integer-expression
An expression evaluating to a fixed binary number representing a decimal integer. Whether signed or not, this integer is treated by the function as unsigned.

radix-expression
An expression that evaluates to any integer from 2 through 16.

Examples

```
DECLARE (X,Y) CHARACTER(5) VARYING;
X = ENCODE(53,8);
Y = ENCODE(10,2);
```

The character-string variable X is assigned the value '65', which is the character equivalent of the octal number 65, which is the equivalent of the decimal number 53. The character-string variable Y is assigned the value '1010', which is the character equivalent of the binary number 1010, which is the equivalent of the decimal number 10.

11.4.33 ERROR

The ERROR preprocessor built-in function returns the number of preprocessor diagnostic error messages issued during compilation up to that particular point in the source program. The format for the ERROR built-in function is:

ERROR();

The function returns a fixed-point result representing the number of compile-time warning messages that were issued up until the point at which the built-in function was encountered.

11.4.34 EVERY

The EVERY built-in function determines whether every bit in a bit string is '1'B. In other words, it performs a logical AND operation on the elements of the bit string. The format of the function is:

```
EVERY(bit-string)
```

The function returns the value '1'B if all bits in the bit-string argument are '1'B. It returns '0'B if one or more bits in the argument are '0'B or if the argument is the null bit string.

11.4.35 EXP

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)
```

11.4.36 FIXED

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 format of the function is:

```
FIXED(x,p[,q])
```

p
The number of bits used to represent the arithmetic value. The precision must be greater than zero and less than or equal to 31.

q
An integer in the range 0 through 31 for decimal data, in the range -31 through 31 for binary data. If q is omitted, it is assumed to be zero. The scale factor q must be less than or equal to the specified precision.

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 PL/I rules. 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 a scale factor of zero. FIXEDOVERFLOW is signaled if appropriate.

11.4.37 FLOAT

The FLOAT built-in function converts a string or arithmetic expression x to floating point, with a specified precision p. The 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, which is:

- For OpenVMS VAX systems: 34 for floating-point decimal data and 113 for floating-point binary data
- For OpenVMS Alpha systems: 15 for floating-point decimal data and 53 for floating-point binary data

If x is a character string, it 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, a plus (+) or minus (-) sign, a decimal point (.), and the letter E. If the character string contains any invalid characters, the CONVERSION condition is signaled.

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 of the result type, following the PL/I conversion rules, and of the specified precision. UNDERFLOW or OVERFLOW is signaled if appropriate.

11.4.38 FLOOR

The FLOOR built-in function returns the largest integer that is less than or equal to an arithmetic expression x. The 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 following attributes:

$precision = min(31, p - q + 1)$

$scalefactor = 0$

Here, p and q are the precision and scale factor of x.

For example:

```
FLOOR_DEMO: PROC OPTIONS(MAIN);
 PUT LIST (FLOOR(3));
 PUT LIST (FLOOR(-3.323));
 PUT LIST (FLOOR(3.456E9));
END;
```

This program prints the following values:

```
3      -4    3.456E+09
```

11.4.39 HBOUND

The HBOUND built-in function returns a fixed-point binary integer that is the upper bound of an array dimension. The format is:

HBOUND(reference[,dimension])

reference
A reference to an array variable.

dimension
An integer constant indicating a dimension of the specified array. If the dimension is not specified, the dimension parameter defaults to 1. Thus, HBOUND(A) is equivalent to HBOUND(A,1).

11.4.40 HIGH

The HIGH built-in function returns a string of specified length that consists of repeated appearances of the highest character in the collating sequence. The format is:

HIGH(length)

length
The specified length of the returned string. The (maximum length of the returned string is 32767 characters.

Returned Value

The string returned is of the length specified. The rank of the highest character that can appear in the collating sequence for PL/I is ASCII 255.

11.4.41 INDEX

The INDEX built-in function returns a fixed-point binary integer that indicates the position of the leftmost occurrence of a specified substring within a string. If the substring is not found, or if the length of either argument is zero, the INDEX function returns zero. This function is case-sensitive.

The format of the function is:

INDEX(string,substring[,starting-position])

string
The string to be searched for the given substring. It can be either a character-string or a bit-string expression.

substring
The substring to be located. It must have the same string data type as the string argument.

starting-position

A positive integer in the range 1 to n+1, where n is the length of the string. It specifies the leftmost position from which the search is to begin. (By default, the search begins at the left end of the string.)

Examples

```
1   DECLARE RESULT FIXED BINARY(31);
    RESULT = INDEX ('ABCDEF','DEF');
```

RESULT is given the value 4 because the substring 'DEF' begins at the fourth position in 'ABCDEF'.

```
2   RESULT = INDEX('SHARP FORTUNE','R');
```

RESULT is given the value 4 because the leftmost occurrence of 'R' is at the fourth position in 'SHARP FORTUNE'.

```
3   RESULT = INDEX('SHARP FORTUNE','R',5);
```

The optional starting-position parameter specifies that the search begins at the fifth position of 'SHARP FORTUNE'. Thus, RESULT is given the value 9: the first R is ignored, so the first recognized occurrence of 'R' is found in the ninth position.

```
4   RESULT = INDEX('0000101100001011','1011');
```

RESULT is given the value 5 because the leftmost occurrence of '1011' is at the fifth position in '0000101100001011'.

```
5   NEW_STRING = '315-54-3159';
    IF INDEX(NEW_STRING,'-')=4 THEN
      PUT LIST('SOCIAL SECURITY NUMBER');
```

The INDEX function is used to determine whether or not a string is a Social Security number. The function finds the location of the first hyphen in the string.

11.4.42 INFORM

The INFORM preprocessor built-in function returns the number of diagnostic informational messages issued during compilation up to that point in the source program. The format for the INFORM built-in function is:

INFORM();

The function returns a FIXED result representing the number of compile-time warning messages that were issued up until the INFORM built-in function was encountered.

11.4.43 INT

The INT built-in function treats specified storage as a signed integer, and returns the value of the integer. The format is:

INT(expression[,position[,length]])

expression

A scalar expression or reference to connected storage. This reference cannot be an array, structure, or named constant. If position and length are not specified, the length of the referenced storage must not exceed 32 bits. If it exceeds 32 bits, a fatal run-time error results.

position

A positive integer value that denotes the position of the first bit in the field. If omitted, position defaults to 1, signifying the first bit of storage denoted by the expression. If specified, position must satisfy the following condition:

$$1 <= position <= size(expression)$$

Here, size(expression) is the length in bits of the storage denoted by expression. A position equal to size(expression) implies a zero-length field.

length

An integer value in the range 0 through 32 that specifies the length of the field. If omitted, length is the number of bits from the bit denoted by position through the end of the storage denoted by expression. If specified, length must satisfy the following condition:

$$0 <= length <= size(expression) - position$$

Here, size(expression) is the length in bits of the storage denoted by expression.

Returned Value

The value returned by INT is of the type FIXED BINARY (31). If the field has a length of zero, INT returns zero.

Examples

The following example shows the use of the INT built-in function to interpret the storage occupied by a bit string as an integer:

```
B16 = '0000000000001101'B;     /* 16-bit string */

I = BIN(B16);                  /* I = 13 */
I = INT(B16);                  /* I = -20480 */

B64 = '5076ABCD00000000'B4;    /* 64-bit string */

I = INT(B64,1,32);             /* First 32 bits; I = -1277858294 */
I = INT(B64,33);               /* Second 32 bits; I = 0 */
I = INT(B64);                  /* Field too large, run-time error */
```

Notice that, unlike the BIN built-in function, the INT built-in function performs no conversion. It simply treats the contents of the designated storage as a signed integer. Therefore, the value returned by INT depends on the data type (and therefore the internal representation) of the variable occupying the storage. For example:

```
INTEXM: PROCEDURE OPTIONS (MAIN);

DECLARE D FIXED DECIMAL (3,2),
        C CHARACTER (4),
        F FLOAT;
```

```
D = 2.54;
C = '2.54';
F = 2.54;

PUT SKIP LIST ( INT(D),
                INT(C),
                INT (F) );

END;
```

The output of this example is:

```
19493     875900466     -1889779422
```

11.4.44 LBOUND

The LBOUND built-in function returns a fixed-point binary integer that is the lower bound of an array dimension. The format is:

LBOUND(reference[,dimension])

reference
A reference to an array variable.

dimension
An integer constant indicating the dimension of the specified array. If the dimension is not specified, the dimension parameter defaults to 1. Thus, LBOUND(A) is equivalent to LBOUND(A,1).

11.4.45 LENGTH

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. (To determine the maximum length of a varying-length character string, use the MAXLENGTH built-in function.)

The format of the function is:

LENGTH(string)

11.4.46 LINE

The LINE preprocessor built-in function returns the line number of the source program text containing the end of the preprocessor statement that calls the LINE built-in function.

The format of the function within a preprocessor expression is:

LINE()

11.4.47 LINENO

The LINENO built-in function returns a FIXED BINARY(15) integer that is the current line number of the referenced print file. The 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.

11.4.48 LOG

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)
```

11.4.49 LOG10

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)
```

11.4.50 LOG2

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)
```

11.4.51 LOW

The LOW built-in function returns a string of specified length that consists of repeated appearances of the lowest character in the collating sequence. The format is:

```
LOW(length)
```

length

The specified length of the returned string. The maximum length permitted is 32767 characters.

Returned String

The string returned is of the length specified. The rank of the lowest character that can appear in the collating sequence for PL/I is ASCII 0.

11.4.52 LTRIM

The LTRIM built-in function accepts a character string as an argument and returns a character string that consists of the input string with the specified characters removed from the left. If you supply only one argument, white spaces (form feeds, carriage returns, tabs, vertical tabs, line feeds, and spaces) are removed from the left.

The format of the LTRIM built-in function is:

```
LTRIM (input-string, [beginning-chars])
```

input-string

A character-string variable or constant. This argument supplies the string from which the characters or blanks are to be trimmed.

beginning-chars

A character-string variable or constant. This argument specifies characters to be trimmed from the left of the input string. If a character that is in the first position in the input string is also present anywhere in beginning-chars, that character is removed from the input string. This process is repeated until a character is encountered on the left of the input string that is not present in beginning-chars, or until the characters in the input string are exhausted. If no argument is supplied, all white spaces (form feeds, carriage returns, tabs, vertical tabs, line feeds, and spaces) are removed from the left of the input-string.

11.4.53 MAX

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 Section 6.4.2. 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. The value has the following attributes:

$$precision = min(31, max(px - qx, py - qy) + max(qx, qy))$$

$$scalefactor = max(qx, qy)$$

Here, px,qx and py,qy are the converted precisions and scale factors of x and y, respectively.

The MAX built-in function is also a preprocessor built-in function; however, the preprocessor does not permit scale factors.

11.4.54 MAXLENGTH

The MAXLENGTH built-in function returns a fixed binary number representing the maximum possible length of a varying-length character string. The format is:

MAXLENGTH (string)

string

A reference to a character string or a bit string. If it is anything other than a varying-length character string, the MAXLENGTH function returns a result identical to the result that would be returned by the LENGTH built-in function.

For example:

```
MAXLENGTH_EXAMPLE: PROCEDURE OPTIONS(MAIN);
  DCL CHAR_VAR CHARACTER(10) VARYING;
  CHAR_VAR = 'String';
  CALL SAMPLE(CHAR_VAR);
END MAXLENGTH_EXAMPLE;
SAMPLE: PROCEDURE(STRING);
  DCL STRING CHAR(*) VARYING;
  PUT LIST(LENGTH(STRING),MAXLENGTH(STRING));
END SAMPLE;
```

The program prints the following:

```
6             10
```

11.4.55 MIN

The MIN built-in function returns the smaller of two arithmetic expressions x and y. The 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 Section 6.4.2. 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. The value has the following attributes:

$$precision = min(31, max(px - qx, py - qy) + max(qx, qy))$$

$$scalefactor = max(qx, qy)$$

Here, px,qx and py,qy are the converted precisions and scale factors of x and y.

The MIN built-in function is also a preprocessor built-in function; however, the preprocessor does not permit scale factors.

11.4.56 MOD

The MOD built-in function returns, for an arithmetic expression x and nonnegative 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 of x divided by y exactly 0.

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.

If the derived type is unscaled fixed point, then the precision of the result 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 of the precisions of the two converted arguments.

The value returned is:

$u - w * floor(u/w)$

The arguments u and w become 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 can signal FIXEDOVERFLOW.

If x and y are fixed-point expressions, a fixed-point value is returned. The value has the following attributes:

$precision = min(31, pw - qw + max(qu, qw))$

$scalefactor = max(qu, qw)$

Here, 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 the following is true:

$pw - qw + max(qu, qw) > 31$

The MOD built-in function is also a preprocessor built-in function; however, the preprocessor does not permit scale factors.

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. The value has the following attributes:

$precision = min(31, 1 - 0 + max(0, 0)) = 1$

$scalefactor = max(0, 0) = 0$

Although 28 modulo 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.)

11.4.57 MULTIPLY

The MULTIPLY built-in function multiplies two arithmetic expressions x and y, and returns the product of the two values with a specified precision p and an optionally specified scale factor q.

The format of the function is:

MULTIPLY(x,y,p[,q])

p

An integer constant greater than zero and less than or equal to the maximum precision of the result type, which is:

- For OpenVMS Alpha systems: 31for fixed-point data, 15 for floating-point decimal data, and 53 for floating-point binary data
- For OpenVMS VAX systems: 31 for fixed-point data, 34 for floating-point decimal data, and 113 for floating-point binary data

q

An integer in the range -31 through p when used with fixed-point binary multiplication. The scale factor for fixed-point decimal multiplication has a range 0 through p. A scale factor is not to be used with floating-point arithmetic. If no scale factor is designated, q defaults to zero.

Expressions x and y are converted to their derived type before the multiplication is performed.

For example:

```
MULT:   PROCEDURE OPTIONS (MAIN);

DECLARE I_RATE  FIXED DECIMAL(31,4),
        PRINCIPAL  FIXED DECIMAL(31,2),
        OWED FIXED DECIMAL(31,6);

I_RATE = .1514;

PRINCIPAL = 27688.25;

OWED = MULTIPLY (I_RATE,PRINCIPAL,31,6);

PUT SKIP LIST ('INTEREST OWED =',OWED);

END;
```

Interest rates are calculated to six decimal places and the following string is printed:

```
INTEREST OWED = 4192.001050
```

11.4.58 NULL

The NULL built-in function returns a null pointer value. The format is:

NULL()

Examples

```
IF NEXT_POINTER = NULL() THEN CALL FINISH;
```

The IF statement checks whether the pointer variable NEXT_POINTER is null; if so, the CALL statement is executed.

The NULL built-in function can be used for offset variables as well as pointer variables, because the compiler automatically performs conversions between pointer and offset values.

11.4.59 OFFSET

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.

Examples

```
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.

11.4.60 ONARGSLIST

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. The format is:

ONARGSLIST()

What the return pointer points to depends on the host system. See the *Kednos PL/I for OpenVMS Systems User Manual*.

11.4.61 ONCHAR

The ONCHAR built-in function returns the character that caused a CONVERSION condition to be raised. If there is no active CONVERSION condition, the return value is a single space.

The format of the function is:

ONCHAR()

The ONCHAR value is actually a single character substring of the ONSOURCE built-in function value, unless there is no active CONVERSION condition.

11.4.62 ONCODE

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. You can use the function 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. The format is:

```
ONCODE()
```

11.4.63 ONFILE

The ONFILE built-in function returns the name of the file constant for which the current file-related condition was signaled. The 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 I/O 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
- A CONVERSION ON-unit that was entered because of an error that occurred during conversion of data in a GET statement

Returned Value

The returned value is a varying-length character string. The ONFILE function returns a null string if referenced outside an ON-unit, within an ON-unit that is executed as a result of a SIGNAL statement, or within a CONVERSION ON-unit that was not entered because of a conversion in a GET statement.

11.4.64 ONKEY

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:

- KEY, ENDFILE, or UNDEFINEDFILE
- 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. The ONKEY built-in function returns a null string if referenced outside an ON-unit or within an ON-unit executed as a result of the SIGNAL statement.

11.4.65 ONSOURCE

The ONSOURCE built-in function returns the source string that was being converted when the CONVERSION condition was raised. If no CONVERSION condition is active, the return value is a null string.

The format of the function is:

ONSOURCE()

11.4.66 PAGENO

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)

11.4.67 POINTER

The POINTER built-in function returns a pointer to the location identified by the referenced offset and area. The format is:

{ POINTER / PTR } (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.

Examples

```
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.

11.4.68 POSINT

The POSINT built-in function treats specified storage as an unsigned integer, and returns the value of the integer. The format is:

POSINT(expression[,position[,length]])

expression

A scalar expression or reference to connected storage. This reference must not be an array, structure, or named constant. If position and length are not specified, the length of the referenced storage must not exceed 32 bits. (If it exceeds 32 bits, a FATAL run-time error results.)

position

A positive integer value that denotes the position of the first bit in the field. If omitted, position defaults to 1, signifying the first bit of the storage denoted by the expression. If specified, position must satisfy the following condition:

$1 <= position <= size(expression)$

Size(expression) is the length in bits of the storage denoted by expression. A position equal to size(expression) implies a zero-length field.

length

An integer value in the range 0 through 32 that specifies the length of the field. If omitted, length is the number of bits from the bit denoted by position through the end of the storage denoted by expression. If specified, length must satisfy the following condition:

$0 <= length <= size(expression) - position$

Size(expression) is the length in bits of the storage denoted by expression.

Returned Value

The value returned by POSINT is of the type FIXED BINARY (31) for OpenVMS VAX and RISC ULTRIX, or FIXED BINARY (63) for OpenVMS Alpha. If the field has a length of zero, POSINT returns zero.

Because the POSINT built-in function treats storage as if it contained an unsigned integer, the value returned can be larger than the maximum positive value that can be contained in the signed integer that is stored in the same number of bits. For example, if the argument to POSINT is 32 bits long and has the high-order (sign) bit set, then the resulting value is too large for assignment to a FIXED BIN (31) variable. The result of such an operation is undefined.

Examples

The use of the POSINT built-in function is identical to the use of the INT built-in function, except that POSINT treats its argument as an unsigned integer. The following example illustrates this difference:

```
DECLARE (X15,Y15,I15,P15) FIXED BIN (15),
        P31 FIXED BIN (31);

    X15 = 585;
    Y15 = -585;
    I15 = INT(X15);    /* I15 = 585      */
    I15 = INT(Y15);    /* I15 = -585    */
    P15 = POSINT(X15); /* P15 = 585      */
    P31 = POSINT(Y15); /* P31 = 64951    */
    P15 = POSINT(Y15); /* ERROR signaled */
```

In this example, POSINT first assigns the storage referenced by X15 to P15. Because this storage is occupied by a positive integer and therefore has the sign bit clear, POSINT behaves exactly like INT. However, when POSINT is applied to storage occupied by a negative integer, it interprets the set sign bit as representing part of the integer. When the resulting value is assigned to a FIXED BIN (31) variable, it is seen to be larger than the largest possible FIXED BIN (15) value, 32767. An attempt to assign the same value to a FIXED BIN (15) variable results in PL/I signaling an ERROR condition.

11.4.69 PRESENT

The PRESENT built-in function allows you to determine whether a given parameter was specified in a call. It can simplify the task of writing procedures with optional parameters.

The PRESENT built-in function takes one argument, the parameter name. It returns the bit value '1'B if the parameter was specified and '0'B if it was not.

The format of this function is:

PRESENT(parameter-name)

Note that the result returned by the PRESENT built-in function for an optional parameter passed by value is unpredictable (if a zero is passed, '0'B is returned). A warning is generated for this use.

11.4.70 PROD

The PROD built-in function takes an array as an argument and returns the arithmetic product of all the elements in the array. The array must have the FIXED or the FLOAT attribute. The format of the PROD built-in function is:

PROD(array-variable);

If the array has the attributes FIXED(p,0), the result will have the attributes FIXED(p,0). If the array has the attributes FLOAT(p), the result will also have the attributes FLOAT(p). If the array has the attributes FIXED(p,q) with q not equal to 0, the result will have the attributes FLOAT(p).

The result will have the same base attribute as the array, either DECIMAL or BINARY.

Note that the PROD built-in function does not perform matrix multiplication of two arrays.

11.4.71 RANK

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 15. The format of the function is:

RANK(character)

character
Any expression yielding a 1-character value.

Examples

```
CODE = RANK('A');  /* CODE = 65 */
CODE = RANK('a');  /* CODE = 97 */
CODE = RANK('$');  /* CODE = 36 */
```

The ASCII characters are the first 128 characters of the DEC Multinational Character Set. See Appendix B for a table of these characters and their corresponding numeric codes.

11.4.72 REFERENCE

The REFERENCE built-in function is used to force a parameter to be passed by reference, rather than by whatever mechanism is specified by the declaration of the formal parameter.

The type of the argument specified with the REFERENCE built-in function is used for the parameter; thus, the type of the parameter declaration is ignored when the REFERENCE built-in function is used.

The format of the REFERENCE built-in function is:

{ REFERENCE / REF } (variable-reference)

variable-reference
The name of a scalar or aggregate variable.

11.4.73 REVERSE

The REVERSE built-in function reverses the characters or bits in a string. It takes one argument, which is either a character string (fixed or varying) or a bit string. It returns a string of the same type and size as its argument, with all the characters (bytes) or bits reversed.

The format of the function is:

REVERSE(string-expression);

string-expression

An expression that evaluates to a character string or a bit string.

Examples

```
DECLARE X CHARACTER(4) VARYING,
        Y BIT(8);
X = REVERSE('abc')
Y = REVERSE('00010101'B)
```

The character-string variable X is assigned the value 'cba'. The bit-string variable Y is assigned the value '10101000'B.

11.4.74 ROUND

The ROUND built-in function rounds a fixed-point binary expression, fixed-point decimal expression, or pictured value to a specified number of binary or decimal places respectively. The format is:

ROUND(expression,position)

expression

An arithmetic expression that yields a fixed-point binary or decimal value; or a pictured value with fractional digits. A binary value can have a positive or negative non-zero scale factor, but a decimal value must have a positive non-zero scale factor.

position

A nonnegative integer constant specifying the number of binary or decimal places in the rounded result.

Returned Value

Where the arguments are an expression of type FIXED BINARY(p,q) or type FIXED DECIMAL(p,q) and position k, the returned value is the rounded value with the following attributes:

$precision = max(1, min(p - q + k + 1, 31))$

$scalefactor = k$

For a fixed binary number, the rounded value is:

$ROUND(x, k) = sign(x) * (2^{-k+1}) * floor(abs(x) * (2^k) + 1)$

For a fixed decimal number, the rounded value is:

$ROUND(x, k) = sign(x) * (10^{-k}) * floor(abs(x) * (10^k) + 0.5)$

Example 1

The following sample program shows rounding of scaled fixed binary numbers:

```
r: procedure options(main);

declare
  fb1 fixed binary(31, 8),
  fb2 fixed binary(31, 4),
  fb3 fixed binary(31, 2),
  fb4 fixed binary(31, 2),
  fb5 fixed binary(31, 2);

  fb1 = 16.8750;  /* 7/8 */
  fb2 = 15.4375;  /* 7/16 */
  fb3 = 128.25;   /* 128 1/4 */
  fb4 = 128.75;   /* 128 3/4 */
  fb5 = -128.75;  /* -128 3/4 */

  put skip edit ('round(16.8750, 2)=',round(fb1,2)) (a,col(20),f(15,5));
  put skip edit ('round(16.8750, 3)=',round(fb1,3)) (a,col(20),f(15,5));
  put skip edit ('round(15.4375, 1)=',round(fb2,1))  (a,col(20),f(15,5));
  put skip edit ('round(128.25,0)=',round(fb3,0)) (a,col(20),f(15,5));
  put skip edit ('round(128.75,0)=',round(fb4,0)) (a,col(20),f(15,5));
  put skip edit ('round(-128.75,0)=',round(fb5,0)) (a,col(20),f(15,5));

end r;
```

This program produces the following output. Note that 16.87500 equals 16 7/8 in fractional notation and that 15.50000 equals 15 1/2 in fractional notation.

```
round(16.8750, 2)=        17.00000
round(16.8750, 3)=        16.87500
round(15.4375, 1)=        15.50000
round(128.25,0)=         128.00000
round(128.75,0)=         129.00000
round(-128.75,0)=       -129.00000
```

Example 2

The following example shows rounding of scaled fixed decimal numbers:

```
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  */
```

11.4.75 RTRIM

The RTRIM built-in function accepts a character string as an argument and returns a character string that consists of the input string with the specified characters removed from the right. If you supply only one argument, white spaces (form feeds, carriage returns, tabs, vertical tabs, line feeds, and spaces) are removed from the end.

The format of the RTRIM built-in function is:

RTRIM (input-string, [end-chars])

input-string

A character-string variable or constant. This argument supplies the string from which blanks are to be trimmed.

end-chars

A character-string variable or constant. This argument specifies characters to be trimmed from the right of the input string. If a character that is in the last position in the input string is also present anywhere in end-chars, that character is removed from the input string. This process is repeated until a character is encountered on the right of the input string that is not present in end-chars, or until the characters in the input string are exhausted. If no argument is supplied, all white spaces (form feeds, carriage returns, tabs, vertical tabs, line feeds, and spaces) are removed from the right of the input-string.

11.4.76 SEARCH

The SEARCH built-in function takes two character-string arguments and attempts to locate the first character in the first string that is also present in the second string. The search for a match is carried out from left to right. If one character is matched, the function returns the position of that character in the first string. This function is case sensitive.

The format is:

SEARCH(string-1,string-2[,starting-position])

string-1

A character-string expression. One character in the string is to be matched, if possible, in the second string.

string-2

A character-string expression to be compared, character by character, with each character in the first string, in order, until one matching character is found.

starting-position

A positive integer in the range 1 to n+1, where n is the length of the first string. It specifies the leftmost character in the first string from which the search is to begin. If starting-position is specified, any characters to the left of that position in the first string are ignored. (By default, the search begins with the leftmost character in the first string.)

Returned Value

The returned value is a positive integer representing the position in string-1 of the first character that is also found in string-2. If no match is found, the returned value is zero.

Examples

```
DECLARE STR1 CHARACTER(20) VARYING,
        STR2 CHARACTER(10) INITIAL ('ABCDEFGHIJ'),
        X FIXED DECIMAL(2);
STR1 = 'BARBARA';
X = SEARCH (STR1,STR2);
```

In this example, X is given the value 1 because the first character ('B') in STR1 ('BARBARA') is found in STR2 ('ABCDEFGHIJ').

```
STR1 = '12-GEORGE';
X = SEARCH (STR1,STR2);
```

Here, X is given the value 4. 'G' is in the fourth position in '12-GEORGE' and is the first character in STR1 that is also present in STR2 ('ABCDEFGHIJ').

```
X = SEARCH (STR1,STR2,6);
```

X is given the value 8. The starting-position parameter, 6, causes the search to begin with the sixth character in '12-GEORGE', and thus the first matching character is the second 'G', which is in the eighth position.

```
PUT LIST (SEARCH('ZZZBAD','ABCD'));
```

The function returns the value 4 because the position of 'B' in 'ZZZBAD' is 4, and 'B' is the leftmost matching character. Here, constants are used instead of variables.

```
PUT LIST (SEARCH('ABCD','ZZZBAD'));
```

This statement is the same as the preceding one except that the parameters are reversed. Now the value returned is 1 instead of 4 because 'A', the first character in 'ABCD', is matched. Note that the order in which the parameters are given is crucial. Note also that duplicate characters in the second string never change the result.

```
PUT LIST (SEARCH ('  TEST  123','0123456789'));
```

The function returns the value 9 because '1', which is in the ninth position, is the first character matched in the second string.

11.4.77 SIGN

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)

11.4.78 SIN

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)

11.4.79 SIND

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)

11.4.80 SINH

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)

11.4.81 SIZE

The SIZE built-in function returns a fixed-point binary integer that is the number of bytes allocated to a referenced variable. The format is:

SIZE (reference)

reference
The name of a variable known to this block. The reference can be to a scalar variable, an array or structure, or a structure member. The reference cannot be to a constant or expression. Although references to individual array elements are allowed, the returned value in this instance is the size of the entire array, not the element.

Returned Value

The returned value is the variable's allocated size in bytes. For bit strings that do not exactly fill an integral number of bytes, the value is rounded up to the next byte.

For varying character-string variables, note that the returned value is two bytes greater than the declared length of the string. These extra two bytes are allocated by PL/I to contain the current length of the string. (If you want the value of the maximum length of a varying character string, use the MAXLENGTH built-in function. If you want the value of the current length of a varying character string, use the LENGTH built-in function.)

Examples

The following example illustrates the use of the SIZE built-in function on some scalar variables.

```
DECLARE S FIXED BINARY(31),
        INT FIXED BINARY(15),
        CHAR1 CHARACTER(5),
        CHAR2 CHARACTER(5) VARYING,
        BITSTRING BIT(10),
        P POINTER;
```

```
    S = SIZE(INT);          /* S = 2 */
    S = SIZE(CHAR1);        /* S = 5 */
    S = SIZE(CHAR2);        /* S = 7 */
    S = SIZE(BITSTRING);    /* S = 2 */
    S = SIZE(P);            /* S = 4 */
```

Note the difference between the allocated size for the fixed-length and varying character strings. Note also that the returned value for the bit string is rounded up to 2 bytes, the integral number of bytes required to contain 10 bits.

```
DECLARE 1 STRUC,
          2 CHARSTR CHARACTER(5),
          2 BITSTR BIT(10),
        ARRAY(5) FIXED BINARY(31),
        S FIXED BINARY(31);

    S = SIZE(STRUC);        /* S = 7  */
    S = SIZE(CHRSTR);       /* S = 5  */
    S = SIZE(ARRAY);        /* S = 20 */
    S = SIZE(ARRAY(2));     /* S = 20 */
```

In this example, the SIZE built-in function is applied to a structure, to one of its members, to an array, and to an element of the array. Note that a reference to an array element returns the same value as a reference to the entire array.

```
DECLARE 1 TARGET,
          2 A BIT(9),
          2 B BIT(10),
          2 C BIT(1),
        1 ALIGNED_TARGET,
          2 A BIT(9) ALIGNED,
          2 B BIT(10) ALIGNED,
          2 C BIT(1) ALIGNED,
        S FIXED BINARY(31);

    S = SIZE(TARGET);         /* S = 3 */
    S = SIZE(ALIGNED_TARGET); /* S = 5 */
```

This example illustrates the difference in PL/I's storage of unaligned and aligned bit strings. The structure TARGET consists of three bit strings that are unaligned (the default storage mechanism). The three bit strings occupy 20 consecutive bits in memory. Therefore, only three bytes are required to hold the structure. The structure ALIGNED_TARGET consists of the same three strings, except each is declared with the ALIGNED attribute, forcing the structure to start on a byte boundary. In this structure, A and B each require two bytes while C requires one byte, for a total of five bytes. A similar situation exists with arrays of bit strings.

```
T: PROC OPTIONS(MAIN);

DCL P PTR;
DCL 1 S BASED(P),
    2 I FIXED,
    2 A(10 REFER(I)) FIXED;

ALLOCATE S;
PUT SKIP LIST(SIZE(S));          /* Returns 44 */
I = 5;
PUT SKIP LIST(SIZE(S));          /* Returns 24 */
END;
```

This example shows how the SIZE built-in function works on a structure containing the REFER option. SIZE returns the current size.

```
DECLARE STR CHARACTER(10) VARYING;
   .
   .
   .
    CALL SUB(STR);
   .
   .
   .
SUB: PROCEDURE(X);
DECLARE X CHARACTER(*) VARYING;

    PUT SKIP LIST (SIZE(X));
```

Here, the SIZE built-in function is used to determine the size of a parameter that is passed to a procedure. This PUT statement prints the value 12.

```
    CALL MACRO_ROUTINE(
        ADDR(OUTSTRING),SIZE(OUTSTRING) );
```

Here, the SIZE built-in function is used to supply an argument to a procedure (possibly one written in another language) that requires the size in bytes of a data structure.

11.4.82 SOME

The SOME built-in function allows you to determine whether at least one bit in a bit string is '1'B. In other words, it performs a logical OR operation on the elements of the bit string. The format of the SOME built-in function is:

SOME(bit-string)

The function returns the value '1'B if one or more bits in the bit-string argument are '1'B. It returns '0'B if every bit in the argument is '0'B or if the argument is the null bit string.

11.4.83 SQRT

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)

11.4.84 STRING

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.

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;
```

11.4.85 SUBSTR

The SUBSTR built-in function returns a specified substring from a string. The 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 1 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:

$LENGTH(string) - position + 1$

In other words, if length is not specified, the substring is extracted beginning at the indicated position and ending at the end of the string.

The length must satisfy the following condition:

$0 <= length <= LENGTH(string) - position + 1$

If it does not, and the module was compiled with /CHECK=BOUNDS, the STRINGRANGE condition is raised.

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             ' */
```

11.4.86 SUBTRACT

The SUBTRACT built-in function returns the difference of two arithmetic expressions x and y, with a specified precision p and an optionally specified scale factor q. The format of the function is:

SUBTRACT(x,y,p[,q])

p

An unsigned integer constant greater than zero and less than or equal to the maximum precision of the result type, which is:

- For OpenVMS Alpha systems: 31for fixed-point data, 15 for floating-point decimal data, and 53 for floating-point binary data
- For OpenVMS VAX systems: 31 for fixed-point data, 34 for floating-point decimal data, and 113 for floating-point binary data

q

An integer constant less than or equal to the specified precision. The scale factor can be optionally signed when used in fixed-point binary subtraction. The scale factor for fixed-point binary must be in the range -31 through p. The scale factor for fixed-point decimal data must be in the range 0 through p. If you omit q, the default value is zero. Do not use a scale factor for floating-point arithmetic.

Expressions x and y are converted to their derived type before the subtraction is performed.

For example:

```
SUBTRACTBIF: PROCEDURE OPTIONS (MAIN);

DECLARE X FIXED DECIMAL (8,3),
        Y FIXED DECIMAL (8,3),
        Z FIXED DECIMAL (9,3);

X=9500.374;
Y=2278.897;
Z = SUBTRACT (X,Y,9,3);

PUT SKIP LIST ('DIFFERENCE =',Z);

END;
```

This program prints:

```
DIFFERENCE = 7221.477
```

11.4.87 SUM

The SUM built-in function takes an array as an argument and returns the arithmetic sum of all the elements in the array. The array must have the FIXED or the FLOAT attribute. The format of the SUM built-in function is:

SUM(array-variable)

If the array has the attributes FIXED(p,q), the result will have the attributes FIXED(p,q). If the array has the attributes FLOAT(p), the result will also have the attributes FLOAT(p).

The result will have the same base attribute as the array, either DECIMAL or BINARY.

11.4.88 TAN

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)

11.4.89 TAND

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)

11.4.90 TANH

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)
```

11.4.91 TIME

The TIME built-in function returns an 8-character string representing the current time of day in the following form:

```
hhmmssxx
```

hh
The current hour (00-23)

mm
The minutes (00-59)

ss
The seconds (00-59)

xx
Hundredths of seconds (00-99)

The format of the TIME built-in function is:

```
TIME()
```

Returned Value

If TIME is used as a preprocessor built-in function, the time returned is the time when the program was compiled; otherwise the function returns the time at run time.

11.4.92 TRANSLATE

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. The format is:

```
TRANSLATE(original,translation[,old-chars])
```

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 old-chars.

old-chars
A character-string expression indicating which characters in the original are to be replaced. If old-chars is not specified, the default is COLLATE( ).

If the translation is shorter than old-chars, the translation is padded on the right with spaces to the length of old-chars before any translation occurs. If the translation is longer than old-chars, its excess characters (on the right) are ignored.

The following steps are performed for each character (beginning at the left) 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 old-chars for the leftmost occurrence of original(i).

3 If old-chars 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 old-chars, 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 <BIT_STRING>(1 2) to <BIT_STRING>(102). The second reference translates <BIT_STRING>(BFFLZFBUB) to <BIT_STRING>(BEELZEBUB). The third reference produces the following new sentence:

```
'the quick brown FOX jumps over the lazy DOG'
```

11.4.93 TRIM

The TRIM built-in function accepts a character string as an argument and returns a character string that consists of the input string with specified characters removed from the left and right. If you supply only one argument, TRIM removes blanks from the left and right of the argument. If you supply second and third arguments, TRIM removes characters specified by those arguments from the left and right of the string, respectively.

The format of the TRIM built-in function is:

TRIM (input-string,[beginning-chars,end-chars])

input-string

A character-string variable or constant. This argument supplies the string from which characters are to be trimmed.

beginning-chars

A character-string variable or constant. This argument specifies characters to be trimmed from the left of the input string. If a character that is in the first position in the input string is also present anywhere in beginning-chars, that character is removed from the input string. This process is repeated until a character is encountered on the left of the input string that is not present in beginning-chars, or until the characters in the input string are exhausted.

end-chars

A character-string variable or constant. This argument specifies characters to be trimmed from the right of the input string. The process of removing characters from the right is identical to that of removing characters from the left, except that the character in the last position is examined.

The TRIM built-in function accepts either one or three arguments. Any of the arguments can consist of a null string; specifically, if beginning-chars or end-chars is null, no characters are removed from the corresponding end of the input string.

When only one argument is supplied, TRIM removes blanks from both ends of that argument. In other words, the following two expressions are equivalent:

```
TRIM(S)
TRIM(S,' ',' ')
```

Returned Value

The returned value is a character string with characters removed from the ends.

Examples

The following examples illustrate the use of the TRIM built-in function.

Text	Returned String
TRIM ('ABC')	*'ABC'*
TRIM(' ABC')	*'ABC'*
TRIM(' ABC ')	*'ABC'*
TRIM('ABC ')	*'ABC'*
TRIM(' ABCDEF','','E')	*' ABCDEF'*
TRIM(' ABCDEF','','FE')	*' ABCD'*
TRIM(' ABCDEF ','ABC','EDF')	*' ABCDEF '*
TRIM('ABCDEF','CADB','FE')	*''*
TRIM(' ABCDEF ','ABC ',' EDF')	*''*
TRIM('AAAABCCXCCDDDDEFFFF','ABC','DF')	*'XCCDDDDE'*

11.4.94 TRUNC

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. The value has the following attributes:

$precision = min(31, p - q + 1)$

$scalefactor = 0$

$scalefactor = 0$

Here, p and q are the precision and scale factor of x.

11.4.95 UNSPEC

The UNSPEC built-in function returns a bit string representing the internal coded value of the referenced variable, or a specified part of that variable. The variable can be an aggregate or a scalar variable of any type. The format of the function is:

UNSPEC(reference[,position[,length]])

Returned Value

The returned value is a bit string whose length is the number of bits occupied by the referenced variable or by that part of the variable specified by the optional parameters, position and length. The length of the bit string must be less than or equal to the maximum length for bit-string data. The returned bit string contains the contents of the storage of the referenced variable (or the specified part of the variable), the first bit in storage being the first bit in the returned value. The actual value is specific to VAX hardware systems and Alpha hardware systems, 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.

Examples

```
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 <BIT_STRING>(A) and <BIT_STRING>(B). The PUT LIST statement containing UNSPEC(I) prints the following string:

```
'0100000000000000'B
```

11.4.96 VALID

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 <BIT_STRING>(0)B if x has an invalid value and <BIT_STRING>(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 CONVERSION 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.

Examples

```
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;
```

Asume that the file MASTER.DAT contains the following data:

```
 $15000.50
  50000.50
```

The program VALP will write out the following:

```
Invalid input:    50000.50
```

The picture <BIT_STRING>($$$$$$V.$$) 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.

11.4.97 VALUE

The VALUE built-in function is used to force a parameter to be passed by immediate value, rather than by whatever mechanism is specified by the declaration of the formal parameter.

The syntax of the function is:

$$\left\{ \begin{matrix} \text{VALUE} \\ \text{VAL} \end{matrix} \right\} \text{(expression)}$$

expression

An expression or scalar variable that is valid to be passed by value. It must fit into a longword (32 bits). The valid data types are:

FIXED BINARY (m) where m is less than or equal to 31
FLOAT BINARY (n) where n is less than or equal to 24
BIT (o) ALIGNED where o is less than or equal to 32
ENTRY
OFFSET
POINTER

Examples

```
DECLARE FOO ENTRY (ANY) EXTERNAL;
DECLARE X FIXED BINARY (31);
X = 15;
   .
   .
   .
CALL FOO(VALUE(X));
```

As with the REFERENCE and DESCRIPTOR built-in functions, VALUE is not designed for use with other PL/I procedures; it is intended for use only with routines written in languages other than PL/I.

11.4.98 VARIANT

The VARIANT preprocessor built-in function returns a string representing the value of the variant qualifier in the command that invoked the compilation.

The format in a preprocessor expression is:

VARIANT()

The /VARIANT qualifier permits specification of compilation variants. The value specified is available to the VARIANT preprocessor built-in function at compile time. The format of compilation variants is:

/VARIANT { [=alphanumeric-string] | [="alphanumeric-string"] }

For example, if a program is to be compiled with one of three different INCLUDE files, you can use the /VARIANT command qualifier to specify which file is to be included. In the following example, the file SPECIAL.SRC is included in the program only if /VARIANT=SPECIAL appears in the **pli** command line.

For example:

```
%IF VARIANT() = 'SPECIAL'
%THEN
    %INCLUDE 'SPECIAL.SRC';
%IF VARIANT() = 'NONE'
%THEN;
```

No action is taken if /VARIANT=NONE appears on the **pli** command line.

If /VARIANT is not specified, or if it is specified without a value, the default value is /VARIANT ="".

11.4.99 VERIFY

The VERIFY built-in function compares a string with a character-set string and verifies that all characters appearing in the string also appear in the character-set string. The function returns the value zero if they all appear. If not, the function returns a fixed-point binary integer that indicates the position of the first character in the string that is not present in the character-set string. The comparison is done character by character and left to right, and as soon as one nonmatching character is found in the first string, no more characters are compared. The function is case sensitive.

The format of the function is:

VERIFY(string,character-set-string[,starting-position])

string
A character-string expression representing the string to be checked.

character-set-string
A character-string expression containing the set of characters with which the characters in the first string are to be compared.

starting-position
A positive integer in the range 1 to n+1, where n is the length of the first string. It specifies the leftmost position in the first string to be compared with the character-set-string. (By default, the comparison starts at the left end of the first string.)

Examples

```
1   STRING = 'HOW MUCH IS 1 PLUS 2';
    ALPHABET = 'abcdefghijklmnopqrstuvwxyz
                ABCDEFGHIJKLMNOPQRSTUVWXYZ ';
    A = VERIFY(STRING,ALPHABET);
```

The value of the variable ALPHABET is a string containing the 26 lowercase letters, the 26 uppercase letters, and the space character. The function returns a value of 13, indicating the position of the character '1', which is the first nonalphabetic and nonspace character in STRING.

```
2   A = VERIFY(STRING,' ');
```

This example finds the first nonspace character in a string by using the space character as a test string. Note that constants can be used as the string parameters.

```
3   NEWSTRING = 'ALL LETTERS';
    A = VERIFY(NEWSTRING,ALPHABET);
```

VERIFY returns a value of zero because all characters in the string NEWSTRING are present in the string ALPHABET.

```
4    NEWSTRING = '9 LETTERS';
     A = VERIFY (NEWSTRING,ALPHABET,2);
```

The optional starting-position parameter specifies that the comparison begins at position 2 in NEWSTRING. VERIFY returns a value of zero because all characters beginning with the second character in the string NEWSTRING are present in the string ALPHABET. If the starting-position parameter had not been specified, VERIFY would have returned a value of 1, because the first character ('9') in NEWSTRING is not present in ALPHABET.

11.4.100 WARN

The WARN preprocessor built-in function returns the number of diagnostic warning messages issued during compilation up to that particular point in the source program. The format for the WARN built-in function is:

WARN();

The function returns a fixed result representing the number of compile-time warning messages that were issued up to the point at which the WARN built-in function was encountered.

11.5 Built-In Subroutines

Built-in subroutines are specific routines for the platform that provide various added capabilities. These routines can be used in a CALL statement. All arguments are evaluated as for normal subroutines.

The built-in subroutines are summarized in Table 11–2, according to the following functional categories:

- Condition-handling built-in subroutines assist in performing condition-related operations.
- File-control built-in subroutines perform file operations not supported by the standard PL/I language.
- Record-locking built-in subroutines allow extended record-locking control in conjunction with the OPTIONS clause of the READ statement.

For more information on the PL/I subroutines in Table 11–2, see the *Kednos PL/I for OpenVMS Systems User Manual*.

Table 11–2 Summary of PL/I Built-In Subroutines

Category	Routine Reference	Action
Condition-handling	RESIGNAL( )	Allows more processing of a signal.
File-control	DISPLAY(f,i)	Returns information on file f into i
	EXTEND(f,b)	Extends file f by b blocks
	FLUSH(f)	Forces all buffers for file f to be flushed

Table 11–2 (Cont.) Summary of PL/I Built-In Subroutines

Category	Routine Reference	Action
	NEXT_ VOLUME(f)	Performs magnetic tape volume processing on file f
	REWIND(f)	Resets file f to the beginning
	SPACE_ BLOCK(f,b)	Positions file f forward or backward b blocks
Record-locking	FREE(f)	Frees all locks for file f
	RELEASE(f,r)	Releases locked record r in file f

11.6 Pseudovariables

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 <BIT_STRING>(A) to a 1-character substring of S, beginning at the second character of S.

A pseudovariable can be used wherever the following three conditions are true:

- The syntax specifies a variable reference.
- The context is one that explicitly assigns a value to the variable.
- 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 preprocessor statements or in an argument list. In the following example, SUBSTR is not interpreted as a pseudovariable:

```
CALL P(SUBSTR(S,2,1));
```

Here, SUBSTR is interpreted as a built-in function reference, rather than as a pseudovariable. The actual argument passed to procedure P is a dummy argument containing the second character of string S.

The following are pseudovariables:

INT
ONSOURCE
ONCHAR
PAGENO
POSINT
STRING
SUBSTR

UNSPEC

The next sections describe these pseudovariables in alphabetic order.

11.6.1 INT Pseudovariable

The INT pseudovariable assigns a signed integer value to specified storage. The format is:

INT(reference[,position[,length]]) = expression;

reference

A reference to connected storage. This reference must not be an array, structure, or named constant. If position and length are not specified, the length of the referenced storage must not exceed 32 bits. If it exceeds 32 bits, a fatal run-time error results.

position

A positive integer value that denotes the position of the first bit in the field. If omitted, position defaults to 1, signifying the first bit of the storage denoted by reference. If specified, position must satisfy the following condition:

$1 <= position <= size(reference)$

where size(reference) is the length in bits of the storage denoted by reference. A position equal to size(reference) implies a zero-length field.

length

An integer value in the range 0 through 32 that specifies the length of the field. If omitted, length is the number of bits from the bit denoted by position through the end of the storage denoted by reference. If specified, length must satisfy the following condition:

$0 <= length <= size(reference) - position$

where size(reference) is the length in bits of the storage denoted by reference.

The INT pseudovariable is valid only in an assignment statement. You cannot use it as the target of an input statement or in other instances where pseudovariables are normally acceptable.

The expression to be assigned to the pseudovariable is first converted to the data type FIXED BINARY (31). Then, the internal representation of the resulting integer value is assigned to the storage specified by the arguments to INT. If the representation of the value is too large for assignment to the storage, the most significant bits of the integer are removed and no error is signaled.

Examples

```
DECLARE F FLOAT INITIAL (123.45);

     INT(F,8,8) = 25;     /* Alter the exponent */
     PUT SKIP LIST (F);   /* New value */
```

In this example, the INT pseudovariable is used to modify the exponent field of a floating-point variable. This example prints the following value:

```
9.5102418E-32
```

Proper interpretation of this result requires understanding of the internal representation of floating-point numbers. As such, this example is only valid on the VAX hardware.

The next example demonstrates how the INT pseudovariable treats cases in which the value is too large for the specified storage:

```
INTOVER: PROCEDURE OPTIONS (MAIN);

DECLARE I15 FIXED BINARY (15),
        I31 FIXED BINARY (31);

ON FIXEDOVERFLOW PUT SKIP LIST ('FIXEDOVERFLOW signaled');

    I31 = -876543;       /* Too big for I15 */

    I15 = I31;           /* Arithmetic assignment */
    INT(I15) = I31;      /* No error signaled */
    PUT SKIP LIST (I15);

    END;
```

This example produces the following output:

```
FIXEDOVERFLOW signaled
   -24575
```

The arithmetic assignment to I15 signals FIXEDOVERFLOW because the value of I31 is outside the range of a FIXED BINARY (15) variable. However, the assignment using the INT pseudovariable does not signal an error; it just copies the low-order 16 bits of the value of I31 into the storage for I15.

11.6.2 ONCHAR Pseudovariable

The ONCHAR pseudovariable can be used to replace the single character in the ONSOURCE value that caused a CONVERSION condition to be raised. An attempt to assign a value to the ONCHAR pseudovariable when there is no active CONVERSION condition causes the ERROR condition to be raised.

The format of the pseudovariable is:

ONCHAR()

See Section 8.10.4.4 for more information about CONVERSION condition name.

11.6.3 ONSOURCE Pseudovariable

The ONSOURCE pseudovariable can be used to replace the entire ONSOURCE value that caused a CONVERSION condition to be raised. An attempt to assign a value to the ONSOURCE pseudovariable when there is no active CONVERSION condition causes the ERROR condition to be raised.

The format of the pseudovariable is:

```
ONSOURCE()
```

The ONSOURCE value is a fixed-length string value. An assignment of a longer string is truncated, and an assignment of a shorter string is padded with blanks on the right to the necessary length.

See Section 8.10.4.4 for more information about CONVERSION condition name.

11.6.4 PAGENO Pseudovariable

The PAGENO pseudovariable refers to the page number of the referenced print file. Assignment to the pseudovariable modifies the current page number. The format of the PAGENO pseudovariable 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.

11.6.5 POSINT Pseudovariable

The POSINT pseudovariable assigns an integer value to specified storage. The format is:

```
POSINT(expression1[,position[,length]]) = expression2;
```

expression1
A reference to connected storage. This reference must not be an array, structure, or named constant. If position and length are not specified, the length of the referenced storage must not exceed 32 bits. (If it exceeds 32 bits, a fatal run-time error results.)

position
A positive integer value that denotes the position of the first bit in the field. If omitted, position defaults to 1, signifying the first bit of the storage denoted by expression1. If specified, position must satisfy the following condition:

$$1 <= position <= size(expression1)$$

Size(expression1) is the length in bits of the storage denoted by expression1. A position equal to size(expression1) implies a zero-length field.

length
An integer value in the range 0 through 32 that specifies the length of the field. If omitted, length is the number of bits from the bit denoted by position through the end of the storage denoted by expression1. If specified, length must satisfy the following condition:

$0 <= length <= size(expression1) - position$

Size(expression1) is the length in bits of the storage denoted by expression1.

expression2

Any expression that evaluates to an integer.

The POSINT pseudovariable is valid only in an assignment statement. It cannot be used as the target of an input statement or in other instances where pseudovariables are normally acceptable.

The expression to be assigned to the pseudovariable is first converted to the data type FIXED BINARY (31). Then, the internal representation of the resulting integer value is assigned to the storage specified by the arguments to POSINT. If the representation of the value is too large for assignment to the storage, the most significant bits of the integer are removed and no error is signaled.

The POSINT pseudovariable is identical in operation and use to the INT pseudovariable. For examples, see INT pseudovariable.

11.6.6 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 the aggregate 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 (see Section 5.5.6.1 and Section 5.8.2). 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 */
```

11.6.7 SUBSTR Pseudovariable

The SUBSTR pseudovariable refers to a substring of a specified string variable reference. 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 length must satisfy the following condition:

$1 <= position <= LENGTH(reference) + 1$

length

An integer expression that indicates the length of the substring. If not specified, length is:

$length = LENGTH(reference) - position + 1$

In other words, if length is not specified, the substring begins at the indicated position and ends at the end of the string. The length must satisfy the following condition:

$0 <= length <= LENGTH(reference) - position + 1$

Note that the following two lines are equivalent:

```
SUBSTR(r,p,l) = v;
r = SUBSTR(r,1,p-1)||SUBSTR(v||SUBSTR(r,p+length(v)),1,l)||SUBSTR(r,p+l);
```

Assignment to the SUBSTR pseudovariable does not change the length of reference.

Examples

```
DECLARE (NAME,NEW_NAME) CHARACTER(20) VARYING;

NAME = 'ISAK DINESEN';
NEW_NAME = NAME;
SUBSTR(NEW_NAME,4) = 'AC NEWTON';
/* NEW_NAME = 'ISAAC NEWTON' */
```

11.6.8 UNSPEC Pseudovariable

The UNSPEC pseudovariable interprets a reference to a scalar or aggregate element variable as a reference to a bit string. The format of the pseudovariable (in an assignment statement) is:

UNSPEC(reference[,position[,length]]) = expression;

reference

A reference to a scalar or aggregate 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) = expression;

the value of the expression 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.

To prevent zero-extending a value that is shorter than the variable, you can use the position parameter or both the position parameter and the length parameter. Then only the specified bits in the variable will be assigned a new value, and the other bits will remain as they were. Note that a position parameter of 1 refers to the low-order bit of the variable's storage, not the high-order bit.

Examples

```
DECLARE X FIXED BINARY (15);
UNSPEC(X) = '110'B;
```

The use of the constant <BIT_STRING>(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.

```
UNSPEC(X,1,3) = '101'B;
```

The optional parameters position and length are specified, causing the first three, low-order bits of the variable X to be assigned the value '101'B; the other bits are unaffected.

A Alphabetic Summary of Keywords

Table A–1 summarizes all of the keywords. This alphabetic summary includes both the options for the ENVIRONMENT attribute and the options for I/O statements.

Table A–1 PL/I Keywords

Keyword	Abbreviation	Use
A		Format item
ABS		Preprocessor built-in function, Built-in function
ACOS		Built-in function
%ACTIVATE		Preprocessor statement
ACTUALCOUNT		Built-in function
ADD		Built-in function
ADDR		Built-in function
ADDREL		Built-in function
ALIGNED		Attribute
ALLOCATE	ALLOC	Statement
ALLOCATION	ALLOCN	Built-in function
ANY		Attribute
ANYCONDITION		Condition name
APPEND		Environment option
AREA		Data attribute, Condition name
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
BACKUP_DATE		Environment option
BASED		Attribute
BATCH		Environment option
BEGIN		Statement

Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
BINARY	BIN	Data attribute, Built-in function
BIT		Data attribute, Built-in function
BLOCK_BOUNDARY_ FORMAT		Environment option
BLOCK_IO		Environment option
BLOCK_SIZE		Environment option
BOOL		Built-in function
BUCKET_SIZE		Environment option
BUILTIN		Attribute
BY		DO option
BYTE		Preprocessor built-in function, Built-in function
BYTESIZE		Built-in function
CALL		Statement
CANCEL_CONTROL_O		PUT OPTIONS option
CARRIAGE_RETURN_ FORMAT		Environment option
CEIL		Built-in function
CHARACTER	CHAR	Data attribute, Built-in function
CLOSE		Statement
COLLATE		Built-in function
COLUMN	COL	Format item
CONDITION	COND	Attribute, Condition name
CONTIGUOUS		Environment option
CONTIGUOUS_BEST_TRY		Environment option
CONTROLLED	CTL	Attribute
CONVERSION	CONV	Condition name
COPY		Preprocessor built-in function, Built-in function
COS		Built-in function
COSD		Built-in function
COSH		Built-in function
CREATION_DATE		Environment option
CURRENT_POSITION		Environment option
DATE		Preprocessor built-in function, Built-in function
DATETIME		Preprocessor built-in function, Built-in function
%DEACTIVATE		Preprocessor statement

Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
DECIMAL	DEC	Data attribute, Built-in function
%DECLARE	%DCL	Preprocessor statement
DECLARE	DCL	Statement
DECODE		Preprocessor built-in function, Built-in function
DEFAULT_FILE_NAME		Environment option
DEFERRED_WRITE		Environment option
DEFINED	DEF	Attribute
DELETE		Statement, Environment option
DESCRIPTOR	DESC	Attribute, Built-in function
%DICTIONARY		Preprocessor statement
DIMENSION	DIM	Attribute, Built-in function
DIRECT		File attribute, OPEN option
DISPLAY		Built-in subroutine
DIVIDE		Built-in function
%DO		Preprocessor statement
DO		Statement, GET and PUT I/O specifier
E		Format item
EDIT		GET option, PUT option
%ELSE		Keyword of the %IF statement
ELSE		Keyword of the IF statement
EMPTY		Built-in function
ENCODE		Preprocessor built-in function, Built-in function
%END		Preprocessor statement
END		Statement
ENDFILE		Condition name
ENDPAGE		Condition name
ENTRY		Statement, Attribute
ENVIRONMENT	ENV	File attribute, OPEN option, CLOSE option
%ERROR		Preprocessor statement
ERROR		Condition name, Preprocessor built-in function
EVERY		Built-in function
EXP		Built-in function
EXPIRATION_DATE		Environment option
EXTEND		Built-in subroutine
EXTENSION_SIZE		Environment option

Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
EXTERNAL	EXT	Attribute
F		Format item
FAST_DELETE		DELETE OPTIONS option
%FATAL		Preprocessor statement
FILE		Attribute, Option of the GET, PUT, READ, WRITE, DELETE, REWRITE, OPEN, and CLOSE statements
FILE_ID		Environment option
FILE_ID_TO		Environment option
FILE_SIZE		Environment option
FINISH		Condition name
FIXED		Data attribute, Built-in function
FIXEDOVERFLOW	FOFL	Condition name
FIXED_CONTROL_FROM		REWRITE OPTIONS option, WRITE OPTIONS option
FIXED_CONTROL_SIZE		Environment option
FIXED_CONTROL_SIZE_TO		Environment option
FIXED_CONTROL_TO		READ OPTIONS option
FIXED_LENGTH_RECORDS		Environment option
FLOAT		Data attribute, Built-in function
FLOOR		Built-in function
FLUSH		Built-in subroutine
FORMAT		Statement
FREE		Statement, Built-in subroutine
FROM		WRITE option, REWRITE option
GET		Statement
GLOBALDEF		Attribute
GLOBALREF		Attribute
%GOTO		Preprocessor statement
GOTO	GO TO	Statement
GROUP_PROTECTION		Environment option
HBOUND		Built-in function
HIGH		Built-in function
IDENT		PROCEDURE OPTIONS option
%IF		Preprocessor statement
IF		Statement
IGNORE_LINE_MARKS		Environment option
IN		ALLOCATE option, FREE option
%INCLUDE		Preprocessor statement

Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
INDEX		Preprocessor built-in function, Built-in function
INDEXED		Environment option
INDEX_NUMBER		DELETE OPTIONS option, READ OPTIONS option, REWRITE OPTIONS option, Environment option
%INFORM		Preprocessor statement
INFORM		Preprocessor built-in function
INITIAL	INIT	Attribute
INITIAL_FILL		Environment option
INPUT		File attribute, OPEN option
INT		Built-in function, Pseudovariable
INTERNAL	INT	Attribute
INTO		READ option
KEY		Condition name, READ option, DELETE option, REWRITE option
KEYED		File attribute, OPEN option
KEYFROM		WRITE option
KEYTO		READ option
LABEL		Attribute
LBOUND		Built-in function
LEAVE		Statement
LENGTH		Preprocessor built-in function, Built-in function
LIKE		Attribute
LINE		PUT option, Preprocessor built-in function, Format item
LINENO		Built-in function
LINESIZE		OPEN option
%LIST		Preprocessor statement
LIST		Attribute, GET option, PUT option
LOCK_ON_READ		READ OPTIONS option
LOCK_ON_WRITE		READ OPTIONS option
LOG		Built-in function
LOG10		Built-in function
LOG2		Built-in function
LOW		Built-in function
LTRIM		Preprocessor built-in function, Built-in function
MAIN		PROCEDURE OPTIONS option

Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
MANUAL_UNLOCKING		READ OPTIONS option
MATCH_GREATER		DELETE OPTIONS option, READ OPTIONS option, REWRITE OPTIONS option
MATCH_GREATER_EQUAL		DELETE OPTIONS option, READ OPTIONS option, REWRITE OPTIONS option
MATCH_NEXT		DELETE OPTIONS option, READ OPTIONS option, REWRITE OPTIONS option
MATCH_NEXT_EQUAL		DELETE OPTIONS option, READ OPTIONS option, REWRITE OPTIONS option
MAX		Preprocessor built-in function, Built-in function
MAXIMUM_RECORD_ NUMBER		Environment option
MAXIMUM_RECORD_SIZE		Environment option
MAXLENGTH		Built-in function
MEMBER		Attribute
MIN		Preprocessor built-in function, Built-in function
MOD		Preprocessor built-in function, Built-in function
MULTIBLOCK_COUNT		Environment option
MULTIBUFFER_COUNT		Environment option
MULTIPLY		Built-in function
NEXT_VOLUME		Built-in subroutine
%NOLIST		Preprocessor statement
NOLOCK		READ OPTIONS option
NONEXISTENT_RECORD		READ OPTIONS option
NONRECURSIVE		PROCEDURE option, ENTRY option
NONVARYING	NONVAR	Attribute
NORESCAN		Option of the %ACTIVATE statement
NO_ECHO		GET OPTIONS option
NO_FILTER		GET OPTIONS option
NO_SHARE		Environment option
NULL		Built-in function
OFFSET		Data attribute, Built-in function
ON		Statement
ONARGSLIST		Built-in function

Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
ONCHAR		Built-in function, Pseudovariable
ONCODE		Built-in function
ONFILE		Built-in function
ONKEY		Built-in function
ONSOURCE		Built-in function, Pseudovariable
OPEN		Statement
OPTIONAL		Attribute
OPTIONS		File attribute, Option of the GET, PUT, READ, WRITE, DELETE, REWRITE, and PROCEDURE statements
OTHERWISE	OTHER	Keyword of the SELECT statement
OUTPUT		File attribute, OPEN option
OVERFLOW	OFL	Condition name
OWNER_GROUP		Environment option
OWNER_ID		Environment option
OWNER_MEMBER		Environment option
OWNER_PROTECTION		Environment option
P		Format item
%PAGE		Preprocessor statement
PAGE		PUT option, Format item
PAGENO		Built-in function, Pseudovariable
PAGESIZE		OPEN option
PARAMETER	PARM	Attribute
PICTURE	PIC	Data attribute
POINTER	PTR	Data attribute, Built-in function
POSINT		Built-in function, Pseudovariable
POSITION	POS	Attribute
PRECISION	PREC	Attribute
PRESENT		Built-in function
PRINT		File attribute, OPEN option
PRINTER_FORMAT		Environment option
%PROCEDURE	%PROC	Preprocessor statement
PROCEDURE	PROC	Statement
PROD		Built-in function
PROMPT		GET OPTIONS option
PURGE_TYPE_AHEAD		GET OPTIONS option
PUT		Statement
R		Format item

Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
RANK		Preprocessor built-in function, Built-in function
READ		Statement
READONLY		Attribute
READ_AHEAD		Environment option
READ_CHECK		Environment option
READ_REGARDLESS		READ OPTIONS option
RECORD		File attribute, OPEN option
RECORD_ID		DELETE OPTIONS option, READ OPTIONS option, REWRITE OPTIONS option
RECORD_ID_ACCESS		Environment option
RECORD_ID_TO		READ OPTIONS option, REWRITE OPTIONS option, WRITE OPTIONS option
RECURSIVE		PROCEDURE option, ENTRY option
REFER		Attribute
REFERENCE		Attribute, Built-in function
RELEASE		Built-in subroutine
REPEAT		DO option
%REPLACE		Preprocessor statement
RESCAN		Option of the %ACTIVATE statement
RESIGNAL		Built-in subroutine
RETRIEVAL_POINTERS		Environment option
%RETURN		Preprocessor statement
RETURN		Statement
RETURNS		Entry attribute, PROCEDURE option, ENTRY option
REVERSE		Preprocessor built-in function, Built-in function
REVERT		Statement
REVISION_DATE		Environment option
REWIND		Built-in subroutine
REWIND_ON_CLOSE		Environment option
REWIND_ON_OPEN		Environment option
REWRITE		Statement
ROUND		Built-in function
RTRIM		Preprocessor built-in function, Built-in function
%SBTTL		Preprocessor statement

Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
SCALARVARYING		Environment option
SEARCH		Preprocessor built-in function, Built-in function
SELECT		Statement
SEQUENTIAL	SEQL	File attribute, OPEN option
SET		READ option, ALLOCATE option
SHARED_READ		Environment option
SHARED_WRITE		Environment option
SIGN		Preprocessor built-in function, Built-in function
SIGNAL		Statement
SIN		Built-in function
SIND		Built-in function
SINH		Built-in function
SIZE		Built-in function
SKIP		GET option, PUT option, Format item
SNAP		ON statement option
SOME		Built-in function
SPACEBLOCK		Built-in subroutine
SPOOL		Environment option
SQRT		Built-in function
STATEMENT		Option of the %PROCEDURE statement
STATIC		Attribute
STOP		Statement
STORAGE		Condition name
STREAM		File attribute, OPEN option
STRING		GET option, PUT option, Built-in function, Pseudovariable
STRINGRANGE	STRG	Condition name
STRUCTURE		Attribute
SUBSCRIPTRANGE	SUBRG	Condition name
SUBSTR		Preprocessor built-in function, Built-in function, Pseudovariable
SUBTRACT		Built-in function
SUM		Built-in function
SUPERSEDE		Environment option
SYSIN		Default input file
SYSPRINT		Default output file

Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
SYSTEM		ON statement option
SYSTEM_PROTECTION		Environment option
TAB		Format item
TAN		Built-in function
TAND		Built-in function
TANH		Built-in function
TEMPORARY		Environment option
%THEN		Keyword of the %IF statement
THEN		Keyword of the IF statement
TIME		Built-in function
TIMEOUT_PERIOD		READ OPTIONS option
%TITLE		Preprocessor statement
TITLE		OPEN option
TO		DO option
TRANSLATE		Preprocessor built-in function, Built-in function
TRIM		Preprocessor built-in function, Built-in function
TRUNC		Built-in function
TRUNCATE		Attribute, Environment option
UNALIGNED	UNAL	Attribute
UNDEFINEDFILE	UNDF	Condition name
UNDERFLOW	UFL	Condition name, PROCEDURE OPTIONS option
UNION		Attribute
UNSPEC		Built-in function, Pseudovariable
UNTIL		DO option
UPDATE		File attribute, OPEN option
USER_OPEN		Environment option
VALID		Built-in function
VALUE	VAL	Attribute, Built-in function
VARIABLE		Attribute, OPTIONS option
VARIANT		Preprocessor built-in function
VARYING	VAR	Attribute
VAXCONDITION		Condition name
VERIFY		Preprocessor built-in function, Built-in function
WAIT_FOR_RECORD		READ OPTIONS option
%WARN		Preprocessor statement

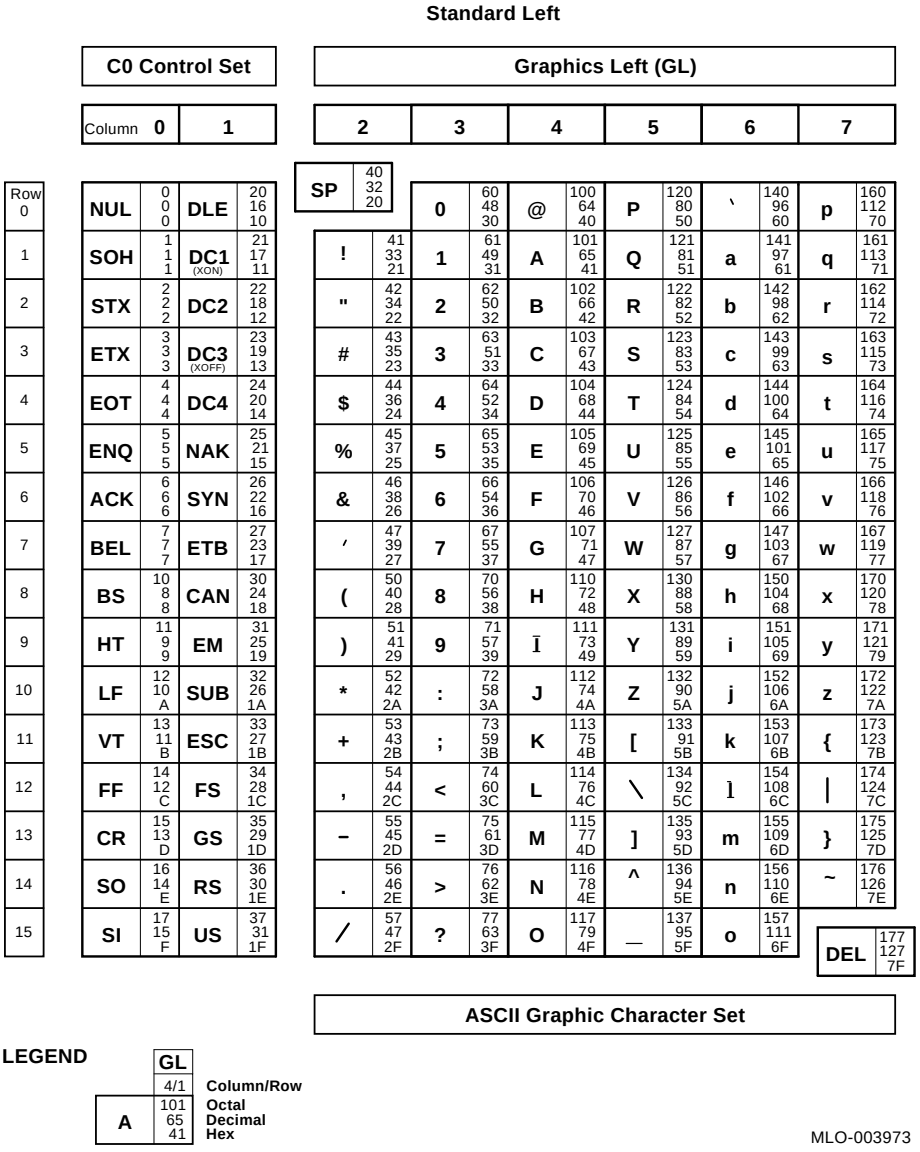
Table A–1 (Cont.) PL/I Keywords

Keyword	Abbreviation	Use
WARN		Preprocessor built-in function
WHEN		Keyword of the SELECT statement
WHILE		DO option
WORLD_PROTECTION		Environment option
WRITE		Statement
WRITE_BEHIND		Environment option
WRITE_CHECK		Environment option
X		Format item
ZERODIVIDE	ZDIV	Condition name

B Digital Multinational Character Set

The Digital Multinational Character Set is a set of 8-bit numeric values representing the alphabet, numerals, punctuation, and other symbols. The first 128 characters of the set (with decimal values from 0 through 127) are the American Standard Code for Information Interchange (ASCII) characters. The remaining characters (with values from 128 through 255) are non-ASCII characters and can be used only in string constants and data with I/O statements.

The following table shows the first half of the Digital Multinational Character Set, which is the ASCII character set. The first half of each of the numbered columns identifies the character as you would enter it on a VT200 or VT100 series terminal or as you would see it on a printer (except for the nonprintable characters). The remaining half of each column identifies the character by the binary value of the byte; the value is stated in three radixes—octal, decimal, and hexadecimal. For example, the uppercase letter A has, under ASCII conventions, a storage value of hexadecimal 41 (a bit configuration of 01000001), equivalent to 101 in octal notation and 65 in decimal notation.



The following table shows the second half of the Digital Multinational Character Set (the non-ASCII characters, with decimal values 128 through 255). The first half of each of the numbered columns identifies the character as you would see it on a VT200 series terminal or printer; these characters cannot be output on a VT100 series terminal.

Standard Right

	C1 Control Set				Graphics Right (GR)											
Column	8		9		10		11		12		13		14		15	
Row 0		200 128 80	DCS	220 144 90		240 160 A0	°	260 176 B0	À	300 192 C0		320 208 D0	à	340 224 E0		360 240 F0
1		201 129 81	PU1	221 145 91	¡	241 161 A1	±	261 177 B1	Á	301 193 C1	Ñ	321 209 D1	á	341 225 E1	ñ	361 241 F1
2	BPH	202 130 82	PU2	222 146 92	¢	242 162 A2	2	262 178 B2	Â	302 194 C2	Ò	322 210 D2	â	342 226 E2	ò	362 242 F2
3	NBH	203 131 83	STS	223 147 93	£	243 163 A3	3	263 179 B3	Ã	303 195 C3	Ó	323 211 D3	ã	343 227 E3	ó	363 243 F3
4	IND	204 132 84	CCH	224 148 94		244 164 A4		264 180 B4	Ä	304 196 C4	Ô	324 212 D4	ä	344 228 E4	ô	364 244 F4
5	NEL	205 133 85	MW	225 149 95	¥	245 165 A5	µ	265 181 B5	Å	305 197 C5	Õ	325 213 D5	å	345 229 E5	õ	365 245 F5
6	SSA	206 134 86	SPA	226 150 96		246 166 A6	¶	266 182 B6	Æ	306 198 C6	Ö	326 214 D6	æ	346 230 E6	ö	366 246 F6
7	ESA	207 135 87	EPA	227 151 97	§	247 167 A7	•	267 183 B7	Ç	307 199 C7	Œ	327 215 D7	ç	347 231 E7	œ	367 247 F7
8	HTS	210 136 88	SOS	230 152 98	¤	250 168 A8		270 184 B8	È	310 200 C8	Ø	330 216 D8	è	350 232 E8	ø	370 248 F8
9	HTJ	211 137 89		231 153 99	©	251 169 A9	1	271 185 B9	É	311 201 C9	Ù	331 217 D9	é	351 233 E9	ù	371 249 F9
10	VTS	212 138 8A	SCI	232 154 9A	ª	252 170 AA	º	272 186 BA	Ê	312 202 CA	Ú	332 218 DA	ê	352 234 EA	ú	372 250 FA
11	PLD	213 139 8B	CSI	233 155 9B	«	253 171 AB	»	273 187 BB	Ë	313 203 CB	Û	333 219 DB	ë	353 235 EB	û	373 251 FB
12	PLU	214 140 8C	ST	234 156 9C		254 172 AC	¼	274 188 BC	Ì	314 204 CC	Ü	334 220 DC	ì	354 236 EC	ü	374 252 FC
13	RI	215 141 8D	OSC	235 157 9D		255 173 AD	½	275 189 BD	Í	315 205 CD	Ý	335 221 DD	í	355 237 ED	ý	375 253 FD
14	SS2	216 142 8E	PM	236 158 9E		256 174 AE		276 190 BE	Î	316 206 CE		336 222 DE	î	356 238 EE		376 254 FE
15	SS3	217 143 8F	APC	237 159 9F		257 175 AF	¿	277 191 BF	Ï	317 207 CF	ß	337 223 DF	ï	357 239 EF		377 255 FF

DEC Supplemental Graphic Character Set

LEGEND

GR
12/1
Á
301
193
C1

Column/Row
Octal
Decimal
Hex

MLO-003974

C Compatibility with PL/I Standards

This appendix describes the relationship between Kednos PL/I for OpenVMS VAX and and Kednos PL/I for OpenVMS Alpha to each other and to the various PL/I language standards that are currently in force.

Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha are strict supersets of the ANSI X3.74-1981 PL/I General Purpose Subset. Both contain many features from larger or more recent PL/I standards and implementations. Most of the features implemented in Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha that go beyond the language defined by ANSI X3.74-1981 are contained in either the ANSI X3.53-1976 (full) PL/I language standard or the new ANSI X3.74-198x PL/I General Purpose Subset.

C.1 Differences and Similarities Between Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha

- Kednos PL/I for OpenVMS VAX and Alpha both support floating-point variables and constants up to 53 bits of precision. Attempts to declare variables with greater precision result in a compilation error on VAX. On Alpha, such attempts result in a warning that a value of 53 has been used. On both OpenVMS VAX and OpenVMS Alpha, using a constant with more than 15 decimal digits results in a warning indicating possible loss of precision.
- In both Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha, the default condition-handling capabilities of a module without OPTIONS(MAIN) are not correct.
- Neither the VAX nor the Alpha Run-Time Library (RTL) signals ENDPAGE for SYSPRINT.
- Kednos PL/I for OpenVMS VAX supports the OPTIONS(UNDERFLOW) option and checks for UNDERFLOW. Kednos PL/I for OpenVMS Alpha ignores OPTIONS(UNDERFLOW).

C.2 Relation to the 1981 PL/I General-Purpose Subset

The 1981 PL/I General-Purpose Subset (ANSI X3.74-1981) 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 the following:

- 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 ANSI X3.74-1981 standard.

C.2.1 Program Structure

The General-Purpose 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.

C.2.2 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 can contain unlabeled THEN and ELSE clauses.

An ON statement can specify a single condition. The condition names supported are ERROR, ENDFILE, ENDPAGE, FIXEDOVERFLOW, KEY, OVERFLOW, UNDEFINEDFILE, UNDERFLOW, and ZERODIVIDE.

C.2.3 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 can be initialized.

The ALLOCATE statement with the SET option and the FREE statement are included in the subset.

C.2.4 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.

C.2.5 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 as follows: 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.

C.2.6 Built-In Functions and Pseudovariables

The built-in functions in the subset are as follows: 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.

C.2.7 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.

C.3 198x PL/I General-Purpose Subset Features Supported

The 198x PL/I General-Purpose Subset (ANSI X3.74-198x) was designed to extend the previous subset standard on the basis of experience with subset implementations and the desire for more capabilities in subset-conforming implementations.

The following sections describe features in this standard that have been implemented to date in Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha.

C.3.1 Lexical Constructs

The character pair /* is permitted within comments.

Both uppercase and lowercase characters are permitted in source programs.

No space is required after the P for picture constants.

C.3.2 Program Control

RETURNS(CHAR(*)) is supported.

The statements following THEN and ELSE can be labeled.

The NONRECURSIVE procedure option is supported.

The SELECT statement is supported.

The LEAVE statement is supported.

The UNTIL clause for DO groups and clauses is supported.

C.3.3 Storage Control

The IN option can be used for the ALLOCATE and FREE statements, and language controlled allocation in areas is supported.

The SET option is optional for ALLOCATE if the based variable being allocated was declared with a base pointer.

The ALLOCATE and FREE statements can specify a comma list of items.

String assignment can have the source and target overlapped.

C.3.4 Input/Output

Expressions can be used in GET and PUT FORMAT lists.

You can 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.

The OPEN and CLOSE statements can contain a list of file specifications.

The FROM option of the REWRITE statement can be omitted.

C.3.5 Attributes and Pictures

The INITIAL attribute is allowed with AUTOMATIC storage. The initial items can contain asterisks to denote uninitialized values. The initial values can be expressions. The NULL built-in function can be used in both STATIC and AUTOMATIC INITIAL attributes. The initial iteration factor can be an asterisk.

Restricted expressions can be used for static extents, parameter extents, and returns descriptor extents.

The AREA and OFFSET data types are supported.

The REFER attribute can be used at the end of a structure.

The DIMENSION, PARAMETER, and NONVARYING keywords can be specified.

The UNALIGNED attribute can be specified, but only for BIT and CHARACTER variables.

SYSIN and SYSPRINT can be contextually declared as files.

The CONDITION attribute is supported.

The UNION attribute is supported.

C.3.6 Program Control

The AREA, CONDITION, CONVERSION, FINISH, and STORAGE conditions are supported.

Multiple conditions can be specified for ON and REVERT.

The SNAP and SYSTEM options of the ON statement are supported.

C.3.7 Built-In Functions and Pseudovariables

The following built-in functions are supported: ADD, EMPTY, EVERY, HIGH, LOW, MULTIPLY, OFFSET, ONSOURCE, POINTER, PROD, REVERSE, SOME, SUBTRACT, and SUM.

The ONSOURCE pseudovariable is supported.

The DIMENSION, HBOUND, and LBOUND built-in functions have a default of one for the second parameter if it is not specified.

The INDEX and VERIFY built-in functions have an optional starting position parameter.

The UNSPEC built-in function and pseudovariable can be used on aggregates.

C.3.8 Expressions

The operators AND THEN (short-circuiting AND, specified as &:) and OR ELSE (short-circuiting OR, specified as |:) are supported.

EXCLUSIVE OR (infix or dyadic ^) is supported.

C.4 Full PL/I Features Supported

The items discussed in this section are features that are explicitly excluded from both the old subset standard (ANSI X3.74-1981) and the new subset standard (ANSI X3.74-198x) but that have been implemented in Kednos PL/I. These features all exist in full PL/I.

C.4.1 Program Structure

The STRINGRANGE and SUBSCRIPTRANGE conditions are supported.

Replication factors for string constants are supported.

A comma list can be specified on the left-hand side of an assignment statement.

C.4.2 Program Control

The ENTRY statement is supported.

C.4.3 Storage Control

CONTROLLED storage is supported.

C.4.4 Attributes and Pictures

The CONTROLLED, LIKE, MEMBER, POSITION, PRECISION, REFER, and STRUCTURE attributes are supported. (The REFER attribute is restricted to BASED and CONTROLLED variables.)

The picture characters Y, T, I, and R are supported, and pictures can include iteration factors.

Scaled fixed binary numbers are supported. They can have a scale factor within the range -31 through 31 on OpenVMS VAX systems or the range -63 through 63 on OpenVMS Alpha systems.

C.4.5 Built-In Functions and Pseudovariables

The OFFSET and POINTER built-in functions are not restricted to ADDR.

The ALLOCATION and ONCHAR built-in functions are supported.

The ONCHAR pseudovariable is supported.

C.4.6 Expressions

The expression in a WHILE or UNTIL clause or in an IF statement can 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 1 and in a false value if all bits in the string expression are 0s.

The control variable and the expressions in the TO, BY, and REPEAT options of the DO statement are not restricted to integers and pointers.

C.5 Nonstandard Features from Other Implementations

The features discussed in this section are not described in any ANSI PL/I standard. They are, however, provided by some other implementations.

C.5.1 Preprocessor

Both Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha support an embedded lexical preprocessor for compilation control. The following preprocessor statements are included: %ACTIVATE, %DEACTIVATE, %DECLARE, %DICTIONARY %DO, %END, %ERROR, %FATAL, %GOTO, %INFORM, %IF, %LIST, %NOLIST, %PAGE, %PROCEDURE, %REPLACE, %RETURN, %SBTTL, %TITLE, and %WARN.

An %IF statement can contain unlabeled %THEN and %ELSE clauses.

The following preprocessor built-in functions are included: ABS, BYTE, COPY, DATE, DATETIME, DECODE, ENCODE, ERROR, INDEX, INFORM, LENGTH, LINE, LTRIM, MAX, MIN, MOD, RANK, REVERSE, RTRIM (Kednos PL/I for OpenVMS VAX only), SEARCH, SIGN, SUBSTR, TIME, TRANSLATE, TRIM, VARIANT, VERIFY, and WARN.

C.5.2 Built-In Functions

The following nonstandard built-in functions are included: ACTUALCOUNT, ADDREL, BYTE, BYTESIZE, DATETIME, DECODE, DESCRIPTOR, ENCODE, INT, LTRIM, MAXLENGTH, ONARGLIST, POSINT, PRESENT, RANK, REFERENCE, RTRIM, SEARCH, SIZE, TRIM, and VALUE.

C.5.3 LIKE Extension

Both Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha allow you to use the LIKE attribute on a structure already containing LIKE.

C.5.4 Declarations

Variables can be declared outside procedures.

C.6 PL/I-Specific Extensions for OpenVMS VAX and OpenVMS Alpha Platforms

The extensions in the following sections are enhancements for PL/I programs executing on the OpenVMS VAX and OpenVMS Alpha platforms. These extensions are provided for procedure calling, condition handling, support of OpenVMS Record Management Services (RMS), compilation control, and miscellaneous purposes.

C.6.1 Procedure-Calling and Condition-Handling Extensions

The following extensions to PL/I were made to allow procedures to call procedures written in any other programming language that also supports the OpenVMS calling standard.

- The ANY, VALUE, REFERENCE, and DESCRIPTOR attributes describe how data is to be passed to a called procedure.
- The OPTIONAL attribute indicates that a parameter need not be specified in a call; and the TRUNCATE attribute indicates the point at which an actual parameter list can be truncated.
- The LIST attribute can be used for the parameter descriptor in an external entry declaration to denote that a list of parameters may be specified.
- The ACTUALCOUNT built-in function returns the number of parameters the current procedure was called with. The PRESENT built-in function determines whether a parameter was specified in a call.
- 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.
- The VALUE, REFERENCE, and DESCRIPTOR built-in functions can be used to pass an argument by the specified mechanism to a non-PL/I procedure.

The following attributes provide storage classes for PL/I variables. These attributes permit PL/I programs to take advantage of features of the OpenVMS Linker and to combine PL/I procedures with other procedures that use these storage classes.

- 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.
- The READONLY attribute can be applied to a static computational variable whose value does not change.
- The VALUE attribute defines a variable that is, in effect, a constant whose value is supplied by the linker. The value attribute can also be used to allow a procedure to receive constants passed by immediate value.

The following extensions to condition handling provide support for condition handling in the OpenVMS environment:

- The ANYCONDITION condition name can be used in an ON-unit to handle any condition that is signaled that does not explicitly have an ON-unit of its own.
- The VAXCONDITION condition name can be used in ON, SIGNAL, and REVERT statements to process conditions specific to OpenVMS systems.

- The RESIGNAL built-in subroutine permits an ON-unit to keep a signal active.
- The ONARGSLIST built-in function provides an ON-unit with access to the mechanism and signal arguments of a system-specific exception condition.

C.6.2 Support of OpenVMS Record Management Services

The options of the ENVIRONMENT attribute provide support for many of the features and control values of the OpenVMS Record Management Services (RMS). Additional extensions have been made to the PL/I language to augment this support, as follows:

- The USER_OPEN ENVIRONMENT option allows access to the RMS FAB and RAB control structures during a PL/I file OPEN operation.
- The OPTIONS option is supported on the GET, PUT, READ, WRITE, REWRITE, and DELETE statements.
- The FLUSH, FREE, RELEASE, and REWIND built-in subroutines provide file handling and control functions. PL/I provides these additional built-in subroutines: DISPLAY, EXTEND, NEXT_VOLUME, and SPACEBLOCK.

C.6.3 Miscellaneous Extensions

PL/I supports the OpenVMS Common Data Dictionary (CDD). Data definitions are included in source programs with the %DICTIONARY statement.

The following built-in functions are supported for BYTE, DECODE, ENCODE, INT, POSINT, RANK, and SIZE.

C.7 Implementation-Defined Values and Features

The following values and features are implementation-defined:

- PL/I supports the full 256-character DEC Multinational Character Set (a superset of ASCII) for CHARACTER data (including character string constants in PL/I source programs). All identifiers in a source program are restricted to the ASCII character set.
- The default precisions for arithmetic data are as follows:

  FIXED BINARY (31)
  FIXED DECIMAL (10)
  FLOAT BINARY (24)
  FLOAT DECIMAL (7)

- The maximum record size for SEQUENTIAL files is 32767 bytes minus the length of any fixed-length control area.
- The maximum key size is 255 bytes for character keys.

- 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.

- The default value for the PAGESIZE option is as follows:
  - If the logical name SYS$LP_LINES is defined, the default page size is 6, the numeric value of SYS$LP_LINES.
  - 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.
- The values for TAB positions are columns beginning with column 1 and every eight columns thereafter.
- The maximum length allowed for a file title is 128 characters.
- The maximum number of digits in editing fixed-point data is 34.
- The maximum numbers of digits for each combination of base and scale are as follows:

  FIXED BINARY - 31
  FIXED DECIMAL - 31
  FLOAT BINARY - 113 for OpenVMS VAX and 53 for OpenVMS Alpha
  FLOAT DECIMAL - 34 for OpenVMS VAX and 15 for OpenVMS Alpha
- The maximum length of CHARACTER, CHARACTER VARYING, and BIT strings is 32767.
- The default precision for integer values is 31.
- The maximum number of arguments that can be passed to an entry point is 253.
- The second parameter of the F format item (the optional parameter specifying the number of fractional digits in the stream representation) must have a value less than or equal to 31.

D Migration Notes

This appendix contains notes and comments about migration issues. It lists and describes keywords and functions of the PL/I language that are available in other implementations of PL/I but are not included in Kednos PL/I. It also outlines the differences between Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha.

The information in this appendix is not intended to represent either a complete or a formal description of migration issues. The information is presented informally and has not been subjected to extensive testing and verification.

D.1 Keywords Not Supported

Table D–1 lists keywords used in other implementations of PL/I but not in Kednos PL/I. The table does not include ENVIRONMENT keywords or implementation-specific language extensions.

Table D–1 PL/I Keywords Not Supported

Keyword	Abbreviation	Use
AFTER		Built-in function
ALL		Built-in function
ANY		Built-in function
ATTENTION	ATTN	Condition
BACKWARDS		Attribute, Option of OPEN statement
BEFORE		Built-in function
BUFFERED	BUF	Attribute, Option of OPEN statement
BY NAME		Option of assignment statement
C		Format item
CALL		Option of INITIAL attribute
CASE		Option of DO statement
CHECK		Statement, Condition, Condition prefix
COMPLETION	CPLN	Built-in function, Pseudovariable
COMPLEX	CPLX	Attribute, Built-in function, Pseudovariable
CONJG		Built-in function
CONNECTED	CONN	Attribute
CONSTANT		Attribute
CONVERSION	CONV	Condition prefix
COPY		Option of GET statement
COUNT		Built-in function

Table D–1 (Cont.) PL/I Keywords Not Supported

Keyword	Abbreviation	Use
CURRENTSTORAGE	CSTG	Built-in function
DATA		Stream I/O transmission mode
DATAFIELD		Built-in function
DECAT		Built-in function
DEFAULT	DFT	Statement
DELAY		Statement
DESCRIPTORS		Option of DEFAULT statement
DISPLAY		Statement
DOT		Built-in function
ERF		Built-in function
ERFC		Built-in function
EVENT		Attribute, Option of several statements
EXCLUSIVE	EXCL	Attribute
EXIT		Statement
FETCH		Statement
FORMAT		Attribute
GENERIC		Attribute
HALT		Statement
IGNORE		Option of READ statement
IMAG		Built-in function, pseudovariable
IRREDUCIBLE	IRRED	Attribute
LIST		Option of OPEN statement
LOCAL		Attribute
LOCATE		Statement
NAME		Condition
NOCHECK		Statement, Condition prefix
NOCONVERSION	NOCONV	Condition prefix
NOFIXEDOVERFLOW	NOFOFL	Condition prefix
NOLOCK		Option of READ statement
NONE		Option of DEFAULT statement
NOOVERFLOW		Condition prefix
NOSIZE		Condition prefix
NOSTRINGRANGE	NOSTRG	Condition prefix
NOSTRINGSIZE	NOSTRZ	Condition prefix
NOSUBSCRIPTRANGE	NOSUBRG	Condition prefix
NOZERODIVIDE	NOZDIV	Condition prefix
ONCOUNT		Built-in function
ONFIELD		Built-in function

Table D–1 (Cont.) PL/I Keywords Not Supported

Keyword	Abbreviation	Use
ONLOC		Built-in function
ORDER		Option of BEGIN and PROCEDURE statements
OVERFLOW	OFL	Condition prefix
PENDING		Condition
POLY		Built-in function
PRIORITY		Option of CALL statement, Built-in function, Pseudovariable
RANGE		Option of DEFAULT statement
REAL		Attribute, Built-in function, Pseudovariable
RECORD		Condition
REDUCIBLE	RED	Attribute
REENTRANT		Option of OPTIONS option
RELEASE		Statement
REORDER		Option of BEGIN and PROCEDURE statements
REPEAT		Built-in function
REPLY		Option of DISPLAY statement
SAMEKEY		Built-in function
SIZE		Condition, Condition prefix
SNAP		Option of PUT statement
STATUS		Built-in function, Pseudovariable
STORAGE	STG	Built-in function
STRINGRANGE	STRG	Condition prefix
STRINGSIZE	STRZ	Condition, Condition prefix
SUB		Dummy variable of DEFINED attribute
SUBSCRIPTRANGE	SUBRG	Condition prefix
SYSTEM		Option of DECLARE statement
TAB		Option of OPEN statement
TASK		Attribute, Option of OPTIONS option, Option of CALL statement
TRANSIENT		Attribute, option of OPEN statement
TRANSMIT		Condition
UNBUFFERED	UNBUF	Attribute, Option of OPEN statement
UNLOCK		Statement
WAIT		Statement

D.2 Differences Between Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha

This section lists the differences between Kednos PL/I for OpenVMS VAX, Kednos PL/I for OpenVMS Alpha, and other PL/I compilers that require you to modify your source files to avoid compilation errors. In some cases, differences require reprogramming.

- The at sign (@) and number sign (#) characters are not allowed in identifiers; thus, you must change all identifiers that contain either of these characters.
- You must explicitly declare all names (except internal procedure and label constants) There is no *I through N rule* that provides an implicit declaration of FIXED BINARY to undeclared names. In fact, the compiler defaults all undeclared names to FIXED BINARY and issues a warning message to that effect.
- If the attribute FLOAT is specified and neither of the attributes BINARY or DECIMAL is specified with it, the compiler provides a default of BINARY. This should not present a problem; however, if a precision was specified under the assumption that the default floating-point base was DECIMAL, overflow conditions can result if you do not correct the declaration.
- You cannot explicitly declare internal entry constants and subscripted label constants; these names are implicitly declared by their appearance. You must remove the declarations from the source file.
- You must reprogram ON-units for unsupported conditions. For example, ON-units for SIZE, RECORD, and TRANSMIT should be modified so that they are invoked for the ERROR condition, which is the condition that is signaled for any of these errors.
- You cannot specify the ALIGNED and UNALIGNED attributes for a structure in which string variables are to be aligned or unaligned. The attribute must be specified in the declaration of each variable that is to be aligned.
- You cannot pass parameters directly to main procedures from the command stream. For examples of techniques for passing values or data to a main procedure, see the *Kednos PL/I for OpenVMS Systems User Manual*.
- Changes in virtual memory requirements

  The virtual memory requirements for Kednos PL/I may be extensive according to the types of PL/I language constructs that you use. Processing the PUT and GET statements, BEGIN blocks and PROCEDURE blocks with the Kednos PL/I for OpenVMS Alpha compiler tends to require more virtual memory than the same operation would if you were using the Kednos PL/I for OpenVMS VAX compiler.

Heavy usage of these constructs in your programs may exhaust your user limits for virtual memory. The typical compiler errors that you receive are:

```
%PLIG-F-TEXT, Compiler abort - virtual memory limits exceeded.
%SYSTEM-F-ABORT, abort
%LIB-F-INSVIRMEM, insufficient virtual memory
```

You can recover by increasing your user page-file quota (PGFLQUOTA). See the *Kednos PL/I for OpenVMS Alpha Installation Guide* or the appropriate OpenVMS Alpha system manual for more information on increasing your user page-file quota.

- Changes in behavior for calls to bound procedures

  With Kednos PL/I, nested procedures can be called by other routines using the PL/I language's entry variable. However, in order to successfully reach these bound-procedure routines and ensure proper execution of uplevel referenced variables within the bound procedures, the user must keep the parent invocation of the bound procedure active.

  This ensures that the stack references to uplevel variables are still valid.

  The same situation exists with Kednos PL/I for OpenVMS Alpha. However, you may experience differences in behavior between Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha if you do not keep the parent invocation active. This is because bound procedures are not created in the same way on the two compilers.

  Kednos PL/I for OpenVMS Alpha creates the bound procedure values on the stack of the parent of the bound procedure. The compiler sends an access violation message or may exhibit other undefined behavior when the parent invocation of the bound procedure is not kept active. Because the parent's stack has been destroyed, the bound procedure code found on the parent's stack may also have been destroyed or corrupted, and the user may not be able to reach the bound procedure.

  With Kednos PL/I for OpenVMS VAX you may receive the same undefined behavior. Cases may exist in which you are able to reach the bound procedure; however, you may still receive errors in results from the uplevel referenced variables within the bound procedure. This is because the bound procedure code on the parents stack may have been corrupted.

  The following example shows behavior where the bound procedure entry6a cannot be reached by Kednos PL/I for OpenVMS Alpha, because the parent procedure, entry6, has exited and its stack is free to be recovered and used by other procedures. Therefore, the bound procedure code has been destroyed.

```
program: proc options(main);
```

```
program: proc options(main);
dcl (ent2) entry variable;
dcl fnc entry returns(entry) variable;
dcl evar char (25) var init(' ');

  fnc=entry6;
  ent2=fnc();
  call destroy_stack();
  call ent2;

  put skip list('Evar is =>',evar);

  entry6: proc returns (entry);
          return(entry6a);

          entry6a: proc;
                  evar=evar||'entry6a*';
                  return;
          end entry6a;
  end entry6;

  destroy_stack: proc;

          /* Declare enough space to destroy previous
           * stack values before this call.
           */
          dcl temp_space char(1000);

          temp_space = 'hello';
  end;
end program;
```

Note that this program also fails with Kednos PL/I for OpenVMS VAX. However, it fails after having reached the bound procedure entry6a and while trying to access the uplevel referenced variables from the parent, entry6.

- Changes in behavior for overlapping static storage initialization

  In both Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha, overlapping initialization of static storage is not supported. When Kednos PL/I for OpenVMS VAX finds an overlapping static initialization, the compiler uses the value of the last found initialization as the value for the static variable.

  In Kednos PL/I for OpenVMS Alpha, however, you cannot depend on the value of the last found initialization being assigned to the static variable. This behavior may affect the following user constructs:

  — Changes in behavior for PL/I external variable initialization

    If a PL/I external variable is declared with the attributes EXTERNAL STATIC INITIAL, all blocks that declare the variable MUST be initialized with the same value.

    Because Kednos PL/I for OpenVMS VAX allowed overlapping static storage initialization, you could specify different initial values for the SAME external variable within containing blocks. The last initial value encountered by the Kednos PL/I for OpenVMS VAX compiler was the value of the external variable.

In Kednos PL/I for OpenVMS Alpha this rule is more strictly enforced: if the user specifies an initial value in the declaration of a PL/I external variable, the same initial value declaration must be specified at each occurrence of the declaration of the external variable in all blocks that declare the external variable. Failure to do so can cause unpredictable results. The following examples show correct and incorrect declarations:

Correct Declaration of an External Variable Initialization

```
/* A CORRECT declaration of an external
 * variable initialization.
 */
program: proc options(main);

     dcl test fixed external initial(5);

     p:proc;

         dcl test external initial(5);

         /* The value of this variable should be 5. */

         put skip list ('The value of test is =>',test);

     end;
end;
```

Incorrect Declaration of an External Variable Initialization

```
/* INCORRECT declarations of external
 * variable initialization.
 */
program: proc options(main);

     /* The initialization MUST be the same
      * for all declarations of the external
      * variable.
      */
     dcl test fixed external initial(5);
     dcl test1 fixed external initial(5);
     dcl test2 fixed external;

     p:proc;

         /* The initialization MUST be the same
          * for all declarations of the external
          * variable.
          */
         dcl test external initial(6);
         dcl test1 external;
         dcl test2 external initial(6);

         /* The value of these variables is unpredictable. */
         put skip list ('The value of test is =>',test);
         put skip list ('The value of test1 is =>',test1);
         put skip list ('The value of test2 is =>',test2);

     end;
end;
```

Note: As no guarantee exists as to which occurrence of the external variable the Kednos PL/I for OpenVMS Alpha compiler processes first, each occurrence of the variable must contain the same initial value.

- Changes in behavior for the SUBSTR built-in function

  In Kednos PL/I for OpenVMS VAX, the SUBSTR function is defined only when the position attribute (the integer expression that indicates the position of the first bit or character in the substring) is greater than or equal to 1.

  Similarly, in Kednos PL/I for OpenVMS Alpha, the SUBSTR function is defined only when the position of the first bit or character is greater than or equal to 1. Therefore, if the position of the first bit or character to the SUBSTR built-in function is less than 1 (that is, the SUBSTR BIF is undefined), you may observe different results with Kednos PL/I for OpenVMS Alpha than with Kednos PL/I for OpenVMS VAX.

- Changes in underflow detection behavior for Kednos PL/I for OpenVMS Alpha

  The Kednos PL/I for OpenVMS VAX compiler outputs underflow detection code, the Kednos PL/I for OpenVMS Alpha compiler does not.

- Changes for null statements and label behavior in Kednos PL/I for OpenVMS Alpha

  Kednos PL/I for OpenVMS VAX does not compare the multiple labels of many succeeding combinations of labels and null statements so that they will have the same address. However, Kednos PL/I for OpenVMS Alpha optimizes these label-null statement sequences and compares the multiple labels so that they result in the same address. The following example shows this difference:

```
X67:      PROC options(main);

A:
; /* Null statement. */
B:
; /* Null statement. */
C:
; /* Null statement. */

IF A=C THEN
     PUT LIST('NULL STATEMENT LABELS COMPARE EQUAL FOR Kednos PL/I for OpenVMS Al
     ELSE
     PUT LIST('NULL STATEMENT LABELS COMPARE NOT EQUAL FOR Kednos PL/I for OpenVM

D:   IF C=D THEN
     PUT LIST('ERROR THESE LABELS SHOULD NEVER COMPARE EQUAL');
END;
```

- In Kednos PL/I for OpenVMS Alpha you can continue when a GOTO statement with the OTHERWISE option is used to go to an undefined label array element within a BEGIN-END block. Kednos PL/I for OpenVMS VAX does not support this function. The following example works with Kednos PL/I for OpenVMS Alpha but not Kednos PL/I for OpenVMS VAX:

```
program: procedure options(main);
            dcl i fixed binary(31,0);
            begin;
```

```
            i = 2;
            goto part(i) otherwise;
            put skip list('At continue !!');
              end;
part(1):
  end program;
```

- Run-time exception handling is more consistent when processing formats of GET EDITs and PUT EDITs in Kednos PL/I for OpenVMS Alpha than it is in Kednos PL/I for OpenVMS VAX. If Kednos PL/I for OpenVMS Alpha encounters an exception when processing a format such as PLI$_INVFMTPARM, a signal is raised. If the exception is handled, then processing continues with the next format item regardless of the nature of the signal and the format item that was being processed when the exception was detected. This is not always the case with Kednos PL/I for OpenVMS VAX.
- Nonlocal returns work properly under Kednos PL/I for OpenVMS Alpha. A nonlocal return is a RETURN statement that is lexically nested between any number of BEGIN/END pairs; for example:

```
nested_proc : proc returns (fixed);
    begin;
        begin;
            begin;
                begin;
                    begin;
                        return (5);
                    end;
                    put skip list ('shouldnt be here 1');
                end;
                put skip list ('shouldnt be here 2');
            end;
            put skip list ('shouldnt be here 3');
        end;
        put skip list ('shouldnt be here 4');
    end;
    put skip list ('shouldnt be here 5');
end;

nested_proc2 : proc returns (float);
    begin;
        begin;
            begin;
                begin;
                    begin;
                        return (5.1);
                    end;
                    put skip list ('shouldnt be here 1');
                end;
                put skip list ('shouldnt be here 2');
            end;
            put skip list ('shouldnt be here 3');
        end;
        put skip list ('shouldnt be here 4');
    end;
    put skip list ('shouldnt be here 5');
end;
```

```
program: proc options(main);
    dcl result1 fixed;
    dcl result2 float;

    result1 = nested_proc();
    result2 = nested_proc2();

    put skip list (result1);
    put skip list (result2);
end;
```

Two special considerations should be observed for nonlocal returns from OPTIONS(MAIN) procedures. First, an ON unit declared in the scope of one of the BEGIN/END pairs for FINISH will be given a chance to execute. Second, an ON unit declared in the scope of one of the BEGIN/END pairs for VAXCONDITION(ss$_unwind) will NOT be given a chance to execute with the previous construct.

- When a program terminates abnormally due to an unhandled exception, Kednos PL/I for OpenVMS Alpha closes all open files before turning control over to the OpenVMS last-chance condition handler (the utility that prints the error and traceback).

  This means that I/O that was being held in a buffer is allowed to reach its destination before the error dump appears on the screen. This is not the case with Kednos PL/I for OpenVMS VAX, in which I/O held in a buffer is not allowed to reach its destination before the error dump appears on the screen.

- Differences in exception handling between OpenVMS Alpha and OpenVMS VAX

  Because machine instructions on OpenVMS Alpha systems differ from those on OpenVMS VAX systems, any exception handler written to handle VAX hardware exceptions should be examined to ensure that it handles Alpha hardware exceptions similarly. For example, consider a case in which an exception handler has been written to handle an access violation. You may expect different behavior upon normal completion of the handler. In this case the handler should always perform a nonlocal GOTO to exit the handler so that program execution continues in a predictable way.

- Fixed-decimal precision differences between Kednos PL/I for OpenVMS Alpha and Kednos PL/I for OpenVMS VAX

  The precision specified for a PL/I fixed-decimal data type must be in the range of 1 to 31 for Kednos PL/I for OpenVMS Alpha. Kednos PL/I for OpenVMS VAX allows a fixed-point decimal variable to be declared with a precision of zero and also allows built-in functions to specify a fixed-decimal precision of zero. Kednos PL/I for OpenVMS Alpha does not allow zero to be used in either of these situations and issues an error "FIXDPRECZERO" when the precision specified for a fixed decimal is zero.

- Differences in behavior between OpenVMS VAX and OpenVMS Alpha architectures regarding PL/I error conditions

  In general, any PL/I operation that overflows on OpenVMS VAX systems will also overflow with Kednos PL/I for OpenVMS Alpha on OpenVMS Alpha systems. However, the Alpha hardware does not include support for packed decimal instructions that correspond to the PL/I fixed decimal data type; data items of this type are handled on OpenVMS Alpha systems through run-time calls, either to Kednos PL/I for OpenVMS Alpha run-time library routines or to system OTS routines. These emulation routines perform many operations to compute the result of a fixed decimal operation, which in most cases can be done with a single VAX instruction. Any one of these many emulation operations can and will generate an overflow.

  Therefore, Kednos PL/I for OpenVMS Alpha can guarantee at least one overflow only on OpenVMS Alpha systems for every overflow on OpenVMS VAX systems per PL/I statement. Kednos PL/I for OpenVMS Alpha cannot guarantee that the resulting behavior or value produced by a PL/I statement that produces an overflow condition will be the same value or behavior as it was on Kednos PL/I for OpenVMS VAX.

  The following PL/I example shows the difference in overflow detection between Kednos PL/I for OpenVMS VAX and Kednos PL/I for OpenVMS Alpha. This difference occurs when a PL/I fixed-decimal item with precision of 31 and scale factor of 21 [fixed decimal(31,21)] is converted to a PL/I fixed binary item with precision 31 and scale of 30 [fixed binary(31,30)]. On OpenVMS VAX systems this overflow situation results in two overflow conditions being raised. On OpenVMS Alpha systems this situation results in one overflow condition being raised.

  Note that all Kednos PL/I for OpenVMS VAX cases of overflow are detected on OpenVMS Alpha systems here. However, in this case Kednos PL/I for OpenVMS Alpha detects one overflow for the two overflows reported by Kednos PL/I for OpenVMS VAX. This difference is due to a difference in the instruction set between OpenVMS VAX and OpenVMS Alpha systems and a further explanation is detailed below.

The selected program fragment contained here shows two examples of this situation. In each case the fixed-decimal item is converted to a fixed-binary item by a series of three steps.

1 Multiplies the fixed decimal(31,21) item by the decimal representation of 2**30.

2 Shifts the fixed decimal (31,51) created by step 1 right by 21.

3 Converts the fixed decimal (31,30) created by step 2 to a fixed binary (31,30).

The VAX macro instructions output by Kednos PL/I for OpenVMS VAX to perform this conversion are:

```
 23    1        fixb30 = fixd21;
A0 AD 1F C4 AD 1F 00000000* EF 0A 25    0125      mulp    #10,PLI$B_PAC_2_POWER_30,#31,-60(fp),#31,-96(fp)
       90 AD 1F 00 A0 AD 1F EB 8F F8    0132      ashp    #-21,#31,-96(fp),#0,#31,-112(fp)
          54 90 AD 1F 36    013C      cvtpl   #31,-112(fp),r4
             B0 AD 54 D0    0141      movl    r4,-80(fp)
```

In this example fragment, overflows are detected during the VAX *mulp* instruction which causes an overflow to occur and during the VAX *cvtpl* instruction. The OpenVMS Alpha instruction set does not contain decimal instructions, so the OpenVMS VAX decimal instructions are emulated by a series of OpenVMS Alpha instructions and OTS calls. During the instructions generated on by OpenVMS Alpha systems by Kednos PL/I for OpenVMS Alpha to emulate the OpenVMS VAX *mulp* instruction, an overflow is correctly detected. During the instructions to convert packed decimal to integer, an overflow is not detected, however.

Note that after a fixed-overflow condition has been raised, the value resulting from an operation that causes this condition is undefined. In this case, the value from the result of the multiply that caused an overflow is undefined. Therefore, when it is used in the expression no guarantee exists that an overflow will be raised again. This is what is happening when the result of the overflow is shifted right and then converted from decimal to integer. Therefore, in this case it is reasonable to expect a difference in the number of overflows detected from one PL/I statement.

Due to the difference in OpenVMS VAX and OpenVMS Alpha systems instructions, we can not prevent this situation from occurring. If you notice a situation during a conversion in which you receive one overflow on OpenVMS Alpha systems but two on OpenVMS VAX systems this is likely to be the reason.

In general, on a per-statement basis Kednos PL/I for OpenVMS Alpha can be expected to detect overflow, but the number of overflows detected per statement cannot be guaranteed to be the same on OpenVMS VAX and OpenVMS Alpha systems. The following complete example shows the difference:

```
program: procedure options(main);

dcl fixb30  fixed bin(31,30);
dcl fixd18  fixed decimal(31,18);
dcl fixd21  fixed decimal(31,21);
dcl fixd22  fixed decimal(31,22);
dcl fixd24  fixed decimal(31,24);

on fixedoverflow begin;
  put skip list('fixed overflow occurred');
end;

fixd18 = 18.36;
fixd22 = 22.40;

fixd21 = fixd18+fixd22;
fixb30 = fixd21;

fixd18 = 18.42;
fixd24 = 24.58;

fixd21 = fixd24+fixd18;
fixb30 = fixd21;

end;
```

D.3 Implicit Conversions

The Kednos PL/I compilers issue warning-level messages when they perform implicit conversions between arithmetic and string data types and between bit-string and character-string data types. They issue these messages for all such conversions, not just those excluded by the PL/I General-Purpose Subset.

You can avoid the messages by compiling your programs with the /NOWARNINGS qualifier. Otherwise, you can edit your program, locate the occurrences of implied conversions, and change them to explicit conversions, as follows:

- Arithmetic to character-string-use the CHARACTER built-in function.
- Arithmetic to bit-string-use the BIT built-in function.[1]
- Bit-string to arithmetic-use the BINARY built-in function.
- Bit-string to character string-use the CHARACTER built-in function.
- Character-string to arithmetic-use the BINARY, DECIMAL, FIXED, or FLOAT built-in function, according to the target data type.
- Character-string to bit string-use the BIT built-in function.

[1] This conversion is based on the way bit strings are printed by PUT LIST (the first bit of the string is the high-order bit if the printed string is viewed as a binary integer) rather than being based on the internal representation (the first bit of the string is then in the low-order digit position in memory).

D.4 Printing a Hexadecimal Memory Dump

Dump printing routines written for other hardware architectures are not transportable to OpenVMS VAX and OpenVMS Alpha systems. Because the order in which bits are stored on OpenVMS machines is reversed on some other machines, these routines must be entirely rewritten. The program HEXDUMP that follows shows one technique for outputting the contents of memory in hexadecimal:

```
/*
  This procedure illustrates the dumping of memory in
  hexadecimal. The output format is consistent with other
  OpenVMS VAX or OpenVMS Alpha memory dump utilities.
*/

HEXDUMP: PROCEDURE OPTIONS(MAIN);
    DECLARE DUMP_LOCATION POINTER;
    DECLARE (I,J) FIXED BINARY(31);
/* declare and initialize fake memory to dump */
    DECLARE MEMORY(0:255) FIXED BINARY(7);
    DO I = 0 TO 127;
        MEMORY(I) = I;
        MEMORY(I + 128) = I - 128;
        END;
/* dump the pseudomemory on the user's terminal */
    DO I = 0 TO 255 BY 16;
        PUT SKIP;
    DO J = 12 TO 0 BY -4;
    DUMP_LOCATION = ADDR(MEMORY(I+J));
    CALL OUTPUT_HEX(DUMP_LOCATION);
    END;
        PUT EDIT(' ')(A(1));
        CALL OUTPUT_HEX(ADDR(I));
        END;
    STOP;
/* subroutine to output a hexadecimal longword */
OUTPUT_HEX: PROCEDURE(ADDRESS);
    DECLARE ADDRESS POINTER;
    DECLARE F FIXED BIN(31) BASED(ADDRESS);

    PUT EDIT(REVERSE(UNSPEC(F))) (B4(8));

    END OUTPUT_HEX;

END HEXDUMP;
```

E Language Summary

This appendix briefly describes PL/I statements, attributes, expressions, data conversions, built-in functions, pseudovariables, and built-in subroutines.

E.1 Statements

%activate-statement

% { ACTIVATE / ACT } element [RESCAN / NORESCAN] , . . . ;

allocate-statement

{ ALLOCATE / ALLOC } allocate-item, . . . ;

allocate-item:

variable-reference [SET(locator-reference)][IN(area-reference)]

%assignment-statement

%target = expression;

assignment-statement

target, . . . = expression;

begin-statement

BEGIN;

call-statement

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

close-statement

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

%deactivate-statement

% { DEACTIVATE / DEACT } element, . . . ;

element:

```
{ identifier          }
{ (identifier, . . . ) }
```

%declare-statement

```
% { DECLARE } element [ FIXED     ] , . . . ;
  { DCL     }        [ CHARACTER ]
                     [ BIT       ]
```

element:

```
{ identifier          }
{ (identifier, . . . ) }
```

declare-statement

```
{ DECLARE } [level] declaration [,[level] declaration, . . . ];
{ DCL     }
```

declaration:

[level] declaration-item

declaration-item:

```
{ identifier                }[(bound-pair, . . . )] [attribute . . . ]
{ (declaration-item, . . . ) }
```

delete-statement

DELETE FILE(file-reference) [KEY (expression)][OPTIONS(option, . . .)]

%dictionary-statement

%DICTIONARY cdd-path;

%do-statement

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

do-statement

```
     [reference=expression]
     [TO expression [BY expression]]
DO   [REPEAT expression]
     [WHILE(expression)]
     [UNTIL(expression)];
```

%end-statement

```
%END;
```

end-statement

```
END [label-reference];
```

entry-statement

```
entry-name: ENTRY [ (parameter, . . . ) ]
                  [ RECURSIVE    ]
                  [ NONRECURSIVE ]
                [ RETURNS (returns-descriptor) ];
```

%error-statement

```
%ERROR preprocessor-expression;
```

%fatal-statement

```
%FATAL preprocessor-expression;
```

format-statement

```
label:
FORMAT (format-specification, . . . );
```

free-statement

```
FREE variable-reference [IN area-reference], . . . ;
```

get-statement

```
GET EDIT (input-target, . . . )(format-specification, . . . )
  [ FILE(file-reference)          ]
  [     [SKIP[(expression)]]      ]
  [     [OPTIONS(option, . . . )] ]
  [ STRING(expression)            ]
    ;
GET LIST (input-target, . . . )
  [ FILE(file-reference)          ]
  [     [SKIP[(expression)]]      ]
  [     [OPTIONS(option, . . . )] ]
  [ STRING(expression)            ]
    ;
GET [FILE(file-reference)] SKIP [(expression)];
```

%goto-statement

```
%GOTO label-reference;
```

goto-statement

```
{ GOTO    }
{ GO TO   } label-reference ;
```

%if-statement

%IF test-expression %THEN action [%ELSE action];

if-statement

IF test-expression THEN action [ELSE action];

%include-statement

```
%INCLUDE { 'file-spec'                      };
         { module-name                      }
         { 'library-name(module-name)'      }
```

%inform-statement

%INFORM preprocessor-expression;

leave-statement

LEAVE [label-reference];

%[no]list-statement

%[NO]LIST;
%[NO]LIST_ALL;
%[NO]LIST_DICTIONARY;
%[NO]LIST_INCLUDE;
%[NO]LIST_MACHINE;
%[NO]LIST_SOURCE;

%null-statement

%;

null-statement

;

on-statement

```
ON condition-name, . . . [SNAP] { on-unit  }
                               { SYSTEM;   }
```

open-statement

```
OPEN FILE(file-reference) [file-description-attribute . . . ]
        [,FILE(file-reference) [file-description-attribute . . . ]] . . .
```

%page-statement

```
%PAGE;
```

%procedure-statement

```
%label: { PROCEDURE } [(parameter-identifier, . . .
        { PROC      }
)][STATEMENT]
     RETURNS ( { CHARACTER } );
               { FIXED     }
               { BIT       }
   .
   .
   .
[%]RETURN (preprocessor-expression);
   .
   .
   .
[%]END.
```

procedure-statement

```
entry-name: { PROCEDURE } [ (parameter, . . . ) ]
            { PROC      }
          [ OPTIONS (option, . . . ) ]
            [ RECURSIVE    ]
            [ NONRECURSIVE ]
          [ RETURNS (value-descriptor) ];
```

put-statement

```
PUT EDIT (output-source, . . . ) (format-specification, . . . )
  [ FILE(file-reference)          ]
  [      [PAGE]                   ]
  [      [LINE(expression)]       ]
  [      [SKIP[(expression)]]     ]
  [      [OPTIONS(option)]        ]
  [ STRING(reference)             ]
    ;
PUT [FILE(file-reference)] LINE(expression);
PUT LIST (output-source, . . . )
```

```
[ FILE(file-reference)          ]
[     [PAGE]                    ]
[     [LINE(expression)]        ]
[     [SKIP[(expression)]]      ]
[     [OPTIONS(option)]         ]
[ STRING(reference)             ]
  ;
PUT [ FILE(file-reference)] PAGE;
PUT [ FILE(file-reference)] SKIP [(expression)];
```

read-statement

```
READ FILE (file-reference)

{ INTO (variable-reference) }
{ SET (pointer-variable)    }

    [ KEY (expression)              ]
    [ KEYTO (variable-reference)    ]

      [ OPTIONS (option, . . . ) ];
```

%replace-statement

```
%REPLACE identifier BY constant-value;
```

%return-statement

```
[%]RETURN (preprocessor-expression);
```

return-statement

```
RETURN [ (return-value) ];
```

revert-statement

```
REVERT  condition-name, . . .  ;
```

rewrite-statement

```
REWRITE FILE (file-reference)
    [ FROM (variable-reference) [ KEY (expression) ]]
    [ OPTIONS (option, . . . ) ];
```

%sbttl-statement

```
%SBTTL preprocessor-expression
```

select-statement

```
SELECT [(select-expression)];
    [WHEN [ANY | ALL] (expression, . . . ) [action];] . . .
    [{OTHERWISE | OTHER} [action];]
      END;
```

signal-statement

SIGNAL condition-name;

stop-statement

STOP;

%title-statement

%TITLE preprocessor-expression

%warn-statement

%WARN preprocessor-expression;

write-statement

```
WRITE FILE(file-reference) FROM (variable-reference)
                [ KEYFROM (expression) ]
                [ OPTIONS (option, . . . ) ];
```

E.2 Attributes

Computational Data Type Attributes

The following attributes define arithmetic and string data:

```
CHARACTER [ (length) ] [ VARYING    ]
                       [ NONVARYING ]
BIT [ (length) ] [ ALIGNED   ]
                 [ UNALIGNED ]
{ FLOAT } { BINARY  } [ [PRECISION] (precision
{ FIXED } { DECIMAL }                [,scale-factor]) ]
PICTURE 'picture'
```

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

Noncomputational Data Type Attributes

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

AREA
CONDITION
ENTRY [VARIABLE]
FILE [VARIABLE]
LABEL [VARIABLE]
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)][INITIAL(initial-element, . . .)]
CONTROLLED [INITIAL(initial-element, . . .)]
DEFINED(variable-reference) [POSITION(expression)]
STATIC [READONLY] [INITIAL(initial-element, . . .)]
PARAMETER
INTERNAL

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

Member Attributes

The following attributes can be applied to the major or minor members of a structure:

LIKE
MEMBER
REFER
STRUCTURE
TYPE
UNION

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)]
   [RETURNS (returns-descriptor)]
BUILTIN
```

Non-Data Type Attributes

The following attributes can be applied to data declarations:

```
ALIGNED
DIMENSION
UNALIGNED
```

E.3 Expressions and Data Conversions

The following table lists the categories of operators, their symbols, and their meanings.

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	^ (prefix)	Logical NOT
	&	Logical AND
	|	Logical OR
	&:	Logical AND THEN
	|:	Logical OR ELSE
	^ (infix)	Logical EXCLUSIVE OR
Concatenation operator	||	String concatenation

Note: For any of the operators, the tilde character (~) can be used instead of a circumflex (^), and an exclamation point (!) can be used instead of a vertical bar (|).

The following table gives the priority of PL/I operators. Low numbers indicate high priority. For example, the exponentiation operator (**) has the highest priority (1), so it is performed first, and the OR ELSE operator (|:) has the lowest priority (9), so it is performed last.

Precedence of Operators

Operator	Priority	Left/Right Associative	Order of Evaluation
()	0	N/A	deepest first
**	1	right	left to right
+ (prefix)	1	N/A	N/A
- (prefix)	1	N/A	N/A
^ (prefix)	1	N/A	N/A
*	2	left	left to right
/	2	left	left to right

Operator	Priority	Left/Right Associative	Order of Evaluation
+ (infix)	3	left	left to right
- (infix)	3	left	left to right
||	4	left	left to right
>	5	left	left to right
<	5	left	left to right
^>	5	left	left to right
^<	5	left	left to right
=	5	left	left to right
^=	5	left	left to right
<=	5	left	left to right
>=	5	left	left to right
&	6	left	left to right
|	7	left	left to right
^ (infix)	7	left	left to right
&:	8	left	left to right across entire expression
|:	9	left	left to right across entire expression

The following table discusses the contexts in which PL/I performs data conversion.

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.

Context	Conversion Performed
x + y x - y x * y x / y x**y x||y x & y x | y x&:y x|:y x ^ y x > y x < y x = y x^=y	In any expression, if operands do not have the required data type, they are converted to a common data type before the operation. For most operators, the data types of all operands must be identical. A warning message is issued in the case of a concatenation conversion.
BINARY (expression) BIT (expression) CHARACTER (expression) DECIMAL (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) SKIP (expression) LINE (expression) COLUMN (expression) format items A, B, E, F, and X TAB (expression)	Values specified for various options to PL/I statements must be converted to integer values.
DO control-variable . . .	Values are converted to the attributes of the control variable.
parameter	Actual parameters are converted to the type of the formal parameter if necessary.
INITIAL attribute	Initial values are converted to the type of the variable being initialized.

E.4 Pseudovariables

PL/I has the following pseudovariables:

INT
ONSOURCE
ONCHAR
PAGENO
POSINT

STRING
SUBSTR
UNSPEC

A pseudovariable can be used, in certain assignment contexts, in place of an ordinary variable reference. For example:

```
SUBSTR(S,2,1) = 'A';
```

This assigns the character <BIT_STRING>(A) to a 1-character substring of S, beginning at the second character of S.

A pseudovariable can be used wherever the following three conditions are true:

- The syntax specifies a variable reference.
- A value is explicitly assigned to the variable.
- The context does not require the variable to be addressable.

Pseudovariables are used most often in the following locations:

- The left side of an assignment statement
- The input target of a GET statement

Note that a pseudovariable cannot be used in preprocessor statements or 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.

E.5 Built-In Subroutines

The following table summarizes the file-handling built-in subroutines.

Summary of File-Handling Built-In Subroutines

Subroutine	Function
DISPLAY	Returns information about a file.
EXTEND	Allocates additional disk blocks for a file.
FLUSH	Requests the file system to write all buffers onto disk to preserve the current status of a file.
FREE	Unlocks all the locked records in a file.
NEXT_VOLUME	Begins processing the next volume in a multivolume tape set.
RELEASE	Unlocks a specified record in a file.
REWIND	Positions a file at its beginning or at a specific record.
SPACEBLOCK	Positions a file forward or backward a specified number of blocks.

PL/I also has the condition-handling subroutine RESIGNAL. This subroutine allows an ON-unit to *pass* on a condition signal and causes the condition to be resignaled for handling by a different ON-unit.

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