digital

Introduction to VAX–11 PL/I

Order No. AA-H950A-TE

VAX11

July 1980

Provides an overview of the VAX-11 PL/I programming language and its implementation and operation on the VAX/VMS operating system.

Introduction to VAX-11 PL/I

Order No. AA-H950A-TE

digital equipment corporation · maynard, massachusetts

First Printing, July 1980

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

6/80-15

Contents

Page

Preface vii

Chapter 1 PL/I Concepts

1.1 Program Structure and Statement Syntax . . . 1–1

1.1.1 Statements . . . 1–1
1.1.2 Blocks . . . 1–3

1.2 Data Declarations and Data Types . . . 1–3

1.2.1 Computational Data Types . . . 1–4
1.2.2 Noncomputational Data Types . . . 1–6
1.2.3 The Data Types of Constants . . . 1–6
1.2.4 Automatic and Static Storage for Data . . . 1–8
1.2.5 Internal and External Variables . . . 1–8

1.3 Aggregates . . . 1–9

1.3.1 Arrays . . . 1–9
1.3.2 Structures . . . 1–10

1.4 Assignment and Expressions . . . 1–12

1.4.1 Expressions . . . 1–12
1.4.2 Conversion of Data . . . 1–12
1.4.3 Operators . . . 1–13
1.4.4 Aggregate Assignments . . . 1–15

1.5 Input/Output Concepts . . . 1–16

1.5.1 Types of Input/Output . . . 1–16
1.5.2 Attributes of Files . . . 1–17
1.5.3 Associating File Constants with External Files . . . 1–17

1.6 Stream Input/Output . . . 1–18

1.6.1 List-Directed Stream I/O . . . 1–18
1.6.2 Edit-Directed Stream I/O . . . 1–19
1.6.3 Declaration of Stream Files . . . 1–20
1.6.4 Print Files . . . 1–21

1.7 Program Control . . . 1–21

1.7.1 IF Statement . . . 1–21
1.7.2 DO Statement . . . 1–22
1.7.3 GOTO Statement . . . 1–24

1.8 Procedures . . . 1–24

1.8.1 Internal Procedures . . . 1–25
1.8.2 Parameters and Arguments . . . 1–26
1.8.3 Subroutines and Functions . . . 1–27
1.8.4 Declaring External Procedures . . . 1–28
1.8.5 Entry Points . . . 1–29
1.8.6 Recursion . . . 1–29
1.8.7 Built-In Functions . . . 1–29
1.8.8 Terminating Procedures . . . 1–30

1.9 Condition Handling 1–31

1.9.1 Condition Names 1–31
1.9.2 The Execution of ON-Units 1–32
1.9.3 The ONCODE Built-In Function 1–33

1.10 Record Input/Output. 1–33

1.10.1 File Organizations 1–34
1.10.2 Access Modes 1–35
1.10.3 Declaring and Opening Record Files 1–35
1.10.4 Sequential Access. 1–35
1.10.5 Random Access. 1–36
1.10.6 Error Handling 1–36

1.11 Storage Allocation and Control 1–37

1.11.1 Locator-Qualified References 1–38
1.11.2 Pointers 1–39
1.11.3 Areas and Offsets. 1–40
1.11.4 Defined Variables. 1–41

Chapter 2 VAX–11 Extensions to PL/I

2.1 Extensions for Input/Output Processing 2–1

2.1.1 Indexed Sequential Files 2–2
2.1.2 Relative Files. 2–2
2.1.3 File Disposition Options for Closing a File 2–3
2.1.4 File Ownership and Protection. 2–3
2.1.5 Terminal I/O 2–4
2.1.6 Fixed-Control Files 2–4
2.1.7 Magnetic Tape File Processing 2–5
2.1.8 Block I/O 2–6
2.1.9 Record Id Access 2–6

2.2 Extensions for Calling and Condition Handling 2–6

2.2.1 Argument-Passing Mechanisms 2–7
2.2.2 Variable-Length Argument Lists. 2–8
2.2.3 Global Symbols. 2–8
2.2.4 New Storage Classes 2–9
2.2.5 VAXCONDITION and ANYCONDITION 2–10
2.2.6 Resignaling. 2–10
2.2.7 ONCODE and ONARGSLIST. 2–11

Chapter 3 Using PL/I in the VAX/VMS Environment

3.1 Sample Terminal Session. 3–1
3.2 The File System 3–10

3.2.1 Subdirectories 3–11
3.2.2 Logical Names 3–12

3.3 The VAX/VMS Librarian 3–13

3.3.1 INCLUDE Files and Libraries. 3–13
3.3.2 Object Module Libraries 3–14

3.4 Command Procedures 3–15

VAX–11 PL/I Language Summary

Glossary

Index

Figures

1 Documentation for VAX–11 PL/I Programmers viii
1–1 Block Activations 1–3
1–2 Internal Variables 1–8
1–3 Static External Variables. 1–9
1–4 An Internal Procedure 1–25
1–5 List Processing in PL/I. 1–40
3–1 Creating a Directory Hierarchy 3–11
3–2 Creating and Using an INCLUDE File Library 3–14

Tables

1–1 Computational Constants 1–7
1–2 Summary of ON Conditions 1–32

Preface

Manual Objectives

This manual provides introductory information on the VAX-11 PL/I programming language. This language is an implementation of the PL/I General-Purpose (G) Subset, defined by the proposed ANSI standard BSR X3.74, with extensions to support the execution of PL/I programs in the VAX/VMS operating system environment.

Audience Assumptions

It is assumed that readers are programmers who have used PL/I or some other high-level programming language. This manual does not attempt to explain programming concepts to the novice.

The information in this manual should be of particular value to the two following audiences:

- Programmers who are familiar with the VAX/VMS operating system and who wish to understand the concepts of programming in PL/I.
- Programmers having prior experience with other implementations of PL/I who wish to understand the differences between those implementations and VAX-11 PL/I.

Structure of this Document

This manual has three chapters, a glossary, and a language summary:

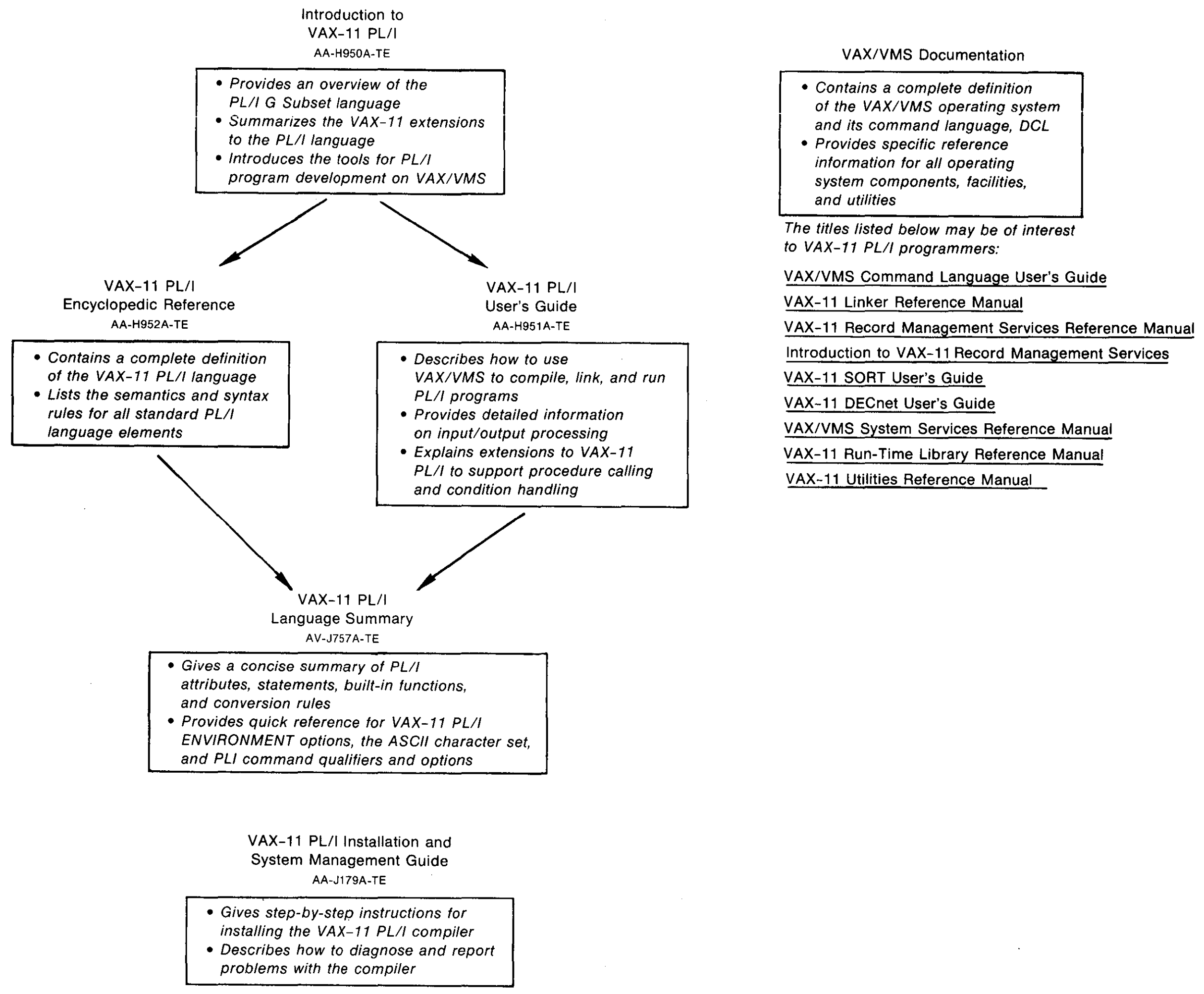
- Chapter 1, "PL/I Concepts," provides an overview of the PL/I programming language. This chapter emphasizes only those aspects of VAX-11 PL/I that are common among implementations of the ANSI standard PL/I language.
- Chapter 2, "VAX-11 Extensions to PL/I," describes the areas in which the PL/I language has been extended for programs that execute under the control of the VAX/VMS operating system.

- Chapter 3, "Using PL/I in the VAX/VMS Environment," shows a sample terminal session in which a PL/I program is written and executed. It also discusses the program development tools provided by the VAX/VMS operating system.
- The "VAX-11 PL/I Language Summary" is a syntactic summary of the VAX-11 PL/I language.
- The Glossary defines the terms that are introduced in this manual and that are pertinent to an understanding of the PL/I language.

Related Documents

This manual introduces concepts and techniques that are described fully in the primary VAX-11 PL/I documents. Figure 1 shows the relationship between the PL/I documents and related material in the VAX/VMS documentation set.

For abstracts of the VAX/VMS documents shown in Figure 1 and their order numbers, see the *VAX/VMS Information Directory and Index.*



Introduction to VAX-11 PL/I
AA-H950A-TE

- *Provides an overview of the PL/I G Subset language*
- *Summarizes the VAX-11 extensions to the PL/I language*
- *Introduces the tools for PL/I program development on VAX/VMS*

VAX-11 PL/I Encyclopedic Reference
AA-H952A-TE

- *Contains a complete definition of the VAX-11 PL/I language*
- *Lists the semantics and syntax rules for all standard PL/I language elements*

VAX-11 PL/I User's Guide
AA-H951A-TE

- *Describes how to use VAX/VMS to compile, link, and run PL/I programs*
- *Provides detailed information on input/output processing*
- *Explains extensions to VAX-11 PL/I to support procedure calling and condition handling*

VAX-11 PL/I Language Summary
AV-J757A-TE

- *Gives a concise summary of PL/I attributes, statements, built-in functions, and conversion rules*
- *Provides quick reference for VAX-11 PL/I ENVIRONMENT options, the ASCII character set, and PLI command qualifiers and options*

VAX-11 PL/I Installation and System Management Guide
AA-J179A-TE

- *Gives step-by-step instructions for installing the VAX-11 PL/I compiler*
- *Describes how to diagnose and report problems with the compiler*

VAX/VMS Documentation

- *Contains a complete definition of the VAX/VMS operating system and its command language, DCL*
- *Provides specific reference information for all operating system components, facilities, and utilities*

The titles listed below may be of interest to VAX-11 PL/I programmers:

VAX/VMS Command Language User's Guide
VAX-11 Linker Reference Manual
VAX-11 Record Management Services Reference Manual
Introduction to VAX-11 Record Management Services
VAX-11 SORT User's Guide
VAX-11 DECnet User's Guide
VAX/VMS System Services Reference Manual
VAX-11 Run-Time Library Reference Manual
VAX-11 Utilities Reference Manual

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

Conventions Used in This Document

A *computational* data item	An italicized word or phrase indicates a term that is being introduced. These terms are defined in the Glossary.
RET	A symbol with a one- to three-character abbreviation indicates that you press a key on the terminal, for example, RET or ESC.
CTRL/x	The symbol CTRL/x indicates that you press the key "x" while holding down the key labeled CTRL, for example, CTRL/C. In examples, this control key sequence is shown as ^x, for example ^C, because that is how the VAX/VMS system prints control key sequences.
$ RUN METRIC RET Enter conversion mode:	Command examples show all interactive examples in two colors. Program output and prompting characters that the system prints or displays are shown in black letters. All user-entered commands and data are shown in red letters.
option,...	Horizontal ellipses indicate that additional parameters, options, or values can be entered. When a comma precedes the ellipses, it indicates that successive items must be separated by commas.
DECLARE X FIXED ; . . . X = 5 ;	Vertical ellipses indicate that not all of the text of a program or program output is illustrated. Only relevant material is shown in the example.
quotation mark apostrophe	The term "quotation mark" is used to refer to the quotation mark (") symbol. The term "apostrophe" is used to refer to the single quotation mark (′) symbol.
[/qualifier...]	Square brackets indicate that a syntactic element is optional and you need not specify it. Square brackets are not optional, however, when used to delimit a directory name in a VAX/VMS file specification.
$\left\{\begin{array}{c}\text{'file-spec'}\\ \text{module-name}\end{array}\right\}$	Braces surrounding two or more stacked items indicate a choice; you must choose one of the two syntactic elements.

Chapter 1
PL/I Concepts

This chapter summarizes and provides brief examples of programming in PL/I. It discusses PL/I in the following contexts:

- Program structure and statement syntax
- Data declarations and data types
- Aggregates (arrays and structures)
- Assignment and expressions
- Input/output concepts
- Stream input/output
- Program control
- Procedures (subroutines and functions)
- Condition handling
- Record input/output
- Storage allocation and control

For complete reference information on any of the PL/I concepts or language elements discussed in this chapter, see the *VAX–11 PL/I Encyclopedic Reference*.

1.1 Program Structure and Statement Syntax

PL/I is a block-structured language in which *statements* are grouped in units called *blocks*.

1.1.1 Statements

Statements are used to declare variables and constants and to express actions to be performed by a PL/I program. A statement consists of PL/I *keywords*, user-specified *identifiers*, literal *constants*, *operators*, and punctuation. An example of a PL/I statement is:

```
DECLARE (TOTAL,COUNTER) FIXED BINARY;
```

The elements DECLARE, FIXED, and BINARY are PL/I keywords; the elements TOTAL and COUNTER are user-specified identifiers for variables. The parentheses, comma, spaces, and semicolon are punctuation that delimit these elements and give the statement its semantic meaning.

In PL/I, keywords are not reserved words; PL/I recognizes them as keywords within the context in which they are used. Thus, these words can be used as identifiers.

Identifiers are user-defined names that can be created from any of the characters A through Z, a through z, 0 through 9, $, and _. Uppercase letters and lowercase letters are equivalent (for example, the identifier A is the same as the identifier a). An identifier cannot have more than 31 characters or any embedded spaces.

Each PL/I statement must be terminated with a semicolon. Other than that, the text formatting requirements of PL/I are quite loose. Any number of spaces, tabs, or new-line characters can be entered in any position in a statement where a space is required or permitted; more than one statement can be entered on a single line. Most PL/I programmers prefer to specify only one statement per line and to use indention within the text of a source file to indicate nesting levels of blocks and groups of statements.

Here is an example of a simple PL/I program that shows one method of formatting a program. This program, called ADDUP, obtains integer values from the terminal and sums them:

```
ADDUP: PROCEDURE OPTIONS (MAIN);
DECLARE (TOTAL,COUNTER) FIXED BINARY;

        COUNTER = 1;
        TOTAL = 0;
        DO WHILE (COUNTER ^= 0);
            PUT SKIP LIST ('Enter number:');
            GET LIST (COUNTER);
            TOTAL = TOTAL + COUNTER;
            END;
        PUT LIST ('Total is',TOTAL);
        RETURN;
        END;
```

A *comment* can appear in any position in a PL/I statement in which punctuation is permitted or required. A comment is signified by the character pairs /* and */. For example, a statement in the program ADDUP may be commented as shown below:

```
COUNTER = 1;         /* initialize to any nonzero value */
```

The text between the comment characters can contain any characters except the character pair */.

A *label* may appear on most PL/I statements (the notable exceptions are DECLARE statements and target statements for an ON statement or for an

ELSE or THEN clause in an IF statement). A label is a user-specified identifier that is terminated with a colon (:). For example:

```
C: BEGIN;
```

The identifier C is the label of the BEGIN statement that heads a begin block. A statement can have at most one label.

1.1.2 Blocks

PL/I recognizes two types of block: *procedure blocks* and *begin blocks.* A procedure is the basic executable program unit of PL/I. The statements in a procedure block are contained within the limits of the statement pair PROCEDURE and END. In the program ADDUP, shown above, the entire program consists of a single procedure block.

A begin block is a program unit into which control flows during normal execution of a procedure. The statements of a begin block are contained within the limits of a pair of BEGIN and END statements.

When either a procedure block or a begin block is entered, a *block activation* is created for it. A block activation consists of the allocation of storage for variables declared within the block and hardware information that connects the block to the previous block. Figure 1-1 illustrates the relationship between the PL/I statements PROCEDURE and BEGIN and block activations created for procedure blocks and a begin block.

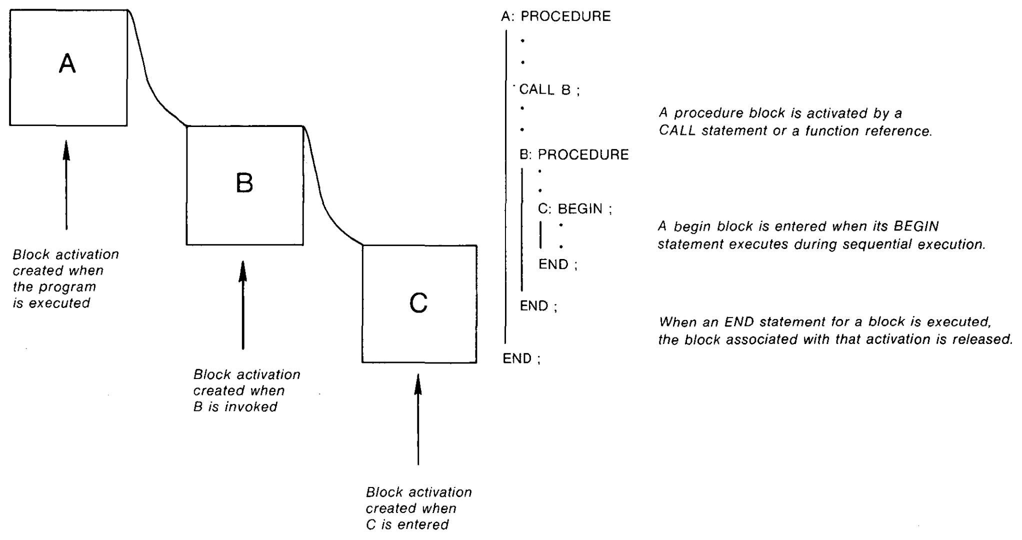

A
B
C
Block activation created when the program is executed
Block activation created when B is invoked
Block activation created when C is entered
A: PROCEDURE
CALL B ;
B: PROCEDURE
C: BEGIN ;
END ;
END ;
END ;
A procedure block is activated by a CALL statement or a function reference.
A begin block is entered when its BEGIN statement executes during sequential execution.
When an END statement for a block is executed, the block associated with that activation is released.

Figure 1-1: Block Activations

1.2 Data Declarations and Data Types

A *variable* is a data item whose value may change during the execution of a PL/I program. All variables used in a PL/I program must be explicitly specified in a DECLARE statement. In a DECLARE statement the *declaration* of a variable consists of:

- An identifier that names the variable. Within the program, the data associated with the variable is accessed by *references* to this name.
- The *data type*, or characteristics, of the data that the variable represents.
- The *storage* requirements of the variable.

The data type and storage requirements of a variable are specified in a DECLARE statement by PL/I keywords called *attributes*. Some examples of variable declarations are:

```
DECLARE CODE CHARACTER;
DECLARE TOTAL FIXED BINARY;
```

In these declarations, the variables named by the identifiers CODE and TOTAL are given the attributes described by the keywords CHARACTER and FIXED BINARY. By default, these variables are allocated storage each time the block that contains these declarations is activated.

Declarations of variables of the same data type may be factored. For example, variables that have the data type BIT may be declared as follows:

```
DECLARE (YES,NO) BIT;
```

The variables YES and NO are both given the attribute BIT.

All declarations in a block may be grouped into a single DECLARE statement by separating declarations with commas, as shown below:

```
DECLARE CODE CHARACTER,
        TOTAL FIXED BINARY,
        (YES,NO) BIT;
```

There are two categories of data types: computational and noncomputational. Each of these is discussed next.

1.2.1 Computational Data Types

The *computational* data types represent data items whose values can be manipulated. For example, arithmetic operations can be performed on *arithmetic data*, and string operations can be performed on *string data*. PL/I provides the following attribute keywords to specify computational data types:

- FIXED BINARY, for *fixed-point binary*, or *integer*, data.
- FIXED DECIMAL, for *fixed-point decimal* data.
- FLOAT BINARY or FLOAT DECIMAL, for *floating-point* data.
- CHARACTER, for *character-string* data. A character string consists of zero or more *characters*. The valid characters in a VAX-11 PL/I character string are defined in the *ASCII character set*.
- BIT, for *bit-string* data or for *Boolean* values. A bit string consists of zero or more of the binary digits, or *bits*, 0 and 1. A one-bit string with a value of

zero represents a Boolean value of false; a one-bit string with a value of one represents a Boolean value of true.

- PICTURE, for *pictured data*. Pictured data is data maintained in a character form that may contain nonnumeric characters (for example, dollar signs or actual decimal points).

When a variable is declared with any of the attributes listed above, the declaration normally includes additional information that further describes the data. For example, a character-string variable has an *extent*, or length. The following is an example of a character-string declaration:

```
DECLARE TEXT_MESSAGE CHARACTER (10);
```

This declaration specifies that 10 characters, each occupying a byte of storage, are required to hold the value of the variable TEXT_MESSAGE. When no extent is given in a variable declaration, PL/I provides a default. For example, the default extent for a character-string variable is one character. Character-string variables may also be declared with the VARYING attribute. The length of a varying-length character string is, at any given time, the length of the current value of the variable.

Arithmetic data has a *precision*. The precision of an arithmetic data item is simply the number of binary or decimal digits used to hold the data item's value. For example, the following declaration reserves seven binary digits for a fixed-point binary variable:

```
DECLARE INDEX FIXED BINARY(7);
```

PL/I always allocates a bit for the sign of a fixed binary value; thus, eight binary digits (bits) are required to hold the value of the variable INDEX, which can have a value in the range of –128 to 127. When no precision is specified for a fixed-point binary variable, PL/I supplies a default precision of 31 bits.

In addition to a precision, a fixed-point decimal variable has a *scale factor*. The scale factor specifies the number of digits of the value that are fractional, for example:

```
DECLARE TOTAL_PRICE FIXED DECIMAL (9,2);
```

This declaration indicates a fixed-point decimal value with a precision of nine decimal digits, two of which are fractional. When no precision is specified for a fixed-point decimal variable, PL/I supplies a default precision of 10 digits. If no scale factor is specified, PL/I supplies a default scale factor of 0.

When a precision of 24 or less is specified for a floating-point variable, VAX–11 PL/I creates a single-precision floating-point variable. When the precision is in the range of 25 through 53, VAX–11 PL/I creates a double-precision floating-point variable. A floating-point variable may be declared as in the following example:

```
DECLARE DISTANCE_TO_STAR FLOAT (53);
```

A *picture*, specified for pictured data, is a representation of an arithmetic value stored as a character string. A picture is like a fixed-point decimal value, including a precision and scale factor. However, a picture may also contain special characters such as dollar signs and decimal points. For example:

```
DECLARE PRICE PICTURE '$ZZZZ9V.99';
```

This declaration describes a fixed-point decimal value that, when output, will be formatted with a dollar sign and decimal point. The precision and scale factor of the value of PRICE are (7,2).

1.2.2 Noncomputational Data Types

Sometimes called program-control data types, the *noncomputational* data types refer to data used within the program for control structures and linkage.

The attributes for noncomputational data types are:

- ENTRY. Entry data identifies a point of invocation for a procedure.
- FILE. File data is used to define a source of input data or a destination for output data.
- LABEL. Label data represents statement labels.
- POINTER, AREA, and OFFSET. These data types describe other data items in terms of their storage locations.

The ENTRY, FILE, and LABEL attributes may be used in declarations as in the following example:

```
DECLARE WHICH ENTRY VARIABLE;
```

The VARIABLE attribute indicates that the variable named WHICH may be assigned different entry values during the execution of the program.

Pointer, area, and offset variables must be declared with the POINTER, AREA, and OFFSET attributes, respectively. These attributes are described in Section 1.11, "Storage Allocation and Control."

1.2.3 The Data Types of Constants

A constant is a data item whose value does not change. Computational data items and some noncomputational data items can be specified using constants.

A computational constant is a literal representation of a value. Table 1-1 summarizes the types of constant recognized by VAX-11 PL/I for computational data.

VAX-11 PL/I has no constants of binary or pictured types. Binary and pictured variables are given values by assigning them constant values or by assigning them the values of variables of other data types.

Table 1-1: Computational Constants

Data Type of Constant	Examples	Comments
Fixed-point decimal	55.1 23	All numeric constants are assumed to have a decimal base. When the constant has a decimal point and fractional digits, it is scaled. Otherwise, it is an integer.
Floating point	4E33 01.E-2	The letter E follows the mantissa and precedes the exponent.
Character string	'string'	Apostrophes delimit the string.
Bit string or Boolean	'0101'B	The letter B indicates a bit string.
Hexadecimal Octal	'4F'B4 '77'B3	The notation B4, B3, B2, and B1 permits bit-string constants to be specified in other bases.

The terms entry constant, file constant, and label constant are used to refer to names that are declared as follows:

- Entry and label constants may be declared implicitly by context. For example:

  ```
  PRINT_REPORT: BEGIN;
  ```

  In this statement, PRINT_REPORT represents a label constant; it is a label whose value does not change. Similarly, a PROCEDURE statement or ENTRY statement defines an entry constant:

  ```
  METRIC: PROCEDURE;
  ```

- File constants, except the default file constants SYSIN and SYSPRINT, must be explicitly declared. For example:

  ```
  DECLARE STATE_FILE FILE;
  ```

  This declaration represents a file constant. All I/O statements that reference STATE_FILE affect the same file.

- An entry constant may be declared explicitly if the constant represents an entry that is not in the current procedure. See Section 1.8.4, "Declaring External Procedures."

Any declaration of a file or entry data item is by default considered a constant, unless the VARIABLE attribute is also specified.

PL/I also recognizes compile-time symbolic constants, called *constant identifiers*. A constant identifier is defined and given a value by a %REPLACE statement, as follows:

```
%REPLACE SLENGTH BY 80;
```

After this statement is read during program compilation, all occurrences of the identifier SLENGTH are replaced by the constant 80. Constant identifiers are valid for arithmetic, character-string, and bit-string constants.

1.2.4 Automatic and Static Storage for Data

A PL/I programmer can effectively manage the use of storage by selecting the appropriate *storage class* for a variable. Two of the attributes that specify how PL/I allocates storage for a variable are AUTOMATIC and STATIC.

An *automatic variable* is not allocated storage by PL/I until the block in which it is declared is activated. When the block is deactivated, the storage is released. All variables, except those declared with the ENTRY and FILE attributes, are automatic by default.

A *static variable*, on the other hand, is allocated storage only once, and maintains its value from one activation of the block to the next.

Static and automatic variables may be initialized. They may be declared with values in the INITIAL attribute and PL/I will assign them these values at compile time. For example:

```
DECLARE PASSWORD STATIC CHARACTER(5) INITIAL('XXYYZ');
```

Other storage classes are based, defined, and parameter. Based and defined variables are described in Section 1.11, "Storage Allocation and Control." Parameter is the storage class of variables whose values are determined by arguments specified when a procedure is invoked. Parameters are described in Section 1.8.2, "Parameters and Arguments."

1.2.5 Internal and External Variables

An *internal variable* is one whose value is known only within the block that declared it and all blocks whose source text is contained within that block. The range of blocks in a program within which a name is known is called the *scope* of the name.

Figure 1–2 illustrates internal variables.

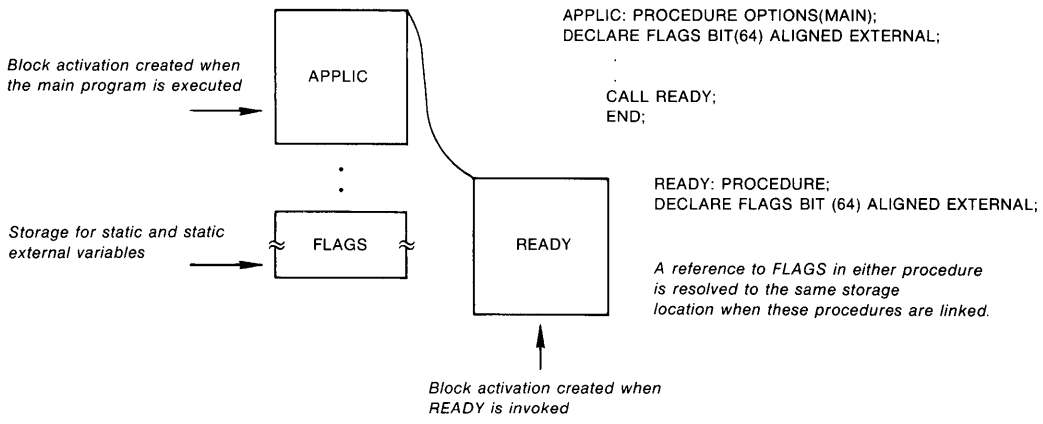
```
A:   PROCEDURE;
|    DECLARE INDEX FIXED BINARY;
|
|      B:  PROCEDURE;
|      |   C: BEGIN;
|      |   |  DECLARE COUNTER FIXED BINARY;
|      |   |
|      |   END;
|      |
|      END;
|
END;
```

Figure 1–2: Internal Variables

In Figure 1–2, the internal variable INDEX is known in the block that declares it, A, and in the contained blocks B and C. The internal variable COUNTER, however, is known only within the block C; C has no contained blocks. All variables that are not explicitly declared with the EXTERNAL attribute have the INTERNAL attribute by default.

An *external variable* is one whose value can be known in separately compiled procedures that are bound together by linking. All external variables have the STATIC attribute.

Figure 1–3 illustrates the declaration of PL/I external variables. Two separately compiled procedures, APPLIC and READY, declare the external variable FLAGS. In each procedure, FLAGS is declared with the same data type attributes and with the EXTERNAL attribute.



Block activation created when the main program is executed
APPLIC
APPLIC: PROCEDURE OPTIONS(MAIN);
DECLARE FLAGS BIT(64) ALIGNED EXTERNAL;
CALL READY;
END;
READY: PROCEDURE;
DECLARE FLAGS BIT (64) ALIGNED EXTERNAL;
Storage for static and static external variables
FLAGS
READY
A reference to FLAGS in either procedure is resolved to the same storage location when these procedures are linked.
Block activation created when READY is invoked

Figure 1–3: Static External Variables

External variables provide a way for modular procedures to share common data. A PL/I external variable is the same, functionally, as a FORTRAN COMMON. For an example of a PL/I program and a FORTRAN program sharing external variables, see the *VAX-11 PL/I User's Guide.*

1.3 Aggregates

A variable that represents a single element, or item, of data is called a *scalar* variable. Scalar variables can be grouped into *aggregates*. There are two types of aggregate:

- An *array* is an aggregate in which all items, called *elements*, have the same data type attributes, including extent and precision. Individual elements of an array are referred to by position, or order, in the array.
- A *structure* is an aggregate in which individual items, or *members*, are organized in a hierarchical manner. Each member of a structure may be of a different data type and each has a unique identifier within the structure.

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

1.3.1 Arrays

The number of *dimensions* of an array and the *extent*, that is the number of elements, in each are specified in parentheses following the variable name in a

DECLARE statement. The following example shows the declaration of an array of fixed binary integers:

```
DECLARE POP_VALUES (100,20) FIXED BINARY(31);
```

This statement declares an array named POP_VALUES that has two dimensions. The first dimension has 100 elements and the second dimension has 20 elements.

A reference to any element in the array is made by *subscripts*. Subscripts are integer expressions that specify the position of an element within each dimension of the array. For example:

```
POP_VALUES (50,4)
```

This reference specifies the element in row 50, column 4. Subscripts can be specified using any expressions that result in integer values.

An array declaration specifies the range of subscripts to be used to refer to array elements; these are the *bounds* of the array. When only one value is given for a dimension, PL/I uses that value for the *high bound* and uses a value of one for the *low bound*. In the array POP_VALUES, the bounds are 1 to 100 for the first dimension and 1 to 20 for the second dimension.

Bounds may be declared explicitly as well. For example:

```
DECLARE TEMPERATURES (-40:120) FIXED BINARY(7);
```

The first element of this array, that is, its low bound, is TEMPERATURES(-40), the second element is TEMPERATURES(-39), and so on. The upper limit, or high bound, is 120.

An array can have up to eight dimensions. Arrays are stored in memory in *row-major order*, which means that the rightmost array subscript varies the most rapidly.

Arrays can consist of noncomputational as well as computational data. For example:

```
DECLARE PATH_CHOICES (10) LABEL;
```

This declaration represents an array of label variables, each of which may be assigned values that represent label constants.

1.3.2 Structures

The hierarchy of the data items in a structure in PL/I is denoted by specifying a *level number* preceding the declaration of each member of the structure; the declarations of the members must be separated by commas. For example:

```
DECLARE 1 STATE,
          2 NAME CHARACTER (20),
          2 POPULATION PICTURE 'ZZ,ZZZ,ZZZ',
          2 CAPITAL,
            3 NAME CHARACTER (30),
            3 POPULATION PICTURE 'ZZ,ZZZ,ZZZ',
          2 SYMBOLS,
            3 FLOWER CHARACTER (20),
            3 BIRD CHARACTER (20);
```

In the preceding example, the variable name that has the level number 1 is the name of the *major structure*. Any reference to STATE will apply to the entire structure. The subsequent numbers define additional levels of the hierarchy. At the second level, the variable CAPITAL is further structured into two third-level variables, NAME and POPULATION. CAPITAL, in this example, is a *minor structure*, since it exists within a major structure and is itself a structure.

A structure can have up to 15 levels. The level numbers that precede each member name serve only to define the ordered relationship between the members. The actual number has no other meaning.

An individual member of a structure can be referred to by means of a *structure-qualified reference*. A structure-qualified reference gives the names of the minor and major structures at higher levels, separated by periods. For example, the identifier NAME appears twice in the declaration of STATE, so a reference to NAME would be ambiguous. However, the qualified reference STATE.CAPITAL.NAME precisely references the element NAME at the third level of the structure STATE.

A reference to an item in a structure must be qualified sufficiently to make it unique within the block containing the reference. In the structure STATE above, there are two members referred to as POPULATION. To distinguish between the two, they must be referred to as shown below:

```
STATE.POPULATION
STATE.CAPITAL.POPULATION or CAPITAL.POPULATION
```

If, however, there is only one variable of a given name known within a block containing the structure, this variable can be referred to without qualification. For example, if the identifier FLOWER is not used elsewhere in the block that contains the declaration of the structure STATE or in a previous block, then FLOWER may be referenced in any of the following ways:

```
FLOWER
SYMBOLS.FLOWER
STATE.SYMBOLS.FLOWER
```

A structure may itself be an element of an array, or contain array declarations. For example:

```
DECLARE 1 STATE(50),
        2 NAME CHARACTER(20),
        2 POPULATION PICTURE 'ZZ,ZZZ,ZZZ',
        2 LARGEST_CITIES (10),
          3 NAME CHARACTER(20),
              .
              .
```

This declaration of STATE creates an array of 50 elements, each of which is a structure. Now, members of the structure must be referred to using a subscript, for example:

```
STATE.POPULATION(50)
```

This references the member STATE.POPULATION in the structure STATE(50). A subscript in a structure-qualified reference may appear following any member name in the reference. For example, the reference above may also be written STATE(50).POPULATION.

The third member of STATE in the declaration shown above, LARGEST_CITIES, is an array that is also structured. At this level, two subscripts must be used to refer to members of this structure. For example:

```
STATE.LARGEST_CITIES.NAME(2,2)
```

This reference refers to the second element of the array LARGEST_CITIES in the second element of the array STATE.

1.4 Assignment and Expressions

The *assignment statement* gives a value to a variable. For example:

```
X = 5;
```

This statement gives the value 5 to the variable X. In the assignment of computational data, the source value of an assignment statement, that is, the value or expression on the right of the equals sign, can be any valid PL/I computational constant or expression.

Noncomputational variables may also be assigned values; however, they may be assigned values only by variables or constants of the same types or by expressions that yield the same types.

Variables can also be assigned values from a source outside of the program by an input statement. The types of input/output statement provided by PL/I are discussed in Section 1.5, "Input/Output Concepts."

1.4.1 Expressions

An *expression* is a representation of a value or a computation of a value. In PL/I, an expression can consist of variable names, constants, operators, and references to functions. A function is a procedure that is invoked when a reference to its name appears in an expression. The function returns a value to its point of invocation (functions are described in Section 1.8.3, "Subroutines and Functions").

1.4.2 Conversion of Data

PL/I freely allows assignments between data of different computational types, and most computations are valid as long as operands of the correct types are used. This section gives examples of common conversions that PL/I performs. The complete rules for conversions are given under the heading "Conversion of Data" in the *VAX-11 PL/I Encyclopedic Reference.*

Every expression in a PL/I program has a data type that determines the context in which it can be used. When two values or expressions that do not have the same data type are used in an expression, PL/I uses a well-defined set of rules to perform an implicit *conversion* of the operands to a common data type before performing the conversion. For example:

```
DECLARE A FIXED DECIMAL (10,2),
        B FLOAT DECIMAL (8),
        C FIXED BINARY;
C = A + B;
```

In this example, PL/I converts A to FLOAT DECIMAL before adding it to B. Then, it converts the FLOAT DECIMAL sum to FIXED BINARY before assigning this value to C.

In assignment statements, PL/I performs automatic conversions between any two computational data types. For example:

```
DECLARE A CHARACTER (7),
        B FIXED DECIMAL (5,2);

A = '-124.99';
B = A;
```

Here, PL/I converts the value of the variable A from a character string to fixed decimal at the time of the assignment of the value of A to the variable B.

The compiler also converts constants. For example:

```
DECLARE I FIXED BINARY;
DECLARE A FLOAT;

A = A / (I + 1);
```

In this example, 1 is a fixed-point decimal constant. It is converted to fixed-point binary before being added to the variable I. The fixed-point binary sum, I + 1, is converted to floating-point binary before the division is performed. The result of the division is floating-point binary. Most constants are converted during compilation, so that only conversion of variables is performed during execution.

1.4.3 Operators

An *operator* is a punctuation symbol or symbols used to represent a computation or operation. Each operator takes expressions, or operands, of specific data types and returns a value of a specific data type. The operators are summarized below:

- *Arithmetic operators* perform arithmetic operations. The arithmetic operators are:

  + (addition or prefix plus)
  – (subtraction or prefix minus)
  * (multiplication)
  / (division)
  ** (exponentiation)

  Operands of arithmetic operators must be arithmetic; the result is always arithmetic.

- *Relational operators* perform comparisons of data. The relational operators are:

  = (equal to)
  < (less than)
  > (greater than)
  <= (less than or equal to)
  >= (greater than or equal to)
  ^= (not equal to)
  ^< (not less than)
  ^> (not greater than)

  Computational operands must be both arithmetic or they must both be of the same data type. The result is either the bit-string value '0'B (false) or the bit-string value '1'B (true). Comparisons of the noncomputational data types are restricted to the = and ^= operators.

- *Logical operators* perform logical operations on Boolean values. The logical operators are:

  ^ (logical NOT prefix)
  | (logical OR)[1]
  & (logical AND)

  Operands must be bit strings. The result is a bit string.

- The string *concatenation* operator concatenates character strings or bit strings. The concatenation operator is:

  ||

  Operands must both be character strings or they must both be bit strings. The result is a string with the same data type as the operands.

1. The ! character may be used in place of the |.

An expression that contains more than one operator is evaluated on the basis of the priorities, or *precedence*, of the operators. The operators have the following priorities, where a lower number indicates an operation that is performed first:

Operator	Priority	Operator	Priority
**	1	>	5
+ (prefix)	1	<	5
− (prefix)	1	^>	5
^	1	^<	5
*	2	^=	5
/	2	<=	5
+	3	>=	5
−	3	&	6
||	4	|	7
=	5		

The PL/I rules of precedence tend to produce reasonable results. For instance, in the expression:

```
A + B ^= C * D
```

the addition and multiplication are performed before the results are tested for inequality. Operations involving operators of equal priority are performed from left to right in all operations except exponentiation. Exponents are evaluated from right to left.

Parentheses may be used to specify the order of evaluation. For example:

```
( (A+B) / C ) / 2
```

In this expression, the addition of A and B is performed before the division of the result by C; then the result of this operation is divided by 2. In any expression that includes multiple operations, the expressions within the innermost set of parentheses are evaluated first.

1.4.4 Aggregate Assignments

An array variable may be specified as the target of an assignment statement with a source expression that yields a scalar value. Every element of the array is assigned the resulting value. For example:

```
DECLARE A (50,50) FLOAT;

A = 0;
```

This statement sets each element of the array A to zero.

Both arrays and structures can be used in assignments according to the following rules:

- An array variable can be assigned to another array with the same number of dimensions, extents, and attributes.
- A structure variable can be assigned to another structure that has the same hierarchical organization and in which all members have the same attributes.

For example:

```
DECLARE X (10,10) FIXED BINARY,
        SAVE_X (10,10) FIXED BINARY;

SAVE_X = X;
```

In this example, the value of each element of the array X is assigned to the corresponding element in the array SAVE_X.

1.5 Input/Output Concepts

In PL/I, a *file* is a source of input data or a target for data output. A *file reference* in an input/output statement is actually a reference to a file constant or to a file variable that has been assigned a value. For example:

```
DECLARE INFILE FILE;
```

In the block that contains this declaration, the identifier INFILE may be used in an I/O statement to perform an operation on this file. File constants are external, by default.

1.5.1 Types of Input/Output

PL/I distinguishes between two types of input/output processing. Each type of processing handles input and output data in a different manner, and each has a unique set of input/output statements and usage. These types of input/output are:

- Stream input/output, or simply *stream I/O.* The stream I/O statements are GET and PUT.
- Record input/output, or simply *record I/O.* The record I/O statements are READ, WRITE, DELETE, and REWRITE.

When reading or writing a file using stream I/O, the data is treated as if it forms a continuous stream of ASCII characters. Individual *fields* of data within the stream are delimited by commas, spaces, and end-of-line indicators, or by explicit format descriptions. A stream I/O statement specifies one or more fields to be processed in a single operation.

When reading or writing a file using record I/O, however, a single record is processed upon the execution of an I/O statement. For example, either stream

I/O or record I/O may be used to accept input data from a terminal. Assume the following data is to be read from a terminal:

```
THIS IS A LINE OF INPUT DATA.
```

If this line is read by a typical stream I/O statement, each word in this sentence may be assigned to a different program variable; that is, the words are fields delimited by spaces. However, if this line is read by a record I/O statement, the entire sentence may be assigned to a single program variable. The end of the line constitutes the end of the record.

Stream I/O and record I/O are described individually in this chapter. Stream I/O is discussed in Section 1.6, which follows; record I/O is discussed in Section 1.10.

1.5.2 Attributes of Files

The declaration of a file constant may specify attributes, called *file description attributes*, that indicate how the file is to be used. For example, the STREAM attribute indicates a file that will be read or written using stream I/O, and the RECORD attribute indicates a file that will be read or written using record I/O.

A file can be opened explicitly with an OPEN statement; the OPEN statement may specify attributes that were not specified in the file's declaration. For example:

```
OPEN FILE (INFILE) RECORD INPUT;
```

This statement opens the file associated with the identifier INFILE and gives it the RECORD and INPUT attributes. The INPUT attribute indicates that the file is an input file.

A file is closed with the CLOSE statement. For example:

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

This statement makes the file INFILE inaccessible until it is reopened.

1.5.3 Associating File Constants with External Files

The TITLE option of the OPEN statement permits the specification of an external file or device on which the input/output operations are to be performed. For example:

```
OPEN FILE (STATE_FILE) INPUT
        TITLE('DB1:[MALCOLM]STATE.DAT');
```

This OPEN statement defines the specific device and file that is to be opened for input. If no TITLE option is specified in an OPEN statement, or if the TITLE option does not completely specify a device or file, VAX-11 PL/I follows a well-defined set of rules to determine the specific VAX/VMS file with which a PL/I file is to be associated.

For example, a file might be opened as follows:

```
OPEN FILE (TEMP) RECORD OUTPUT;
```

This OPEN statement does not specify a TITLE option. In this case, PL/I uses the name of the file constant as a title. In the context of the VAX/VMS operating system, this name can be a logical name. Thus, if a logical name TEMP exists, its current equivalence is used. Otherwise, PL/I uses TEMP as a file name and supplies a default file type of DAT.

For a complete description of the rules for associating PL/I files with VAX/VMS files, see the *VAX-11 PL/I User's Guide*.

1.6 Stream Input/Output

In stream I/O, the data in the external file is treated as a stream of ASCII characters. In a stream I/O operation, a list of program variables is associated with actual input or output fields of data.

There are two forms of the stream I/O statements:

- *List-directed stream I/O* is performed by the GET LIST and PUT LIST statements.
- *Edit-directed stream I/O* is performed by the GET EDIT and PUT EDIT statements.

Both forms of the stream I/O statements use the PL/I default files named SYSIN (for input) and SYSPRINT (for output). For an interactive user, these files are associated with the terminal by default. These files need not be declared.

A third form of the stream I/O statements, GET STRING and PUT STRING, permits the manipulation of character strings.

1.6.1 List-Directed Stream I/O

In list-directed stream input, input fields delimited by spaces, tabs, or commas correspond exactly to the variables specified in the statement. When output, variables are output in fields separated by spaces or tabs.

This form of the I/O statement is most commonly used for simple terminal I/O. The program ADDTHREE, below, shows a GET LIST statement that reads three data items from the current input device and assigns their values to the variables A, B, and C. Then, a PUT statement adds these three variables and writes the result to the current output device.

```
ADDTHREE: PROCEDURE OPTIONS (MAIN);
DECLARE (A,B,C) FIXED BINARY;
      GET LIST (A,B,C);
      PUT LIST ('is',A+B+C);
END;
```

If the current input device is a terminal, the following data items may be entered when the GET statement is executed:

```
55,123     60 RET
```

Following the entry of this data, the variable A has the value 55, the variable B has the value 123, and the variable C has the value 60. The PUT LIST statement writes the string 'is', evaluates the expression that adds these three variables, and writes the total on the terminal. This output may appear as follows:

```
is      238
```

In a GET LIST operation, PL/I automatically converts ASCII input data to the data type of the variables specified in the GET LIST statement. In a PUT LIST statement, PL/I converts a computational expression to its ASCII equivalent for output.

The GET and PUT statements read and write data in fields, and do not perform new line operations unless specifically requested to do so. Thus, when a PUT statement precedes a GET statement, the output and input data may all appear on the same line. For example, the following line may be added to ADDTHREE to precede the GET statement:

```
PUT LIST('Enter numbers:');
```

When the program is run, the output and input appear as:

```
Enter numbers: 55,123     60 RET
is      238
```

The SKIP option causes the GET or PUT statement to skip to a new line. For example:

```
PUT SKIP(2) LIST('Finished');
```

This statement advances two lines and displays a message.

The SKIP option may be placed before or after the LIST option and its list of variables. PL/I always performs the advancement before the input or output operation, regardless of where the option appears in the statement.

The parenthesized list of input or output items must always follow the LIST keyword.

1.6.2 Edit-Directed Stream I/O

In edit-directed stream I/O, a special set of characters, called *format items*, specify how PL/I is to process the data. When multiple items are processed, a list of format items, called a *format list*, specifies the editing to be performed on each item in the data list. Format items control:

- The specific type of conversion to be performed
- The width of an input or output field, that is, the number of characters or digits in the field

For example, the E format item describes the representation of floating-point values. Assume that the result of a floating-point calculation is to be output. The following PUT statement specifies the data to be output and its format:

```
PUT EDIT (XYVAL) ( E(10,2) );
```

The first set of parentheses encloses the output data list; the second set encloses the format list. The EDIT keyword, the output data list of variables, and the format list must appear in this order. This format specification indicates that (1) the corresponding value in the data list, that is, XYVAL, is to be converted to floating-point representation, and (2) the output field is to be 10 characters wide, with two fractional digits.

A value that is printed with this statement might appear as:

```
△△6.32E-14
```

Where △△ indicates that this output field contains two leading spaces.

When a format list is supplied in a GET statement, the GET statement reads a field of the specified width from the input stream. If commas, spaces, or tabs appear in the field, they are treated as input data, not as field separators. For example, the format item F, which represents fixed-point decimal numbers, may be specified as follows in a GET statement:

```
GET EDIT (SALARY) ( F(5,2) );
```

This statement reads a field of five characters from the input stream, and assumes that the last two characters read are fractional. It then converts the characters to fixed-point decimal and assigns the result to the variable SALARY. For example, the input field read by this GET statement may be:

```
21356
```

The resulting value of the variable SALARY is 213.56.

Format items are supplied for each of the computational data types; additional format items provide page and columnar formatting for print files. A detailed description of format items is given under the heading "Format Items and Their Uses" in the *VAX-11 PL/I Encyclopedic Reference.*

1.6.3 Declaration of Stream Files

In GET and PUT statements, no file reference need be specified. By default, PL/I uses the file constants named SYSIN (for input) and SYSPRINT (for output). However, stream I/O can be performed on any type of file or device as long as the input or output data is to be treated as ASCII data.

For example, to prepare to read data from a card reader, the following statements could be written:

```
DECLARE CARD_READER FILE;

OPEN FILE (CARD_READER) STREAM INPUT TITLE('CR:');
```

The TITLE option in this OPEN statement identifies the input device as a card reader. After this OPEN statement, the file CARD_READER may be specified in a GET statement, for example:

```
GET FILE (CARD_READER) LIST (GRADES);
```

It is good practice to open files explicitly with an OPEN statement, but it is not required. If a file is not opened with an OPEN statement, an I/O statement for the file causes what is called an implicit open. For example, if the file CARD_READER had not been open in the preceding example, PL/I would have opened it with the STREAM and INPUT attributes because the GET statement is a stream input statement.

1.6.4 Print Files

For stream output files directed to a terminal, line printer, or any device that handles output in terms of pages and lines, PL/I performs special output formatting. When PL/I outputs data to a file declared as a *print file*, it aligns data in columns specified by predefined tab stops, and it counts lines and pages.

The PRINT attribute indicates that a file is a print file. For example, the print file with the identifier LINE_PRINTER may be declared as follows:

```
DECLARE LINE_PRINTER FILE PRINT;
```

The PRINT attribute implies the STREAM and OUTPUT attributes.

When a file is declared as a print file, the LINESIZE and PAGESIZE options may be specified when the file is opened. These options specify the maximum width of an output line and the maximum number of lines per page, respectively. For example, the file LINE_PRINTER may be opened like this:

```
OPEN FILE(LINE_PRINTER)
                    LINESIZE(30) PAGESIZE(40);
```

As PUT statements write data to this file, PL/I keeps track of the current column position of data and performs new line operations as needed to keep the data within the specified line size. When 40 output lines have been written, PL/I notifies the program, by a signal, that the end of the page has been reached. The ways in which a PL/I program can respond to such signals are described in Section 1.9, "Condition Handling." For a print file, PL/I also maintains a current page number and a current line number, which may be accessed by the program through the PL/I built-in functions PAGENO and LINENO. Built-in functions are described in Section 1.8.7, "Built-In Functions."

1.7 Program Control

The statements in a PL/I program are normally executed in the sequence in which they appear in the source program. The IF, DO, and GOTO statements alter this flow.

1.7.1 IF Statement

The IF statement evaluates an expression for a Boolean value of true or false. It executes a statement contained in a THEN clause if the result is true or executes a statement in an optional ELSE clause if the result is false. The IF statement must have the THEN clause and may have the ELSE clause. The statement in each clause must be terminated by a semicolon. For example:

```
IF A < B THEN PUT SKIP LIST('Less');
   ELSE PUT SKIP LIST('Not less');
```

Either clause may consist of a null statement. For example:

```
IF A < B THEN;
   ELSE PUT SKIP LIST ('Not less than');
```

When PL/I evaluates the expression in an IF statement, a resulting bit-string value of zero indicates false and a bit string with a nonzero value indicates true. Thus, if any bit of a value is set to one, the entire expression has a true value. The result of any expression that contains a relational operator is a Boolean value. For example:

```
IF (A < B) & ( C = D) THEN RESULT = OKAY;
```

This IF statement provides comparisons of the variables A and B and of the variables C and D. If both comparisons result in true values, the entire expression is true and the assignment statement in the THEN clause is executed.

THEN and ELSE clauses can consist of multiple statements headed by a BEGIN statement or a DO statement. For example:

```
IF A < B THEN DO;        /* if true, execute DO-group */
   .
   .
   END;
   ELSE BEGIN;      /* otherwise, enter begin block */
     DECLARE TEMP FIXED BINARY (31);
       .
       .
     END;
```

A begin block is appropriate when the statements to be executed require the declaration of automatic variables to be used solely for those statements. Otherwise, a DO-group is used. DO-groups are described next.

1.7.2 DO Statement

The PL/I DO statement begins a sequence of statements called a *DO-group*. A DO-group consists of all the statements between the DO statement itself and a corresponding END statement. In its simplest form, a DO statement results in the unconditional execution of the statements in the DO-group a single time. For example:

```
IF A ^= B THEN DO;
    PUT SKIP LIST('Continue:');
    GET LIST (NEXT);
    END;
```

The DO statement may contain various options that control the execution of the DO-group. A common form of the DO statement, called the iterative DO, initializes a variable, called a *control variable*, and executes a DO-group a number of times, incrementing the control variable until it reaches a certain value. For example:

```
DECLARE ARRAY(10) FIXED BINARY,
        INDEX FIXED;

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

The statements in this DO-group are executed 10 times. The first time the statements are executed, the control variable INDEX has the value 1, the second time it has the value 2, and so on. This form of the DO statement may also specify a value by which the control variable is to be modified, for example:

```
DO INDEX = 10 TO 100 BY 10;
```

This DO statement increments the value of INDEX by 10 in each subsequent execution of the DO-group.

The values in the iterative DO statement may be specified using variables of any computational data type; in most cases, these values are integers.

The DO statement also has a WHILE option, which defines a condition that must be satisfied. The condition must be placed in parentheses. The expression is always evaluated as a Boolean expression. Often, it contains a relational expression that yields a Boolean result, as in this example:

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

The statements in the DO-group following this statement are executed as long as the value of the variable A is less than the value of the variable B. If A is equal to or greater than B when this DO statement is first executed, the DO-group is not executed at all.

The DO statement also permits combinations of forms. For example:

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

The statements between this DO statement and its corresponding END statement execute until the value of the variable A is equal to or greater than the value of the variable B, or until the value of the control variable reaches 11, whichever comes first.

The REPEAT option provides another way to modify a control variable. In this form, an expression is evaluated following each execution of the DO-group, and the result is assigned to the control variable. For example:

```
DO INDEX = 1 REPEAT (INDEX + 3) WHILE (INDEX < 100);
```

This DO statement executes the following DO-group with values for INDEX of 1, 4, 7, and so on, as long as INDEX is less than 100. The control variable, initial value, and REPEAT expression can be of any data type (including POINTER) that is valid for assignment. Thus, the REPEAT option provides a convenient way to modify a pointer variable in list processing, as described in Section 1.11.2, "Pointers."

1.7.3 GOTO Statement

The GOTO statement provides an unconditional transfer of control to a labeled statement. For example:

```
IF (A < B) THEN GOTO DONE;
```

The GOTO statement transfers control to the label DONE if the value of A is less than the value of B.

A GOTO statement can also specify a subscripted label name. For example, labels may be written as follows:

```
CHOICE(1):
    .
    .
CHOICE(2):
    .
    .
CHOICE(3):
```

and so on. A GOTO statement can be written:

```
GOTO CHOICE(INDEX);
```

This GOTO statement transfers control to the label CHOICE whose subscript is represented by the value of the variable INDEX.

A GOTO statement can transfer control to a label that exists in an active block outside of the block containing the GOTO statement. When this type of transfer, called a *nonlocal GOTO*, occurs, then the current block, and all blocks between the current block and the block containing the specified label, are released.

1.8 Procedures

A *procedure* is a block that begins with a PROCEDURE statement. A procedure is executed by a RUN command or is invoked by either a CALL statement or a function reference.

The *main procedure* or main entry point is the point at which control begins when the program is executed by a RUN command. A main procedure is designated by specifying OPTIONS (MAIN) on the procedure statement. For example:

```
CONTROLLER: PROCEDURE OPTIONS (MAIN);
```

Only one procedure in a program may specify the MAIN option. If no procedure specifies OPTIONS(MAIN), or if there is only a single procedure in the program, the RUN command executes the first or only procedure.

Other procedures may be contained within the text of the main procedure or may be linked to it as external procedures. These procedures can be invoked, or entered, by a CALL statement. For example, if a program has a procedure named PRINT_ROUTINE, this procedure is entered with a statement like the following:

```
CALL PRINT_ROUTINE;
```

A procedure returns control with a RETURN statement. For example:

```
RETURN;
```

This statement releases the block for the current procedure and transfers control to the next statement to be executed in the calling procedure.

The next few subsections introduce the following aspects of writing and invoking procedures in PL/I:

- Internal procedures
- Parameters and arguments
- Subroutines and functions
- Declaring external procedures
- Entry points
- Recursion
- Built-in functions
- Terminating procedures

1.8.1 Internal Procedures

PL/I provides for calls to either internal or external procedures. An *internal procedure* is one whose text is contained within the text of another procedure. Figure 1–4 illustrates invoking an internal procedure.

```
A: PROCEDURE OPTIONS (MAIN);
|
|   . ❶
|   .
|   B: PROCEDURE;
|   |
|   |  .   ❸
|   |  .
|   |  RETURN;
|   END;
|
|   CALL B; ❷
|
|   . ❹
|   .
END;
```

Figure 1–4: An Internal Procedure

The flow of control in the example in Figure 1–4 is indicated by the circled numbers. As procedure A executes (1), control branches around the statements between the PROCEDURE statement for B and its corresponding END statement. When A executes the CALL statement that invokes B (2), execution continues with the first statement in B (3). When B completes by executing its RETURN statement, control returns to the statement following the CALL statement in A (4). The procedure B can appear anywhere within the text of procedure A.

1.8.2 Parameters and Arguments

A procedure is generally written so that it may act upon different data or values each time it executes. In PL/I, certain values, or *arguments*, are passed by means of an *argument list* specified in the procedure invocation. The arguments must correspond to the *parameters* specified in a *parameter list* declared in the PROCEDURE statement of the invoked procedure.

For example:

```
PROCESS_LIST: PROCEDURE OPTIONS(MAIN);
DECLARE PRINT_LIST CHARACTER(132);
          .
          .
      CALL PRINT_ROUTINE(PRINT_LIST);
          .
          .
PRINT_ROUTINE: PROCEDURE (LINE);
DECLARE LINE CHARACTER(132);
    .
    .
END PRINT_ROUTINE;
END PROCESS_LIST;
```

When the main procedure calls PRINT_ROUTINE, it supplies an argument, PRINT_LIST, that corresponds to PRINT_ROUTINE's parameter LINE. The procedure PRINT_ROUTINE specifies the name of its parameter, LINE, in parentheses following the PROCEDURE statement. This is its parameter list. Within PRINT_ROUTINE, LINE is declared with data type attributes.

In most cases, PL/I passes an argument to an invoked procedure *by reference* to the storage of the actual written argument. The parameter itself occupies no storage within the procedure in which it is declared. When a parameter refers to the actual storage of the written argument, the invoked procedure can modify a parameter and thus pass a value to its invoker through the parameter list. For example:

```
DECLARE (X,Y,SUM) FLOAT;

CALL ADDFLOAT(X,Y,SUM);
    .
    .
ADDFLOAT: PROCEDURE(A,B,TOTAL);
DECLARE (A,B,TOTAL) FLOAT;

      TOTAL = A+B;
      RETURN;
END ADDFLOAT;
```

In this example, the procedure ADDFLOAT sums the two floating-point numbers that are passed as its first two parameters and places the result in its third parameter. This parameter, TOTAL, occupies the same storage as the argument, SUM, declared in the main procedure.

PL/I does not pass arguments by reference to the argument's actual storage in the following cases:

- When the written argument is a constant or expression
- When the written argument is a variable and its data type does not match the data type of the parameter

In either of these cases, PL/I creates a special variable called a *dummy argument*. It then assigns the value of the written argument to the dummy argument and passes the dummy argument by reference. Note what happens in the second case: an assignment of a value to the parameter within the invoked procedure will not modify the value of the written argument variable. When a variable specified for an argument does not match the data type of the corresponding parameter, the situation may represent a programming error. Therefore in these cases the compiler issues a warning message to indicate that it is creating a dummy argument.

Creation of a dummy argument may be forced by enclosing an argument in parentheses. In this case, the compiler does not issue a diagnostic message.

1.8.3 Subroutines and Functions

Procedures are classified as either *subroutines* or *functions*. A subroutine is invoked with a CALL statement; when it returns, program execution normally continues with the next statement following the CALL statement. The examples thus far in this section have shown subroutines.

A function, on the other hand, is invoked when a reference to its name appears in an expression; it cannot be invoked by a CALL statement. When this reference, called a *function reference*, occurs, PL/I passes control to the function. When the function returns control, it passes a value, called a *return value*, to its point of invocation. Program execution normally continues with the evaluation of the expression in which the function reference occurred.

A function must specify:

- The RETURNS option in its PROCEDURE statement, describing the data type of the value it returns. The attributes specified in the RETURNS option are called the *returns descriptor* of the function.
- A value in the RETURN statement with which it relinquishes control. The value must have the same data type as that specified in the RETURNS option or be valid for conversion to that data type. The value is returned to the point of invocation of the function.

The example below illustrates a function and a function reference. The assignment statement at the beginning of the program OUTER contains two references to the procedure ADDER, an internal function.

```
OUTER: PROCEDURE OPTIONS (MAIN);
DECLARE (TOTAL,A,B,C,D) FLOAT;
          .
          .
     TOTAL = ADDER(C,D) + ADDER(A,B);
          .
          .
ADDER: PROCEDURE (X,Y) RETURNS (FLOAT);
DECLARE (X,Y) FLOAT;

     RETURN (X+Y);
END ADDER;
END OUTER;
```

The function ADDER has two parameters, X and Y. They are floating-point variables declared within the function. The procedure OUTER invokes the function twice in the assignment of a value to TOTAL. Each time this function is invoked, it is passed two arguments that correspond to its parameters. It returns a floating-point value representing the sum of the arguments.

1.8.4 Declaring External Procedures

An *external procedure* is a procedure whose text is not contained within another procedure. An external procedure is separately compiled and is combined with the procedure that references it when the procedures are linked to form an executable program. The name of an external procedure must begin with an alphabetic letter.

In PL/I, all external procedures must be declared in the procedure that references them. The ENTRY attribute indicates that a name is the name of an external procedure. When any external procedure is declared, the declaration must specify the data type attributes for each of its parameters. The list of data type attributes for a given parameter is called a *parameter descriptor*. For example:

```
DECLARE XYZ EXTERNAL ENTRY (FIXED BINARY, CHARACTER(*));
```

This declaration indicates to PL/I that XYZ is the name of an external subroutine that has two parameters. The parameter descriptors for XYZ indicate that XYZ's first parameter is fixed binary and that its second parameter is a character string. The asterisk (*) extent specified for this parameter indicates that the character string may be of any length; this is a PL/I convention for passing strings of undetermined lengths. The length will be provided at the time the procedure is actually invoked. When XYZ is called, it must be passed two arguments that correspond to those parameters.

An external function is declared in a similar manner, but the declaration must also specify the RETURNS attribute. The RETURNS attribute must specify a returns descriptor, which gives the data type attributes of the value that the function returns. This returns descriptor is specified in the same way that the returns descriptor is specified in the RETURNS option of the function's PROCEDURE statement. For example:

```
DECLARE COPYSTRING ENTRY (CHARACTER(40) VARYING,
                          FIXED BINARY(7))
        RETURNS (CHARACTER(400) VARYING);
```

This declaration of the external function COPYSTRING indicates that it has two parameters: (1) a varying-length character string with a maximum length of 40, and (2) a fixed-point binary value. The PROCEDURE statement for the procedure COPYSTRING and its associated parameter declarations might be as follows:

```
COPYSTRING: PROCEDURE (INSTRING,REPCOUNT)
             RETURNS (CHARACTER(400) VARYING);

DECLARE INSTRING CHARACTER(40) VARYING,
        REPCOUNT FIXED BINARY(7);
```

The variables INSTRING and REPCOUNT, specified in the parameter list in the PROCEDURE statement, are declared with data type attributes within the procedure. These attributes must match the attributes of the corresponding arguments.

1.8.5 Entry Points

An *entry point* is a statement at which a program or procedure begins execution. All entry points have user-specified identifiers, or *entry names*, which can be specified in procedure calls or function references.

Only one procedure in a program may specify the MAIN option. Additional entry points to a procedure may be defined with the ENTRY statement. For example:

```
SUB1: ENTRY;
```

This statement defines the entry SUB1 as an entry point to the current procedure. The entry name SUB1 may be declared and invoked as an external entry from another external procedure. When the procedure is invoked at its main entry point and the ENTRY statement is encountered during the flow of sequential execution, control passes to the first executable statement following the ENTRY statement.

1.8.6 Recursion

A PL/I procedure may be *recursive*, that is, it may invoke itself. The entry point at which the procedure will be recursively entered may be declared with the RECURSIVE option. For example:

```
HANOI: PROCEDURE(T1, T2, T3, RINGS) RECURSIVE;
```

This statement declares the procedure HANOI as a recursive procedure.

1.8.7 Built-In Functions

PL/I provides an extensive set of functions that are available without declaration. These *built-in functions* provide the following types of capability:

- Arithmetic built-in functions provide information about the properties of arithmetic values or perform common arithmetic calculations, for example, obtaining the maximum or minimum of two values or rounding a value.
- Trigonometric built-in functions perform standard trigonometric calculations, for example, computing the the sine of an angle. These functions always return floating-point values.
- String-handling built-in functions process character-string and bit-string values, for example, locating a substring within a string or extracting a substring from a given string.
- Conversion built-in functions convert data from one data type to another, for example, obtaining the ASCII representation of a numeric value.
- Condition-handling built-in functions provide information about a specific signaled condition, such as the error code that caused the condition.
- Array-handling built-in functions provide information about arrays, such as the extent of a dimension or the value of the high or low bound of a dimension.

- File control functions return the current page number or current line number of a print file.
- Storage control built-in functions manipulate pointer and offset variables, for example, giving the storage location of a variable.

The sample program FIRST_LAST, below, illustrates the string functions INDEX (which locates a character substring within a string), SUBSTR (which extracts a substring from a string), and LENGTH (which returns the length of a string).

```
FIRST_LAST: PROCEDURE OPTIONS(MAIN);
DECLARE (NAME,FIRST_NAME,LAST_NAME) CHARACTER(30) VARYING,
        BLANK FIXED BINARY(7);

      PUT LIST ('Enter your name:');
      GET LIST (NAME);
      BLANK = INDEX (NAME,' ');    /* locate blank */
      FIRST_NAME = SUBSTR(NAME,1,BLANK-1);
      LAST_NAME = SUBSTR(NAME,BLANK+1,(LENGTH(NAME)-BLANK));

      PUT SKIP LIST('Your first name is',FIRST_NAME,
         'and your last name is',LAST_NAME);
      RETURN;
END;
```

This program assumes that strings are entered in the form:

```
'firstname lastname'
```

The apostrophes are required to ensure that the first name and the last name are read in as a single string to the variable NAME. Using the INDEX built-in function, this procedure locates the space between the first name and the last name. Then, using the SUBSTR built-in function, it extracts the appropriate substrings from the entire given string.

The built-in functions STRING, SUBSTR, PAGENO, and UNSPEC are permitted on the left-hand side of an assignment statement. These *pseudovariables* define the target of an assignment. For example, the SUBSTR pseudovariable may be used as shown below:

```
CHARACTERS = 'AAAAA';
SUBSTR(CHARACTERS,3,2) = 'BB';
```

In this example, the SUBSTR pseudovariable is used to change the third and fourth positions of the variable CHARACTERS from AA to BB. The resulting value of the string is AABBA.

1.8.8 Terminating Procedures

A procedure can be terminated by any of the following:

- A RETURN statement. A function must execute a RETURN statement. If a subroutine does not execute a RETURN statement, the END statement effects a normal return.
- A STOP statement. A STOP statement may appear anywhere in a program or procedure. It differs from the RETURN statement in that it terminates

the entire program, regardless of the procedure in which it is executed. This statement is primarily of use when a procedure detects an error from which it cannot recover.

- A GOTO statement that transfers control to a previous block.
- A run-time error or a program-generated signal that is not handled by the current procedure.

The manner in which PL/I programs can signal, detect, and respond to errors that occur at run time is described next.

1.9 Condition Handling

In PL/I, certain *conditions* that occur during the execution of a program can result in the interruption of the normal sequence of execution. Floating-point overflow, division by zero, end-of-file, and end-of-page are examples of these types of conditions.

When a PL/I program incurs a condition of this nature, a *signal* that indicates the type of condition is generated and PL/I attempts to locate an *ON-unit* to handle the condition. An ON-unit is a single PL/I statement or a begin block written specifically to take special action for a particular condition. For example:

```
ON FIXEDOVERFLOW GOTO PRINT_MESSAGE;
```

When this ON statement is executed, an ON-unit for FIXEDOVERFLOW conditions is established. If a fixed-point overflow occurs after this ON statement is executed, program control will be transferred to the statement labeled PRINT_MESSAGE.

An ON-unit for a specific condition remains in effect until another ON statement that specifies the same condition is executed, until the block that established that ON-unit is released, or until a REVERT statement that specifies that condition name is executed. For example:

```
REVERT FIXEDOVERFLOW;
```

This statement cancels the ON-unit in effect for the FIXEDOVERFLOW condition.

1.9.1 Condition Names

PL/I *ON conditions* have associated language keywords, or *condition names*. Any procedure can establish ON-units for any or all of these conditions. The PL/I condition names are summarized in Table 1–2.

Table 1–2: Summary of ON Conditions

Condition Name	Usage
ENDFILE	Handles end-of-file condition for a specified file
ENDPAGE	Handles an end-of-page for a specified file with OUTPUT and PRINT attributes
ERROR	Handles miscellaneous error conditions and conditions for which no specific ON-unit exists
FINISH	Handles the condition signaled when the main procedure returns, when any procedure in the program executes a STOP statement, or when the program exits due to an error
FIXEDOVERFLOW	Handles fixed-point decimal and integer overflow exception conditions
KEY	Handles any error involving the key when using keyed access to a specified file
OVERFLOW	Handles floating-point overflow exception conditions
UNDEFINEDFILE	Handles any errors opening a specified file
UNDERFLOW	Handles floating-point underflow exception conditions
ZERODIVIDE	Handles divide-by-zero exception conditions

1.9.2 The Execution of ON-Units

When any condition is signaled, PL/I searches for an ON-unit to handle the condition, beginning in the current block. If there is no ON-unit in the current block, PL/I searches the previous block, if any, and so on. It gives control to the first ON-unit it finds for the indicated condition.

When there is no ON-unit to handle the condition, PL/I takes a default action. If there is a procedure in the program that has the MAIN option, the default action taken by PL/I in most cases is to signal the ERROR condition. If no ON-unit exists for the ERROR condition, the program terminates.

When an ON-unit for a signaled condition completes execution, PL/I normally continues the execution of the procedure from the statement that caused the error. For example, when many lines are being printed on a terminal, PL/I signals the ENDPAGE condition after every 60 lines. An ENDPAGE ON-unit can cause PL/I to continue output regardless of the signal, as shown below:

```
ON ENDPAGE (SYSPRINT);
DO INDEX=1 TO 100;
   PUT SKIP LIST (STRING_ARRAY(INDEX));
   END;
```

This ON statement specifies a null action. After the null ON-unit executes, execution continues with the PUT statement.

An ON-unit may transfer control elsewhere in the program. This is the usual action for arithmetic errors, as shown in the program ADDNUMBERS below:

```
ADDNUMBERS: PROCEDURE OPTIONS(MAIN);
DECLARE X FIXED BINARY(7),
        TOTAL FIXED BINARY(31),
        DATA_FILE FILE STREAM INPUT;
/*
   ON-unit for fixed-point overflow conditions
*/
ON FIXEDOVERFLOW GOTO PRINT_MESSAGE;
/*
   Computation
*/
       X = 1;
       TOTAL = 0;
       DO WHILE (X ^= 0);
           GET FILE(DATA_FILE) LIST(X);
           TOTAL = TOTAL + X;
           END;
       PUT LIST('Total is ',TOTAL);
       RETURN;
PRINT_MESSAGE:
       PUT SKIP LIST('VALUE OUT OF RANGE, CORRECT DATA FILE.');
       RETURN;
END;
```

In this example, the variable X is declared with a precision of only seven binary digits. Any input values for X that are not in the range of -128 to 127 cause PL/I to signal the FIXEDOVERFLOW condition during the processing of the GET statement. The ON-unit transfers control to the label PRINT_MESSAGE, where the procedure issues a message and terminates.

1.9.3 The ONCODE Built-In Function

Whenever a condition is signaled, PL/I sets an internally maintained variable to the numeric value of the error condition. This *condition value* can be obtained within the program by referencing the ONCODE built-in function. For example:

```
ON ERROR BEGIN;
   IF ONCODE() = 10 THEN PUT LIST (MESSAGE_10);
     ELSE IF ONCODE() = 12 THEN PUT LIST (MESSAGE_12);
          ELSE PUT LIST ('Unknown signal ',ONCODE());
END;
```

This ON-unit tests the value returned by the ONCODE built-in function following the signaling of the ERROR condition. Given either of two values, it prints a message associated with a specific value. Otherwise, it prints the value of ONCODE.

The meanings of the numeric values of ONCODE are defined by the VAX/VMS system and by VAX-11 PL/I. It is possible to refer to these values in a PL/I program using symbolic names defined by the system. For details on declaring and using these symbolic names, see the *VAX-11 PL/I User's Guide*.

1.10 Record Input/Output

In record I/O, external data is treated in terms of *records*. A file containing records is a *record file*.

1.10.1 File Organizations

When a record file is created, its *file organization* is defined. The organization of the file refers to the physical arrangement of the records in the file and the implied order in which records will be accessed. VAX-11 PL/I recognizes the following file organizations:

- Sequential — in a *sequential file*, records are arranged serially, with one record after another.
- Relative — in a *relative file*, each record has a *relative record number* and the records are ordered on the basis of their relative numbers.
- Indexed sequential — in an *indexed sequential file*, each record has one or more data *keys* embedded within the record and the records are arranged and located on the basis of these keys.

In any type of file organization, the *record format* of individual records — that is, whether they are fixed or variable length, how long they are, and so on — is specified by options in the ENVIRONMENT attribute when the file is created.

For example:

```
DECLARE EMP_RECORDS FILE ENVIRONMENT (
                             FIXED_LENGTH_RECORDS,
                             MAXIMUM_RECORD_SIZE(80) );
```

This declaration describes a file with 80-byte, fixed-length records.

If no ENVIRONMENT options are specified when a file is created, PL/I uses the default record format of variable-length records and a maximum record size of 512 bytes.

When a PL/I program reads or writes records to a file, it specifies the name of a variable whose declaration matches the size of the record. The variable may be a structure whose members match the layout of the data within the file's records. For example:

```
DECLARE 1 STATE,
          2 NAME CHARACTER (20),
          2 POPULATION PICTURE 'ZZ,ZZZ,ZZZ',
          2 CAPITAL,
            3 NAME CHARACTER (30),
            3 POPULATION PICTURE 'ZZ,ZZZ,ZZZ',
          2 SYMBOLS,
            3 FLOWER CHARACTER (20),
            3 BIRD CHARACTER (20),

        STATE_FILE FILE;

    .
    .
READ FILE (STATE_FILE) INTO (STATE);
```

This READ statement reads a record from the file STATE_FILE into the structure variable STATE. This statement assigns a value to each member of STATE.

1.10.2 Access Modes

Each time an existing file is opened, PL/I attributes define the manner in which the records will be accessed, that is the *access mode* of the file. The file description attributes that declare the access mode of a file for a particular opening are:

- SEQUENTIAL — to indicate *sequential access* to a file
- DIRECT — to indicate *random access* only to a file
- KEYED SEQUENTIAL — to indicate sequential and random access to a file

Combinations of these attributes with the attributes INPUT, OUTPUT, and UPDATE determine the specific operations that can be performed on a file. For example, if a file that does not already exist is opened with the DIRECT and OUTPUT attributes, PL/I creates a relative file by default; only WRITE statements may be used to access the file.

If, on the other hand, a file is opened with the SEQUENTIAL and INPUT attributes, the file must already exist; only READ statements may be used to access the file.

1.10.3 Declaring and Opening Record Files

All record files in PL/I must be declared as file constants or associated at open time with the name of a file constant. For example:

```
DECLARE EMP_RECORDS FILE RECORD INPUT SEQUENTIAL;
```

The attributes RECORD, INPUT, and SEQUENTIAL indicate that the file referred to by the name EMP_RECORDS is an input file that is to be processed sequentially using record I/O statements.

Although the attributes specified in the declaration of a file are considered permanent, they may be augmented when the file is opened. The attributes specified in the OPEN statement must not conflict with the file's permanent attributes. For example, the OPEN statement to open the file EMP_RECORDS may be specified as follows:

```
OPEN FILE (EMP_RECORDS) KEYED;
```

Adding the KEYED attribute to the file's description indicates that the file will be accessed by key as well as sequentially. The file's organization, of course, must be one of the types that permits keyed access.

1.10.4 Sequential Access

The following example illustrates a sequential READ statement in which the records in the file are read into the variable STATE:

```
DECLARE STATE_FILE FILE RECORD;
OPEN FILE(STATE_FILE) INPUT;
READ FILE(STATE_FILE) INTO (STATE);
```

After this READ statement executes, the variable STATE contains the contents of the first record in the file STATE_FILE. The next READ statement that specifies the file STATE_FILE reads the next sequential record, and so on, until the file is closed or until the end-of-file is reached.

When the last record has been read from a file being accessed sequentially, PL/I uses its condition-signaling mechanism to signal an end-of-file condition to the program. Thus, the program can establish an ON-unit to perform special processing when the end-of-file is reached. For example:

```
ON ENDFILE (STATE_FILE) BEGIN;
     CLOSE FILE(STATE_FILE);
     CALL PRINT_LIST(STATE_QUEUE_HEAD);
     END;
```

This ENDFILE ON-unit for the file STATE_FILE closes the file and calls the procedure PRINT_LIST with the argument STATE_QUEUE_HEAD. When this ON-unit completes execution, the program continues execution following the READ statement that caused the ENDFILE condition to be signaled.

1.10.5 Random Access

A file opened with the KEYED attribute can be accessed randomly; that is, individual records in the file can be read, added, deleted, or rewritten by specifying the key associated with the record. Depending on the organization of the file, the key can be:

- A relative record number — this type of key applies to a file with relative organization, in which each record has a number corresponding to its position in the file.
- A data key — this type of key applies to a file with indexed sequential organization, in which data keys are embedded within each record.

For example, an indexed sequential file may be processed as follows:

```
DECLARE STATE_FILE FILE RECORD KEYED
                   ENVIRONMENT(INDEXED),
        INPUT_NAME CHARACTER(20) VARYING;

OPEN FILE (STATE_FILE);
PUT SKIP LIST('Which state?');
GET LIST (INPUT_NAME);
READ FILE (STATE_FILE) INTO (STATE) KEY (INPUT_NAME);
```

In this READ statement, the record that is read into the variable STATE is specified by the key INPUT_NAME. The value of this key is obtained by a GET LIST statement that reads the name of a state.

1.10.6 Error Handling

PL/I signals one of the following conditions when errors occur while processing either a stream or a record file:

- UNDEFINEDFILE — this condition is signaled whenever an error occurs opening a file, for example, if the file cannot be located, or if the file's attributes are incompatible with the access attempted to the file.

- KEY — this condition is signaled whenever a key reference causes an error, for example, if a key does not have an appropriate data type, or if a key cannot be located.
- ERROR — this condition is signaled for all other miscellaneous error conditions that occur during file processing.

ON-units for any of these conditions can reference PL/I built-in functions to obtain information about the specific nature of the error:

- The specific file for which the error occurred (the ONFILE built-in function)
- The key value that caused a KEY condition (the ONKEY built-in function)
- The specific numeric error value (the ONCODE built-in function)

For example:

```
ON KEY(STATE_FILE) BEGIN;
      PUT SKIP
           LIST ('Error',ONCODE(),' processing key ',ONKEY());
      STOP;
      END;
```

This ON-unit is executed if an error is signaled because of an invalid key in any I/O operation on the file STATE_FILE. The ON-unit displays the values returned by ONCODE and ONKEY and stops the program.

1.11 Storage Allocation and Control

PL/I provides a special storage class, called *based*, for variables whose storage allocation is under explicit control of the programmer. Based variables are useful in the following situations:

- Allocation of storage for variables whose extents vary from one execution of the program to another
- Temporary allocation of storage
- List processing

A based variable describes storage that is accessed by means of a *pointer*. The pointer specifies a virtual memory location — it points to the data associated with the based variable's description. The BASED attribute defines a based variable and optionally specifies the pointer to be used to reference it. For example:

```
DECLARE STATE_POINTER POINTER;

DECLARE 1 STATE BASED (STATE_POINTER),
          2 NAME CHARACTER (20),
          2 POPULATION PICTURE 'ZZ,ZZZ,ZZZ';
```

The structure STATE is declared with the BASED attribute and associated with the pointer STATE_POINTER. When the program containing these declarations begins execution, no storage actually exists for the variable STATE and it is invalid to reference STATE until STATE_POINTER is given a value.

STATE_POINTER can be given a value in an explicit *allocation* of storage for the variable. The following example shows the allocation of storage with the ALLOCATE statement:

```
ALLOCATE STATE SET (STATE_POINTER);
```

This statement obtains as much storage as is necessary to contain the variable and sets the value of the variable STATE_POINTER to the location in memory of the allocated storage.

After storage has been allocated for a based variable, it can be referenced:

```
STATE.NAME = 'Alabama';
```

The FREE statement releases an allocation of storage obtained for a based variable. For example:

```
FREE STATE;
```

This FREE statement releases the storage for the structure STATE that is referenced by STATE_POINTER.

1.11.1 Locator-Qualified References

More than one allocation of storage can be obtained for a based variable. Each ALLOCATE statement must specify the SET option to indicate the pointer of reference. For example:

```
DECLARE (FIRST,SECOND) POINTER,
        LIST(10) FLOAT BASED;

ALLOCATE LIST SET (FIRST);
ALLOCATE LIST SET (SECOND);
```

When more than one allocation exists for a based variable, as in the above example, a reference to the variable or any of its members must be a *locator-qualified reference*. This type of reference specifies the pointer that locates the specific allocation of interest by means of the -> symbol. This is a *locator qualifier*.

For example, following the two allocations of storage for the based array variable LIST, a reference to an element in the first allocation must contain the locator-qualified reference as follows:

```
FIRST -> LIST(5) = 0;
```

This assignment statement gives the value of zero to the fifth element of LIST in the allocation pointed to by the pointer variable FIRST.

A locator qualifier may also be used to associate a particular storage location with a based variable that is not bound to a particular pointer. For example:

```
DECLARE DATA CHARACTER(10) BASED,
        DP POINTER,
        LINE CHARACTER(10);

LINE = 'string';
DP = ADDR(LINE);
PUT LIST( DP->DATA );
```

The locator qualifier (->) in this PUT statement associates the based variable DATA with the storage occupied by the variable LINE, pointed to by the pointer DP.

1.11.2 Pointers

All pointer variables must be explicitly declared. They may be assigned values in assignment statements, in ALLOCATE statements (as shown above), and in the following ways:

- The ADDR built-in function returns a pointer value giving the location of a specific variable.
- The SET option of the READ statement copies a record to internal storage and sets a pointer variable to the location of the record. A subsequent REWRITE statement replaces the record in the file from the internal storage buffer.

Pointer variables may also be used in relational expressions of equality and inequality. For example:

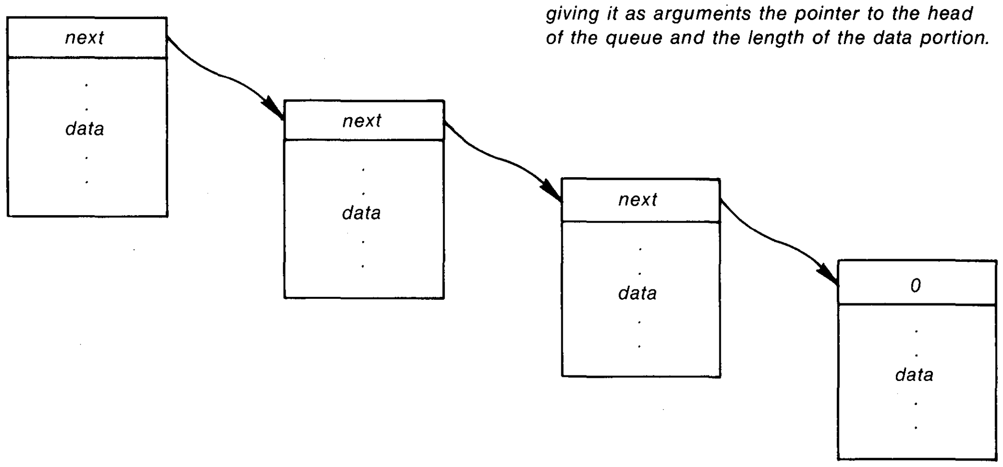
```
DECLARE (NEXT,SAVE) POINTER;
    .
    .
IF NEXT = SAVE THEN CALL FINISH_UP;
```

In this example, the IF statement tests whether the values of two pointers are the same.

PL/I also provides the following ways to test and use pointers:

- The NULL built-in function returns a null pointer value. This value can be used to mark the end of a list of structures that are linked by pointers.
- The REPEAT option of the DO statement provides a convenient way to step through a linked list.

Figure 1–5 illustrates linked-list processing in PL/I using the NULL built-in function and the REPEAT option of the DO statement. It shows a procedure that outputs a linked list with any number of structures. The based structures are linked with pointers; the first member of each structure is a pointer to the next structure in the list.


Allocations of storage for a linked list that were allocated and linked in an external procedure. The procedure invokes PRINT_LIST, giving it as arguments the pointer to the head of the queue and the length of the data portion.
next
data
next
data
next
data
0
data

```
print_list: procedure (queue_head,data_length) ;

declare queue_head pointer,                /*start of queue*/
        data_length fixed binary(31) ;     /*length of data*/
declare 1 list based (p),                  /* structure of queue elements*/
          2 next pointer,
          2 data character(data_length) ;
declare p pointer ;

        do p = queue_head repeat (p->list.next)
                while (p^= null());
           put list (p->list.data);
           end;
return;
end;
```

Figure 1-5: List Processing in PL/I

1.11.3 Areas and Offsets

Allocations of storage for based variables can be explicitly located within larger allocations of storage, called *areas*. Within an area, a specific allocation of storage for a variable can be located by an *offset* variable, which gives the location of the storage relative to the beginning of the area.

When variables are allocated within areas, the contents of an entire area can be either copied to another area in an assignment statement or written to an output file or device in a single WRITE statement. Since an offset gives the location of a variable relative to the start of an area, the values of the offset variables within an area do not change when the contents of an area are moved or assigned. Pointer variables, on the other hand, contain virtual memory addresses and so will not remain valid if they are written out and read back in another execution of the program.

Area and offset variables must be declared with the AREA and OFFSET attributes. For example:

```
DECLARE MAP_SPACE AREA (4096),
        BLOCK_A OFFSET (MAP_SPACE);
```

These declarations define an area of 4096 bytes and an offset variable to be associated with that area.

In some implementations of PL/I, the ALLOCATE statement allocates storage within an area and sets an offset variable to the position of the storage within the area. In VAX-11 PL/I, the actual placement of data within an area and the assignment of values to offset variables must be performed by a user-written procedure. For an example of such a procedure, see the *VAX-11 PL/I User's Guide.*

1.11.4 Defined Variables

PL/I also permits a variable to share the storage of another variable, so that a reference to either variable accesses the same data. This type of variable is a *defined variable* and it is declared with the DEFINED attribute.

For example:

```
DECLARE X CHARACTER(20) DEFINED (STATE.NAME);
```

This declaration defines the variable X but does not allocate any storage for it. When X is referenced, the current value of the variable STATE.NAME is obtained.

The use of defined variables is allowed only when the data types of the variables specified are the same or can be overlaid in a meaningful way. For complete details on the criteria under which storage may be shared by defined variables, see the *VAX-11 PL/I Encyclopedic Reference.*

Chapter 2 VAX–11 Extensions to PL/I

The implementation of the PL/I Subset G for the VAX–11 computer contains extensions that may be used by PL/I programs that will execute exclusively under the control of the VAX/VMS operating system. The extensions are primarily in the following areas:

- Extensions for input/output processing — these provide VAX–11 PL/I programs with full access to the file organizations, access modes, and file placement and control capabilities of the VAX–11 file system, the Record Management Services (RMS).
- Extensions to support VAX–11 calling and condition-handling conventions — these provide support for PL/I procedures that call or are called by procedures written in programming languages other than PL/I.

This chapter describes these extensions in general terms and provides some examples of the capabilities they offer. For a complete description of any of these items, see the *VAX–11 PL/I User's Guide.*

2.1 Extensions for Input/Output Processing

The VAX–11 PL/I language provides extensions to input/output support in the following ways:

- Options to the ENVIRONMENT attribute that can be specified when a file is created, opened, or closed. ENVIRONMENT options are specified following the ENVIRONMENT keyword on a DECLARE, OPEN, or CLOSE statement for a file.
- Options to I/O statements that provide specific processing for an I/O operation. I/O statement options are specified following the OPTIONS keyword in an I/O statement.
- Built-in subroutines that perform file control operations.

Examples in the subsections 2.1.1 through 2.1.9 illustrate ENVIRONMENT options, statement options, and built-in subroutines.

2.1.1 Indexed Sequential Files

Records in an indexed sequential file are accessed by specification of a data key embedded in the record. In VAX–11 PL/I, the key may be character, fixed-point binary, or fixed-point decimal with a zero scale factor. Records may have more than one key; each key has a separate index.

In a file with multiple indexes, each key field has a user-specified *index number*. The primary index number is zero, the secondary index is one, and so on. The record I/O statements READ, REWRITE, and DELETE may specify the INDEX_NUMBER option to specify which key number applies to the current operation. For example:

```
READ FILE (STATE_FILE) INTO (INREC)
                   OPTIONS(INDEX_NUMBER(2));
```

This READ statement accesses the indexed file STATE_FILE sequentially using the second alternate key.

After an index number is specified, it applies to subsequent sequential or keyed I/O operations on the file, until the INDEX_NUMBER option is again specified. The INDEX_NUMBER option may also be specified in the ENVIRONMENT attribute to define an initial index number value. For example:

```
DECLARE STATE_FILE FILE KEYED ENVIRONMENT
                       (INDEX_NUMBER(2));
```

When no index number is specified, VAX–11 PL/I uses index zero, the primary index.

By default, a key specified in an I/O statement must exactly match the key in a record for the operation to be successful. However, VAX–11 PL/I provides the MATCH_GREATER and MATCH_GREATER_EQUAL options for the READ, REWRITE, and DELETE statements. These options specify greater-than key matches or greater-than-or-equal-to key matches.

2.1.2 Relative Files

To create a relative file in VAX–11 PL/I, no ENVIRONMENT options are required. When a file with the KEYED attribute is opened for output, PL/I creates a relative file unless ENVIRONMENT options specify otherwise. In each WRITE statement, the KEYFROM option specifies the relative record number of the record. When a relative file is accessed, the KEY option on the READ statement specifies the relative record number of a record to be read. For example:

```
DECLARE EMPLOYEE_NUMBER FIXED,
        EMP_RECORDS FILE,
        EMP_REC CHARACTER(80);

GET LIST(EMPLOYEE_NUMBER);
READ FILE(EMP_RECORDS) INTO(EMP_REC)
             KEY (EMPLOYEE_NUMBER);
```

This READ statement reads the record whose relative record number is obtained by a GET statement. It places the contents of the record in the variable EMP_REC.

When a relative file is created, the ENVIRONMENT option MAXIMUM_RECORD_NUMBER can be specified to define the maximum number of records that the file can have. This value, once set, cannot be changed. If no value is specified for the maximum record number, VAX-11 PL/I does not provide a default maximum; that is, the file can have any number of records. However, when a relative file has no maximum record number, the file system does not check the validity of records that are added to the file to ensure that they are within a specified range.

2.1.3 File Disposition Options for Closing a File

The ENVIRONMENT attribute can be used on a CLOSE statement to specify the disposition of a file as it is closed. Options that can be specified on the CLOSE statement are:

- DELETE — specifies that the file be deleted.
- SPOOL and BATCH — specify that the file be submitted to the system printer or batch job queue, respectively.
- REWIND_ON_CLOSE — specifies that a magnetic tape file be rewound when it is closed.
- TRUNCATE — truncates the file at its logical end-of-file.

These ENVIRONMENT options may be enabled or disabled at run time. For example:

```
DECLARE IFDELETE BIT(1);

CLOSE FILE(TEMP) ENVIRONMENT ( DELETE (IFDELETE) );
```

This CLOSE statement deletes the file TEMP only if the current value of the Boolean variable IFDELETE is true.

These options may also be specified when a file is opened; if an option is respecified on the CLOSE statement, it can override the effect of the option on OPEN.

2.1.4 File Ownership and Protection

When a file is declared or opened, the following options may be specified to define the file's owner and to specify the type of access permitted to other system users:

- The OWNER_GROUP and OWNER_MEMBER options together specify the files' owner.
- The options GROUP_PROTECTION, OWNER_PROTECTION, SYSTEM_PROTECTION, and WORLD_PROTECTION specify the types of access (read, write, execute, or delete) permitted each category of user.

For example:

```
OPEN FILE (MEMO) OUTPUT RECORD ENVIRONMENT (
                             GROUP_PROTECTION('RWED'));
```

This OPEN statement creates a file that can be accessed for reading, writing, or deleting by any member of the owner's group. When no options are specified, PL/I supplies default values based on the current system- and user-defined defaults.

2.1.5 Terminal I/O

The stream input/output statements, GET and PUT, have options that provide special processing when the I/O device is an interactive terminal. The GET statement provides the following options:

- PROMPT — specifies a prompting string to be displayed on the output device before the GET statement accepts an input list.
- NO_ECHO — suppresses the display of data as it is entered, for example, in order to mask the entering of sensitive information.
- NO_FILTER — passes the ASCII codes for the CTRL/U, CTRL/R, and DEL function keys to the program for processing. These characters are normally interpreted by the terminal.
- PURGE_TYPE_AHEAD — clears the terminal's type ahead buffer before the GET statement reads the input list.

The following example illustrates several GET statement options:

```
GET LIST (PASSWORD) OPTIONS (
                              PROMPT ('Enter Password: '),
                              NO_ECHO,
                              PURGE_TYPE_AHEAD );
```

The PUT statement option CANCEL_CONTROL_O overrides the effect of the CTRL/O function key on the terminal to ensure that the beginning of the output list is displayed.

2.1.6 Fixed-Control Files

A special record format, called variable-length with fixed-length control, can be processed using VAX-11 PL/I record I/O statements. In this record format, each record has associated with it a data area that is not a part of the actual record. This area, called a *fixed-control area*, can contain any type of data, such as line sequence numbers, carriage control information, and so on.

The ENVIRONMENT option FIXED_CONTROL_SIZE defines the size of the fixed-control area. The I/O statement options FIXED_CONTROL_TO and FIXED_CONTROL_FROM may be specified on READ and WRITE statements, respectively, to read and write the fixed-control area of a record.

For example, to create and write a record to a file whose records have an eight-byte fixed-control area, the OPEN and WRITE statements might appear as follows:

```
DECLARE LINE_NUM PICTURE '99999999';

OPEN FILE (OUTFILE) RECORD SEQUENTIAL OUTPUT
          ENVIRONMENT (FIXED_CONTROL_SIZE(8) );

WRITE FILE (OUTFILE) FROM (DATA_RECORD)
            OPTIONS (FIXED_CONTROL_FROM (LINE_NUM));
```

Each WRITE statement that outputs a record to this file specifies the FIXED_CONTROL_FROM option. In this example, the option specifies the eight-byte pictured variable LINE_NUM.

2.1.7 Magnetic Tape File Processing

VAX-11 PL/I provides the following features for processing magnetic tape files:

- ENVIRONMENT options that control the positioning of a magnetic tape. These are:

  ```
  REWIND_ON_OPEN
  REWIND_ON_CLOSE
  CURRENT_POSITION
  ```

- The NXTVOL built-in subroutine. This subroutine performs end-of-volume switching for files that span more than one physical tape.

  This subroutine can be invoked for input or output operations. If necessary, the subroutine sends a message to the system operator requesting that a new volume be mounted. The subroutine does not return control to the program until the next tape volume is mounted and ready.

- The EXPIRATION_DATE option of ENVIRONMENT. This option permits the specification of an expiration date and time for a magnetic tape file.

- The ENVIRONMENT options BLOCK_IO and BLOCK_SIZE permit a magnetic tape file to be read or written in terms of blocks of a user-specified size.

For example:

```
OPEN FILE (TAPEFILE) OUTPUT RECORD ENVIRONMENT (
                        BLOCK_IO,
                        BLOCK_SIZE(2048))
```

This OPEN statement opens a magnetic tape in which records are to be blocked in 2048-byte blocks. The actual blocking is performed by RMS as the program writes individual records to the file.

2.1.8 Block I/O

Any sequential disk file can be read and/or written in terms of logical disk blocks. When a file is opened for *block I/O*, both PL/I and RMS ignore record lengths and record formats. The PL/I program itself must interpret all data in the file in units of 512-byte blocks. For example:

```
OPEN FILE (FAST_COPY) KEYED ENVIRONMENT (BLOCK_IO);
```

This OPEN statement opens the file FAST_COPY for block I/O with keyed access. The program may use the READ and WRITE statements to randomly access blocks in the file.

2.1.9 Record Id Access

For faster input/output operations on record files, a file can be opened for *record id access* by specifying the RECORD_ID_ACCESS option on the ENVIRONMENT attribute. When a file is opened with this option, the RECORD_ID_TO option can be specified on any record I/O statement that performs an operation on a record. The value returned by RECORD_ID_TO may be used in a subsequent operation involving the same record, thus eliminating the overhead required for the run-time system to search for the record. For example:

```
DECLARE ID_VAL (2) FIXED BINARY(31);

READ FILE (DATA_FILE) INTO (STATE)
          OPTIONS (RECORD_ID_TO (ID_VAL));
    .
    .
REWRITE FILE (DATA_FILE) FROM (STATE)
          OPTIONS (RECORD_ID (ID_VAL));
```

In this example, the array ID_VAL is used to obtain the record identification of a record read into the variable STATE. When the same record is subsequently rewritten, the RECORD_ID option specifies the same variable.

2.2 Extensions for Calling and Condition Handling

Extensions to enable PL/I procedures to interact with system procedures and programs not written in PL/I include:

- Attributes for the declaration of non-PL/I external entry points and their parameters
- Attributes for the declaration of external variables and constants whose values are defined by non-PL/I procedures
- New storage class attributes
- Additional condition name keywords for the ON, SIGNAL, and REVERT statements
- Built-in functions to provide information to an ON-unit about system-specific values

These extensions permit PL/I programs to call:

- VAX/VMS *system services* — system services are operating system procedures that can be used to develop application programs.
- *Run-time library* procedures — the VAX-11 Run-Time Library contains procedures that perform common functions not provided by PL/I.
- Miscellaneous system programs — examples are the generalized message-handling and output facilities and the librarian utility routines.

All of these procedures conform to the conventions described in the next subsections.

2.2.1 Argument-Passing Mechanisms

On the VAX-11 computer, the argument list for a procedure is always represented by an array of longwords (a longword is a storage unit of 32 bits). The first longword in the list contains the number of arguments, that is, the number of longwords remaining in the list. Each remaining longword in the list represents a single argument that has been passed using one of the following mechanisms:

- *By reference.* The argument list longword contains the address of the actual argument.
- *By descriptor.* The argument list longword contains the address of a data structure, called a descriptor, that describes the actual argument, including its extent and its address.
- *By immediate value.* The argument list longword contains the actual argument value, which cannot be longer than 32 bits.

PL/I's standard method is to pass arguments associated with parameters with asterisk extents by descriptor and to pass all other arguments by reference. For calls to non-PL/I procedures, VAX-11 PL/I provides the following attributes that modify the standard PL/I methods:

- The VALUE attribute. This attribute specifies that the argument is to be passed by immediate value. In PL/I terms, this means that a dummy argument is always created — the dummy argument is created directly in the argument list longword.
- The ANY attribute. This attribute specifies that the argument is to be passed by reference and also that the argument may have any data type. A dummy argument is never created for a variable reference. This attribute is convenient for passing arrays with nonconstant extents to FORTRAN programs that do not use the descriptor mechanism.
- The ANY and VALUE attributes together. These attributes specified together indicate that the argument is to be passed by immediate value and that the data type of the argument is any type that will fit in 32 bits.

The ANY attribute may be specified only in the parameter descriptor in the declaration of an external entry. For example:

```
DECLARE SYS$GETTIM ENTRY (ANY);
```

This declaration indicates that the procedure SYS$GETTIM requires one argument to be passed by reference. When the ANY attribute is specified, no data type attributes are allowed in the parameter descriptor.

2.2.2 Variable-Length Argument Lists

In VAX-11 PL/I, the declaration of an external entry may specify OPTIONS(VARIABLE). This option can indicate one or more of the following:

- The specified entry point can be invoked with a variable number of arguments.
- Not all of the parameters of the entry point are listed in the parameter descriptor.
- Arguments for which the invoked procedure provides default values may be omitted from an argument list.

For example:

```
DECLARE SYS$ASCTIM ENTRY (ANY, CHARACTER(*))
                         OPTIONS (VARIABLE);
    .
    .
CALL SYS$ASCTIM (,TIME_BUFFER,,);
```

In this example, the system service SYS$ASCTIM is declared with two parameter descriptors and the VARIABLE option. The procedure actually has four parameters, but it provides default values for the first, third, and fourth parameters if they are not specified. The omission of these arguments is indicated in the procedure call by the commas that would separate the arguments if they were present. PL/I places zeros in the argument list longwords for these arguments.

When parameter descriptors are omitted, all parameters beyond those explicitly listed are considered to have the same attributes as the last parameter for which a descriptor is entered. In the preceding example, the final two arguments are assumed to have the CHARACTER(*) data type. However, since the arguments are not specified at the time of the call, PL/I places zeros in the argument list.

2.2.3 Global Symbols

A *global symbol* is a special type of external variable; it permits a PL/I procedure to share a variable with another PL/I procedure or with a procedure written in another language. The GLOBALDEF and GLOBALREF attributes define and declare global symbols. One PL/I procedure declares the global symbol using the GLOBALDEF attribute, as shown below:

```
DECLARE BUFFER CHARACTER (2048) GLOBALDEF;
```

Other procedures must declare this variable using the GLOBALREF attribute as follows:

```
DECLARE BUFFER CHARACTER(2048) GLOBALREF;
```

Global symbols with similar attributes may be specifically grouped into the same program section. This grouping provides more efficient use of memory than external variables declared using the PL/I EXTERNAL attribute, since each PL/I external variable requires a unique program section.

2.2.4 New Storage Classes

VAX-11 PL/I provides the READONLY and VALUE attributes to declare special storage classes for variables. The VALUE attribute specifies that a reference to the variable is a reference to the actual value of the variable, and not a reference to the variable's memory location.

Within the VAX/VMS operating system, global symbol values are used to represent such things as:

- Return values from system procedures or functions
- Offsets to values within system data structures
- Condition values signaled for exception conditions

The values of these symbols can be referenced in a PL/I program by declaring the names of these symbols with the GLOBALREF and VALUE attributes. For example, the names SS$_WASSET and SS$_WASCLR are system global symbol names. These names may be declared in PL/I as follows:

```
DECLARE (SS$_WASSET,SS$_WASCLR) GLOBALREF VALUE
            FIXED BINARY (31);
```

When a name is defined with the GLOBALREF and VALUE attributes, its actual value is not determined until the program is linked. The linker automatically locates the definitions of these symbols in the default system libraries.

The VALUE attribute may be applied only to fixed-point binary variables or to aligned bit-string variables with a length of 32 bits or less.

Any static variable in a VAX-11 PL/I program that will not be modified during the execution of the program can be declared with the READONLY attribute. For example:

```
DECLARE AM_MESSAGE CHARACTER(20) STATIC READONLY
          INITIAL ('Good morning');
```

The use of this attribute permits PL/I to allocate program sections efficiently. PL/I places all variables with the READONLY attribute in the same program section as the procedure's code. When the READONLY and GLOBALDEF attributes are combined, PL/I creates a read-only program section.

2.2.5 VAXCONDITION and ANYCONDITION

The normal PL/I mechanism for error signaling and error handling is compatible with the VAX–11 condition-signaling and condition-handling mechanism. VAX–11 PL/I has added the keywords listed below to extend these capabilities.

- VAXCONDITION — this keyword can specify a condition value for which the ON-unit is established.
- ANYCONDITION — this keyword can define an ON-unit that is executed when any condition is signaled for which no specific ON-unit exists.

These keywords can be specified in the ON, SIGNAL, and REVERT statements.

The VAX-specific or user-specific conditions that can be specified in the VAXCONDITION keyword can be defined as global symbol names. Thus, to define an ON-unit for a specific condition value, the condition name can be declared with the GLOBALREF attribute, as shown below:

```
DECLARE SS$_DECOVF GLOBALREF FIXED BINARY(31) VALUE;
ON VAXCONDITION (SS$_DECOVF) BEGIN;
    .
    .
   END;
```

This ON-unit receives control when the VAX condition SS$_DECOVF (fixed-point decimal overflow) is signaled.

2.2.6 Resignaling

Standard PL/I considers a condition "handled" when any ON-unit is found for the condition. Within the VAX/VMS condition-handling facility, a condition handler, that is, an ON-unit, can *resignal* a condition. When a condition is resignaled, the condition-handling facility continues its search for an ON-unit to handle the condition. In VAX–11 PL/I, an ON-unit can call the RESIGNAL built-in subroutine to resignal a condition.

For example, an ON-unit that handles more than one condition may want to give other ON-units a chance to gain control. This ON-unit may contain lines like the following:

```
ON ANYCONDITION BEGIN;
     IF (ONCODE() ^= SS$_DECOVF) & (ONCODE() ^= SS$_INTOVF)
         THEN CALL RESIGNAL();
         ELSE
              BEGIN;
                 .
                 .
                 END;
    END;
```

This ON-unit receives control when any condition is signaled. It checks the value of ONCODE for two specific conditions. If it is neither of these, the ON-unit calls the RESIGNAL built-in subroutine. PL/I then searches for another ON-unit, beginning in this block.

2.2.7 ONCODE and ONARGSLIST

When the ONCODE built-in function is referenced in an ON-unit in a PL/I program executing under the control of the VAX/VMS operating system, the value returned is always a unique system condition value. All system-defined condition values have symbolic names.

VAX-11 PL/I also provides the ONARGSLIST built-in function, which provides additional information to the ON-unit. For example, some conditions may have arguments associated with them; these arguments can be accessed by invoking the ONARGSLIST built-in function. The ONARGSLIST built-in function also provides access to hardware information associated with the condition, for example, the program counter (PC), and the contents of the general registers R0 and R1.

For details on using ON-units in VAX-11 PL/I, including information on declaring symbolic names for condition values, see the *VAX-11 PL/I User's Guide*.

Chapter 3
Using PL/I in the VAX/VMS Environment

This chapter describes the capabilities offered by the VAX/VMS operating system for developing PL/I programs. It contains:

- A sample terminal session that illustrates the commands to create, compile, link, and run a simple PL/I program
- A description of the VAX/VMS file system and of its logical naming capability, which provides device and file independence for program input/output
- An overview of the VAX/VMS librarian, which can be used for PL/I compile-time INCLUDE file libraries as well as for object module libraries
- An overview of command procedures, which can be used to catalog frequently executed sequences of commands

For a tutorial introduction to VAX/VMS and its command language, *DCL*, see the *VAX/VMS Primer*. Additional tutorial and reference information on any of the DCL commands presented in this chapter is contained in the *VAX/VMS Command Language User's Guide*. For more information on PL/I program development, see the *VAX-11 PL/I User's Guide*.

3.1 Sample Terminal Session

The sample terminal session begins on the next page. Terminal input/output, as it would appear if you entered all these lines, is shown on the left-hand page. Explanations of the commands in the examples are shown on the right.

```
(RET)                 ❶
Username: MALCOLM(RET)
Password:    (RET)

                             VAX/VMS Version 2.0


***** Time Sharing Until 20:00 *****
      ❷

$
$ EDIT METRIC.PLI(RET) ❸
Input:[DBA1:[MALCOLM]METRIC.PLI;1]
00100    * METRIC CONVERSION PROGRAM */(RET)
00200    CONVERT: PROCEDURE;(RET)
00300    DECLARE (INVALUE,OUTVALUE) FIXED DECIMAL (10,2),(RET)
00400            (INUNIT,OUTUNIT) CHARACTER(2);(RET)
00500    DECLARE UNITS (6,2) CHARACTER (2) STATIC INTERNAL(RET)
00600          INITIAL('in','cm','cm','in','ft','m ','m ',(RET)
00700          'ft','mi','Km','Km','mi');(RET)
00800    DECLARE FACTORS (6) FIXED DECIMAL (10,2) INTERNAL STATIC(RET)
00900          INITIAL (2.54,0.39,0.30,3.28,1.61,0.62);(RET)
01000    DECLARE INDEX FIXED BINARY;(RET)
01100    (RET)
01200            ON ENDPAGE(SYSPRINT);(RET)
01300    (RET)
01400            INDEX = 1;(RET)
01500            DO WHILE (INDEX ^= 0);(RET)
01600                PUT SKIP LIST ('Enter conversion mode:');(RET)
01700                PUT SKIP;(RET)
01800                PUT SKIP LIST ('1 - inches to centimeters');(RET)
01900                PUT SKIP LIST ('2 - centimeters to inches');(RET)
02000                PUT SKIP LIST ('3 - feet to meters');(RET)
02100                PUT SKIP LIST ('4 - meters to feet');(RET)
02200                PUT SKIP LIST ('5 - miles to Kilometers');(RET)
02300                PUT SKIP LIST ('6 - Kilometers to miles');(RET)
02400                PUT SKIP LIST ('0 - exit');(RET)
02500                PUT SKIP;(RET)
02600                PUT SKIP LIST ('mode? ');(RET)
02700                GET LIST INDEX;(RET)
02800                IF INDEX ^= 0 THEN DO;(RET)
02900                    IF (INDEX > 0) & (INDEX < 7) THEN DO;(RET)
03000                        PUT SKIP LIST ('Enter value to convert: ');(RET)
03100                        GET LIST (INVALUE);(RET)
03200                        OUTVALUE = INVALUE * FACTORS(INDEX);(RET)
03300                        INUNIT = UNITS(INDEX,1);(RET)
03400                        OUTUNIT = UNITS(INDEX,2);(RET)
03500                        PUT SKIP LIST (INVALUE,INUNIT,(RET)
03600                                     ' = ',OUTVALUE,OUTUNIT);(RET)
03700                        END;(RET)
03800                        ELSE PUT SKIP LIST( 'Invalid code -- retry');(RET)
03900          END;(RET)
04000    RETURN;(RET)
04100    END;(ESC)
*E(RET)
[DBA1:[MALCOLM]METRIC.PLI;1] ❹
$
```

1. *Login* to the system.

   To log in, press RET. The system prompts for a *user name*, then for a *password*. It does not display (echo) the password. After validating the name and password, it displays a system identification and daily messages, if any.

2. Wait for a command prompt.

   All requests to the system are entered by words that give a *command*. A dollar sign ($) prompt indicates that you can begin entering commands that will be interpreted by the system.

   All commands are English-language words that describe a request. Usually, a command is accompanied by *parameters* and *qualifiers* that limit the scope of the command. For example, if a command modifies a file in some way, the command parameter specifies the particular file of interest.

3. Create a PL/I source program.

   The EDIT command invokes the default system *editor*, SOS, to create the program METRIC.PLI. The message from SOS indicates that no file with the name METRIC.PLI currently exists and that one is being created. SOS begins prompting you to enter input lines.

   Each line must be terminated with RET.

4. Leave the editor.

   After the last line of input, press ESC. The asterisk (*) prompt indicates you can enter a command to SOS. The E (Exit) command terminates the editing session. SOS displays the name of the file that it has now saved on disk.

```
$ PLI METRIC(RET)             ❺
%PLIG-E-STMTSYNTOK, Invalid syntax in a 'get' statement.          ❻
                    'INDEX' was found where a '(' was expected.
                    Source file line number 27.

%PLIG-W-ENDGIVEN,  An END statement has been supplied to close
                  a do-group, begin-block, or procedure.
                  Source file line number 41.

%PLIG-E-TEXT, Completed with severe diagnostics. No object produced.
$
$ EDIT METRIC.PLI(RET)                       ❼
Edit:DBA1:[MALCOLM]METRIC.PLI;1]
*fGET(ESC)(RET)                                  ❽
02700                   GET LIST INDEX;
sINDEX(ESC)(INDEX)(ESC)(RET)                         ❾
02700                   GET LIST (INDEX);
*p.:*(RET)                                             ❿
02700                   GET LIST (INDEX);
02800                   IF INDEX ^= 0 THEN DO;
02900                       IF (INDEX > 0) & (INDEX < 7) THEN DO;
03000                           PUT SKIP LIST ('Enter value to convert: ');
03100                           GET LIST (INVALUE);
03200                           OUTVALUE = INVALUE * FACTORS(INDEX);
03300                           INUNIT = UNITS(INDEX,1);
03400                           OUTUNIT = UNITS(INDEX,2);
03500                           PUT SKIP LIST (INVALUE,INUNIT,
03600                                        ' = ',OUTVALUE,OUTUNIT);
03700                           END;
03800                          ELSE PUT SKIP LIST( 'Invalid code -- retry');
03900             END;
04000       RETURN;
04100       END;
*i3800(RET)            ⓫
03850                   END;(RET)
*E(RET)              ⓬
[DBA1:[MALCOLM]METRIC.PLI;2]
$
```

5. Invoke the VAX-11 PL/I *compiler.*

   The PLI command invokes the VAX-11 PL/I compiler to compile METRIC. The PLI command assumes, when no file type is specifed, that the file type is PLI.

6. Examine the *diagnostic* messages.

   The messages from the compiler indicate that it detected a syntax error in a GET statement and a missing END statement.

7. Correct the source program.

   The EDIT command again invokes the editor, this time to modify the existing file METRIC.PLI.

8. Locate a line in the file.

   To locate a line in an SOS file, use the F (Find) command to specify a string to locate. The search string must be terminated with ESC.

9. Change a character string in the line.

   The S (Substitute) command changes character strings. This command puts parentheses around the string INDEX. In a substitute command, ESC delimits the string to be changed and the new string.

10. Display the remaining lines in the file.

    The P.:* command requests SOS to display the lines in the file between the current line (symbolized by a period) and the end of the file (symbolized by the asterisk). The colon symbol denotes a range of lines on which SOS is to act.

11. Insert a line in the file.

    The I (Input) command adds a line to the file. The i3800 command indicates that a line is to be added after the line numbered 3800 SOS prompts for the line 3850. The END statement is added.

12. Leave the editor.

    The E (Exit) command terminates this session. The editor does not change the original version of the file, but creates a new copy. This is indicated by the version number of 2 in the file specification displayed by SOS.

```
$
$
$ PLI METRIC(RET)  ⓭
$
$
$
$ LINK METRIC(RET)  ⓮
$
$
$
$ RUN METRIC(RET)  ⓯

Enter conversion mode:

1 - inches to centimeters
2 - centimeters to inches
3 - feet to meters
4 - meters to feet
5 - miles to Kilometers
6 - Kilometers to miles
0 - exit

mode?  5(RET)

Enter value to convert:  32(RET)

        32.00   mi       =               51.52   Km

Enter conversion mode:

1 - inches to centimeters
2 - centimeters to inches
3 - feet to meters
4 - meters to feet
5 - miles to Kilometers
6 - Kilometers to miles
0 - exit

mode?  8(RET)

Invalid code -- retry

Enter conversion mode:
1 - inches to centimeters
2 - centimeters to inches
3 - feet to meters
4 - meters to feet
5 - miles to Kilometers
6 - Kilometers to miles

0 - exit

mode?  2(RET)

Enter value to convert:  86(RET)

        86.00   cm       =               33.54   in
```

13. Invoke the compiler.

    The PLI command again invokes the PL/I compiler. This time, there are no messages. When the compilation is successful, the compiler creates an *object module* consisting of the machine language instructions to execute the program. This object module is written to the file named METRIC.OBJ.

    To obtain a *listing* of the program, you must explicitly request a listing when you enter the PLI command interactively. Specify:

    ```
    $ PLI METRIC/LIST RET
    ```

    where /LIST is a qualifier for the PLI command. The PLI command creates a listing file named METRIC.LIS, which can be printed with the command:

    ```
    $ PRINT METRIC RET
    ```

14. Link the object module.

    The LINK command invokes the *linker* to bind object modules into an executable *image*. In this example, only one object module is supplied. When multiple files are being linked, you separate them with commas, for example:

    ```
    $ LINK A,B,C
    ```

    The LINK command assumes that the file type of an input file is OBJ. The linker automatically searches the run-time library to locate any PL/I routines that are referenced in the object module(s).

    The executable image file created by the linker from METRIC.OBJ is METRIC.EXE.

15. Execute the program.

    The RUN command initiates the execution of the image. The program METRIC continues to prompt for a type of metric conversion and to calculate a result until a 0 is entered in response to the prompt.

```
Enter conversion mode:

1 - inches to centimeters
2 - centimeters to inches
3 - feet to meters
4 - meters to feet
5 - miles to kilometers
6 - kilometers to miles
0 - exit

mode?  0(RET)
$ logoff(RET)  ⓰
  MALCOLM         logged out at 14-MAR-1980 14:32:17.31
```

16. *Logoff* the system.

    The LOGOFF command terminates the connection with the computer.

3.2 The File System

In VAX/VMS, a file can be uniquely identified in terms of:

- The *device* on which the file resides, for example, a disk or a tape
- If the device is a disk, the *directory* in which the file is cataloged
- A *file name*, a zero- to nine-character name given by the user who created the file
- The *file type*, a zero- to three-character name that is either supplied by the user who created the file or supplied by default by a program that created the file as an output file
- The *file version number*, a numeric value indicating the version of the file

All of these elements comprise a *file specification*. A file specification has the format:

```
device:[directory]filename.filetype;version-number
```

The colon, brackets, period, and semicolon are required syntactic delimiters. A file specification may also be preceded with a node name followed by a double colon, if the current system is connected to a network.

When a file specification does not specify all of these items, the system or the program that is processing the file provides default values.

Some examples of file specifications are shown below. The examples also illustrate some of the more frequently used VAX/VMS commands that process files.

```
$ DIRECTORY/FULL  DBA1:[MALCOLM]METRIC.PLI (RET)
```

This file specification indicates the PLI source file cataloged in the directory MALCOLM on the disk device DBA1. The DIRECTORY command with the /FULL qualifier lists a complete set of information about the file and its attributes.

```
$ TYPE [MUGGSIE]ALPHA.SRT (RET)
```

The TYPE command displays the contents of a file on the terminal. This file specification indicates a file ALPHA.SRT that is cataloged in the directory [MUGGSIE]. Since no device is given, a default value is provided. The absence of a device name always defaults to the default device defined for a user; this is almost always a disk device.

```
$ DELETE METRIC.OBJ;1 (RET)
```

The DELETE command releases all the disk space occupied by a file and makes the file inaccessible. The explicit version number indicates which version of the file METRIC.OBJ in the current default directory is to be deleted. The DELETE command requires either a version number or a ; with no number. When no number is specified, the DELETE command deletes the most recent version of a file.

```
$ PRINT METRIC RET
```

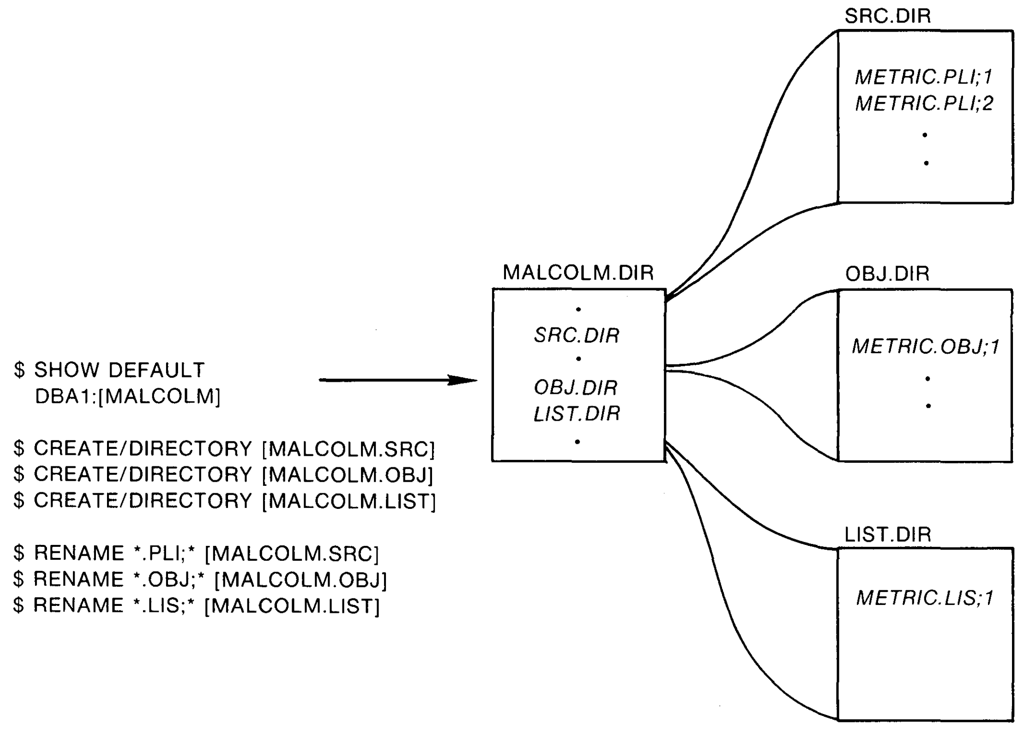
The PRINT command uses the current default device and provides a default file type of LIS. This command prints the file METRIC.LIS on the system line printer.

3.2.1 Subdirectories

When a user is given an account on a VAX/VMS system, the system manager generally provides a default directory for the user's personal use. Within a private directory, a user can create a set of *subdirectories*, and place the entries for related files in the same subdirectory.

The CREATE/DIRECTORY command creates a subdirectory. Figure 3-1 illustrates the creation of a simple directory hierarchy. The user MALCOLM creates separate directories for source files, object modules, and listings. After the creation of the three directories, the RENAME command changes the directories associated with the files in the default directory [MALCOLM].

The asterisks (*) in the file specifications for the RENAME command indicate "all files" — thus, all files with file types of PLI are renamed to the directory [MALCOLM.SRC]; all files with file types of OBJ are renamed to the directory [MALCOLM.OBJ]; and all files with the file type LIS are renamed to the directory [MALCOLM.LIST]. When the files are renamed and the RENAME command does not specify output file names and file types, the output files have the same file names and file types as the input files.



```
$ SHOW DEFAULT
  DBA1:[MALCOLM]

$ CREATE/DIRECTORY [MALCOLM.SRC]
$ CREATE/DIRECTORY [MALCOLM.OBJ]
$ CREATE/DIRECTORY [MALCOLM.LIST]

$ RENAME *.PLI;* [MALCOLM.SRC]
$ RENAME *.OBJ;* [MALCOLM.OBJ]
$ RENAME *.LIS;* [MALCOLM.LIST]
```

Figure 3-1: Creating a Directory Hierarchy

3.2.2 Logical Names

A *logical name* represents a device or file specification. When a logical name is used in a command or in a program, the system translates the logical name and uses the file specification with which the logical name is associated. For example:

```
OPEN FILE (INFILE) RECORD INPUT;
```

When this OPEN statement is executed in a PL/I program, PL/I uses the default title INFILE. Then, the run-time system will attempt to translate the logical name INFILE. If such a logical name exists, the related file specification, or *equivalence name*, is the name that is actually used.

A logical name and its equivalence name become associated by a DEFINE command:

```
$ DEFINE INFILE ALPHA.SRT RET
```

This DEFINE command creates the logical name INFILE and associates it with the file specification ALPHA.SRT. Each time a program is run, a different equivalence can be made for a logical name. When no logical name assignment exists for INFILE, VAX-11 PL/I provides a default file specification of INFILE.DAT.

VAX/VMS provides default logical name equivalences for the default disk device and for the default terminal. These names are:

- SYS$DISK — the default disk device.
- SYS$INPUT — the default input device. This logical name is associated with the default PL/I file SYSIN.
- SYS$OUTPUT — the default output device. This logical name is associated with the default PL/I file SYSPRINT.
- SYS$ERROR — the default diagnostic and error message output device.
- SYS$COMMAND — the default command input device.

These names can be used as equivalence names in logical name assignments as well. Consider the OPEN statement:

```
OPEN FILE (REPORT_DATA) OUTPUT RECORD SEQUENTIAL;
```

To verify the output of this program for testing purposes, the following equivalence can be made for the file REPORT_DATA:

```
$ DEFINE REPORT_DATA SYS$OUTPUT RET
```

When WRITE statements in this program write records to the file REPORT_DATA, the records are actually displayed on the terminal.

3.3 The VAX/VMS Librarian

A *library* is a collection of files contained within another, larger file. The library contains its own directory of files within it. There are two types of library file of interest to PL/I programmers:

- Text libraries — these libraries contain modules of PL/I source statements to be copied at compile time by %INCLUDE statements. A text library file has the file type TLB.
- Object module libraries — these libraries contain external subroutines and functions that are frequently invoked by PL/I programs. An object module library file has the file type OLB.

You can create either of these types of library with the DCL command LIBRARY. This command also lists the modules in a library and permits the addition, deletion, extraction, and replacement of modules within a library.

3.3.1 INCLUDE Files and Libraries

An *INCLUDE file* is an external file containing PL/I source text. The contents of the file can be copied into any PL/I program during compilation by means of the PL/I statement %INCLUDE. For example:

```
%INCLUDE 'APPLIC.SYM';
```

This %INCLUDE statement copies the contents of the file APPLIC.SYM from the current default directory into the PL/I source program. The default file type for INCLUDE files is PLI. For example, the following statement includes the contents of DECLARE.PLI:

```
%INCLUDE 'DECLARE';
```

When individual INCLUDE files are combined into libraries, the name of the library may be specified on the PLI command. When a %INCLUDE statement in the program specifies a module name, the compiler will search that library for the module. For example:

```
%INCLUDE APPLIC;
```

This statement requests the module named APPLIC. When a module is included in a source file from a library, its name must not be enclosed in apostrophes. Figure 3–2 illustrates creating the library that contains the module APPLIC and specifying the library in a PLI command.

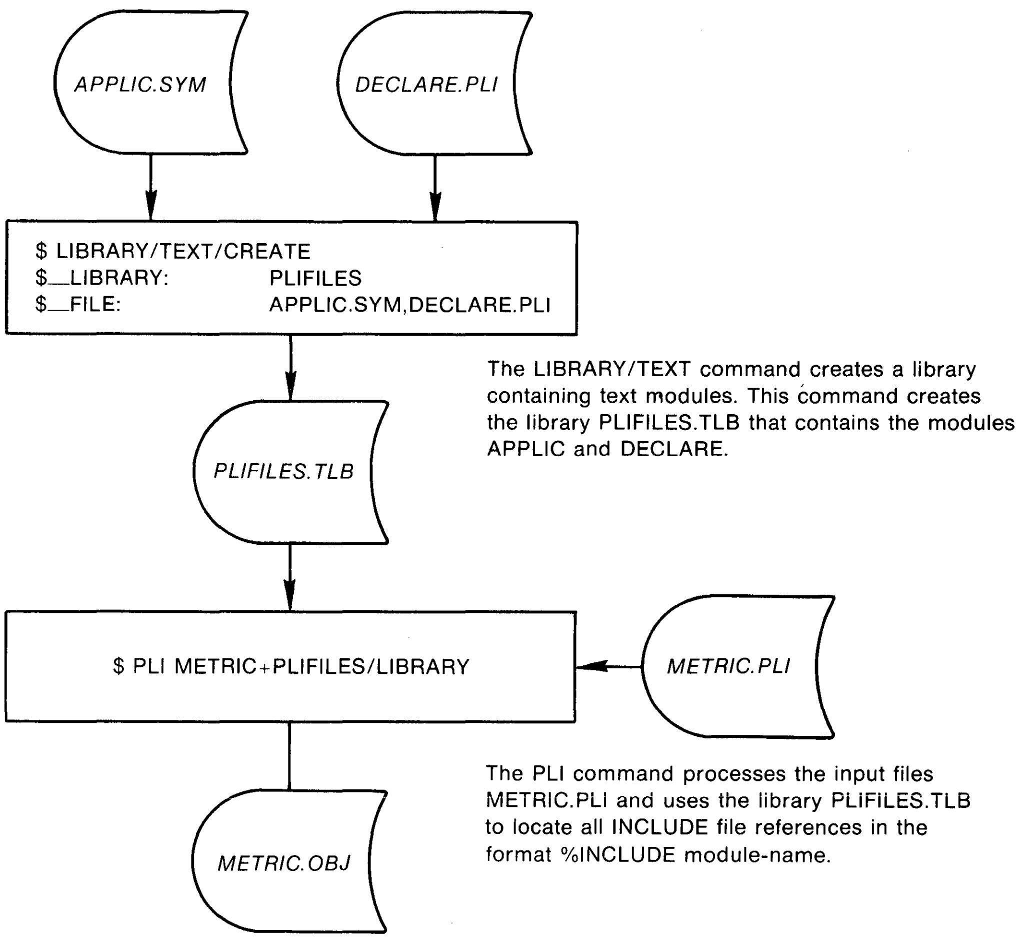

APPLIC.SYM
DECLARE.PLI
$ LIBRARY/TEXT/CREATE
$_LIBRARY: PLIFILES
$_FILE: APPLIC.SYM,DECLARE.PLI
The LIBRARY/TEXT command creates a library containing text modules. This command creates the library PLIFILES.TLB that contains the modules APPLIC and DECLARE.
PLIFILES.TLB
$ PLI METRIC+PLIFILES/LIBRARY
METRIC.PLI
The PLI command processes the input files METRIC.PLI and uses the library PLIFILES.TLB to locate all INCLUDE file references in the format %INCLUDE module-name.
METRIC.OBJ

Figure 3-2: Creating and Using an INCLUDE File Library

3.3.2 Object Module Libraries

An object module library can contain external procedures and functions. When the library is specified on a LINK command, the linker automatically searches the library if it cannot resolve references to external variables, procedures, or functions. For example, if the program METRIC contains a reference to an external entry named KILOMETERS that is in the library DEFLIB.OLB, the module METRIC may be linked as follows:

```
$ LINK METRIC,DEFLIB/LIBRARY RET
```

This LINK command specifies that the object module library DEFLIB.OLB be searched for modules that are referenced but not contained in the module METRIC.OBJ.

In VAX/VMS, you can define a default user library for the linker to search by equating the logical name LNK$LIBRARY with the name of an object module library. For example:

```
$ DEFINE LNK$LIBRARY APPLIC.OLB RET
```

When this logical name assignment is in effect, the linker will search the library APPLIC.OLB after it searches all object modules and libraries specified on the LINK command. Additional default user libraries can be defined with the logical names LNK$LIBRARY_1, LNK$LIBRARY_2, and so on.

After searching object modules and libraries specified on the LINK command and then any default user libraries, the linker searches the default system libraries named VMSRTL.EXE and STARLET.OLB.

3.4 Command Procedures

Sequences of commands that are frequently executed in a particular order can be placed in files called *command procedures*. A command procedure is simply a file that contains DCL commands. For example, a command file named PLIMETRIC.COM can contain the lines:

```
$ PLI METRIC
$ LINK METRIC
```

Each line in the file represents a command for the system to process. The lines are executed when the name of the procedure is specified following an @ (Execute Procedure) command. The procedure PLIMETRIC is executed as follows:

```
$ @PLIMETRIC RET
```

Command procedures can be very simple, like the two-line example shown above, or they can be very complex. For example, arguments can be passed to a command procedure in much the same way that arguments are passed within a PL/I program. Each argument is associated with one of the symbolic names P1, P2, and so on. For example:

```
$ PLI 'P1'
$ LINK 'P1'
```

This procedure compiles and links any PL/I source program whose name is passed to it as a parameter. If the procedure's name is PLILINK.COM, it could be invoked as follows:

```
$ @PLILINK METRIC RET
```

In this example, the parameter METRIC is substituted for the symbol P1. This execution of the procedure compiles METRIC.PLI and links METRIC.OBJ.

The VAX/VMS command language provides additional capabilities for use in command procedures, including:

- Symbol definition and assignment for character strings and integers
- Error processing by means of an ON statement
- Procedure control by means of IF and GOTO statements
- Options for processing command procedures as batch jobs

For additional information on creating and using command procedures, see the *VAX/VMS Guide to Using Command Procedures*.

VAX–11 PL/I Language Summary

1.0 Introduction

This document summarizes the syntax of VAX–11 PL/I statements.

This summary is intended as a quick reference to use when writing PL/I programs and not as a formal or complete description of the language. More detailed information on VAX–11 PL/I features can be found in the *VAX–11 PL/I Encyclopedic Reference* and *VAX–11 PL/I User's Guide.*

2.0 Symbols and Conventions

- Brackets ([ ]) enclose optional language elements. Long brackets enclose lists of elements from which one and only one element may be chosen.
- Braces ({ }) enclose lists of items from which one and only one item must be chosen.
- Text in green ink describes language features that are not in the proposed ANSI General Purpose Subset (BSR X3.74). All features of the subset are in VAX–11 PL/I.

3.0 Statements

allocate-statement:

$$\left\{\begin{array}{c}\text{ALLOCATE}\\ \text{ALLOC}\end{array}\right\} \text{identifier [SET(reference)];}$$

assignment-statement:

reference = expression;

See also Section 5.0, "Expressions and References."

begin-statement:

BEGIN;

call-statement:

CALL reference [(argument,...)];

The reference must be to a built-in subroutine or to an entry point defined without the RETURNS option.

close-statement:

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

declare-statement:

$$\left\{\begin{array}{c}\text{DECLARE}\\ \text{DCL}\end{array}\right\} \text{declaration,...;}$$

declaration:

$$\text{[level]} \left\{\begin{array}{c}\text{identifier}\\ \text{(declaration,...)}\end{array}\right\} \left[\begin{array}{c}\text{([lower-bound:]upper-bound)}\\ \text{(*)}\end{array} \quad \text{[attribute ...]}\right]$$

The bound expressions must have integer values.

delete-statement:

```
DELETE FILE(reference) [KEY(expression)] [OPTIONS(option,...)];

option:
        FAST_DELETE
        INDEX_NUMBER(expression)
        [ MATCH_GREATER             ]
        [ MATCH_GREATER_EQUAL       ]
        RECORD_ID(reference)
```

do-statement:

```
DO [reference=expression] [[TO expression] [BY expression]] [WHILE(expression)];
                          [      REPEAT expression        ]
```

end-statement:

```
END [identifier];
```

entry-statement:

```
ENTRY [(parameter-identifier,...)] [RETURNS(data-attribute ...)];
```

See also Section 4.0, "Attributes."

format-statement:

```
label: FORMAT (format-specification,...);

     format-specification:
             { format-item                                        }
             { format-iteration-factor format-item                }
             { format-iteration-factor(format-specification,...)  }
```

Format Item	Format Specified	Remarks
F(w[,d])	fixed point format	w and d must be integers
E(w[,d])	floating point format	w and d must be integers
P 'picture'	picture format	see PICTURE attribute
A[(w)]	character format	w required on input
B[(w)]	bit string format	w required on input
B1[(w)]	bit string format	w required on input
B2[(w)]	base-4 string format	w required on input
B3[(w)]	octal string format	w required on input
B4[(w)]	hexadecimal format	w required on input
TAB[(count)]	tab control	count must be an integer; print files only
LINE(n)	line control	n must be an integer; print files only
SKIP[(count)]	line control	count must be an integer
COL[UMN](n)	column control	n must be an integer
X [(count)]	spacing control	count must be an integer
PAGE	page control	print files only
R(label)	remote format	label is of FORMAT statement

free-statement:

```
    FREE [reference->] identifier;
```

get-statement:

```
    GET ┌ LIST(input-target,...)                                    ┐
        └ EDIT(input-target,...) (format-specification,...)         ┘

    ┌ [FILE(reference)] [SKIP[(expression)]] [OPTIONS(option,...)] ┐
    └                   STRING(expression)                         ┘

    ;
```

options:

```
        NO_ECHO
        NO_FILTER
        PROMPT (expression)
        PURGE_TYPE_AHEAD
```

For the definition of format specifications, **see** "format-statement."

input-target:

```
{                                              reference                                         }
{ ( input-target,... DO [ reference=expression [[TO expression] [BY expression]  ]] [WHILE(expression)] ) }
{                                              [           REPEAT expression    ]                }
```

goto-statement:

```
   { GOTO  }
   { GO TO } label;
```

if-statement:

```
   IF expression THEN   { do-group    }   [ ELSE { do-group    } ]
                        { begin-block }          { begin-block }
                        { statement   }          { statement   }
```

include:

```
   %INCLUDE {    'file-spec'     }   ;
            { text-module-name   }
```

The file-spec must be a valid VAX/VMS file specification identifying a file of valid PL/I source text. The text-module name must be a 1- to 31-character name of a module in a library of INCLUDE modules.

on-statement:

```
       ⎧ ANYCONDITION                  ⎫
       ⎪ ENDFILE (reference)           ⎪
       ⎪ ENDPAGE (reference)           ⎪
       ⎪ FINISH                        ⎪
       ⎪ KEY (reference)               ⎪
       ⎪ UNDEFINEDFILE (reference)     ⎪  {statement  }
   ON  ⎨ ERROR                         ⎬  {begin-block}
       ⎪ FIXEDOVERFLOW                 ⎪
       ⎪ OVERFLOW                      ⎪
       ⎪ UNDERFLOW                     ⎪
       ⎪ VAXCONDITION (expression)     ⎪
       ⎩ ZERODIVIDE                    ⎭
```

open-statement:

```
   OPEN FILE (reference) [TITLE(expression)]

       ⎡ [STREAM] ⎡[INPUT]  [LINESIZE(integer)] [ PRINT [PAGESIZE(integer)] ] ⎤ ⎤
       ⎢          ⎣OUTPUT                                                      ⎦ ⎥
       ⎢                                                                         ⎥
       ⎢          ⎡[INPUT] ⎤ ⎡DIRECT          ⎤                                  ⎥
       ⎢ RECORD   ⎢OUTPUT  ⎥ ⎣[SEQ[UENTIA]L]  ⎦  [KEYED]                         ⎥
       ⎣          ⎣UPDATE  ⎦                                                     ⎦

       [ENVIRONMENT(environment-option,...)]
   ;
```

The reference is a reference to a declared file constant or variable. For a description of environment options, **see** "attribute." If a file has certain individual attributes, additional attributes are implied, as follows:

Has:	**Implied:**
DIRECT	RECORD KEYED
KEYED	RECORD
PRINT	STREAM OUTPUT
SEQUENTIAL	RECORD
UPDATE	RECORD

If the statement has RECORD but not DIRECT or SEQUENTIAL, SEQUENTIAL is implied. Certain attributes are implied if the statement lacks either of two alternatives, as follows:

Lacks:	**Implied:**
STREAM or RECORD	STREAM
INPUT, OUTPUT, or UPDATE	INPUT

procedure-statement:

```
    {PROCEDURE} [(parameter-identifier,...)] [OPTIONS(option,...)] [RETURNS(data-attribute ...)] [RECURSIVE];
    {  PROC   }
```

options:

```
    MAIN
    IDENT(expression)
    UNDERFLOW
```

put-statement:

```
PUT ⎡ LIST (output-source,...)                               ⎤
    ⎣ EDIT (output-source,...) (format-specification,...)    ⎦

⎡ [FILE(reference)] [PAGE] [LINE[(expression)]] [SKIP[(expression)]] [OPTIONS(CANCEL_CONTROL_O)] ⎤
⎢                                                                                              ⎥
⎣                                  STRING(reference)                                           ⎦

;
```

For definition of format specification, **see** "format-statement."

output-source:

```
⎧                                         expression                                                          ⎫
⎨( output-source,... DO [ reference = expression ⎡[TO expression] [BY expression]⎤ ] [WHILE (expression)] ) ⎬
⎩                                                ⎣      REPEAT expression        ⎦                            ⎭
```

read-statement:

```
READ FILE (reference) ⎧INTO(reference)⎫ ⎡KEY(expression)  ⎤ [OPTIONS(option,...)];
                      ⎩SET(reference) ⎭ ⎣KEYTO(reference) ⎦
```

options:

```
FIXED_CONTROL_FROM(reference)
INDEX_NUMBER (expression)
⎡ MATCH_GREATER        ⎤
⎣ MATCH_GREATER_EQUAL  ⎦
RECORD_ID(reference)
RECORD_ID_TO (reference)
```

replace:

```
%REPLACE identifier BY constant;
```

return-statement:

```
RETURN [(expression)];
```

revert-statement:
    REVERT condition-name;

    For definition of condition name, **see** "on-statement."

rewrite-statement:
    REWRITE FILE(reference) [FROM(reference) [KEY(expression)]] [OPTIONS(option,...)];

        options:
            FIXED_CONTROL_FROM(reference)
            INDEX_NUMBER(expression)
            $\begin{bmatrix}\text{MATCH\_GREATER}\\ \text{MATCH\_GREATER\_EQUAL}\end{bmatrix}$
            RECORD_ID(reference)
            RECORD_ID_TO(reference)

signal-statement:
    SIGNAL condition-name;

        For definition of condition name, **see** "on-statement."

stop-statement:
    STOP;

write-statement:
    WRITE FILE(reference) FROM(reference) [KEYFROM(expression)] [OPTIONS(option,...)];

        options:
            FIXED_CONTROL_FROM(reference)
            INDEX_NUMBER (expression)
            RECORD_ID_TO (reference)

4.0 Attributes

attribute:

Computational data:

```
⎧ CHAR[ACTER] [(length)] [VAR[YING]]                      ⎫
⎪ BIT [(length)] [ALIGNED]                                ⎪
⎪ PICTURE 'picture'                                       ⎪
⎨ FLOAT [BIN[ARY]] [(precision)]                          ⎬
⎪ FIXED [BIN[ARY]] [(precision)]                          ⎪
⎪ BIN[ARY] [FLOAT] [(precision)]                          ⎪
⎪ DEC[IMAL] [FIXED] [(precision[,scale-factor])]          ⎪
⎩ FLOAT DEC[IMAL] [(precision)]                           ⎭
```

Picture:

Picture Character	Meaning
9	Decimal digit, including leading zeros
Z	Decimal digit with leading-zero suppression
*	Decimal digit with asterisk for leading zero
Y	Decimal digit with space for any zero
V	Position of assumed decimal point
(n)	Iteration factor for subsequent character
T	Position of digit and encoded plus sign or minus sign
I	Position of digit and encoded plus sign if number >= 0
R	Position of digit and encoded minus sign if number < 0
.	Position at which decimal point is inserted
,	Position at which comma is inserted
/	Position at which slash is inserted
B	Position at which space is inserted

(Continued on next page)

Picture Cont:

Picture Character	Meaning
$	Position[s] of [drifting] dollar sign
+	Position[s] of [drifting] plus sign if number >=0
–	Position[s] of [drifting] minus sign if number < 0
S	Position[s] of [drifting] plus sign or minus sign
CR	Positions at which 'CR' is inserted if number < 0
DB	Positions at which 'DB' is inserted if number < 0

After all its iterations are expanded and all its insertion characters are removed, a picture must satisfy the following syntax rules (the notation character... indicates a series of the same character, with no embedded characters).

picture:
'[left-part]center-part[right-part]'

left-part:

$$\begin{Bmatrix} \$ \\ + \\ - \\ \text{S} \end{Bmatrix}$$

right-part:

$$\begin{Bmatrix} \text{left-part} \\ \text{CR} \\ \text{DB} \end{Bmatrix}$$

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 may appear wherever 9 is valid except that only one character T, I, or R may appear in a picture, and a picture may not contain T, I, or R if it also contains S, +, -, CR, or DB.

Noncomputational data:

```
/ AREA                   \
| ENTRY [VARIABLE]       |
| FILE [VARIABLE]        |
| LABEL                  |
| OFFSET                 |
\ POINTER                /
```

Storage class:

```
[ [AUTO[MATIC]] [INIT[IAL] (initial-element,...)]                ]
[ BASED [(reference)]                                            ]
[ DEF[INED] (reference) [POSITION(expression)]                   ]
[ STATIC [INIT[IAL](initial-element,...)] [READONLY]             ]
```

initial-element:

```
/ string-constant                               \
| [(iteration-factor)] arithmetic-constant      |
| (iteration-factor) scalar-reference           |
| (iteration-factor) (scalar-expression)        |
\ [(iteration-factor)] *                        /
```

STATIC variables may be initialized only with constants and with the NULL built-in function.

Scope:

```
[ EXT[ERNAL] [ GLOBALDEF[(psect-name)]   [ READONLY ] ] ]
[            [ GLOBALREF                 [ VALUE    ] ] ]
[                                                       ]
[ [INT[ERNAL]]                                          ]
```

File description:

```
┌ [STREAM] ┌[INPUT]      [LINESIZE(integer)] [PRINT [PAGESIZE(integer)] ] ┐ ┐
│          └OUTPUT                                                        ┘ │
│                                                                           │
│           ┌[INPUT] ┐                                                      │
│ RECORD    │OUTPUT  │   ┌DIRECT            ┐                               │
└           └UPDATE  ┘   └[SEQ[UENTIAL]L]   ┘ [KEYED]                       ┘

  [ENVIRONMENT(environment-option,...)]
```

Argument-passing:

```
  [ANY] [VALUE]
```

Entry-name attributes:

```
  {BUILTIN                                                                  }
  {ENTRY [VARIABLE] [OPTIONS(VARIABLE)] [RETURNS (data-attribute ...)]      }
```

Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
APPEND	Places output for a file at the end of an existing file.	Create Open	Record Stream	Disabled	BIT(1)
BATCH	Submits a copy of the file to the system batch job queue on close.	Create Open Close	Record Stream	Disabled	BIT(1)
BLOCK_BOUNDARY_FORMAT	Indicates that records must not cross block boundaries.	Create	Record Stream	Disabled	BIT(1)
BLOCK_IO	Specifies a file will be read or written by blocks instead of records.	Create Open	Record	Disabled	BIT(1)
BLOCK_SIZE(expression)	Specifies the size of a block for the creation of a magnetic tape file.	Create	Record Stream	Mount value	FIXED BINARY(31)
BUCKET_SIZE(expression)	Defines the number of 512-byte blocks in a bucket for an indexed sequential or a relative file.	Create	Record	Maximum record size	FIXED BINARY(31)
CARRIAGE_RETURN_FORMAT	Indicates that records in the file will be printed with default carriage control.	Create	Record	Enabled	BIT(1)
CONTIGUOUS	Specifies that an output file must be placed in a physically contiguous extent on disk.	Create	Record Stream	Disabled	BIT(1)
CONTIGUOUS_BEST_TRY	Requests that if possible an output file be placed in a physically contiguous extent on disk.	Create	Record Stream	Disabled	BIT(1)
CREATION_DATE(variable)	Overrides default creation date of file.	Create	Record Stream	Current date and time	BIT (64) ALIGNED
CURRENT_POSITION	Leaves magnetic tape positioned at last close.	Create Open	Record Stream	Disabled	BIT(1)

(Continued on next page)

Summary of ENVIRONMENT Options (Cont.)

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
DEFAULT_FILE_NAME(expression)	Defines a default file specification for a file.	Create Open	Record Stream	'.DAT'	CHAR(128)
DEFERRED_WRITE	Requests file system optimization of output.	Create Update	Record	Disabled	BIT(1)
DELETE	Specifies that the file be deleted when it is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
EXPIRATION_DATE(variable)	Defines the expiration date for a magnetic tape file.	Create	Record Stream	Creation date	BIT (64) ALIGNED
EXTENSION_SIZE(expression)	Specifies a default extension size for a disk file.	Create Update	Record Stream	System default	FIXED BINARY(31)

FILE_ID(variable)
FILE_ID_TO(variable)
	Identifies a file by its internal file identification.	Create Open	Record Stream	n/a
				n/a
					(6) FIXED BINARY(31)
					(6) FIXED BINARY(31)

FILE_SIZE(expression)	Defines the initial number of blocks to allocate for a file.	Create	Record Stream	n/a	FIXED BINARY(31)

FIXED_CONTROL_SIZE(expression)
FIXED_CONTROL_SIZE_TO(variable)
	Defines records as variable length with fixed-length control and specifies the size of the fixed control area. On open, returns the length of the fixed control area.	Create Open	Record	Disabled	FIXED BINARY(31)

FIXED_LENGTH_RECORDS	Specifies a file with fixed-length records of a maximum record size.	Create	Record	Disabled	BIT(1)
GROUP_PROTECTION(expression)	Defines the type of file access allowed to members of the owner's group.	Create	Record Stream	Current process default	CHAR(4)

(Continued on next page)

Summary of ENVIRONMENT Options (Cont.)

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
IGNORE_LINE_MARKS	Specifies that end-of-line characters are not to be treated as field delimiters in GET LIST statements.	Create Open	Stream	Disabled	BIT(1)
INDEX_NUMBER(expression)	Specifies the initial index to use in accessing records in an indexed sequential file.	Create Open	Record	0	FIXED BINARY(31)
INDEXED	Defines an indexed sequential file.	Create Open	Record	Disabled	BIT(1)
INITIAL_FILL	Requests the file system to leave unused space in file index overflow buckets.	Open	Record	Disabled	BIT(1)
MAXIMUM_RECORD_NUMBER(expression)	Specifies the largest record number that will be valid for records in a relative file.	Create	Record	0	FIXED BINARY(31)
MAXIMUM_RECORD_SIZE(expression)	Specifies the maximum size that is valid for any record in the file.	Create	Record	512 bytes	FIXED BINARY(31)
MULTIBLOCK_COUNT(expression)	Specifies the number of blocks to allocate for file system buffering.	Create Open	Record	Current process default	FIXED BINARY(31)
MULTIBUFFER_COUNT(expression)	Specifies the number of buffers to allocate for file system buffering.	Create Open	Record	Current process default	FIXED BINARY(31)
NO_SHARE	Prohibits all type of shared access to the file.	Create Open	Record	*	BIT(1)
OWNER_GROUP(expression)	Specifies the group number in the user identification code (UIC) of the owner of the file.	Create	Record Stream	Current process group number	FIXED BINARY(31)

* Disabled if the file is opened for input, enabled if opened for output or update.

(Continued on next page)

Summary of ENVIRONMENT Options (Cont.)

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
OWNER_MEMBER(expression)	Specifies the member number in the user identification code (UIC) of the owner of the file.	Create	Record Stream	Current process member number	FIXED BINARY(31)
OWNER_PROTECTION(expression)	Specifies the type of file access allowed the owner of the file.	Create	Record Stream	Current process default	CHAR(4)
PRINTER_FORMAT	Specifies that records in the file will be printed using printer format carriage control embedded in the fixed control area of the records.	Create	Record	Disabled	BIT(1)
READ_AHEAD	Requests file system optimization on read operations.	Open	Record Stream	Enabled	BIT(1)
READ_CHECK	Requests verification of read operations.	Create Open	Record Stream	Disabled	BIT(1)
RECORD_ID_ACCESS	Indicates that records will be accessed by internal file system identification.	Create Open	Record	Disabled	BIT(1)
RETRIEVAL_POINTERS(expression)	Specifies the number of file system extent pointers to maintain for file access.	Create Open	Record Stream	Current system default	FIXED BINARY(31)
REWIND_ON_CLOSE	Requests that a magnetic tape volume be rewound when the file is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
REWIND_ON_OPEN	Requests that a magnetic tape volume be rewound when the file is opened.	Create Open	Record Stream	Enabled	BIT(1)

(Continued on next page)

Summary of ENVIRONMENT Options (Cont.)

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
SCALARVARYING	Specifies that varying character strings will be read/written using the entire storage of the variable.	Create Open	Record	Disabled	BIT(1)
SHARED_READ	Allows other users to read records in the file.	Create Open	Record	*	BIT(1)
SHARED_WRITE	Allows other users to read and write records in the file.	Create Open	Record	Disabled	BIT(1)
SPOOL	Queues a copy of the file to the system printer when the file is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
SUPERSEDE	Replaces an existing file with the same file name, file type, and version number.	Create	Record Stream	Disabled	BIT(1)
SYSTEM_PROTECTION(expression)	Defines the type of file access allowed to users with system user identification codes.	Create	Record Stream	Current process default	CHAR(4)
TEMPORARY	Specifies a temporary file for which no directory entry is made.	Create	Record Stream	Disabled	BIT(1)
TRUNCATE	Truncates a sequential file at its logical end-of-file when it is closed.	Create Update Close	Record Stream	Disabled	BIT(1)
WORLD_PROTECTION(expression)	Specifies the type of file access allowed to general system users.	Create	Record Stream	Current process default	CHAR(4)
WRITE_BEHIND	Requests file system optimization on output operations.	Create Update	Record Stream	Disabled	BIT(1)
WRITE_CHECK	Requests verification of output operations.	Create Update	Record Stream	Disabled	BIT(1)

* Enabled if the file is opened for input, otherwise disabled.

5.0 Expressions and References

expression:

```
 / [(]logical-expression[)]          \
|  [(]relational-expression[)]        |
{  [(]concatenation-expression[)]     }
|  [(]arithmetic-expression[)]        |
|  [(]reference[)]                    |
 \ [(]constant[)]                    /
```

logical-expression:

```
/ expression  { | }  expression \
{             { & }             }
\        ^expression            /
```

All operands must be bit-string expressions. *VAX-11 PL/I also permits the use of the exclamation point (!) as the OR operator.*

All logical expressions result in bit strings.

relational-expression:

```
              / >  \
             |  >=  |
             |  =   |
expression   {  <   }   expression
             |  <=  |
             |  ^>  |
             |  ^=  |
              \ ^< /
```

Both operands must be arithmetic, or both must be of the same type. All relational expressions have Boolean results of type BIT(1).

concatenation-expression:

expression || expression

Both operands must be bit-string expressions, or both must be character-string expressions. Concatenation expressions have results of the same type as the operands.

VAX-11 PL/I also permits the use of a double exclamation point (!!) as the concatenation operator.

arithmetic-expression:

```
        ⎧ expression /  expression ⎫
        ⎪ expression *  expression ⎪
  [ - ] ⎨ expression -  expression ⎬
  [ + ] ⎪ expression +  expression ⎪
        ⎪ expression ** expression ⎪
        ⎩ expression               ⎭
```

All operands must be arithmetic expressions.

reference:
[reference->] [structure-qualification] identifier [(subscript-expression,...)]
A subscript-expression is any valid expression with an integer result.

structure-qualification:
[structure-qualification] identifier [(subscript-expression,...)].

Priority of Operators

Operator	**Priority**	**Operator**	**Priority**
**	1	=	5
+ (prefix)	1	>	5
- (prefix)	1	<	5
^	1	^=	5
*	2	^>	5
/	2	^<	5
+ (infix)	3	>=	5
- (infix)	3	<=	5
||	4	&	6
		|	7

In the evaluation of expressions, parentheses may be used to group operands so that they are evaluated irrespective of the priority of operators.

6.0 Built-In Functions and Pseudovariables

6.1 Built-In Functions

A built-in function reference may be used wherever a reference of the same type is valid. The following built-in functions are supported.

1. Arithmetic Functions
   - ABS(x) — returns the absolute value of x.
   - CEIL(x) — returns smallest integer greater than or equal to x.
   - DIVIDE(x,y,p[,q]) — returns x/y to precision p and scale factor q.
   - FLOOR(x) — returns largest integer less than or equal to x.
   - MAX(x,y) — returns larger of the values x and y.
   - MIN(x,y) — returns smaller of the values x and y.
   - MOD(x,y) — returns x modulo y.
   - ROUND(x,k) — returns value of a scaled fixed-point decimal x rounded to k digits.
   - SIGN(x) — returns -1, 0, or 1 to indicate sign of x.
   - TRUNC(x) — returns truncated (integer) form of x.
2. Mathematical Functions
   - ACOS(x) — returns arc cosine of x in radians.
   - ASIN(x) — returns arc sine of x in radians.
   - ATAN(y,[x]) — returns arc tangent in radians.
   - ATAND(y[,x]) — returns arc tangent in degrees.
   - ATANH(x) — returns inverse hyperbolic tangent of x.
   - COS(x) — returns cosine of radian-angle x.
   - COSD(x) — returns cosine of degree-angle x.
   - COSH(x) — returns hyperbolic cosine of x.
   - EXP(x) — returns *e* to the power x.
   - LOG(x) — returns base-*e* logarithm of x.
   - LOG10(x) — returns base-10 logarithm of x.
   - LOG2(x) — returns base-2 logarithm of x.
   - SIN(x) — returns sine of radian-angle x.
   - SIND(x) — returns sine of degree-angle x.
   - SINH(x) — returns hyperbolic sine of x.

- SQRT(x) — returns square root of x.
- TAN(x) — returns tangent of radian-angle x.
- TAND(x) — returns tangent of degree-angle x.
- TANH(x) — returns hyperbolic tangent of x.

3. String Functions

- BOOL(x,y,z) — returns bit-string result of Boolean operation z on bit strings x and y. The argument z is a 4-bit string giving the Boolean results of bitwise comparisons of x and y:
  - Bit 1 of z: result of x-bit = 0, y-bit = 0
  - Bit 2 of z: result of x-bit = 0, y-bit = 1
  - Bit 3 of z: result of x-bit = 1, y-bit = 0
  - Bit 4 of z: result of x-bit = 1, y-bit = 1
- COLLATE( ) — returns string containing ASCII character set in collating order.
- COPY(s,c) — returns string containing c concatenated copies of string s
- INDEX(s,c) — returns position of character c in string s.
- LENGTH(s) — returns number of characters or bits in string s.
- STRING(s) — returns string containing concatenated string representations of values in array or structure s.
- SUBSTR(s,i[,j]) — returns part of string s beginning at position i and j characters in length.
- TRANSLATE(s,c[,d]) — returns translation of string s, such that characters in string d are replaced with characters from string c; if d is omitted, its value defaults to COLLATE()
- VALID(x) — returns a Boolean value indicating whether the character-string contents of x are valid with respect to the picture declared for x.
- VERIFY(s,c) — returns position of first character in string s that is not found in string c.

4. Conversion Functions

- BINARY(x[,p[,0]]) — returns arithmetic value of x converted to binary precision p and scale factor 0.
- BIT(s[,l]) — returns value of s converted to bit string of length l.
- BYTE(x) — returns ASCII character represented by integer x.
- CHARACTER(s[,l]) — returns value of s converted to character string of length l.

- DECIMAL(x[,p[,q]]) — returns value of x converted to decimal value of precision p and scale factor q.
- FIXED(x,p[,q]) — returns value of x as a fixed-point number of precision p and scale factor q.
- FLOAT(x,p) — returns value of x as a floating-point number of precision p.
- RANK(c) — returns integer representation (ASCII code) of character c.
- UNSPEC(x) — returns internally coded form of x as a bit string.

5. Condition-Handling Functions
   - ONARGSLIST( ) — returns pointer to argument lists of exception condition.
   - ONCODE( ) — returns error code of the most recent run-time error.
   - ONFILE( ) — returns name of file constant for which the most recent ENDFILE, ENDPAGE, KEY, or UNDEFINEDFILE condition was signaled.
   - ONKEY( ) — returns value of the key that signaled the KEY condition.

6. Array-Handling Functions
   - DIMENSION(x,n) — returns number of elements in the nth dimension of the array variable x.
   - HBOUND(x,n) — returns upper bound of the nth dimension of the array variable x.
   - LBOUND(x,n) — returns lower bound of the nth dimension of the array variable x.

7. Storage Functions
   - ADDR(x) — returns pointer identifying the storage referenced by x.
   - NULL( ) — returns the null pointer value.
   - OFFSET(p,a) — returns the offset (into area a) for the location indicated by pointer p.
   - POINTER(o,a) — returns pointer to the location at offset o within area a.

8. Timekeeping Functions
   - DATE( ) — returns string containing system date in format YYMMDD.
   - TIME( ) — returns string containing system time of day in format HHMMSSXX.

9. File Control Functions
   - LINENO(reference) — returns line number of the referenced print file.
   - PAGENO(reference) — returns page number of the referenced print file.

10. Argument-Passing Function
    - DESCRIPTOR(x) — forces the argument x to be passed by descriptor to a non-PL/I procedure.

6.2 Pseudovariables

Pseudovariables may be used on the left-hand side (that is, as the reference) of an assignment statement and in certain other assignment contexts.

1. PAGENO(reference) — changes the current page number of the referenced print file.
2. STRING(reference) — assigns substrings to elements/members of the referenced array/structure.
3. SUBSTR(s,i[,j]) — replaces the indicated substring with a string expression.
4. UNSPEC(reference) — replaces the internal representation of the referenced variable with a bit-string expression.

7.0 Built-In Subroutines

A built-in subroutine is used as the reference in a CALL statement. For detailed information on built-in subroutines, **see** the *VAX-11 PL/I User's Guide.*

The following built-in subroutines are supported.

1. File-Handling Subroutines
   - DISPLAY (file,structure) — returns, in the referenced structure, the attributes of the referenced file.
   - EXTEND (reference,expression) — increases the space allocated for the referenced file by the number of disk blocks specified by the expression (which must be an integer).
   - FLUSH (reference) — preserves all RMS buffers and attributes for the referenced file.
   - NXTVOL (reference) — sends request to system operator to mount the next volume of the referenced multivolume tape file.

- REWIND (reference) — repositions the referenced file such that the next record to be read is the first record in the file, relative record 1, or the lowest key value in the current index, for sequential, relative, and indexed sequential files, respectively.
- SPACEBLOCK (reference,expression) — positions the referenced file forward or backward the number of blocks specified by the expression (which must be an integer).

2. Condition-Handling Subroutines

- RESIGNAL( ) — allows an ON-unit to "pass" on a condition signal and causes the condition to be resignaled for handling by a different ON-unit.

8.0 ASCII Characters

The ASCII Character Set

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
0	NUL	Null	33	!	Exclamation mark
1	SOH	Start of heading	34	"	Quotation mark
2	STX	Start of text	35	#	Number sign
3	ETX	End of text	36	$	Dollar sign
4	EOT	End of transmission	37	%	Percent sign
5	ENQ	Enquiry	38	&	Ampersand
6	ACK	Acknowledgement	39	'	Apostrophe
7	BEL	Bell	40	(	Left parenthesis
8	BS	Backspace	41	)	Right parenthesis
9	HT	Horizontal tab	42	*	Asterisk
10	LF	Line feed	43	+	Plus sign
11	VT	Vertical tab	44	,	Comma
12	FF	Form feed	45	-	Minus sign or hyphen
13	CR	Carriage return	46	.	Period or decimal point
14	SO	Shift out	47	/	Slash
15	SI	Shift in	48	0	Zero
16	DLE	Data link escape	49	1	One
17	DC1	Device control 1	50	2	Two
18	DC2	Device control 2	51	3	Three
19	DC3	Device control 3	52	4	Four
20	DC4	Device control 4	53	5	Five
21	NAK	Negative acknowledgement	54	6	Six
22	SYN	Synchronous idle	55	7	Seven
23	ETB	End of transmission block	56	8	Eight
24	CAN	Cancel	57	9	Nine
25	EM	End of medium	58	:	Colon
26	SUB	Substitute	59	;	Semicolon
27	ESC	Escape	60	<	Left angle bracket
28	FS	File separator	61	=	Equal sign
29	GS	Group separator	62	>	Right angle bracket
30	RS	Record separator	63	?	Question mark
31	US	Unit separator	64	@	At sign
32	SP	Space or blank	65	A	Upper case A

(Continued on next page)

The ASCII Character Set (Cont.)

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
66	B	Upper case B	97	a	Lower case a
67	C	Upper case C	98	b	Lower case b
68	D	Upper case D	99	c	Lower case c
69	E	Upper case E	100	d	Lower case d
70	F	Upper case F	101	e	Lower case e
71	G	Upper case G	102	f	Lower case f
72	H	Upper case H	103	g	Lower case g
73	I	Upper case I	104	h	Lower case h
74	J	Upper case J	105	i	Lower case i
75	K	Upper case K	106	j	Lower case j
76	L	Upper case L	107	k	Lower case k
77	M	Upper case M	108	l	Lower case l
78	N	Upper case N	109	m	Lower case m
79	O	Upper case O	110	n	Lower case n
80	P	Upper case P	111	o	Lower case o
81	Q	Upper case Q	112	p	Lower case p
82	R	Upper case R	113	q	Lower case q
83	S	Upper case S	114	r	Lower case r
84	T	Upper case T	115	s	Lower case s
85	U	Upper case U	116	t	Lower case t
86	V	Upper case V	117	u	Lower case u
87	W	Upper case W	118	v	Lower case v
88	X	Upper case X	119	w	Lower case w
89	Y	Upper case Y	120	x	Lower case x
90	Z	Upper case Z	121	y	Lower case y
91	[	Left square bracket	122	z	Lower case z
92	\	Back slash	123	{	Left brace
93	]	Right square bracket	124	|	Vertical line
94	^ or ↑	Circumflex or up arrow	125	}	Right brace
95	← or _	Back arrow or underscore	126	~	Tilde
96	`	Grave accent	127	DEL	Delete

Glossary

This Glossary defines the terms that are used to describe the VAX-11 PL/I language. The terms defined here are used throughout the VAX-11 PL/I documentation. The definitions in this glossary are followed by one of the following codes to indicate sources that provide more detailed information than is provided in this manual:

Code	**Document**
'Pedia	*VAX-11 PL/I Encyclopedic Reference*
User's Guide	*VAX-11 PL/I User's Guide*
VAX/VMS	*VAX/VMS Command Language User's Guide* or related VAX/VMS documentation

Consult the index of the appropriate document for page number references.

access mode

Manner in which records in a record file will be read or written. The access modes are sequential, direct, or keyed sequential. *(User's Guide)*

aggregate

A collection of related data items that can be referred to individually or collectively. See *array* and *structure. ('Pedia)*

allocation

(1) Specific unit of storage obtained for a based variable. (2) Activity of obtaining storage for a variable. *('Pedia)*

area

Unit of storage in which based variables may be allocated. *('Pedia)*

argument

Variable or expression value that is passed to an invoked subroutine or function. *('Pedia, User's Guide)*

argument list

Zero or more arguments, specified in the invocation of a procedure or a built-in function. *('Pedia, User's Guide)*

arithmetic data

Data of the fixed-point binary, fixed-point decimal, floating-point, or pictured types. *('Pedia)*

arithmetic operator

One of the punctuation symbols (+, - , /, * , or **) that requests an arithmetic operation. *('Pedia)*

array

A named collection of data items that have the same attributes and in which individual items, called elements, are accessed by subscripts. *('Pedia)*

array reference

A variable reference that denotes an entire array (as opposed to an element of an array). *('Pedia)*

ASCII character set

Numeric values used to represent characters and control information. *('Pedia)*

assignment statement

An executable statement that gives a value to a variable. *('Pedia)*

attribute

Characteristic of a data item, such as fixed- or floating-point, decimal or binary, extent, and so on. *('Pedia)*

automatic variable

A variable for which storage is allocated when the block that declares it is activated. The storage is released when the block is deactivated. *('Pedia)*

based variable

A variable that is used to describe storage that is accessed using a pointer. *('Pedia)*

begin block

A sequence of statements headed by a BEGIN statement and terminated by a corresponding END statement. A begin block is entered when control flows into the BEGIN statement. When a begin block is entered, a block activation is created for it and for the variables declared within it. *('Pedia)*

bit

(1) A unit of storage that can hold either of the binary digits 0 or 1. (2) Data type applied to variables consisting of bit values. *('Pedia)*

bit string

Zero or more of the binary digits 0 or 1. *('Pedia)*

block

A sequence of PL/I statements that is delimited by one of the statement pairs PROCEDURE and END or BEGIN and END. *('Pedia)*

block activation

Hardware context created each time a block is entered, including the allocation of storage for automatic variables and hardware information that connects the block to the previous block. *('Pedia)*

block I/O

The performance of I/O in which each physical 512-byte block in a file is treated as a record, regardless of the structure of the records in the file. *(User's Guide)*

Boolean

A logical value that can be either true or false. In PL/I, a one-bit string with a value of zero indicates a Boolean value of false and a one-bit string with a value of one indicates a Boolean value of true. *('Pedia)*

bound

Upper or lower limit to the subscript values of a dimension of an array. *('Pedia)*

built-in function

A function provided by the PL/I language. *('Pedia)*

by descriptor

Argument-passing mechanism used to pass arguments to a parameter declared with asterisk extents. *(User's Guide)*

by immediate value

Argument-passing mechanism used to pass arguments to parameters declared with the VALUE attribute. *(User's Guide)*

by reference

Conventional PL/I method for passing arguments. *(User's Guide)*

character

(1) A single element of the ASCII character set. (2) Data type applied to variables consisting of characters. *('Pedia)*

character string

Zero or more characters. *('Pedia)*

command

An instruction or request for the system or a system program to perform a particular action. *(VAX/VMS)*

command procedure

A file containing a sequence of operating system commands to be executed. *(VAX/VMS)*

comment

Any sequence of characters appearing between the character pairs /* and */. Comments are for documentation purposes and have no special meaning to PL/I. *('Pedia)*

comparison operator

See *relational operator.*

compiler

A program that translates source statements in a high-level programming language into an object module. The object module consists of machine instructions and relocation information to be used by the linker to form an executable image. *(User's Guide)*

computational

One of the data types on which operations can be performed. The computational data types are arithmetic (including pictured) and string. *('Pedia)*

concatenation operator

Punctuation symbol (¦¦ or !!) that joins two string values to form a single string. *('Pedia)*

condition

An occurrence that causes the interruption of the program and initiates a search for a sequence of statements to be executed in response. See also *ON condition. ('Pedia)*

condition name

Keyword associated with a specific ON condition, whose name suggests the nature of the condition. *('Pedia)*

condition value

A unique 32-bit number that identifies a specific operating system error, warning, or informational condition. *(User's Guide)*

constant

(1) A literal value specified to represent a computational data item. (2) An entry or label name that is declared implicitly by context. (3) A name declared with one of the attributes ENTRY or FILE and without the VARIABLE attribute. *('Pedia)*

constant identifier

An identifier that is replaced by a constant at compile time. *('Pedia)*

control variable

A variable whose value is modified for each iteration of a DO-group and which may be tested to determine whether or not the statements in the DO-group are to be executed. *('Pedia)*

conversion

Transformation of a value from one data type to another. *('Pedia)*

data type

Class to which a data item belongs, for example, fixed-point decimal or character string. The data type of a variable determines the operations that can be performed on it. *('Pedia)*

DCL

DIGITAL Command Language. Set of commands and utilities that invoke programs provided by the VAX/VMS operating system. *(VAX/VMS)*

debugger

Interactive program that permits the display and modification of program variables during the execution of the program. *(User's Guide)*

declaration

Explicit or contextual specification of an identifier and its data type. *('Pedia)*

defined variable

A variable declared with the DEFINED attribute that refers to all or part of another variable's storage. *('Pedia)*

descriptor

See *by descriptor*, *parameter descriptor*, *returns descriptor.*

device

General term for a physical system component or a link connected to the computer that is capable of storing or transmitting data. *(VAX/VMS)*

diagnostic

Message from the compiler indicating that a statement contains a syntax error or a violation of the language rules. *(User's Guide)*

dimension

A set of bounds describing one extent of an array. *('Pedia)*

directory

File that contains a list of files cataloged on a particular device. *(VAX/VMS)*

DO-group

A sequence of statements headed by a DO statement and terminated with a corresponding END statement. *('Pedia)*

dummy argument

A unique variable allocated by the compiler to contain a copy of an argument specified in a procedure invocation. *('Pedia, User's Guide)*

edit-directed stream I/O

Transmission of data between a program and an external input/output device for which the formatting and conversion of data are controlled by format specifications in a GET or PUT statement. *('Pedia)*

editor

Program designed for the interactive creation and modification of files. *(VAX/VMS)*

element

Individual data item in an array. *('Pedia)*

entry name

Identifier on a PROCEDURE or ENTRY statement that defines an entry point for that procedure. *('Pedia)*

entry point

Statement or instruction at which the execution of a procedure can commence. *('Pedia)*

equivalence name

Character string equated to a logical name to be used in a VAX/VMS file specification. *(VAX/VMS)*

expression

A variable reference or constant, or any expression involving variable references, constants, operators, built-in functions, or procedure function references. *('Pedia)*

extent

(1) The range comprising the low-bound:high-bound for one dimension of an array. (2) Length of a string. *('Pedia)*

external procedure

A procedure that is not contained in another procedure. *('Pedia)*

external variable

A variable that is known in all blocks that declare it with the EXTERNAL attribute. *('Pedia)*

field

A string of characters that corresponds to an input or output variable in a stream I/O statement. *('Pedia)*

file

(1) In PL/I, the input source or output target specified in an I/O statement. *('Pedia)*
(2) In VAX/VMS, a physical device or named collection of records on a mass storage device such as a disk or magnetic tape. *(VAX/VMS)*

file constant

A name declared with the FILE attribute but not the VARIABLE attribute. *('Pedia)*

file description attribute

One of the PL/I attribute keywords that can be specified in the declaration of a file constant or used in an OPEN statement. The file description attributes indicate the properties of the file and the manner in which a file will be used. *('Pedia)*

file name

A zero- to nine-character component of a file specification that is generally a name supplied by the user. *(VAX/VMS)*

file organization

Manner in which the records in a record file are arranged. The file organizations that may be used in VAX-11 PL/I are sequential, relative, and indexed sequential. *(User's Guide)*

file reference

The use of an identifier declared as a file constant, a scalar reference to a variable with the FILE attribute, or a function that returns a file value. *('Pedia)*

file specification

Unique identification of a file to the VAX/VMS operating system. *(VAX/VMS)*

file type

A zero- to three-character component of a file specification that generally describes the usage of the file. *(VAX/VMS)*

file version number

Numeric component of a file specification, indicating the number of times a file has been updated. *(VAX/VMS)*

fixed-control area

A data area associated with a record that is maintained separately from the data portion of the record and that can be read or written in an I/O operation. *(User's Guide)*

fixed-point binary

Data type for integer values. *('Pedia)*

fixed-point decimal

Data type for decimal data with a fixed number of fractional digits. *('Pedia)*

floating point

Data type used for very small or very large numbers. A floating-point value has a mantissa and an optionally signed integer exponent. *('Pedia)*

format item

A character code and possibly associated value or values indicating input or output data representation and formatting. *('Pedia)*

format list

A list of format items corresponding to variable references or output data items for edit-directed stream I/O. *('Pedia)*

function

A procedure that is entered when a reference to its name appears in an expression and that returns a value to its point of reference. *('Pedia)*

function reference

Appearance of the name of a user-written or built-in function in a PL/I statement. *('Pedia)*

global symbol

External static variable declared with the GLOBALDEF or GLOBALREF attribute. Global symbols may also have the VALUE attribute; symbols so declared are constants that do not occupy any storage, but whose values are resolved at link time. *(User's Guide)*

high bound

Upper limit of one dimension of an array. *('Pedia)*

identifier

A user-supplied name of from 1 to 31 characters that denotes the name of a variable, statement label, entry point, or file constant. *('Pedia)*

image

Output from the linker; created by processing one or more object modules. An image is the executable version of a program. *(User's Guide)*

INCLUDE file

External file from which the compiler reads source text during the compilation of a PL/I program. *(User's Guide)*

index number

The key field to which a key specified in a given I/O operation applies. An indexed file can have multiple keys; each of which has an index. The first, or primary, index is always zero. *(User's Guide)*

indexed sequential file

A record file in which each record has one or more data keys embedded in it. Records in the file are individually accessible by specifying a key associated with a record. *(User's Guide)*

infix operator

An operator that is positioned between two operands in an expression to define the operation.

integer constant

Optionally signed string of decimal digits. *('Pedia)*

integer data

Data declared as FIXED BINARY or FIXED DECIMAL with a zero scale factor. *('Pedia)*

internal procedure

A procedure that is contained within another procedure. *('Pedia)*

internal variable

A variable whose value can be referenced within the block that declared it and any blocks contained within the block that declared it. *('Pedia)*

iteration factor

An integer constant written in parentheses that specifies the number of times to use a value in the initializing of array elements, or the number of times to use a given format item or picture specification. *('Pedia)*

key

(1) A value used in I/O statements to specify a particular record in a file. (2) A data item embedded within a record in an indexed sequential file, or the relative record number of a record in a relative file. *('Pedia, User's Guide)*

keyed access

See *random access.*

keyword

An identifier that has a specific meaning to PL/I when used in the appropriate context. *('Pedia)*

label

A PL/I identifier, terminated by a colon (:), which is used to identify a statement. *('Pedia)*

level number

An integer constant that defines the relationship of a name within the hierarchy of a structure with respect to other names in the structure. *('Pedia)*

library

A file that contains modules and a directory listing those modules. The two types of library used in PL/I program development are text libraries of INCLUDE files and object module libraries. *(User's Guide)*

linker

The program that arranges object modules into an executable image and that resolves references among external variables declared in the modules. *(User's Guide)*

list-directed stream I/O

Transmission of data between a program and an input/output device, for which PL/I provides automatic data conversion and formatting. *('Pedia)*

listing

Output file created by the compiler that lists the statements in the source program, the line numbers it has assigned to them, the names of variables and constants referenced in the program, and additional optional information. *(User's Guide)*

locator-qualified reference

Specification of a based variable in terms of a pointer or offset value that indicates the location of the variable. *('Pedia)*

locator qualifier

Pointer reference and punctuation symbol (->) that associate a storage location with a based variable. *('Pedia)*

logical name

Alternate name used to refer to VAX/VMS files or devices by other than their unique file specifications. *(VAX/VMS)*

logical operator

One of the punctuation symbols (^, &, !, or ¦) that performs a logical operation on bit-string values. *('Pedia)*

login

Sequence of terminal interaction that establishes a user's communication with the operating system. *(VAX/VMS)*

logoff

Termination of a user's communication with the operating system. *(VAX/VMS)*

low bound

Lower limit of one dimension of an array. *('Pedia)*

main procedure

Procedure that defines the primary entry point for a program. *('Pedia)*

major structure

Name given to an entire structure, by which all members of the structure can be specified in a single reference. A major structure always has a level number of 1. *('Pedia)*

member

A data item in a structure that may itself be a structure. *('Pedia)*

memory

Addressable locations into which storage acquired for variables is mapped. *(User's Guide)*

minor structure

A member of a structure that is itself a structure. *('Pedia)*

noncomputational

A data item that is not string or arithmetic. The noncomputational data types are entry, file, label, pointer, area, and offset. *('Pedia)*

nonlocal GOTO

A GOTO statement that results in a transfer of program control to a statement in a previous block. *('Pedia)*

object module

Output from a language compiler or assembler that can be linked with other modules to form an executable image. *(User's Guide)*

offset

A data item whose value represents a displacement from the beginning of an area. *('Pedia)*

ON condition

Any one of several named conditions that can interrupt a program and generate a signal, such as a fixed-point or a floating-point overflow. *('Pedia)*

ON-unit

PL/I statement or begin block specifying the action to be taken when a specific ON condition is signaled during the execution of a PL/I program. *('Pedia)*

operator

Punctuation symbol that requests or causes PL/I to perform a specific function such as addition or comparison. *('Pedia)*

parameter

(1) A variable that is matched to an argument when the procedure is invoked. (2) The storage class of a variable whose value is obtained from an invoking procedure. (3) In VAX/VMS, a value passed to a command or command procedure. *('Pedia)*

parameter descriptor

Set of attributes in a PL/I program describing a parameter to be passed to an external procedure. The attributes in the parameter descriptor can specify data type and argument-passing mechanism. *('Pedia)*

parameter list

Specification of the names of variables whose values will be determined when a procedure is invoked. The parameter list is specified on the PROCEDURE or ENTRY statement for the procedure's entry point. *('Pedia)*

password

A string of characters associated with a user name. A user logging into the system must supply the correct password before the system will allow access. *(VAX/VMS)*

picture

(1) A specification of the character-string representation of an arithmetic value. The specification is given as a character-string constant that defines the position of a decimal point, zero suppression, sign conventions, and so on. (2) A data type for which fixed-point decimal values are stored internally as character strings, in accordance with a picture specification. *('Pedia)*

pointer

A data item whose value is the address of a location in memory. *('Pedia)*

precedence

The priority of an operator applied to the evaluation of operations in an expression. An operation with a higher precedence is performed before an operation with a lower precedence. *('Pedia)*

precision

The number of digits in an arithmetic data item. *('Pedia)*

prefix operator

One of the operators + or – that precedes a variable or constant to indicate or change its sign. *('Pedia)*

print file

A stream output file for which PL/I aligns certain data on predefined tab stops, controls output by a page size and line size, and does not enclose strings in apostrophes. *('Pedia)*

procedure

A sequence of statements, headed by a PROCEDURE statement and terminated by an END statement, that define an executable set of program instructions. A procedure can be a subroutine that is invoked by a CALL statement or a function that is invoked by a function reference. *('Pedia)*

procedure block

A sequence of statements headed by a PROCEDURE statement and terminated by an END statement. A procedure block is entered when its name is specified in a CALL statement or a function reference. When a procedure block is entered, a block activation is created for it and for the internal variables declared within it. *('Pedia)*

pseudovariable

A built-in function that can be used on the left-hand side of an assignment to give a special meaning to the assignment. *('Pedia)*

qualified reference

See *locator-qualified reference*, *structure-qualified reference*.

qualifier

Keyword that modifies the operation of a DCL command. Qualifiers are always preceded by slash (/) characters. *(VAX/VMS)*

random access

The performance of I/O to a record file by specifying individual records to be read, written, rewritten, or deleted. Records are specified by a key that can be either a data key embedded in the record or a relative record number. *('Pedia, User's Guide)*

record

An organized collection of data transmitted by a record I/O statement. *('Pedia, User's Guide)*

record file

A file that is processed in terms of records. *('Pedia)*

record format

The properties of the records in a specific file, including the record length and variability. *(User's Guide)*

record id access

The specification of a record by its internal file identification. *(User's Guide)*

record I/O

The transmission and interpretation of data grouped in well-defined units called records. *('Pedia)*

recursive procedure

A procedure that may invoke itself. *('Pedia)*

reference

The appearance of an identifier in any context except the one in which it is declared. *('Pedia)*

relational operator

One of the punctuation symbols (>, <, =, <=, >=, ^<, ^>, or ^=) that states a relationship between two expressions and results in a one-bit Boolean value indicating whether the relationship is true or false. *('Pedia)*

relative file

A record file in which each record occupies a fixed-length, numbered cell. Records in the file are individually accessed by specifying the number of a cell, relative to the first record in the file. The first cell in the file is numbered 1. *(User's Guide)*

relative record number

(1) The position of a specific record in a relative file. (2) The key by which a record in a relative file is accessed randomly. *(User's Guide)*

resignal

Mechanism by which a condition handler, or ON-unit, indicates that a signal is still active. *(User's Guide)*

return value

Value returned by a function to be used at its point of invocation. *('Pedia)*

returns descriptor

Set of attributes describing the data type of the return value of a function. *('Pedia)*

row-major order

The order of storage of an array's elements, and the order of assignment of values to an array. In row-major order, the rightmost subscript varies the most rapidly. *('Pedia)*

run-time library

Collection of procedures that support the execution of a PL/I program. *(User's Guide)*

scalar

A data item that is neither an array nor a structure. *('Pedia)*

scale factor

The number of fractional digits for a fixed-point decimal data item. *('Pedia)*

scope

The range within a program in which the declaration of an identifier is known. *('Pedia)*

sequential access

The performance of I/O to a file by accessing records serially. *('Pedia, User's Guide)*

sequential file

A record file in which the records are arranged serially, to which new records can be added only at the end of the file and from which records cannot be deleted. *('Pedia, User's Guide)*

signal

Mechanism by which PL/I indicates that an error or other special condition occurred. *('Pedia)*

statement

A sequence of PL/I keywords, user-specified identifiers, and punctuation marks that specifies an executable instruction or data declaration in a program. *('Pedia)*

static variable

A variable whose storage is allocated for the entire execution of a program. *('Pedia)*

storage

Contiguous region of the computer's memory that is associated with a particular variable. *('Pedia)*

storage class

The attribute of a variable that describes how its storage is allocated and released by PL/I. The storage classes are automatic, static, based, defined, and parameter. *('Pedia)*

stream I/O

The transmission and interpretation of input or output data in terms of sequences of ASCII characters that are delimited by spaces, tabs, commas, or fields defined by format items. *('Pedia)*

string data

One of the data types character string or bit string. *('Pedia)*

structure

A hierarchical arrangement of related data items, called members, that need not have the same data type. *('Pedia)*

structure-qualified reference

Naming of a member of a structure by specifying each higher-level name within the structure and separating the names with periods. *('Pedia)*

structure reference

A variable reference denoting an entire structure (as opposed to a member of a structure). *('Pedia)*

subdirectory

Directory file cataloged in a higher-level directory that lists additional files. *(VAX/VMS)*

subroutine

A procedure that is entered by a CALL statement and that does not return a value to its point of invocation. *('Pedia)*

subscript

An integer expression that specifies an element in an array or a label. *('Pedia)*

system service

Operating system procedure used by VAX/VMS for controlling resources of the system. *(User's Guide)*

user name

Name by which the system identifies a particular user and under which a user gains access to the system. *(VAX/VMS)*

variable

A data item whose value may change during the execution of a program. *('Pedia)*

variable reference

A reference to all or part of a variable. The reference may include qualification by member names and subscripts. *('Pedia)*

Index

A

Access mode, 1-35, G-1
Addition operator, 1-13
ADDR built-in function, 1-39
Aggregate, 1-9, G-1
  assignments, 1-15
ALLOCATE statement, 1-38
Allocation, 1-38, G-1
AND operator, 1-14
ANY attribute, 2-7
ANYCONDITION condition, 2-10
Area, 1-40, G-1
  data type, 1-6
AREA attribute, 1-6, 1-41
Argument, 1-26, G-2
  dummy, 1-27
Argument list, 1-26, G-2
  hardware representation, 2-7
  variable length, 2-8
Argument passing,
  PL/I conventions, 1-26
  VAX-11 conventions, 2-7
Arithmetic data, 1-4, G-2
Arithmetic operator, 1-13, G-2
Array, 1-9, G-2
  assignment, 1-15
  order of assignment, 1-10
Array reference, G-2
ASCII character set, 1-4, G-2
Assignment statement, 1-12, G-2
  aggregates, 1-15
  conversion of data, 1-13
Asterisk,
  extent, 1-28
  wild card character, 3-11
Attribute, 1-4, G-2
AUTOMATIC attribute, 1-8
Automatic variable, 1-8, G-2

B

BASED attribute, 1-37
Based storage, 1-37
Based variable, 1-37, G-2
BATCH option, 2-3
Begin block, 1-3, G-3
  compared to DO-group,1-22
BEGIN statement, 1-3
  following IF, 1-22
BINARY attribute, 1-4
Bit, 1-4, G-3
BIT attribute, 1-4
Bit string, 1-4, G-3
  constant, 1-7
Block, 1-1, G-3
Block activation, 1-3, G-3
  effect of GOTO, 1-24
Block I/O, 2-6, G-3
BLOCK_IO option, 2-5 to 2-6
BLOCK_SIZE option, 2-5
Boolean, 1-4, G-3
Boolean expression,
  in DO statement, 1-23
Bound, 1-10, G-3
Built-in function, 1-29, G-3
Built-in subroutines, 2-1
By descriptor, 2-7, G-3
By immediate value, 2-7, G-4
By reference, 1-26, 2-7, G-4

C

CALL statement, 1-24
CANCEL_CONTROL_O option, 2-4
Character, 1-4, G-4
CHARACTER attribute, 1-4
Character string, 1-4, G-4
  constant, 1-7
CLOSE statement, 1-17
Closing a file, 1-17
Command, 3-3, G-4
Command procedure, 3-15, G-4
Comment, 1-2, G-4
Comparison operator,
  see relational operator
Compiler, 3-5, G-4
Computational data, 1-4, G-4
Concatenation operator, 1-14, G-5
Condition, 1-31, G-5
  handling, 1-31
Condition name, 1-31, G-5

Condition value, 1-33, G-5
Constant, G-5
  bit string, 1-7
  character string, 1-7
  conversion, 1-13
  data types, 1-6
  fixed-point decimal, 1-7
  floating point, 1-7
Constant identifier, 1-7, G-5
Control variable, 1-23, G-5
Conversion, 1-13, G-5
CREATE/DIRECTORY
  command, 3-11

D

DAT file type, 1-18
Data type, 1-4, G-5
DCL, 3-1, G-5
Debugger, G-6
DECIMAL attribute, 1-4
Declaration, 1-4, G-6
  factored, 1-4
DECLARE statement, 1-2, 1-4
  array declaration, 1-10
DEFINE command, 3-12
DEFINED attribute, 1-41
Defined variable, 1-41, G-6
DELETE command, 3-10
DELETE option, 2-3
DELETE statement, 1-16
Descriptor, *see*
  by descriptor
  parameter descriptor
  returns descriptor
Device, 3-10, G-6
Diagnostic, 3-5, G-6
Dimension, 1-9, G-6
DIRECT attribute, 1-35
Directory, 3-10, G-6
  subdirectory, 3-11
DIRECTORY command, 3-10
Division,
  divide-by-zero condition, 1-32
Division operator, 1-13
DO-group, 1-22, G-6
DO statement, 1-22
  following IF, 1-22
  REPEAT option, 1-23, 1-39
  WHILE option, 1-23
Double-precision floating point, 1-5
Dummy argument, 1-27, G-6
  passed by immediate value, 2-7

E

EDIT command, 3-3
Edit-directed stream I/O, 1-18, G-6
  examples, 1-19
Editor, 3-3, G-7
Element, 1-9, G-7
END statement, 1-3
  end DO-group, 1-22
End-of-file, 1-32, 1-36
End-of-page, 1-21, 1-32
End-of-volume switching, 2-5
ENDFILE condition, 1-32, 1-36
ENDPAGE condition, 1-32
ENTRY attribute, 1-6, 1-8, 1-28
  options, 2-8
Entry constant, 1-7
Entry data, 1-6
  default storage class, 1-8
Entry name, 1-29, G-7
Entry point, 1-29, G-7
ENTRY statement, 1-7, 1-29
Entry variable, 1-6
ENVIRONMENT attribute, 1-34
ENVIRONMENT options, 2-1
Equal operators, 1-14
Equivalence name, 3-12, G-7
ERROR condition, 1-32 to 1-33, 1-37
Error handling, 1-31
Execute a program, 3-7
@ (Execute Procedure) command, 3-15
EXPIRATION_DATE option, 2-5
Exponentiation operator, 1-13
Expression, 1-12, G-7
  in an IF statement, 1-22
Extent, G-7
  array, 1-9
  character string, 1-5
EXTERNAL attribute, 1-8 to 1-9
External procedure, 1-28, G-7
External variable, 1-9, G-7

F

Field, 1-16, G-7
  edit-directed stream I/O, 1-20
  key, 2-2
  list-directed stream I/O, 1-18
File, 1-16, G-8
  ownership, 2-3
  protection, 2-3
FILE attribute, 1-6 to 1-8
File constant, 1-7, 1-35, G-8

File data, 1-6
  default storage class, 1-8
File description attribute, 1-17, G-8
File name, 3-10, G-8
File organization, 1-34, G-8
File reference, 1-16, G-8
File specification, 3-10, G-8
  associate with PL/I file, 1-17
  logical name, 3-12
File type, 3-10, G-8
File variable, 1-35
File version number, 3-10, G-8
FINISH condition, 1-32
FIXED attribute, 1-4
Fixed-control area, 2-4, G-8
FIXED_CONTROL_FROM option, 2-4 to 2-5
FIXED_CONTROL_SIZE option, 2-4 to 2-5
FIXED_CONTROL_TO option, 2-4
FIXED_LENGTH_RECORDS option, 1-34
Fixed-point binary data, 1-4, G-9
  precision, 1-5
Fixed-point decimal data, 1-4, G-9
  constant, 1-7
  overflow condition, 1-32
  precision, 1-5
  scale factor, 1-5
FIXEDOVERFLOW condition, 1-31 to 1-32
  example of ON-unit, 1-33
FLOAT attribute, 1-4
Floating-point data, 1-4, G-9
  constant, 1-7
  overflow condition, 1-32
  precision, 1-5
  underflow condition, 1-32
Format,
  of PL/I statement, 1-2
  of records, 1-34
Format item, 1-19, G-9
Format list, 1-19, G-9
FORTRAN COMMON, 1-9
FREE statement, 1-38
Function, 1-27, G-9
Function reference, 1-27, G-9

G

GET statement, 1-16
  GET EDIT, 1-20
  GET LIST, 1-18
  GET STRING, 1-18
  I/O options, 2-4
Global symbol, G-9
GLOBALREF attribute, 2-9 to 2-10
GOTO statement, 1-24
  in ON-unit, 1-33
Greater-than operators, 1-14
GROUP_PROTECTION option, 2-3

H

Hexadecimal notation, 1-7
High bound, 1-10, G-9

I

I/O,
  block I/O, 2-6
  error handling, 1-36
  record id access, 2-6
I/O concepts, 1-16
I/O statements,
  options, 2-1
Identifier, 1-2, G-9
  rules, 1-2
IF statement, 1-21
Image, 3-7, G-10
INCLUDE file, 3-13, G-10
  %INCLUDE statement, 3-13
  libraries, 3-13
%INCLUDE statement, 3-13
INDEX built-in function, 1-30
Index number, 2-2, G-10
INDEX_NUMBER option, 2-2
Indexed sequential file, 1-34, G-10
  I/O options, 2-2
Infix operator, G-10
INITIAL attribute, 1-8
INPUT attribute, 1-17, 1-35
Integer constant, 1-7, G-10
Integer data, 1-4, G-10
Integer overflow, 1-33
INTERNAL attribute, 1-8
Internal procedure, 1-25, G-10
Internal variable, 1-8, G-10
Iteration factor, G-10

K

Key, 1-34, G-11
KEY condition, 1-32, 1-37
KEY option, 2-2
Keyed access,
  see random access
KEYED attribute, 1-35

KEYFROM option, 2-2
Keyword, 1-1, G-11

L

Label, 1-2, G-11
  array, 1-20
  constant, 1-7
  data type, 1-6
  subscripted, 1-24
LABEL attribute, 1-6, 1-10
Length,
  of character string, 1-5
LENGTH built-in function, 1-30
Less-than operators, 1-14
Level number, 1-10, G-11
Library, 3-13, G-11
  INCLUDE files, 3-13
  object module, 3-14
LIBRARY command, 3-13
Line number of a print file, 1-21
LINENO built-in function, 1-21
LINESIZE option, 1-21
LINK command, 3-7
  specify libraries, 3-14
Linker, 3-7, G-11
  specify libraries, 3-14
/LIST qualifier, 3-7
List-directed stream I/O, 1-18, G-11
  examples, 1-18
Listing, 3-7, G-11
LNK$LIBRARY, 3-14
Locator-qualified reference, 1-38, G-12
Locator qualifier, 1-38, G-12
Logical name, 3-12, G-12
  in TITLE option, 1-18
Logical operator, 1-14, G-12
Login, 3-3, G-12
Logoff, 3-9, G-12
LOGOFF command, 3-9
Low bound, 1-10, G-12

M

MAIN option, 1-24
Main procedure, 1-24, G-12
Major structure, 1-11, G-12
MATCH_GREATER option, 2-2
MATCH_GREATER_EQUAL option, 2-2
MAXIMUM_RECORD_NUMBER option, 2-3
MAXIMUM_RECORD_SIZE option, 1-34
Member, 1-9, G-12
Memory, 1-37, G-13
Minor structure, 1-11, G-13
Multiplication operator, 1-13

N

NO_ECHO option, 2-4
NO_FILTER option, 2-4
Noncomputational data, 1-6, G-13
Nonlocal GOTO, 1-24, G-13
NOT operator, 1-14
NULL built-in function, 1-39
NXTVOL built-in subroutine, 2-5

O

OBJ file type, 3-7
Object module, 3-7, G-13
  library, 3-14
Octal notation, 1-7
Offset, 1-40, G-13
  data type, 1-6
OFFSET attribute, 1-6, 1-41
OLB file type, 3-13
ON condition, 1-31, G-13
ON statement, 1-31, 2-10
  action, 1-32
  ANYCONDITION, 2-10
  for VAXCONDITION, 2-10
ON-unit, 1-31, G-13
  FIXEDOVERFLOW condition, 1-33
  for any condition, 2-10
  for VAX-specific condition, 2-10
  search, 1-32
ONARGSLIST built-in subroutine, 2-11
ONCODE built-in function, 1-33, 1-37, 2-10 to 2-11
ONFILE built-in function, 1-37
ONKEY built-in function, 1-37
OPEN statement, 1-17, 1-35
Opening a file, 1-17
Operator, 1-2, 1-13, G-13
OR operator, 1-14
OUTPUT attribute, 1-21, 1-35
OVERFLOW condition, 1-32
OWNER_GROUP option, 2-3
OWNER_MEMBER, 2-3
OWNER_PROTECTION option, 2-3
Ownership, file, 2-3

P

Page number of a print file, 1-21
PAGENO built-in function, 1-21
PAGESIZE, 1-21
Parameter, 1-26, G-13
  for DCL command, 3-3
  storage class, 1-26
Parameter descriptor, G-14
Parameter list, 3-3, G-14
Password, 3-3, G-14
Picture, 1-5, G-14
PICTURE attribute, 1-5
Pictured data, 1-5
PLI command, 3-5
PLI file type, 3-5
Pointer, 1-37, G-14
  assign values, 1-39
  data type, 1-6
POINTER attribute, 1-6, 1-37
Precedence, 1-15, G-14
Precision, 1-5, G-14
  of pictured data, 1-6
Prefix operator, G-14
Preprocessor statements,
  %INCLUDE, 3-13
  %REPLACE, 1-7
PRINT attribute, 1-21
PRINT command, 3-7, 3-11
Print file, 1-21, G-14
Procedure, 1-24, 1-30, G-15
  parameter list, 1-26
  recursive, 1-29
  terminating, 1-30
Procedure block, 1-3, G-15
PROCEDURE statement, 1-3, 1-7, 1-24 to 1-26
  function, 1-27
PROMPT option, 2-4
Protection,
  file, 2-3
Pseudovariable, 1-30, G-15
PURGE_TYPE_AHEAD option, 2-4
PUT statement, 1-16
  I/O options, 2-4
  PUT EDIT, 1-20
  PUT LIST, 1-19
  PUT STRING, 1-18

Q

Qualified reference, *see*
  locator-qualified
  structure-qualified
Qualifier, 3-3, G-15
Queues, 1-39

R

Random access, 1-35, G-15
READ statement, 1-16, 1-35, 2-2
  SET option, 1-39
READONLY attribute, 2-9
Record, 1-33, G-15
RECORD attribute, 1-17, 1-35
Record file, 1-33, G-15
Record format, 1-34, G-15
Record id access, 2-6, G-16
RECORD_ID option, 2-6
RECORD_ID_ACCESS option, 2-6
Record I/O, 1-16, G-16
Recursive procedure, 1-29, G-16
Reference, 1-4, G-16
  file, 1-16
  function, 1-27
  locator-qualified, 1-38
  structure-qualified, 1-11
Relational operator, 1-14, G-16
Relative file, 1-34, 2-2, G-16
Relative record number, 1-34, 1-36
    2-2, G-16
REPEAT option, 1-23, 1-39
%REPLACE statement, 1-7
Resignal, 2-10, G-16
RESIGNAL built-in subroutine, 2-10
RETURN statement, 1-25, 1-30
  function, 1-27
Return value, 1-27, G-16
RETURNS attribute, 1-27 to 1-28
Returns descriptor, 1-27 to 1-28, G-16
RETURNS option, 1-27 to 1-28
REVERT statement, 1-31, 2-10
REWIND_ON_CLOSE option, 2-3
REWRITE statement, 1-16
Row-major order, 1-10, G-17
RUN command, 3-7
Run-time library, 2-7, G-17

S

Scalar, 1-9, G-17
Scale factor, 1-5, G-17
Scope, 1-8, G-17
Sequential access, 1-35, G-17
SEQUENTIAL attribute, 1-35
Sequential file, 1-34, G-17

SET option, 1-38
Signal, 1-31, G-17
SIGNAL statement, 2-10
Single-precision floating point, 1-5
SKIP option, 1-19
SOS, 3-3
SPOOL option, 2-3
Statement, 1-1, G-17
  formatting requirements, 1-2
STATIC attribute, 1-8
Static variable, 1-8, G-17
STOP statement, 1-30, 1-32
Storage, 1-4, G-18
Storage class, 1-8, G-18
  VAX-11 PL/I, 2-9
STREAM attribute, 1-17
Stream I/O, 1-16, G-18
String data, 1-4, G-18
  functions, 1-30
Structure, 1-9, G-18
  assignment, 1-16
  based, 1-37
Structure-qualified reference,
    1-11, G-18
Structure reference, G-18
Subdirectory, 3-11, G-18
Subroutine, 1-27, G-18
Subscript, G-18
  array, 1-10
  label, 1-24
SUBSTR built-in function, 1-30
SUBSTR pseudovariable, 1-30
Subtraction operator, 1-13
Symbols,
  global, 2-8
  in command procedures, 3-15
SYS$COMMAND, 3-12
  SYS$DISK, 3-12
SYS$ERROR, 3-12
SYS$INPUT, 3-12
SYS$OUTPUT, 3-12
SYSIN, 1-18, 1-20, 3-12
SYSPRINT, 1-18, 1-20, 3-12
System service, 2-7, G-19
SYSTEM_PROTECTION option, 2-3

T

Tape I/O operations, 2-5
TITLE option, 1-17, 1-20
TLB file type, 3-13
TRUNCATE option, 2-3
TYPE command, 3-10

U

UNDEFINEDFILE condition, 1-32, 1-36
UNDERFLOW condition, 1-32
UPDATE attribute, 1-35
User name, 3-3, G-19

V

VALUE attribute, 2-7, 2-9
Variable, 1-3, G-19
VARIABLE attribute, 1-6 to 1-7
VARIABLE option, 2-8
Variable reference, G-19
VARYING attribute, 1-5
VAX-11 PL/I compiler,
  invoke, 3-5
VAXCONDITION condition, 2-10

W

WHILE option of DO, 1-23
WORLD_PROTECTION option, 2-3
WRITE statement, 1-16, 2-2

Z

ZERODIVIDE condition, 1-32

Introduction to
VAX–11 PL/I
AA-H950A-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)__________

Name__________ Date__________

Organization__________

Street__________

City__________ State__________ Zip Code or Country__________

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

Do Not Tear - Fold Here