OpenVOS PL/I Language Manual

Stratus Technologies

Notice

Manual Name: *OpenVOS PL/I Language Manual*

Part Number: R009
Revision Number: 05
OpenVOS Release Number: 17.0.0
Publication Date: July 2008

Stratus Technologies, Inc.
111 Powdermill Road
Maynard, Massachusetts 01754-3409

Preface

The *OpenVOS PL/I Language Manual (R009)* documents the PL/I language as implemented under OpenVOS. This manual is intended as a reference document, not as a tutorial.

This manual is intended for experienced application programmers working in an OpenVOS PL/I environment.

Manual Version

This manual is a revision. Change bars, which appear in the margin, note the specific changes to text since the previous publication of this manual. Note, however, that change bars are not used in new chapters or appendixes.

This revision incorporates the following changes.

- The `hex64` OpenVOS-supplied function has been added. See "OpenVOS-Supplied Function Descriptions" in Chapter 13.
- Text related to Continuum-series systems has been removed.
- The description of the iterative-do statement has been clarified. See "The Iterative-Do" in Chapter 12.
- The description of the `%include` preprocessor statement has been updated to discuss extended names. See "The `%include` Statement" in Chapter 11.

Manual Organization

This manual has 15 chapters and 4 appendixes.

Chapter 1 briefly describes the history of the PL/I language.

Chapter 2 defines what a PL/I program is and introduces the essential components of the PL/I language.

Chapter 3 discusses the block structure of PL/I programs and addresses such issues as block activation and scope rules.

Chapter 4 explains the OpenVOS PL/I data types and how they are used.

Chapter 5 documents the rules governing conversions between PL/I data types.

Chapter 6 describes the OpenVOS PL/I storage classes. This chapter includes discussions of storage sharing and how based variables and pointers are used.

Chapter 7 explains how names are declared. This chapter also includes an attribute reference guide.

Chapter 8 describes how to refer to PL/I objects within a program, including the use of subscripts and other qualifiers. This chapter also explains the rules the compiler uses to resolve a reference to a specific object.

Chapter 9 documents PL/I operators and describes how operators and operands are combined to form expressions. This chapter also explains the rules for evaluating expressions.

Chapter 10 documents the OpenVOS preprocessor statements.

Chapter 11 documents the PL/I preprocessor statements.

Chapter 12 describes the PL/I language statements.

Chapter 13 describes the OpenVOS PL/I built-in functions, pseudovariables, and OpenVOS-supplied functions.

Chapter 14 explains how to perform I/O using PL/I language statements.

Chapter 15 documents PL/I conditions and condition handlers.

Appendix A lists the abbreviations for certain OpenVOS PL/I keywords.

Appendix B describes how data is stored internally in OpenVOS PL/I.

Appendix C summarizes the nonstandard OpenVOS PL/I features.

Appendix D provides the OpenVOS internal character code set.

Related Manuals

See the following Stratus manuals for related documentation.

- *OpenVOS PL/I Subroutines Manual* (R005)
- *OpenVOS PL/I Transaction Processing Facility Reference Manual* (R015)
- *VOS PL/I Forms Management System* (R016)
- *VOS PL/I User's Guide* (R145)

Notation Conventions

This manual uses the following notation conventions.

- Italics introduces or defines new terms. For example:

  The *master disk* is the name of the member disk from which the module was booted.

- Boldface emphasizes words in text. For example:

  Every module **must** have a copy of the `module_start_up.cm` file.

- Monospace represents text that would appear on your terminal's screen (such as commands, subroutines, code fragments, and names of files and directories). For example:

  ```
  change_current_dir (master_disk)>system>doc
  ```

- Monospace italic represents terms that are to be replaced by literal values. In the following example, the user must replace the monospace-italic term with a literal value.

  ```
  list_users -module module_name
  ```

- Monospace bold represents user input in examples and figures that contain both user input and system output (which appears in monospace). For example:

  ```
  display_access_list system_default

  %dev#m1>system>acl>system_default

  w  *.*
  ```

Syntax Notation

A *language format* shows the syntax of an OpenVOS PL/I statement, portion of a statement, declaration, or definition. When OpenVOS PL/I allows more than one format for a language construct, the documentation presents each format consecutively. For complex language constructs, the text may supply additional information about the syntax.

The following table explains the notation used in language formats.

The Notation Used in Language Formats

Notation	Meaning
`element`	Required element.
`element...`	Required element that can be repeated.
{`element_1 element_2`}	List of required elements.
{`element_1 element_2`}`...`	List of required elements that can be repeated.
$\left\{\begin{array}{l}\textit{element\_1}\\ \textit{element\_2}\end{array}\right\}$	Set of elements that are mutually exclusive; you must specify one of these elements.
[`element`]	Optional element.
[`element`]`...`	Optional element that can be repeated.
[`element_1 element_2`]	List of optional elements.
[`element_1 element_2`]`...`	List of optional elements that can be repeated.
$\left[\begin{array}{l}\textit{element\_1}\\ \textit{element\_2}\end{array}\right]$	Set of optional elements that are mutually exclusive; you can specify only one of these elements.
Note: Dots, brackets, and braces are not literal characters; you should **not** type them. Any list or set of elements can contain more than two elements. Brackets and braces are sometimes nested.	

In the preceding table, *`element`* represents one of the following OpenVOS PL/I language constructs.

- keywords (which appear in monospace)
- generic terms (which appear in monospace italic) that are to be replaced by items such as expressions, identifiers, literals, constants, or statements
- statements or portions of statements

The elements in a list of elements must be entered in the order shown, unless the text specifies otherwise. An element or a list of elements followed by a set of three dots indicates that the element(s) can be repeated.

The following example shows a sample language format.

```
delete file(file_reference) [key(key_expression)] ;
```

In examples, a set of three vertically aligned dots indicates that a portion of a language construct or program has been omitted. The following example illustrates this concept.

```
a:   procedure;
declare   b    entry;
      .
      .
      .
      call b;
      .
      .
      .
end a;
```

The manual uses a special character, ␣ , to represent a space character. For example, the following output line has three leading space characters.

```
␣␣␣cat
```

Online Documentation

The OpenVOS StrataDOC Web site is an online-documentation service provided by Stratus. It enables Stratus customers to view, search, download, print, and comment on OpenVOS technical manuals via a common Web browser. It also provides the latest updates and corrections available for the OpenVOS document set.

You can access the OpenVOS StrataDOC Web site, at no charge, at `http://stratadoc.stratus.com`. A copy of OpenVOS StrataDOC on supported media is included with this release. You can also order additional copies from Stratus.

This manual is available on the OpenVOS StrataDOC Web site.

For information about ordering OpenVOS StrataDOC on supported media, see the next section, “Ordering Manuals.”

Ordering Manuals

You can order manuals in the following ways.

- If your system is connected to the Remote Service Network (RSN™), issue the `maint_request` command at the system prompt. Complete the on-screen form with all of the information necessary to process your manual order.

- Customers in North America can call the Stratus Customer Assistance Center (CAC) at (800) 221-6588 or (800) 828-8513, 24 hours a day, 7 days a week. All other customers can contact their nearest Stratus sales office, CAC office, or distributor; see `http://www.stratus.com/support/cac/index.htm` for CAC phone numbers outside the U.S.

Manual orders will be forwarded to Order Administration.

Commenting on This Manual

You can comment on this manual by using the command `comment_on_manual`. To use the `comment_on_manual` command, your system must be connected to the RSN. Alternatively, you can email comments on this manual to `comments@stratus.com`.

The `comment_on_manual` command is documented in the manual *OpenVOS System Administration: Administering and Customizing a System* (R281) and the *OpenVOS Commands Reference Manual* (R098). There are two ways you can use this command to send your comments.

- If your comments are brief, type `comment_on_manual`, press Enter or Return, and complete the data-entry form that appears on your screen. When you have completed the form, press Enter.
- If your comments are lengthy, save them in a file before you issue the command. Type `comment_on_manual` followed by `-form`, then press Enter or Return. Enter this manual's part number, `R009`, then enter the name of your comments file in the `-comments_path` field. Press the key that performs the `CYCLE` function to change the value of `-use_form` to `no` and then press Enter.

  Note: If `comment_on_manual` does not accept the part number of this manual (which may occur if the manual is not yet registered in the `manual_info.table` file), you can use the `mail` request of the `maint_request` command to send your comments.

Your comments (along with your name) are sent to Stratus over the RSN.

Stratus welcomes any corrections and suggestions for improving this manual.

Contents

Figures

Tables

Chapter 1:
OpenVOS PL/I

In 1963, an effort was begun to extend the FORTRAN programming language. That project soon turned into the development of a new language. This new language, known as "programming language one" or, more simply, PL/I, was first implemented commercially in 1966. On August 9, 1976, the American National Standards Institute (ANSI) approved a PL/I language standard, which was published as ANSI X3.53-1976.

The full PL/I language is powerful, versatile, and more than a little unwieldy. In 1976, an ANSI committee began working on a proper subset of full PL/I that would be more practical for general use. The standard for the Programming Language PL/I General-Purpose Subset, or Subset G, was accepted by ANSI on July 27, 1981, and published as ANSI X3.74-1981.

The ANSI standards leave some aspects of PL/I undefined. These aspects of the language might differ between various implementations of PL/I. For example, the order in which the two sides of an assignment statement are evaluated is undefined.

A program that depends on a particular implementation of an undefined aspect of the language might or might not produce expected results or consistent results. Even if the application runs successfully once, it might not run successfully at a later time or on a different implementation of PL/I. Therefore, the results of such a program are said to be *unpredictable*.

This manual documents OpenVOS PL/I, which is an implementation of PL/I Subset G. While it generally follows the ANSI standard, OpenVOS PL/I has many nonstandard features. These features are documented in Appendix C.

Chapter 2:
The Elements of OpenVOS PL/I

This chapter provides an overview of the OpenVOS PL/I language and defines some fundamental terms. Specifically, the chapter discusses the following topics.

- "Programs"
- "Identifiers"
- "Constants"
- "Punctuation"
- "Comments"
- "Variables and Storage"
- "Statements"

Programs

A PL/I source file is a text file. You can write and update source files using a text editor such as Emacs. The source file is called a *source module*. PL/I source module names have the suffix `.pl1`.

The text of a source module need not conform to any particular format. Text lines can be any length up to 300 characters. A source module can contain up to 32,767 lines of text, including blank lines, comments, and include files. The compiler ignores all blank lines. To improve readability, most programmers follow certain indentation conventions as illustrated by the examples in this manual.

The PL/I compiler processes the source module to produce an *object module*. An object module is a form of a program that the binder can read. Object module names have the suffix `.obj`.

The binder combines one or more object modules to produce a *program module*. A program module is an executable form of a program. Program module names have the suffix `.pm`.

For information on compiling and binding source modules, see the *VOS PL/I User's Guide* (R145).

Identifiers

An *identifier* is a sequence of 1 to 32 letters, digits, underline characters (_), and dollar-sign characters (`$`); the first character of an identifier must be a letter. The dollar-sign character is normally reserved for use in external system-related names.

The following examples are identifiers.

```
x
next
employee_name
s$error
t27
```

Identifiers can contain uppercase letters, lowercase letters, or both. An uppercase letter, such as `X`, is not equivalent to its lowercase counterpart, `x`, unless you choose the `-mapcase` argument of the PL/I compiler.

Identifiers cannot contain spaces or hyphens (`-`). A space separates two identifiers; the hyphen is the subtraction operator. Within an identifier, use an underline character in place of a hyphen.

PL/I programs contain two kinds of identifiers: keywords and names. A *keyword* is an identifier that denotes a part of a statement, such as a verb, an option, or a clause. A *name* is an identifier supplied by the programmer to denote an object that the program operates on, such as a variable, file, or label.

The meaning of a name is determined by a declaration of the name. For this reason, names are also known as declared names. See Chapter 7 for information on declaring a name.

The following example is a part of a PL/I program.

```
declare   x    fixed bin(15);
TOP:      x = 25;
```

In the preceding example, the identifiers `declare` and `fixed` are keywords. Both `x` and `TOP` are declared names: `x` is declared explicitly and `TOP` is declared contextually. Chapter 7 discusses how names are declared.

Keywords are not reserved in PL/I, which means that you can use a keyword as a name. However, this practice makes programs difficult to read and is generally discouraged.

PL/I recognizes abbreviations for certain keywords. See Appendix A for a list of these abbreviations.

Constants

PL/I recognizes two kinds of constants: named constants and literal constants.

A *named constant* is a declared name that represents a file, an entry point, a format, or a statement label. See Chapter 4 for information about file, entry, and label data.

A *literal constant* is a set of characters that always represents a particular value. PL/I recognizes three general types of literal constants: arithmetic constants, character-string constants, and bit-string constants. These three types of constants are described later in this section.

Chapter 4 discusses how the data type of a literal constant is determined. See Chapter 5 for information about data-type conversion.

The next three sections discuss the following topics.

- "Arithmetic Constants"
- "Character-String Constants"
- "Bit-String Constants"

Arithmetic Constants

Arithmetic constants represent decimal (base-10) values. In OpenVOS PL/I, binary arithmetic values have no constant representation. However, when you use a decimal value in a context where a binary arithmetic value is expected, the decimal value is automatically converted to the binary equivalent.

The following examples are arithmetic constants.

```
25
-7.5
3.12E-01
```

The first two of the preceding examples are fixed-point constants, and the third is a floating-point constant. The first is also an integer constant.

See Chapter 4 for a full description of arithmetic data.

Character-String Constants

Character-string constants consist of a series of ASCII characters (except the apostrophe) that is always enclosed in apostrophes (`'`). The apostrophes delimit the character-string value, but they are not a part of the value. If an apostrophe is required within a character-string constant, type two apostrophes (`''`). Note that the quotation-mark character (`"`) is **not** equivalent to the apostrophe or two apostrophes. The quotation-mark character has no special significance; it is treated the same as any other character within a constant.

The following examples are character-string constants.

```
'This is a character-string constant.'
'He said, "I don''t know."'
```

See Chapter 4 for a full description of character-string data. Appendix D contains a table showing the complete OpenVOS internal character set.

Bit-String Constants

Bit-string constants consist of a series of characters enclosed in apostrophes and immediately followed by the character `b`. The character `b` might be immediately followed by the character `1`, `2`, `3`, or `4`. The character following the `b` determines how the compiler should interpret the characters between the apostrophes. Table 2-1 summarizes how different types of bit-string constants are interpreted. The base determines the set of valid characters for bit-string constants.

Table 2-1. Bit-String Types

Bit-String Type	Interpretation of String
`b`	Base-2 (binary) digits
`b1`	Base-2 (binary) digits
`b2`	Base-4 digits
`b3`	Base-8 (octal) digits
`b4`	Base-16 (hexadecimal) digits

The following examples are bit-string constants.

```
'1'b
'1011'b
'775'b3
'a70'b4
```

The first two of the preceding examples are written in binary notation. The third is written in octal notation. The last example is written in hexadecimal notation.

Table 4-1 lists the expanded binary form of nonbinary bit strings. Chapter 4 also fully describes bit-string data.

Punctuation

All identifiers and constants must be separated from one another by one or more spaces or by a punctuation symbol. You can include additional spaces around punctuation symbols; this practice often improves readability, but it is not required.

Punctuation symbols can be divided into two classes: operators and separators.

Figure 2-1 shows the PL/I operators.

```
+    -    *    /    **   =    ^=   <    >    <=

>=   ^<   ^>   |    !    &    ^    ||   !!   ->
```

Figure 2-1. PL/I Operators

See Chapter 9 for more information on the PL/I operators.

Figure 2-2 shows the PL/I separators. Note that the space character is also a separator.

```
(   )   ,   .   :   ;
```

Figure 2-2. PL/I Separators

The following examples illustrate the use of punctuation in PL/I statements.

```
a=b+c*d;
a  = b+  c * d;
do k  =  n to m by k;
```

The first of the preceding examples uses only operators to separate identifiers. The second example uses arbitrarily placed extra spaces. The third example uses extra spaces around the equals sign, but otherwise uses the minimum number of spaces.

When a character-string or bit-string constant follows an identifier, as in `picture'9v99'`, you need not separate the identifier from the constant by a space. The apostrophe that opens the string constant effectively serves as a separator. Arithmetic constants must always be separated from identifiers by a space character or punctuation symbol.

Comments

A *comment* is a remark included by the programmer within the source module. The compiler ignores the contents of comments.

The beginning of a comment is marked by a slant followed immediately by an asterisk (`/*`). The end of a comment is marked by an asterisk followed immediately by a slant (`*/`). Any sequence of characters can appear between these delimiters, provided the sequence does not contain an asterisk followed immediately by a slant.

A comment can appear anywhere within a program that a space can appear. The compiler treats a comment as the equivalent of a space.

The following examples illustrate how to format comments.

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

if a>25/* This is a comment, too. */then

/* This is a
   comment written on
   more than one line. */
```

If you omit the terminating asterisk-slant combination from a comment, the compiler treats all program text up to the next asterisk-slant combination as part of the comment. That portion of the source module is ignored by the compiler. This might result in a misleading compiler error message. If you compile with the `-list` argument, the compiler puts an asterisk into the line number field of the listing for each line on which a comment is continued from the preceding line. This can aid you in debugging the program.

See the *VOS PL/I User's Guide* (R145) for an explanation of the various compiling and binding arguments.

Variables and Storage

A *variable* is a named object that can hold various values. Every variable has a data type and a storage class. The *data type* determines the kind of values the variable can hold and the operations that can be performed on those values. The *storage class* determines when and where storage for the variable is allocated.

This section discusses the following topics.

- "Data Types and Conversion"
- "Storage Classes"

Data Types and Conversion

Each variable must have a data type. If you do not specifically declare the data type of a variable, the compiler issues a warning and gives the variable the data type `fixed bin(15)`.

Ordinarily, every variable is given a data type in a `declare` statement. In the following example, four variables are declared.

```
declare   average   float dec(7);
declare   length    fixed bin(15);
declare   c_string  char(5);
declare   b_string  bit(1) aligned;
```

In the preceding example, `average` is a variable that can hold floating-point decimal values with a precision of 7; `length` can hold fixed-point binary integer values with a precision of 15; `c_string` can hold any string of 5 ASCII characters; `b_string` can hold either `'1'b` or `'0'b`.

Although each variable can hold only a specific type of value, the PL/I language allows a value to be converted from one data type to another. Whenever a value is assigned to a variable, the value is converted to the data type of that variable before assignment.

The following example uses the declarations from the preceding example to illustrate some common data-type conversions.

```
average = 1;    /* Converts 1 to a floating-point number         */
length  = 2.5;  /* Converts 2.5 to the binary integer equivalent
                   of 2                                          */
c_string = -4.5;/* Converts -4.5 to the character string ' -4.5' */
b_string = 0;   /* Converts 0 to the bit string '0'b             */
c_string = 256; /* Converts 256 to the character string '   25'  */
```

Note that in the preceding example, when the assignment `b_string = 0` is compiled, the compiler issues a level-1 warning that an arithmetic value has been implicitly converted to a bit string.

Chapter 4 discusses data types. Chapter 5 explains data-type conversions.

Storage Classes

The storage class of a variable determines when storage is allocated for the variable and when that storage is accessible.

Unless a different storage class is specifically declared, the compiler gives variables the `automatic` storage class. Storage is allocated for an automatic variable each time its containing procedure or begin block is activated. Every time the procedure returns to its caller, the storage allocated for that activation is freed. Consequently, an automatic variable does not retain its value after the procedure containing it returns.

If a variable must retain its value between activations of the procedure that contains it, declare the variable to have the `static` storage class. Storage for static variables is allocated prior to program execution. The storage remains allocated throughout program execution.

The following example illustrates how storage classes are specified in variable declarations.

```
declare   i         float dec(7);
declare   k         fixed bin(15) static;
declare   table(4)  char(2) static initial('ab','cd','ef','gh');
```

In the preceding example, because the variable `i` is declared without a specific storage class, the compiler gives it the `automatic` storage class. Both `k` and `table` are declared as static variables. Note also that each element of the array `table` is given an initial value. Only static variables can be initialized in this way.

Chapter 6 discusses the PL/I storage classes. Chapter 7 describes the `initial` attribute.

Statements

The text of a PL/I program is a series of tokens. A *token* is an identifier, literal constant, punctuation symbol, comment, or compile-time statement. Tokens are described in the remaining sections of this chapter.

In English, you use words and punctuation to form sentences; in PL/I, you use tokens to form statements. A PL/I *statement* is a sequence of tokens ending with a semicolon (`;`). Most statements begin with a keyword that identifies the purpose of the statement; typically, programmers use that initial keyword to refer to a statement. The following examples illustrate the `read`, `call`, `stop`, `return`, and `do` statements, respectively.

```
read file(f) into(x);
call p(a,b,c);
stop;
return;
do k = 1 to 10;
```

Two PL/I statements do not have keywords: the assignment statement and the null statement. The first statement in the following example is an assignment statement; the second, which contains no text except the terminating semicolon, is a null statement.

```
a = -c;
;
```

Chapter 12 describes the assignment and null statements.

The remainder of this section discusses the following topics.

- "Compound Statements"
- "Order of Execution"

Compound Statements

The statements discussed thus far have all been *simple statements*, meaning that they contain only one statement. PL/I has two compound statements: the `if` statement and the `on` statement. These are called *compound statements* because they contain one or more other statements.

The following example contains an `if` statement and an `on` statement.

```
if a > b
then read file(f) into(x);

on endfile(f) stop;
```

The `if` statement introduces a condition that determines whether a statement should be executed. The `if` statement has the following general form.

```
if expression
then then_clause;
[else else_clause;]
```

Since each embedded statement ends with a semicolon, the `if` statement requires no terminating semicolon of its own. Because the embedded statements can themselves be `if` statements, you can create nested `if` statements, as shown in the following example.

```
if a > b
then if d < c
     then read file(f) into(x);
     else stop;
```

In the preceding example, the second `if` statement has both a `then` clause and an `else` clause. An `else` clause always corresponds to the nearest preceding unclosed `then` clause. A `then` clause is closed by an `else` clause or the end of a containing group. In the example, the first `if` statement has only a `then` clause. If you want the `stop` statement to be executed whenever `a` is not greater than `b`, you can either write a null `else` clause or enclose the second `if` statement in a do-group. The following examples illustrate both methods.

```
if a > b
then if d < c
     then read file(f) into(x);
     else;        /* Example using a null else clause */
else stop;


if a > b
then do;          /* Example using a do-group */
     if d < c
     then read file(f) into(x);
     end;
else stop;
```

For more information on the `if` statement, see Chapter 12.

Unlike `if` statements, `on` statements **cannot** be nested. You use an `on` statement to establish an action to be taken if an exceptional condition occurs. This action is called an *on-unit*. In the following example, an on-unit is created for the `endfile(f)` condition.

```
on endfile(f) stop;
```

For further information about the `if` statement and the `on` statement, see Chapter 12. For information about exception handling, see Chapter 15.

Order of Execution

Generally, statements are executed in the order in which they appear in the program. That is, the first executable statement is executed first, followed by the second, and so forth. However, certain PL/I statements alter this flow. These statements are summarized in Table 2-2.

Table 2-2. Statements That Alter the Order of Execution

Statement	Description
`call`	Transfers control to a specific procedure entry point
`do`	Causes a group of statements to be executed zero, one, or more times
`end`	Transfers control back to the point from which the current procedure was activated (or marks the end of a do-group)
`goto`	Transfers control to a specific statement
`return`	Transfers control back to the point from which the current procedure was activated
`signal`	Transfers control to an on-unit
`stop`	Halts program execution

> **Note:** The effect of a function reference is similar to the effect of a `call` statement; see Chapter 3 for information.

All of the statements shown in Table 2-2 are fully described in Chapter 12. In addition, the `call` and `return` statements are discussed in Chapter 3. The `goto` statement is also discussed in Chapter 4 and Chapter 7.

Chapter 3:
Procedures and Blocks

This chapter discusses the following topics related to procedures and blocks.

- "Overview"
- "Scope of a Name"
- "Procedure Calls and Returns"
- "Entry Points"
- "Block Activation and Recursion"
- "Inline Procedures"
- "Parameters and Arguments"
- "Functions"
- "Begin Blocks"

Overview

A *procedure* is a sequence of statements that begins with a `procedure` statement and terminates with an `end` statement. The group of statements within a procedure constitutes a *block*. The following example shows two procedures: `a` and `b`.

```
a:   procedure;
        .
        .
        .
end a;

b:   procedure;
        .
        .
        .
end b;
```

Procedures can contain other procedures. Contained procedures are called *nested* or *internal* procedures. Procedures that are not contained within other procedures are called *external* procedures. A source module consists of one or more external procedures.

The following example contains one external procedure and three internal procedures.

```
a:   procedure;
               .
               .
               .
     b:   procedure;
               .
               .
               .
     end b;

     c:   procedure;
               .
               .
               .
          d:   procedure;
               .
               .
               .
          end d;
     end c;
end a;
```

In the preceding example, procedure `a` is external. The internal procedures `b` and `c` are nested within `a`. Procedure `d` is nested within procedure `c`.

Scope of a Name

Names declared within a procedure are recognized throughout a distinct region of the program text. This region is called the *scope* of the name. A name can be referenced anywhere within its scope.

Most variables have *internal* scope. The scope of an internal variable is the procedure in which it is declared and all procedures nested within that procedure, except those nested procedures in which the same name is redeclared.

The following example illustrates internal scope.

```
a:   procedure;

declare   x    float bin(24);
declare   y    float bin(24);
          .
          .
          .
     b:   procedure;

     declare   x    char(4);
     declare   z    fixed dec(7,2);
               .
               .
               .
     end b;
end a;
```

The scope of variable `y` includes both procedure `a` and procedure `b`, because it is declared in the external procedure `a`. The scope of `x` as a floating-point number is procedure `a`, excluding procedure `b`, because `x` is redefined in procedure `b`. The scope of `x` as a character string is limited to procedure `b`, as is the scope of variable `z`.

For more information on scope, see Chapter 6 and Chapter 7.

Procedure Calls and Returns

The statements within a procedure are executed only when the procedure is activated by another procedure. You can use the `call` statement to activate all procedures except functions. Functions are discussed later in this chapter.

In the following example, procedure `b` is activated within procedure `a`.

```
a:   procedure;
       .
       .
       .
     call b;
       .
       .
       .
end a;

b:   procedure;
       .
       .
       .
end b;
```

In the preceding example, the statements in procedure b are executed when b is called from within procedure a. Procedure a resumes execution when b is completed, even if b is nested within a, as illustrated in the following example.

```
a: procedure;
        .
        .
        .
b: procedure;
        .
        .
        .
end b;
        .
        .
        .
end a;
```

In the preceding example, when the procedure statement of b is encountered as the result of normal program flow in procedure a, that statement and all statements up to and including the end statement that terminates b are ignored. Execution resumes with the statement following the end of b. If a does not call b, the statements in b will never execute.

A procedure returns to its caller when either a return statement or the procedure's end statement is executed. Procedure b in the following example can return in either of these ways.

```
b:   procedure;
           .
           .
           .
     if x < 0
     then return;
           .
           .
           .
end b;
```

In the preceding example, the expression x < 0 is evaluated in the middle of procedure b. If the expression is true, the activation of b terminates immediately, and the procedure returns to its caller. If the expression is false, execution continues until the end of b.

Entry Points

This section describes the following topics.

- "Primary Entry Points"
- "Secondary Entry Points"
- "External Entry Points"

Primary Entry Points

When a procedure is activated, execution usually begins from the `procedure` statement. The `procedure` statement is the *primary entry point* into the block. The following example illustrates a primary entry point.

```
a:   procedure;
          .
          .
          .
     call b;
          .
          .
          .
     b:   procedure;
          .
          .
          .
     end b;
end a;
```

In the preceding example, when the `call` statement is executed, procedure `b` is activated and control is transferred to the `procedure` statement of procedure `b`.

Secondary Entry Points

You can use the `entry` statement to establish *secondary entry points* within a procedure. In the following example, procedure `b` has a secondary entry point named `b2`.

```
a:   procedure;

     call b;
     call b2;

     b:   procedure;
            .
            .
            .
     b2:  entry;
            .
            .
            .
     end b;
end a;
```

In the preceding example, the statement `call b` activates procedure `b`. Control is transferred to the `procedure` statement. The statement `call b2` also activates procedure `b`. In this case, control transfers to the `entry` statement with the label prefix `b2`. Statements between the `procedure` statement and `entry` statement in procedure `b` are not executed as a result of the second call.

If an `entry` statement is encountered as the result of normal program flow, it is ignored. Execution continues with the statement immediately following the `entry` statement.

External Entry Points

All entry points to an external procedure can be referenced in other source modules. Such entry points, called *external entry points*, must be explicitly declared in the module in which they are referenced. In the following example, the entry point `b` is in another source module.

```
a:   procedure;

declare    b    entry;
           .
           .
           .
      call b;
           .
           .
           .
end a;
```

The binder must combine the object module that contains the entry point `b` with the current object module. For information on binding a program, see the *VOS PL/I User's Guide* (R145).

For additional information on entry points, see Chapter 4 and Chapter 7.

Block Activation and Recursion

Each time a procedure is called, it becomes active. The procedure remains active until it returns from the call. A recursive procedure can be activated more than once during the execution of a single program.

This section describes the following topics.

- "Stacks and Stack Frames"
- "Recursive Procedures"

Stacks and Stack Frames

When a program runs, a series of storage areas is created. This series is called a *stack*. Each time a procedure is activated, an associated area of storage is allocated on the stack. This area of storage is called a *stack frame*. The stack frame holds information that is unique to that activation. This information includes the storage of automatic variables declared in the procedure and the location to which control returns when the activation is complete.

You can visualize a stack as a series of stack frames set one on top of the other. The stack frame on the top of the stack is associated with the most recently activated procedure. Note that there may not be a stack frame associated with each activation of a procedure. Frameless procedures and inline procedures, for example, do not have their own stack frames. See the *VOS PL/I User's Guide* (R145) for additional information about stack frames and the stack.

Stack frames can be added to or removed from the top of the stack only. A stack frame that is added to the stack is said to be *pushed* onto the stack; a stack frame that is removed from the stack is said to be *popped* from the stack.

If procedure `a` calls procedure `b`, the stack frame associated with the activation of `b` is pushed onto the stack on top of the stack frame associated with the activation of `a`. The stack frame associated with the activation of `b` becomes the current stack frame. The stack frame of procedure `a` is saved until the activation of `b` completes. When `b` returns, its stack frame is popped from the stack. The stack frame of `a` then becomes the current stack frame.

Consider the following example.

```
a: procedure;
     call b;

b: procedure;
     .
     .
     .
     call c;
end b;

c: procedure;
     .
     .
     .
end c;

end a;
```

In the previous example, procedure `a` calls procedure `b`, which, in turn, calls procedure `c`. Using the `trace` debugger request, you can get a representation of the stack during program execution. The following is a representation of the stack from the previous example, with a breakpoint set within procedure `c`.

```
# 4: c (line 11 in module a)
# 3: b (line 7 in module a)
# 2: a (line 3 in module a)
```

When procedure `c` completes execution, its stack frame is popped from the stack. Control then returns to procedure `b`. When procedure `b` completes execution, its stack frame is popped and control returns to procedure `a`. When procedure `a` completes execution, its stack frame is popped and control returns to the operating system. If you set another breakpoint in procedure `b` after the completion of procedure `c`, the stack appears as shown in the following example.

```
# 3: b (line 7 in module a)
# 2: a (line 14 in module a)
```

For information on using the debugger, see the *VOS PL/I User's Guide* (R145) and the *VOS Symbolic Debugger User's Guide* (R308).

Recursive Procedures

In most cases, only one activation of a given procedure appears on the stack at any one time. However, some procedures call, and thus can activate, themselves. Such procedures are *recursive*. For example, if procedure `a` calls itself, or calls any other procedure that calls `a`, `a` is a recursive procedure, regardless of how indirect the chain of calls might be. The `procedure` statement of a recursive procedure must include the `recursive` option.

In the following example, `a` is a recursive procedure.

```
a:   procedure recursive;
declare   x    fixed bin(15);
declare   y    fixed bin(15) static initial(0);

     y = y + 1;
       .
       .
       .
     if y = 1
     then call a;
       .
       .
       .
end a;
```

In the preceding example, when the `call` statement is executed, the stack frame associated with the initial activation of `a` is pushed down by a new stack frame associated with a new activation of `a`. All automatic variables declared in `a`, such as `x`, have distinct storage allocated within each of these stack frames. If the value of `x` is altered in the second activation of `a`, the value of `x` in the first activation remains unchanged. Storage for static variables, such as `y`, is only allocated once for the program; the value of such a variable is retained between calls to the procedure. When the second activation of `a` completes, its stack frame is popped and the original activation of `a` becomes current again.

Recursive procedures are often used in applications that process linked lists, trees, or other list structures. An example of a recursive procedure that processes a linked list appears in Chapter 6.

Inline Procedures

Each time you activate a procedure, there is a cost involved in executing the procedure itself and also in executing the call. When the body of a procedure is small, the amount of code in the calling sequences may be more than the amount of code in the procedure body. It is, therefore, more efficient to use the `inline` option of the `procedure` statement to cause inline expansion of the procedure into the caller's code. Since inline expansion does slow compilation, it is only available with optimization level 3 or higher.

The `inline` option causes the body of the procedure to be substituted for the call, and the actual arguments are substituted for the formal parameters. Procedures containing the `inline` option behave just as they would without the option except that they execute more quickly. Because they are part of the enclosing procedure's stack frame, they have no separate stack frame of their own.

A procedure can qualify for inline expansion **only** if the following conditions are met. If any of these conditions are not met, the procedure is treated as a normal procedure, no inline expansion is performed, and the compiler issues a warning.

- The procedure must not call itself recursively.
- The procedure must not contain another block.
- The entry point must not be assigned to an entry variable.
- The argument matching a parameter with asterisk extents must have constant extents unless the parameter is an `in` parameter or the argument is passed by reference.
- The procedure must **not** contain any of the following:
  - a label constant array
  - automatic or defined variables with adjustable extents
  - `on`, `signal`, or `revert` statements
  - calls to the `paramptr` or `entryinfo` function
  - calls to the `s$enable_condition` or `s$revert_condition` subroutine
  - more than one entry point
  - I/O statements or `format` statements—calls to OpenVOS I/O subroutines, however, are allowed
- The program must be compiled with optimization level 3 or greater.

In the following example, `a` is an inline procedure.

```
main: procedure;

     call a;    /* The code for procedure a is expanded here. */

a: procedure inline;
      .
      .
      .
end a;
end main;
```

A `procedure` statement can contain both the `inline` option and the `returns` option, in any order, but the parameter list must precede them both.

If the program is compiled with the `-table` argument, no inline expansion is performed. If the program is compiled with the `-production_table` argument and an optimization level of 3 or greater, inline expansion is performed and the procedure is also compiled as an "out of line" procedure, which allows you to call the procedure from the debugger.

Inline procedures are restricted in their use of the `unspec`, `maxlength`, `size`, and `bytesize` built-in functions as described in the following list.

- The `unspec` function requires a scalar variable reference; therefore, a formal parameter to an inline procedure cannot be specified as an argument to the `unspec` function unless the corresponding actual argument is a scalar variable reference.
- The `maxlength` function requires a string variable reference; therefore, a formal parameter to an inline procedure cannot be specified as an argument to the `maxlength` function unless the corresponding actual argument is a string variable reference.
- The `size` and `bytesize` functions operate on the storage of variables, and can only accept level-one, unsubscripted variables as arguments; therefore, a formal parameter to an inline procedure cannot be specified as an argument to the `size` and `bytesize` functions unless the corresponding actual argument is a level-one, unsubscripted variable.

See Chapter 12 for more information about the `inline` option.

Parameters and Arguments

The purpose of a procedure is to package a set of declarations and executable statements to form a block that can be executed from one or more places in a program. Often, a procedure is more useful if it can operate on different values each time it is activated. To make this possible, a procedure uses parameters. A *parameter* describes an area of storage on which the procedure operates. A procedure's parameter list can contain from 0 to 127 arguments. A procedure can have **no** parameter list but not an **empty** parameter list.

When a procedure is called, one argument is passed from the caller for each parameter. An *argument* is a variable, constant, or expression with a value that can be assigned by its corresponding parameter. Arguments correspond with parameters positionally, which means that the first argument corresponds with the first parameter, the second argument with the second parameter, and so forth. The number of arguments must be equal to the number of parameters.

The following example demonstrates the use of parameters and arguments.

```
main:     procedure;
declare   a    fixed bin(15);
declare   b    char(256) varying;
declare   d    fixed bin(15);
declare   e    char(256) varying;

     call w(a,b);
          .
          .
          .
     call w(d,e);
          .
          .
          .
w:        procedure(p_code,p_str);
declare   p_code    fixed bin(15);
declare   p_str     char(256) varying;
          .
          .
          .
     put file(f) list(p_code,p_str);

end w;
end main;
```

In the preceding example, `a` and `b` are arguments of the first call to procedure `w`. While `w` is executing as the result of this call, `a` corresponds to `p_code` and `b` corresponds to `p_str`. This means that the parameter `p_code` shares storage with the argument `a`, and the parameter `p_str` shares storage with the argument `b`. Any operation acting upon `p_code` actually acts upon `a`; any operation acting upon `p_str` actually acts upon `b`.

While `w` is executing as the result of the second call, `d` corresponds to `p_code` and `e` corresponds to `p_str`. Therefore, operations acting upon `p_code` and `p_str` actually act upon `d` and `e`, respectively.

No special restrictions apply to parameter names. However, using a prefix like `p_` helps to distinguish parameters from other program objects.

Different entry points to the same procedure can have different parameter lists. Every time you call a procedure, you must provide one argument for each parameter to the entry point you reference. The following example illustrates.

```
a:  procedure;
declare   a    fixed bin(15);
declare   b    char(256) varying;
declare   ss   char(64) varying;

     call w(a,b);
          .
          .
          .
     call little_w(ss);
          .
          .
          .
w:        procedure(p_code, p_str);
declare   p_code    fixed bin(15);
declare   p_str     char(256) varying;
          .
          .
          .
little_w: entry(p_short_str);
declare   p_short_str char(64) varying;
          .
          .
          .
     return;
end w;
```

In the preceding example, when procedure `w` is entered at its primary entry point, it requires two arguments: a 2-byte binary integer followed by a varying-length character string with a maximum length of 256 characters. When `w` is entered at `little_w`, only one argument is required: a varying-length character string with a maximum length of 64 characters.

> **Note:** You must not reference `p_str` and `p_code` in any statements that will be executed when the procedure `w` is entered at `little_w`. Likewise, you must not reference `p_short_str` in any statements that will be executed when `w` is invoked. All automatic variables declared anywhere within `w` are allocated for every activation of `w`, regardless of which entry point is used.

The next seven sections discuss the following topics.

- "Passing Arguments by Reference or by Value"
- "Arguments Declared with the `in` Attribute"
- "Data-Type Matching"
- "Alignment Compatibility"
- "Parenthesized Arguments"
- "Array and Structure Parameters"
- "Input Arguments and Output Arguments"

Passing Arguments by Reference or by Value

You can think of an *argument list* as an array of pointers. Each value in the array is the storage address of an argument. It is this array of pointers that is actually passed to the called procedure.

When possible, arguments are passed *by reference*. This means that the storage address in the argument list is the address of a variable in the calling procedure. The called procedure then operates on that variable storage.

In some cases, arguments are not passed by reference; instead, they are passed *by value*. When an argument is passed by value, its value is copied to a temporary area of storage in the caller's stack frame. The storage area's address, rather than the variable's address, is then included in the argument list. Arguments are passed by value in the following cases.

- The argument is an expression.
- The argument is a reference to a variable whose data type does not match that of the parameter. The compiler issues a warning in this case.

All of the following are expressions.

- a function reference
- a built-in function reference
- a constant
- a parenthesized variable reference

The following example illustrates how arguments are passed by reference and by value.

```
declare   a    fixed bin(15);
     a = 0;

     call subr(a);

     call subr((a));
          .
          .
          .
subr: procedure(p_count);

     declare   p_count   fixed bin(15);

     p_count = p_count + 1;

end subr;
```

As shown in the preceding example, in the first call to `subr`, the argument `a` is passed by reference. This call has the effect of changing the value of `a` to `1`. In the second call, because the argument `a` is enclosed in parentheses, it is treated as an expression and is passed by value. The value of `a` is copied to an area of temporary storage and `subr` operates on that storage. The value of `a` is not changed by the second call.

If you pass an argument by reference and need to reference its storage in a subsequent invocation of the procedure in which you do not pass the same argument, you must declare both the argument and the corresponding formal parameter with the `volatile` attribute. Consider the following example.

```
a: procedure;

declare num1        fixed bin(15) volatile;
declare num2        fixed bin(15);
declare first_time  bit(1);
declare s$write     entry(char(*) varying);
declare ltrim       builtin;

     first_time = '1'b;
     num1 = 5;

     call b(num1);
     call s$write('num1 + 1 = ' !! ltrim(num1 + 1));

     first_time = '0'b;
     num2 = 15;

     call b(num2);
     call s$write('num1 + 1 = ' !! ltrim(num1 + 1));

b: procedure(pnum);

     declare pnum        fixed bin(15) volatile;
     declare remember    pointer static;
     declare based_num   fixed bin(15) based;

     if first_time = '1'b
          then remember = addr(pnum);
     else remember->based_num = 10;

end b;
end a;
```

The preceding example produces the following output.

```
num1 + 1 = 6
num1 + 1 = 11
```

In the preceding program, `num1` and its corresponding parameter, `pnum`, are declared with the `volatile` attribute. Specifying this attribute ensures that the compiler will neither optimize expressions containing references to that variable nor store the value of that variable in a register between references. (Note, however, that depending on the hardware platform, if you do not specify `volatile`, such a value could be stored either in a register or in memory; you must specify `volatile` to **guarantee** that the value will be stored in memory.)

In the program, the first time procedure `b` is called, the value of `num1` is stored in memory, not in a register. The second time `b` is called, the value stored in memory is changed from 5 to 10. Thus, when `s$write` is called the second time, the compiler re-evaluates the expression `num1 + 1`. The result, therefore, of `num1 + 1` is 11.

If you had **not** specified the `volatile` attribute for `num1` and `pnum`, the output would be as shown in the following example.

```
num1 + 1 = 6
num1 + 1 = 6
```

In this case, the compiler holds onto the value of `num1 + 1` rather than re-evaluating it. Therefore, the result of `num1 + 1` is 6, not 11.

See Chapter 7 for more information on the `volatile` attribute. See the *OpenVOS PL/I Subroutines Manual* (R005) for more information on the `s$write` subroutine.

Arguments Declared with the `in` Attribute

It is possible to declare a parameter with the `in` attribute, specifying that it is input-only. If you specify a parameter as input-only, you must not modify its value; if you attempt to do so, the compiler issues an error message. Using the `in` attribute can make your code more efficient, and it causes the compiler to check for errors such as the caller passing uninitialized variables and the callee changing values. With this information, the compiler may be better able to optimize code.

For additional information about the `in` attribute, see Chapter 7.

Data-Type Matching

In order for an argument to be passed by reference, its data type must exactly match the data type of the corresponding parameter. If the data types do not match exactly, the compiler attempts to convert the argument to the proper type and pass it by value. The compiler issues a warning in such cases.

Table 3-1 describes what constitutes a match between an argument and a parameter. Note that all parameters and arguments must be alignment-compatible. See "Alignment Compatibility" later in this chapter. For additional information about alignment, see Chapter 4.

Table 3-1. Requirements for Passing Arguments by Reference

Parameter Type	Attributes That Must Match
Arithmetic	Base, scale, and precision
Character string with constant extents	`aligned` and `varying` attributes and length
Character string with asterisk extents	`aligned` and `varying` attributes
Bit string with constant extents	`aligned` attribute and length
Bit string with asterisk extents	`aligned` attribute
Pictured value	Picture must be identical
Array with constant extents	Extents and component data type
Array with asterisk extents	Component data type
Structure	Same hierarchic organization and same member data types (additional alignment rules described in text)

If an argument is a reference to a variable whose data type does not match the data type of the corresponding parameter, the argument is passed by value unless it is a reference to an entire array or structure. Attempting to pass an entire array or structure that does not match the corresponding parameter is an error. Attempting to pass any other argument that does not match the corresponding parameter produces a warning message from the compiler. Such warning messages are suppressed when arguments are enclosed in parentheses.

Alignment Compatibility

When passing arguments, **one** of the following conditions must be true in order for the argument and parameter to be compatibly aligned.

- Both the argument and the corresponding parameter are aligned using shortmap rules.
- Both the argument and the corresponding parameter are aligned using longmap rules.
- An argument aligned using longmap rules is being passed to a parameter that is aligned using shortmap rules. In this case, both the argument and the parameter must be nonstructure data types and **not** arrays of aligned strings.

- For a structure, an argument aligned using longmap rules can be passed to a parameter that is aligned using shortmap rules if the size and shape are the same.
- An argument aligned using shortmap rules is being passed to a parameter that is aligned using longmap rules, and both the argument and the parameter are of one of the following data types.
  - fixed bin(15)
  - `bit` (must be an unaligned bit string)
  - `char` (must be an unaligned character string)
  - `char varying`

  If the argument is not one of the preceding data types, the compiler will issue a warning.

  Note: In this discussion, do not confuse shortmap and longmap alignment with the `aligned` attribute. The `aligned` attribute refers only to the alignment of bit strings and character strings. See the description of the `aligned` attribute in Chapter 7 for more information.

Consider the following example.

```
main: procedure;

declare 1 struct1   shortmap,
          2 s_1     fixed bin(15),
          2 t_1     fixed bin(31);
     .
     .
     .
     call w(struct1);

w: procedure(struct2);
declare 1 struct2    shortmap,
          2 s_2      fixed bin(15),
          2 t_2      fixed bin(31);
     .
     .
     .
end w;

end main;
```

Both `struct1` and `struct2` are declared with the `shortmap` attribute. You could have omitted the `shortmap` attribute, if the system default is shortmap alignment or if you specified the `shortmap` option with the `-mapping_rules` argument of the `pl1` command or the `%options` statement. The structures themselves, and their members, `s_1` and `t_1` of `struct1` and `s_2` and `t_2` of `struct2`, are all allocated on even-numbered (mod2) boundaries. (As described in Chapter 4, alignment boundaries are determined with the modulo operation, represented in the table as mod*n*; the *n* specifies the number of bytes by which the boundary is divisible with no remainder.)

In the previous example, the two structures have the same data alignment. Suppose instead, you had declared `struct2` with the `longmap` attribute, as shown in the following example.

```
main: procedure;

declare 1 struct1   shortmap,
          2 s_1     fixed bin(15),
          2 t_1     fixed bin(31);
     .
     .
     .
     call w(struct1);

w: procedure(struct2);
declare 1 struct2    longmap,
          2 s_2      fixed bin(15),
          2 t_2      fixed bin(31);
     .
     .
     .
end w;

end main;
```

In the preceding example, `struct2` is expected to be on a mod4 boundary as required by its largest member; of its members, `s_2` is expected to be on a mod2 boundary and `t_2` is expected to be on a mod4 boundary, so there are two bytes of padding between `s_2` and `t_2`. The two structures do **not** have the same data alignment, and they are not compatible. Therefore, the compiler will issue a warning.

For a complete description of longmap and shortmap alignment, see Chapter 4.

Parenthesized Arguments

If a reference to an argument is enclosed in parentheses, it is treated as an expression and is passed by value. That is, its value is converted to the data type of the corresponding parameter and is assigned to a temporary area of storage.

In the following example, `b` is passed by value.

```
call p(a,(b));
```

In the preceding example, `a` is passed by reference if it matches the corresponding formal parameter defined in `p`; otherwise, it is passed by value (a compiler message will warn of this). Because `b` is in parentheses, it is passed by value.

Array and Structure Parameters

Parameters are not restricted to scalar types. Arrays and structures can also be passed as parameters, but they **must** be passed by reference, not by value. When you pass a structure as an argument, the size and the shape of the structure in the argument must be the same as that of the structure in the corresponding parameter.

The following example uses array and structure parameters.

```
a:
     procedure;

declare   x(5)           fixed bin(15);

declare   1 employee     ,
            2  first     char(12) varying,
            2  mid       char(1),
            2  last      char(12) varying;

declare   spouse         type(employee);

     call b(x, employee, spouse);
     .
     .
     .
b:   procedure(p_array, p_emp, p_spouse);

declare   p_array(*)     fixed bin(15);

declare   p_emp          type(employee);

declare   p_spouse       type(employee);
     .
     .
     .
end b;
end a;
```

In the preceding example, because an asterisk (`*`) is used in the declaration of `p_array`, the array passed to `p_array` can have any extents; `p_array` accepts the extents of the corresponding argument. Further information on arrays and array extents appears in Chapter 4.

The `type` attribute is used in the preceding example to create three structures—`spouse`, `p_emp`, and `p_spouse`—that are of the type `employee`. See Chapter 7 for more information on the `type` attribute. (Note that the `type` attribute is an OpenVOS extension.)

The declaration of `p_emp` in the preceding example could have been written as shown in the following example.

```
declare   1  p_emp like employee;
```

The members of `p_emp` would then have the same names and data types as the members of the structure `employee`. You can use the `like` attribute when declaring parameters in an entry point to an external procedure. For example, if `b` in the preceding example was an external procedure, you could declare `b` and its parameter list as shown in the following example.

```
declare   b    entry((*)fixed bin(15), 1 like employee);
```

The `like` attribute is discussed in Chapter 7, while structures and pointers are discussed in Chapter 4. See Chapter 6 and Chapter 8 for information on using pointers with based variables.

Input Arguments and Output Arguments

The values of certain arguments are never altered by the called procedure. Such arguments are called *input arguments* because they provide input to the called procedure but do not return any output from that procedure. Input arguments can be passed either by reference or by value.

If the value of an argument can be altered by a called procedure, it is an *output argument*. Output arguments must be passed by reference.

See "Passing Arguments by Reference or by Value," earlier in this chapter, for additional information.

Functions

A *function* is a special kind of procedure that returns a single scalar value. The `procedure` statement and any `entry` statements of a function must include a `returns` option that specifies the data type of the value the function returns. The activation of a function must be terminated by a `return` statement that includes an expression representing the returned value. Executing the `end` statement of a function is an error.

You can specify from 0 to 127 parameters in the parameter list for a function declaration (and, in the function call, you must specify one argument for each parameter in a function's parameter list). The number of arguments in a function call **must** be equal to the number of parameters in the parameter list specified in the function declaration. See "Parameters and Arguments," earlier in this chapter, for more information on parameters and arguments.

The following function returns the average of two floating-point numbers.

```
average: procedure(p_first, p_second) returns(float bin(24));

     declare    p_first    float bin(24);
     declare    p_second   float bin(24);

     return((p_first + p_second) / 2);

end average;
```

In most cases, all arguments to a function are input arguments. If a function includes an output argument, the function is said to produce a *side effect*. Usually such functions should be rewritten as subroutines.

A function is not activated by a `call` statement. A function is activated whenever one of its entry points is referenced with an argument list. Such a reference is considered an expression and can appear anywhere that an expression of the same data type can appear.

The following example shows how the function `average` could be used.

```
a:   procedure;
declare    rate_1    float bin(24);
declare    rate_2    float bin(24);
declare    ave_rate  float bin(24);
          .
          .
          .
ave_rate = average(rate_1, rate_2);
          .
          .
          .
average: procedure(p_first, p_second) returns(float bin(24));

     declare    p_first   float bin(24);
     declare    p_second  float bin(24);

     return((p_first + p_second) / 2);

end average;

end a;
```

All entry points to a function need not specify the same returned data type. However, each `return` statement in the function must specify a returned value that can be converted to each of the data types specified in the `returns` options for all entry points.

Certain functions, called *built-in functions*, are predefined by the OpenVOS PL/I compiler. The built-in functions are described in Chapter 13.

For further information on referencing functions and built-in functions, see Chapter 8.

Begin Blocks

A *begin block* is a set of statements initiated by a `begin` statement and terminated by an `end` statement. A begin block is like a procedure **except** in the following respects.

- A begin block is activated whenever its `begin` statement is encountered in the flow of the program.
- A begin block cannot have parameters.
- Executing a `return` statement within a begin block returns control to the **caller** of the procedure that immediately contains the begin block.

Internally, begin blocks are named according to where they appear in the source module. Internal begin block names have the following form.

```
begin.line_number
```

The value of `line_number` is an integer indicating the source-module line number on which the `begin` statement appears.

When a begin block is activated, an associated stack frame is created. The following output from a debugger `trace` request shows a stack containing a stack frame associated with a begin block.

```
#  5:  begin.18 (line 20 in module a)
#  4:  c (line 18 in module a)
#  3:  b (line 11 in module a)
#  2:  a (line 4 in module a)
```

See the *VOS Symbolic Debugger User's Guide* (R308) for more information on the `trace` request.

The activation of a begin block is normally terminated when its `end` statement is executed or when control is transferred out of the block by a `goto` statement.

In the following example, activation of the begin block can end in any of three ways, depending on the value of `x`.

```
         begin;
            if x < 0
            then goto L2;      /* Transfer control out of begin block   */
              else if x > 10;
                 then return;  /* Return to caller of current procedure */
              .
              .
              .
         end;                  /* End of begin block                    */
              .                /* Continue with next statement          */
              .
              .
L2:
              .
              .
              .
```

A begin block is more limited than a procedure. However, because a begin block defines the scope of any declarations that it contains, you can use it to package declarations and executable statements that reference those declarations. You can use a begin block to redeclare a name that has already been declared in a containing block and to cause allocation and freeing of automatic variables declared within the block. The following example illustrates this usage.

```
     declare   index     fixed bin(15);
               .
               .
               .
          begin;
               declare   index     file;
               declare   x(1000)   float bin(24);
                    .
                    .
                    .
          end;      /* End of begin block */
```

In the preceding example, the name `index` is redeclared within the begin block as a file constant. The `index` variable declared in the outer block cannot be accessed within the scope of the begin block. A large array, `x`, is also declared within the begin block. This declaration saves space because storage for the array is allocated only while the begin block is executing.

See Chapter 12 for a full description of the `begin` statement.

Chapter 4: Data Types

This chapter discusses the following topics related to data types.

Overview

Every PL/I value has a data type. The *data type* of a value determines the operations that can be performed on it. The data type also affects how the value is stored internally. Conversely, you can think of a data type as a description of how an area of storage is to be interpreted.

Each data type is described by a list of attributes. The following attributes describe PL/I arithmetic data types.

`fixed binary` (or `fixed bin`)
`fixed decimal` (or `fixed dec`)
`float binary` (or `float bin`)
`float decimal` (or `float dec`)

PL/I supports two kinds of string data: character-string data and bit-string data. The following attributes describe character-string data types.

`character` (or `char`)
`character aligned` (or `char aligned`)
`character varying` (or `char varying`)

The following attributes describe bit-string data types.

```
bit
bit aligned
```

The `picture` attribute describes data stored as character strings but operated on as arithmetic values.

The following attributes describe other data types.

```
pointer
label
entry
file
```

Objects having elementary data types are called *scalars*. Scalar objects can be combined to form arrays or structures.

Table 4-7, at the end of this chapter, shows how OpenVOS PL/I data types correspond to data types in other OpenVOS languages. Appendix B discusses the alignment and storage requirements for each OpenVOS PL/I data type.

Arithmetic Data

In PL/I, three characteristics describe an arithmetic value.

- *Base* indicates whether a value is binary or decimal.
- *Scale* indicates whether a value is fixed-point or floating-point.
- *Precision* indicates the number of digits in the value.

These properties collectively constitute a value's data type. Arithmetic data types are most commonly expressed in the following format.

```
scale base (precision)
```

The following examples illustrate arithmetic data types.

```
fixed bin(15)
fixed dec(7,2)
float bin(24)
float dec(9)
```

The base describes the units in which the precision is measured. A variable declared to be `bin(15)` can hold values having up to 15 base-2 digits; a variable declared to be `dec(15)` can hold values having up to 15 base-10 digits. In OpenVOS PL/I, all arithmetic variables are stored in binary format regardless of the declared base.

All arithmetic values are converted to and from base-10 values on input and output. Arithmetic constants are also written in base 10.

The next two sections discuss the following topics.

- "Fixed-Point Data"
- "Floating-Point Data"

Fixed-Point Data

Fixed-point numbers can be either binary-based or decimal-based. Fixed-point binary numbers are always integers. Fixed-point decimal numbers can have fractional parts.

This section discusses the following topics.

- "Fixed-Point Binary Data"
- "Fixed-Point Decimal Data"
- "Fixed-Point Operations"

Fixed-Point Binary Data

Fixed-point binary values cannot have fractional parts; they are always integers.

The maximum precision of a fixed-point binary number is 31 or 63, depending on the value you specify in the `-max_fixed_bin` command-line argument or the `%options max_fixed_bin` preprocessor statement. The value you specify sets the value of the PL/I preprocessor symbol `$MAX_FIXED_BIN` to 31 or 63. If you specify neither the `-max_fixed_bin` argument nor the `max_fixed_bin` preprocessor option, the maximum precision (and therefore, the value of `$MAX_FIXED_BIN)` is 31. If you specify both the `-max_fixed_bin` argument and the `max_fixed_bin` preprocessor option, the option overrides the command-line argument.

The default precision is 15. If the precision is 15 or less, the value requires 2 bytes of storage: 15 bits for the value and 1 bit for the sign. If the precision is 16 to 31, the value requires 4 bytes of storage: 31 bits for the value and 1 bit for the sign. If the precision is 32 to 63, the value requires 8 bytes of storage: 63 bits for the value and 1 bit for the sign.

The following examples show declarations of fixed-point binary variables.

```
declare   x    fixed bin(15);
declare   y    fixed bin(31);
declare   z    fixed bin(63);
```

In the preceding examples, the variable `x` is a fixed-point binary integer with 15 bits of precision and 1 sign bit. Therefore, `x` can hold any integer value in the range $-(2^{15} - 1)$ to $(2^{15} - 1)$, or -32,767 to 32,767. The variable `y` is a fixed-point binary integer with 31 bits of precision and 1 sign bit. Therefore, `y` can hold any integer value in the range $-(2^{31} - 1)$ to $(2^{31} - 1)$, or -2,147,483,647 to 2,147,483,647. The variable `z` is a fixed-point binary integer with 63 bits of precision and 1 sign bit. Therefore, `z` can hold any integer value in the range $-(2^{63} - 1)$ to $(2^{63} - 1)$, or -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807.

The precision can follow either the `fixed` or the `binary` attribute. For example, the following declarations are equivalent and valid.

```
declare   y    fixed bin(15);
declare   z    fixed (31) bin;
```

Fixed-Point Decimal Data

Some or all of the digits in a fixed-point decimal value can be fractional digits. Therefore, the precision of a fixed-point decimal value has two parts: the total number of digits in the value and the number of those digits that appear to the right of the decimal point. The latter is called the *scaling factor* of the value.

A fixed-point decimal data type has the following syntax.

```
fixed dec( number_of_digits [,scaling_factor] )
```

The precision can follow either the `fixed` or `decimal` attribute. For example, the following declarations are equivalent and valid.

```
declare   y    fixed dec(7,2);
declare   z    fixed (7,2) dec;
```

The value of *`number_of_digits`* represents the total number of digits, integral and fractional, in the value. The value of *`scaling_factor`* is the number of digits that appear to the right of the decimal point.

The maximum number of digits allowed for a fixed-point decimal value is 18. The scaling factor can range from -18 to 18. The default precision is nine digits with a scaling factor of zero. Whenever you do not specify a scaling factor, it defaults to zero.

If the value contains nine digits or fewer, it is stored in four bytes. If the value contains more than nine digits, it is stored in eight bytes. The scaling factor has no effect on the storage size.

The following examples show declarations of fixed-point decimal variables.

```
declare   a    fixed dec(9);
declare   b    fixed dec(7,2);
declare   c    fixed dec(3,18);
declare   d    fixed dec(5,-13);
```

In the preceding examples:

- The variable `a` is a fixed-point decimal number of up to nine digits. Because no scaling factor is provided, the value is an integer. Therefore, `a` can hold any integer value in the range -999,999,999 to 999,999,999.

- The variable `b` is a fixed-point decimal number of up to seven digits, two of which appear to the right of the decimal point. Therefore, `b` can hold any multiple of 0.01 in the range -99,999.99 to 99,999.99.

- The variable `c` is a fixed-point decimal number with three significant digits. The scaling factor is 18. Therefore, `c` can hold any multiple of 0.000000000000000001 (that is,

10^{**-18}) in the range from -0.000000000000000999 to 0.000000000000000999 (that is, -999 * 10^{-18} to 999 * 10^{-18}).

- The variable `d` is a fixed-point decimal number with five significant digits. The scaling factor is -13. Therefore, `d` can hold any multiple of 10,000,000,000,000 in the range -999,990,000,000,000,000 to 999,990,000,000,000,000.

Fixed-point constants with very large or very small magnitudes can be difficult to read and to write accurately. To simplify their presentation, you can use an exponential format.

A constant in exponential format consists of the significant digits of the value (optionally with a preceding sign) and a base-10 exponent (optionally with a preceding sign) providing the scaling factor of the value. The significant digits and the exponents are separated by the upper- or lowercase letter f. The following table provides some examples of this format.

Standard Format	**Exponential Format**
`0.000000000000000999`	`999F-18` or `9.99F-16`
`-0.000000000000000999`	`-999F-18` or `-9.99F-16`
`0.000000000000000001`	`001F-18` or `1F-18`
`999990000000000000`	`99999F+13` or `9.9999F+17`
`-999990000000000000`	`-99999F+13` or `-9.9999F+17`
`100000000000000000`	`10000F+13` or `1F+17`
`-100000000000000000`	`-10000F+13` or `-1F+17`

A plus sign (+) preceding a positive exponent is optional.

Fixed-Point Operations

Fixed-point operations are addition, subtraction, multiplication, division, exponentiation, and comparison. Fixed-point binary division is performed with the `divide` built-in function.

You should use fixed-point binary variables rather than decimal variables whenever possible. For example, using fixed-point binary variables as subscripts, string lengths, and do-loop indexes significantly increases efficiency. Fixed decimal values are normally used in only the following two cases.

- where fractional digits are required
- to store integers with magnitudes of 2^{31} or greater, when you have not specified a maximum precision of 63 for fixed binary values, using either the `-max_fixed_bin` command-line argument or the `%options max_fixed_bin` preprocessor statement—see “Fixed-Point Binary Data” for more information

Except for division and multiplication, operations on fixed-point data align the decimal points and produce results that retain all fractional digits and all integral digits. If possible, these results are within the maximum precision for the declared base: up to 31 or 63[1] binary digits or 18 decimal digits. If the result does not fit within the maximum allowed precision, all fractional digits are saved and integral digits are dropped.

> **Note:** Depending on the context, the resultant value might be converted; data-type conversions are explained in Chapter 5.

See Chapter 9 for the rules that determine expression results.

The following example illustrates some fixed-point operations.

```
declare    a    fixed dec(8,3);
declare    b    fixed dec(2);
declare    c    fixed dec(1,2);

     a = 892.571;
     b = 18;

     a = a + b;     /* a = 910.571   */
     a = a - b;     /* a = 892.571   */
     a = a * b;     /* a = 16066.278 */

     c = 0.01;

     a = a + c;     /* a = 16066.288 */
     a = a - c;     /* a = 16066.278 */
     a = a * c;     /* a = 160.662   */
```

In the preceding example, note that the last result was truncated so that it would fit in the target variable.

If aligning the decimal points requires a scaling operation of greater than 18 decimal places, the `fixedoverflow` condition is signaled. For example, you cannot add `99999F+12` and `612F-09`. Also, if you attempt to calculate a fixed-point value that is too large for the declared base, the `fixedoverflow` condition might be signaled. See Chapter 15 for information on the `fixedoverflow` condition.

---

[1] The maximum precision for the declared base depends on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" for more information.

The result produced by the division operator has a precision that provides for the maximum possible integral quotient plus as many fractional digits as possible within the maximum precision for the declared base. The following example illustrates fixed-point division.

```
declare    a     fixed dec(8,3);
declare    b     fixed dec(2);
declare    c     fixed dec(1,2);

     a = 892.571;
     b = 18;
     c = 0.01;

     a = a / b;   /* a = 49.587    */
     a = a / c;   /* a = 4958.700  */
     a = b / c;   /* a = 1800.000  */
```

Using the division operator (`/`) with fixed-point binary operands always produces an integer result. The compiler detects such usage and issues a warning message. To suppress the warning, use the `divide` built-in function described in Chapter 13, or use (or convert to) floating-point or fixed-point decimal data. Chapter 5 discusses data-type conversion.

If the result of an operation involving a fixed-point binary value contains fractional digits, the result is converted to a fixed-point decimal value. If the compiler detects such a situation, it issues a warning message.

Internally, the number of digits stored for a fixed-point value at any time can exceed the value's stated precision. However, a program that relies on values that exceed the stated precision is invalid; the results of such a program are unpredictable.

If a fixed-point value is converted to a character-string or bit-string value, the length of the string is determined by the stated precision of the value, **not** by the actual current value. See Chapter 5 for a discussion of conversion rules.

Fixed-point values are never rounded unless a built-in function for that specific purpose is used. The `ceil`, `floor`, `trunc`, and `round` built-in functions perform different kinds of roundings; these functions are described in Chapter 13. When a value is assigned to a fixed-point variable, excess low-order fractional digits are truncated.

Floating-Point Data

Floating-point data represents real numbers. A floating-point value consists of a mantissa and an exponent. The *mantissa* is a number that can include a fractional part. The exponent scales the mantissa.

This section discusses the following topics.

- "Floating-Point Values"
- "Floating-Point Operations"

Floating-Point Values

A floating-point value has the following format.

$$\begin{bmatrix} + \\ - \end{bmatrix} \mathtt{\mathit{mantissa}} \begin{Bmatrix} \mathtt{E} \\ \mathtt{e} \end{Bmatrix} \begin{bmatrix} + \\ - \end{bmatrix} \mathtt{\mathit{exponent}}$$

The leading sign applies to the mantissa and determines the sign of the value. The sign immediately preceding the exponent applies to the exponent. The lowercase letter `e` or the uppercase letter `E` separates the exponent from the mantissa.

Typically, the mantissa of a floating-point value has one nonzero integral digit; the exponent provides proper scaling. The value of a floating-point number has the following appearance.

```
mantissa * (10 ** exponent)
```

For example, the following is a floating-point constant.

```
-3.450717E+05
```

In the preceding example, the mantissa is `-3.450717` and the exponent is `+05`. The value is approximately equal to `-345071.7`.

Floating-point values, like fixed-point values, are described by their scale, base, and precision. The following example shows how floating-point variables are declared and referenced.

```
declare   x    float bin(24);
declare   y    float dec(15);

     x = 4.789018E-12;
     y = -8.82904484938376E+56;
```

The precision of a floating-point value is the number of binary or decimal digits the mantissa can hold. The mantissa of a variable declared `float dec(15)` can hold 15 decimal digits; the mantissa of a `float bin(24)` variable can hold 24 binary digits, which corresponds to approximately 7 decimal digits.

The precision can follow the `float` attribute, or it can follow the `binary` or `decimal` attribute, as shown in the following declarations.

```
declare   y    float bin(24);
declare   z    float (24) dec;
```

The following table lists the most common floating-point precisions and the approximate range of values allowed for each.

Data Type	Approximate Range of Values
`float bin(24)`	10^{-38} to 10^{37}
`float bin(53)`	10^{-308} to 10^{307}
`float dec(7)`	10^{-38} to 10^{37}
`float dec(15)`	10^{-308} to 10^{307}

In OpenVOS PL/I, the base of a floating-point number serves only to qualify the precision. A floating-point decimal value with a precision of 7 is internally equivalent to a floating-point binary value with a precision of 24. These values require 4 bytes of storage. Likewise, a floating-point decimal value with a precision of 15 is internally equivalent to a floating-point binary value with a precision of 53 and requires 8 bytes of storage. See Appendix B for information on how data is stored internally.

Floating-point values are approximations only. For example, in the following fragment, the value `0.1` is assigned to a floating-point variable.

```
declare   x    float dec(1);

     x = 1E-01;
```

In the preceding example, the value actually stored for `x` could be anywhere in the range `0.05` to `0.15`. When rounded to a precision of one, the value is as expected; at greater precisions, the results may differ slightly.

Use the following formula to determine the maximum error for a floating-point value *x* .

$$x \,/\, b^{(p\,-\,1)} \,/\, 2$$

In the preceding formula, the value of *b* is 2 for binary data and 10 for decimal data; *p* is the value's precision (not the target's precision).

The following table shows the range of stored values for several floating-point types.

Data Type	Assigned Value	Range of Stored Value
`float dec(1)`	`0.1`	`0.05` to `0.15`
`float dec(5)`	`0.1`	`0.099995` to `0.100005`
`float bin(2)`	`0.1`	`0.075` to `0.125`
`float bin(24)`	`0.1`	`0.099999994` to `0.100000006`

Floating-Point Operations

Floating-point operations are addition, subtraction, multiplication, division, and exponentiation.

If a floating-point variable of precision *p* is assigned a value with a mantissa of fewer than *p* digits, the stored value of the mantissa is right-padded with zero bits to a precision of *p*. If the variable is assigned a value with a mantissa of greater than *p* digits, the stored mantissa value is rounded to *p* digits. Because floating-point values are only approximate, the result of this padding or rounding might not be exactly as you expect.

The following example illustrates how a floating-point mantissa can be padded or rounded.

```
declare   x    float dec(7);
declare   y    float bin(53);

     x = 1.2984987839E+20; /* Rounded to approximately
                              1.298499E+20     */
     y = 0.1E-11;          /* Expanded to approximately
                              0.09999999960042E-11 */
```

Although floating-point values are approximations, any integer value that is converted to a floating-point value and then back to an integer retains its original value, provided the integer's precision does not exceed that of the floating-point value. Likewise, floating-point addition, subtraction, and multiplication return integer results if both operands are integer values, provided the integer's precision does not exceed that of the floating-point value.

If the exponent of the result of a floating-point operation is too large for the allocated storage, the `overflow` condition is signaled. If you return to the point of the overflow, execution continues with the result of positive or negative infinity.

If the exponent of a floating-point result is too small, the `underflow` condition is signaled. All models provide *gentle underflow*. This means that when control returns from an `underflow` on-unit to the computation, it uses a denormalized, near-zero value. Stratus machines support denormalized values; when they return to the computation, they use the denormalized value.

If you divide a number by zero, the `zerodivide` condition is signaled. If the operation involves a floating-point operand, the dividend is not zero; you can return from the `zerodivide` on-unit and the result is a floating-point value representing positive or negative infinity.

> **Note:** Program continuation after the `overflow` or `zerodivide` condition occurs is an OpenVOS extension.

If the result of an operation involving infinity is itself infinity, no condition is signaled and the result is infinity with the proper sign. For example, adding positive infinity to itself produces positive infinity; dividing negative infinity by zero produces negative infinity.

If the result of an operation involving one or more infinite operands is undefined, the `error` condition is signaled. For example, subtracting infinity from itself or multiplying infinity by zero signals the `error` condition.

See Chapter 15 for information on the `overflow`, `underflow`, `zerodivide`, and `error` conditions.

Character-String Data

A character-string value is a sequence of ASCII characters. The number of characters in the string is the *length* of the string. The maximum length of an OpenVOS PL/I character string is 32,767.

A character string with zero length is the *null character string*.

The next three sections discuss the following topics.

- "Character-String Attributes"
- "Character-String Constants"
- "Character-String Operations"

Character-String Attributes

The description of a character-string value includes the `character` attribute and the maximum length, or *extent*, of the string. Optionally, you can specify the `varying` or `aligned` attribute.

```
char(extent) [ varying ]
             [ aligned ]
```

The expression *`extent`* must produce an integer value in the range 0 to 32,767, inclusive. The extent value is the maximum length of all string values that can be held by the variable.

If you specify the `varying` attribute, the values can be of any length from 0 to *`extent`*. The current length of the string is retained as part of the variable value.

If you do not specify the `varying` attribute, the length of the value is always the maximum length. If a shorter value is assigned to the string, the value is padded on the right with space characters to make it the appropriate length. If a string of length greater than *`extent`* is assigned to either a varying or nonvarying character-string variable, only the leftmost *`extent`* characters are assigned. Excess characters are truncated.

The `aligned` attribute forces the character string to be aligned in storage. *Alignment* refers to the allocation of data on a particular storage boundary.

If you specify the `aligned` attribute and **shortmap** rules are in effect, the number of bytes required to store the value must be a multiple of **two** (that is, an even number of bytes). If you specify the `aligned` attribute and **longmap** rules are in effect, the number of bytes required to store the value must be a multiple of **four**. Therefore, a character string of length *`n`* might occupy more than *`n`* bytes. However, this does not increase the maximum size of values that can be stored in the variable.

Varying-length character strings are always mod2-aligned and always occupy an even number of bytes. By default, nonvarying character strings are byte-aligned.

The `aligned` attribute has no effect on operations performed on the variable, but it is considered a part of the data type for purposes of sharing storage or matching arguments with parameters. The `aligned` attribute affects performance, since alignment, as mentioned earlier, makes access to the value more efficient.

> **Note:** The storage size of a character string described with the `varying` or `aligned` attribute is also subject to the mapping rules described in "Data Alignment" later in this chapter.

For information about how values are stored, see Appendix B. See Chapter 6 for a discussion of storage sharing. See Chapter 3 for a discussion of arguments and parameters.

Character-String Constants

A character-string constant begins and ends with an apostrophe. Any ASCII character other than an apostrophe can appear between the apostrophes. The following example illustrates.

```
'Any ASCII characters except the apostrophe.'
```

If an apostrophe is required within a character string, type two apostrophes. The following examples illustrate.

```
'I don''t know'
'He said, "I don''t know."'
''
```

The last of the preceding examples is the null string. Note that the quotation-mark character (`"`) in the second example has no special meaning.

The maximum length of a character-string constant representation is 2048 characters, including the enclosing apostrophes. Therefore, the value of a character-string constant cannot exceed 2046 characters.

Each double internal apostrophe is stored as one apostrophe and counts as only one character in the string length. For example, the length of the value `'don''t'` is five. However, because you must type two apostrophes for each apostrophe in the string, the maximum length allowed for the constant value is reduced. For example, if a character-string constant contains three internal apostrophes, the maximum length of a string value represented by that constant is 2043.

If you type a character-string constant on two or more lines in the source module, the ASCII carriage return character is inserted into the string at the point where the line breaks. The carriage return character is printed as `` `0D `` on output.

For example, the following statement might appear in a program.

```
the_string = 'The first line of the string
               the second line.'
```

In the preceding example, `the_string` is set to the following value.

```
The first line of the string`0D                  the second line.
```

If one or more space characters were typed after the word `string`, those space characters would be included in the value before the carriage return character. If the second line of the string were left-justified, no space characters would appear between the carriage return character and the word `the`. Note that although three character positions are required to print the carriage return character, it is stored in one byte and counts as one character in the string length.

Character-String Operations

You can perform relational comparisons, concatenation, and several built-in functions on character strings. Relational operations (greater-than, less-than, and equal-to) and concatenation are discussed in Chapter 9. The built-in functions that can be used with character strings are described in Chapter 13.

Character strings are compared from left to right using the ASCII collating sequence (see Appendix D). Strings of unequal length are compared by treating the shorter string as though it were padded on the right with space characters.

The following table shows the result of comparing two character strings.

`a`	`b`	Result
`'abcde'`	`'abcde'`	`a = b`
`'abcde'`	`'bcdef'`	`a < b`
`'abcd'`	`'abc.de'`	`a > b`
`'abcd'`	`'1234'`	`a > b`
`'abcde'`	`'abc'`	`a > b`
`'abcd '`	`'abcd    '`	`a = b`

Concatenation enables you to join two character strings. The concatenate operator is `||` or `!!`.

If you compile with the `-system_programming` argument, the compiler issues a warning each time a character-string constant breaks over a line. If a character-string constant is too long to fit on one line, you can type it as two strings and join them with the concatenate operator. Using concatenation suppresses the compiler's warning. The following example demonstrates the use of the concatenate operator.

```
the_string = 'The first line of the string '||
'the second line.'
```

In the preceding example, `the_string` is set to the following value.

```
'The first line of the string the second line.'
```

See Chapter 9 for further information on the concatenate operator. See the *VOS PL/I User's Guide* (R145) for more information on the `-system_programming` argument of the `pl1` command.

Bit-String Data

A *bit string* is a sequence of bits. The *length* of a bit-string value is the number of bits in the sequence.

A bit string whose length is zero is the *null bit string*.

Bit-string values are compared from left to right, one bit at a time. If bit-string values with different lengths are being compared, the shorter operand is treated as if it were right-padded with enough zero bits to make the lengths equal.

The next four sections discuss the following topics.

- "Bit-String Attributes"
- "Bit-String Constants"
- "Boolean Values"
- "Bit-String Operations"

Bit-String Attributes

A bit-string variable is declared with the following attributes.

```
bit(extent)[aligned]
```

The expression *extent* must yield an integer value in the range 0 to 32,767, inclusive. This value specifies the maximum length of string values the variable can hold.

If a bit-string variable has an extent of *n* and a value of less than *n* bits is assigned to it, the value is right-padded with zero bits to make it *n* bits long. If a value of more than *n* bits is assigned, the value is truncated on the right to a length of *n*.

The `aligned` attribute forces the bit string to be aligned in storage. *Alignment* refers to the allocation of data on a particular storage boundary.

If shortmap rules are in effect, an aligned bit string is stored beginning on a boundary of mod2 bytes (mod16 bits); the size allocated for the string is a multiple of two (that is, an even number). If longmap rules are in effect, an aligned bit string is stored beginning on a boundary of mod4 bytes (mod32 bits); the size allocated for the string is a multiple of four. The bit-string value of length *n* might occupy more than *n* bits. This, however, does not increase the maximum size of values that can be stored in the variable. See "Data Alignment," later in this chapter, for additional information.

The `aligned` attribute does not affect operations performed on the variable, but the attribute is considered a part of the data type for purposes of sharing storage or matching arguments with parameters.

Unaligned bit-string values of extent *n* always occupy *n* bits. As elements of arrays or members of structures, they always begin on the next available bit regardless of byte or other storage address boundaries. Therefore, an array or structure of unaligned bit strings can be used to describe objects, such as system control tables and machine instructions, commonly used by systems programmers.

The compiler issues advice if you attempt to reference a bit string contained in a structure that is either based or a parameter and that contains only unaligned bit strings or unaligned character strings. To avoid this advice, if the bit string is a structure member, insert a dummy variable with the attributes `char(0) aligned` or `bit(0) aligned`. The dummy variable ensures that the bit string will begin on an even byte boundary if the program uses shortmap rules, or on a mod4 boundary if the program uses longmap rules, making your program more efficient. Note that you should insert the dummy variable **only** if the actual storage generation begins on the appropriate boundary.

See Appendix B for information on how values are stored. See Chapter 6 for a discussion of storage sharing. See Chapter 3 for a discussion of arguments and parameters.

Bit-String Constants

A bit-string constant is written as a string of the binary digits 0 and 1 enclosed in apostrophes, followed by the letter `b`.

The following examples are bit-string constants.

```
'1'b
'10110'b
'0'b
''b
```

The last of the preceding examples is the null bit string.

The maximum length of bit-string constant representation is 256, including the enclosing apostrophes and the letter `b`. This means the maximum length of a bit-string value that can be represented by a bit-string constant is 253 bits.

An alternative form of bit-string constant is a string of characters enclosed in apostrophes, followed by a `b` character and one of the integers `1`, `2`, `3`, or `4`. The integer specifies the number of bits each character of the constant represents. If the integer is `1`, the characters in the string must be binary digits: `0` or `1`. If the integer is `2`, the characters must be base-4 digits: `0`, `1`, `2`, or `3`. If the integer is `3`, the characters must be octal digits: `0` through `7`. If the integer is `4`, the characters must be hexadecimal digits: `0` through `9` and `a` through `f` or `A` through `F`.

The following examples show various bit-string constant formats.

```
'101001'b1      /* Equivalent to '101001'b             */
'3132'b2        /* Equivalent to '11011110'b           */
'647352'b3      /* Equivalent to '110100111011101010'b */
'c03'b4         /* Equivalent to '110000000011'b       */
```

All bit-string values are converted to binary format before being stored. Table 4-1 summarizes the allowable character values for the different forms of bit-string constants and provides their base-2 equivalents.

Table 4-1. Valid Characters for Bit-String Formats

Character	`b` or `b1`	`b2`	`b3`	`b4`
`0`	`1`	`00`	`000`	`0000`
`1`	Invalid	`01`	`001`	`0001`
`2`	”	`10`	`010`	`0010`
`3`	”	`11`	`011`	`0011`
`4`	”	Invalid	`100`	`0100`
`5`	”	”	`101`	`0101`
`6`	”	”	`110`	`0110`
`7`	”	”	`111`	`0111`
`8`	”	”	Invalid	`1000`
`9`	”	”	”	`1001`
`a`	”	”	”	`1010`
`b`	”	”	”	`1011`
`c`	”	”	”	`1100`
`d`	”	”	”	`1101`
`e`	”	”	”	`1110`
`f`	”	”	”	`1111`

Note: The length of a bit-string constant is determined after it is expanded to binary format. Remember that the maximum allowable length of a bit-string constant is 2045 bits.

Boolean Values

Boolean values are bit strings of length one. A zero bit, `'0'b`, indicates a false value; a one bit, `'1'b`, indicates a true value.

Bit-String Operations

You can perform relational comparisons, logical operations, concatenation, and several built-in functions on bit strings. Relational operations (greater-than, less-than, and equal-to), logical operations (AND, NOT, and inclusive OR), and concatenation are described in Chapter 9. The built-in functions that can be used with bit strings are described in Chapter 13.

Pictured Data

A value whose data type is determined by a `picture` attribute or by the `p` format in a format list is called a *pictured value*. Pictured values are stored as character strings but can be operated on as fixed-point decimal numbers. Pictured values can contain embedded characters such as the following:

- period (`.`)
- comma (`,`)
- dollar sign (`$`)
- credit symbol (`cr`)
- debit symbol (`db`)

The declaration of a pictured variable contains a symbolic description of how values are to be formatted. The following examples are pictured declarations.

```
declare   a    picture '$zz,zzzv.99cr';
declare   b    picture '$$,$$$v.99-';
declare   c    picture '$**,***v.999';
declare   d    picture '----9v.999';
declare   e    picture '9999v9999s';
```

If the value `1234.56` is assigned to each of these variables, the following values result.

```
a    '$ 1,234.56'
b    ' $1,234.56'
c    '$*1,234.560'
d    ' 1234.560'
e    '12345600+'
```

The next four sections discuss the following topics.

- "The Picture Characters"
- "Drifting Fields"
- "Pictured Values"
- "Operations on Pictured Data"

The Picture Characters

Table 4-2 explains the purpose of each picture character.

Table 4-2. Picture Characters

Picture Character	Description
`b`	Indicates that a space is to be inserted into the pictured value. The space is inserted only if preceded by a nonzero digit, or a `9`, `y`, `t`, `i`, `r`, or `v` picture character; otherwise, a zero-suppression character is inserted.
`cr`	Replaced by two spaces if the value is positive. The `cr` picture characters can occur only as a pair and must appear at the rightmost end of the picture.
`db`	Replaced by two spaces if the value is positive. The `db` picture characters can occur only as a pair and must appear at the rightmost end of the picture.
`i`	Acts like a `9` picture character on negative numbers and like a `t` picture character on positive numbers.
`r`	Acts like a `9` picture character on positive numbers and like a `t` picture character on negative numbers.
`s`	A single `s` in a picture causes the insertion of a `+` or `-` sign. If more than one `s` appears in a picture, the entire field, from the first `s` to the last `s`, is a drifting field, as described later in this section.
`t`	Stops zero-suppression and causes the insertion of a character representing a digit with an overpunched sign. See Table 4-3 for information on digits with overpunched signs.
`v`	Acts like a `9` picture character on negative numbers and like a `t` picture character on positive numbers.
`y`	Acts like a `9` picture character except that a zero digit is represented by a space character.
`z`	Causes zero-suppression: leading zeroes are replaced by a space character.
`9`	Stops zero-suppression and causes the insertion of a digit into the pictured value.
`, . /`	These characters are inserted literally into the pictured value only if preceded by a nonzero digit, or a `9`, `y`, `t`, `i`, `r`, or `v` picture character; otherwise, a zero-suppression character is inserted.
`*`	Causes zero-suppression: leading zeroes are replaced by an asterisk.
`-`	Operates exactly like an `s` picture character (single or multiple), but the `+` sign is indicated by a space character.
`+`	Operates exactly like an `s` picture character (single or multiple), but the `-` sign is indicated by a space character.
`$`	Operates exactly like an `s` picture character (single or multiple), but a `$` character is inserted instead of the `+` or `-` sign.

Any digit positions that appear to the right of the `v` picture character represent fractional digits. Any value assigned to a pictured value is first scaled so that its decimal point is aligned with the `v` picture character. If the picture does not contain a `v` picture character, the decimal point is assumed to be at the rightmost end of the picture.

A picture cannot contain more than one sign character unless all of the sign characters are part of a drifting field. The following are the sign characters.

```
s, +, -, cr, db
```

A picture cannot contain two dissimilar sign characters. A picture cannot contain more than one `cr` or `db` picture character.

The `t`, `i`, and `r` picture characters are called *overpunched-sign characters*. Although `t`, `i`, and `r` are not sign characters, they cannot appear in the same picture with a sign character. Furthermore, no more than one instance of an overpunched-sign character can appear in any picture. Table 4-3 shows the digits 0 through 9 with overpunched signs.

Table 4-3. Digits with Overpunched Signs

Sign	Digit	Digit with Overpunched Sign
+	1 2 3 4 5 6 7 8 9	A B C D E F G H I
-	1 2 3 4 5 6 7 8 9	J K L M N O P Q R

+
0
-
	0
	0
	0
		{
		{
		}

The following example illustrates the use of overpunched signs.

```
declare a1      picture 't9999999v.99';
declare a2      picture '999999t9v.99';

     a1 = 1234.56;
     put skip list(a1); /* {0001234.56 */
     a1 = -1234.56;
     put skip list(a1); /* }0001234.56 */
     a1 = 0;
     put skip list(a1); /* {0000000.00 */

     a2 = 1234.56;
     put skip list(a2); /* 000012C4.56 */
     a2 = -1234.56;
     put skip list(a2); /* 000012L4.56 */
     a2 = 0;
     put skip list(a2); /* 000000{0.00 */
```

Drifting Fields

If more than one instance of the `s`, `+`, `-`, or `$` picture character appears in a picture, the part of the picture beginning with the first such character and ending with the last such character is called a *drifting field*. The sign characters or dollar-sign characters within a drifting field are called *drifting characters*.

A drifting field forces the sign or dollar sign to appear immediately before the most significant digit of the value. The picture of a drifting field can contain only the following characters, in any order.

- optionally, a `v` character
- optionally, one or more `b`, period (`.`), comma (`,`), and slant (`/`) characters
- any number of `s`, `+`, `-`, and `$` characters

A drifting field cannot contain or be preceded by a `9`, `y`, `t`, `i`, `r`, `z`, or `*` picture character. A picture cannot contain more than one drifting field.

The number of digits represented by a drifting field is one less than the total number of drifting characters in the field. The digits are zero-suppressed, and the drifting character is inserted immediately before the most significant digit.

Any insertion characters (`b`, `.`, `,`, or `/`) in the drifting field are not printed unless they are preceded by a significant digit or a `v` picture character.

Pictured Values

Pictured values can be interpreted as fixed-point decimal values. Each `9`, `y`, `t`, `i`, `r`, `z`, or `*` picture character is considered to be a digit of precision. Within a drifting field, each `s`, `+`, `-`, or `$` picture character, except the first one, is considered to be a digit of precision.

```
declare   a    picture 'zzzv.zz';
declare   b    picture '---v.--';
```

In the preceding examples, the variable a has a precision of 5. The variable b has a precision of 4 (the first minus sign is not a digit of precision). Both variables have a scaling factor of 2.

If a value of zero is assigned to a pictured variable and the picture does not contain at least one 9, y, t, i, or r picture character, the entire pictured value is filled with zero-suppression characters. Suppression characters are spaces unless the picture contains the * picture character (in which case leading zeroes are replaced by asterisks).

If a nonzero value is assigned to a pictured variable with a v picture character followed by zero-suppression characters or by part of a drifting field, zero-suppression stops at the v.

The following example illustrates zero-suppression.

```
declare    a     picture 'zzzv.zz';
declare    b     picture '***v.**';

     a = .01;  /* a = '    .01' */
     b = .01;  /* b = '***.01' */
```

If zero were assigned to a and b, their values would be '   ', and '******', respectively.

Negative values cannot be assigned to a pictured variable unless the picture contains a cr, db, -, +, or s picture character.

Values assigned to pictured variables are aligned with the implicit decimal point and truncated if necessary, just as they would be if assigned to a fixed-point decimal variable with the same precision.

> **Note:** The implicit decimal point is the v picture character; if the picture does not contain a v, the implicit decimal point is at the right of the picture. Including a period character (.) within the picture has no effect on the position of the implicit decimal point.

If you attempt to access a pictured value that does not fit with the associated picture, the results are unpredictable. The valid built-in function allows you to test whether the current value of a pictured variable fits with the associated picture. The valid function is described in Chapter 13.

Operations on Pictured Data

If a pictured value is used in a context that expects either an arithmetic or relational operator, the pictured value is converted to a fixed-point decimal value, as discussed earlier in this section.

A pictured value is treated as a character string only in the following contexts.

- when it is assigned to a character-string variable
- when it is an operand of the concatenate operator
- when it is the operand of a built-in function that expects character-string operands

In all other contexts, pictured values are treated as fixed-point decimal values.

Pointer Data

A *pointer* is a variable that can hold a pointer value. A *pointer value* is an address of storage.

In the following example, a pointer is declared and, subsequently, assigned a pointer value.

```
declare   a(5)      fixed bin(15);
declare   p         pointer;
          .
          .
          .
     p = addr(a(k));
```

In the preceding example, the variable `p` is a pointer. Therefore, it can hold the storage address of any named object. The assignment statement uses the `addr` built-in function to supply a pointer value; in this case, the address of the array element `a(k)`. The pointer value is assigned to the variable `p`.

The next two sections discuss the following topics.

- "Pointer Operations"
- "Accessing Storage with Pointers"

Pointer Operations

You can perform the following operations on a pointer value.

- Assign it to a pointer variable.
- Compare it to another pointer value.
- Pass it as an argument.
- Return it from a function.
- Set it with an `allocate` statement.
- Use it in a pointer-qualified reference.

You cannot perform conversions on pointer values or transmit pointer values in stream I/O.

The OpenVOS PL/I compiler accepts the relational operators `<`, `<=`, `^<`, `>`, `>=`, and `^>` for comparisons on pointer operands, in addition to the `=` and `^=` equality operators. Relational comparisons are performed using unsigned arithmetic. See Chapter 9 for more information on relational operators. Note that this is a nonstandard extension.

You can use the `addr` built-in function to return the address of a named object.

You can use the `addrel` built-in function to increment or decrement a pointer value.

You can use the `null` built-in function to assign a *null* value to a pointer variable. The *null pointer value* is a unique value that does not address any storage; it indicates that a pointer variable does not currently address anything.

A pointer value cannot be represented by a constant. However, you can use the `null` built-in function in contexts where you might want a constant, such as in an `initial` attribute.

You can use the `rel` built-in function to return the absolute byte offset of a pointer value. You can use the `pointer` built-in function to return the pointer value associated with an absolute byte offset.

> **Note:** If you use the `rel` built-in function to convert pointers to `fixed bin(31)` values for use in comparisons, you should realize that `rel` returns a **signed** value. Since relational comparisons are performed using **unsigned** arithmetic, you should compare the result of `rel(`*`ptr`*`)` with a pointer value directly in order to help prevent unexpected behavior.

See Chapter 13 for information on the `addr`, `addrel`, `null`, `pointer`, and `rel` built-in functions.

Accessing Storage with Pointers

To access storage, you need, in addition to a pointer, a template describing the storage. You use based variables for this purpose, as shown in the following example.

```
declare   nump      pointer;
declare   num       fixed bin(15) based;
          .
          .
          .
/* Allocate storage described by num, setting nump to that address */

     allocate num set(nump);

/* Assign a value to that newly allocated storage                  */

     nump -> num = 3;
```

In the preceding example, the based variable `num` acts as a template to describe the storage to which `nump` points. In the assignment statement, `nump` acts as a pointer qualifier of `num`. The locator qualifier symbol, `->`, is used to associate the pointer with the based variable.

The following example illustrates the use of pointers with based structures.

```
declare 1 structure1     ,
          2 first        fixed bin(15),
          2 second       char(3) varying,
          2 third        float dec(7);

declare 1 template       based,
          2 a            fixed bin(15),
          2 b            char(3) varying,
          2 c            float dec(7);

declare p                pointer;
     .
     .
     .
     p = addr(structure1);
     p->template.c = 3.2E-02;
```

In the preceding example, the first assignment statement returns the address of `structure1` and assigns it to the pointer `p`. The second assignment statement assigns the value `3.2E-02` to the storage addressed by `p` and described by `template.c`; that is, to `structure1.third`.

See Chapter 6 for information on based storage and how it is used. See Chapter 8 for an explanation of how to use pointers to qualify references.

You can use pointers to hold the address of dynamically allocated based storage. If the variable whose storage is addressed by a pointer is freed, the pointer value is no longer valid. The value of the pointer must be reset before it can be used again. Referencing the storage pointed to by an invalid pointer produces unpredictable results.

You cannot take the address of an unaligned bit string, or an array or structure consisting entirely of unaligned bit strings.

Label Data

A *label* identifies a PL/I statement. A *label prefix* on any statement **other** than a `procedure`, `entry`, or `format` statement is a declaration of a statement label.

The following example illustrates how to declare and reference a label.

```
a:      procedure;
                    .
                    .
                    .
        do while ('1'b);
                    .
                    .
                    .
             if x = 0
             then goto EXIT_LOOP;
                    .
                    .
                    .
        end;
EXIT_LOOP:
        x = 15;
                    .
                    .
                    .
end a;
```

In the preceding example, the name `EXIT_LOOP` is declared as a label in procedure `a` because the statement `x = 15;` uses it as a prefix. The `goto` statement in the example transfers control to the statement with the label `EXIT_LOOP` (in this case, an assignment statement).

See Chapter 7 for information on declaring statement labels.

The next four sections discuss the following topics.

- "Label Variables"
- "Label Operations"
- "Label Values"
- "Label Arrays"

Label Variables

You can use the `label` attribute to declare variables and function results whose values are labels. The following example demonstrates how to declare and reference a label variable.

```
a:   procedure;

declare   L1    label;
          .
          .
          .
     L1 = START;
START:
          .
          .
          .
     goto L1;
end a;
```

In the preceding example, the name `L1` is declared to be a statement label variable. In the assignment statement, the statement label value `START` is assigned to `L1`. The `goto` statement transfers control to the statement labeled `START`.

Label Operations

You can perform the following operations on a label value.

- Assign it to a label variable.
- Compare it for equality or inequality with another label value.
- Pass it as an argument.
- Return it from a function.
- Reference it in a `goto` statement.

You cannot perform calculations or conversions on labels, nor can labels be transmitted in stream I/O.

Label Values

A label value consists of two parts. One part designates a statement (more accurately, the instructions compiled for the statement), and the other designates the stack frame of the block activation that immediately contains that statement.

Figure 4-1 is a graphic representation of a label value.

Statement Address	Stack Frame Address

Figure 4-1. Format of a Label Value

In the example shown in "Label Variables," earlier in this chapter, the variable `L1` is assigned two addresses: one is a designator to the statement labeled `START`, and the other is a designator to the current stack frame of procedure `a`. A subsequent `goto` statement uses the stack frame designator only if the `goto` is executed in a different block activation.

The following example illustrates how the second part of a label value is used.

```
a:   procedure recursive;

declare   x    fixed bin(15) static initial(0);
declare   L1   label internal static;
          .
          .
          .
     if x = 0
     then do;
          L1 = RESTART; /* Assign label value in first activation */
          x = x + 1;
          goto L1;
     end;
          .
          .
          .
     call b;
     return;
RESTART:
     call a;
     return;

b: procedure;
          .
          .
          .
     goto L1;
end b;

end a;
```

In the preceding example, the first `goto` statement (within the do-group) transfers control to the statement labeled `RESTART`; the current block activation remains unchanged. The second `goto` statement also transfers control to the statement labeled `RESTART`; however, because `a` has been called recursively, two stack frames on the stack contain that statement. Because the label value (`RESTART`) was assigned in the first activation of `a`, the second part of the label value points to the stack frame associated with that initial activation of `a`. Therefore, the effect of the second `goto` statement is as follows:

1. The current activation of `b` terminates.
2. The stack frame associated with the second activation of `a` is popped.
3. Control returns to the statement labeled `RESTART` in the original activation of `a`.

A reference to a label variable that points to a popped stack frame produces unpredictable results.

Label Arrays

Statement labels can be subscripted by a single-digit integer constant. The constant can, optionally, be signed. (The label prefixes on `procedure`, `entry` and `format` statements **cannot** be subscripted.)

The following example shows how to use a label array.

```
          goto CASE(k);

CASE(1):
                .
                .
                .
          goto END_CASE;
CASE(2):
                .
                .
                .
          goto END_CASE;
CASE(3):
                .
                .
                .
          goto END_CASE;
CASE(6):
                .
                .
                .
END_CASE:
                .
                .
                .
```

The use of subscripted label prefixes, as illustrated in the preceding example, contextually declares an array of label **constants**. Such a label array cannot also be declared in a `declare` statement. A label array is always one-dimensional. In the preceding example, the label array has six elements. Elements `4` and `5` are undefined and cannot be used. Use of an undefined element generates unpredictable results.

You cannot reference an array of labels, such as `CASE` in the preceding example, as a whole array value. However, you can reference the elements of the array in any context that permits a statement label.

As a nonstandard extension, OpenVOS PL/I also accepts a default label for a label array. A default label has the following syntax.

```
label(*):
```

If you do not use a default label, any attempt to reference an undefined element of the label array is an error. If you do use a default label, control is transferred to the default label if the subscript is outside the bounds of the label array or references an undefined element within the array. In this way, a program still retains control even if the array is too small or undefined elements are referenced.

You can use the notation for a default label only as a statement label. You cannot use it in an expression context. To assign a label constant that references the same statement as the default label, you must use a nondefault label, as shown in the following example.

```
default_case:        /* Regular label               */
CASE(7):             /* Regular label array element */
CASE(*):             /* Default label               */
```

The following example shows how to use a default label.

```
     goto CASE(k);
CASE(1):
          .
          .
          .
     goto END_CASE;
CASE(2):
          .
          .
          .
     goto END_CASE;
CASE(*):             /* Default label */
          .
          .
          .
     goto END_CASE;
END_CASE:
          .
          .
          .
```

Do not confuse a label array with an array of label variables. You can declare an array of label variables explicitly, as shown in the following example.

```
declare   labs(4,3)      label;

     labs(3,2) = FIRST;
          .
          .
          .
FIRST:
          .
          .
          .
```

> **Note:** Whenever a label is declared explicitly in a `declare` statement (as in the preceding example), it is a **variable**. Label **constants** are always declared contextually. For further information on contextual and explicit label declarations, see Chapter 7.

Arrays are discussed later in this chapter.

Entry Data

A PL/I procedure is normally entered at its `procedure` statement. You can create additional entry points by using the `entry` statement. Every entry point referenced within a source module must be declared either explicitly or contextually, but not both.

The label prefix on a `procedure` statement is a contextual declaration of the primary entry point of a procedure. All entry points of procedures that are part of the current source module are contextually declared by appearing in `procedure` or `entry` statements and cannot be declared in `declare` statements.

You cannot reference an entry point of a procedure that is not part of the current source module unless you explicitly declare that entry point name with the `entry` attribute in a `declare` statement. If you fail to do so, the compiler issues an error message.

The following example shows how to declare entry points of procedures in other compiled modules.

```
declare   e  entry(fixed bin(15), pointer);
declare   f  entry((5) float dec(7)) returns(float dec(7));
```

In the preceding example, the name `e` is declared to be an entry point to a procedure. When the procedure is entered at `e`, two arguments are required: a 2-byte binary integer and a pointer value. The procedure can only be accessed by a `call` statement.

Also in the preceding example, the name `f` is declared to be an entry point to a function. When the function is entered at `f`, it requires an array of five `float dec(7)` values as its arguments and returns a single `float dec(7)` result.

The next three sections discuss the following topics.

- "Entry Variables"
- "Entry Operations"
- "Entry Values"

Entry Variables

You can use the `entry` attribute together with the `variable` attribute to declare variables whose values are entry points, as shown in the following example.

```
a:  procedure recursive;

declare   g    entry variable;
        .
        .
        .
     g = a;
        .
        .
        .
end a;
```

In the preceding example, the name `g` is declared to be an entry variable. As such, it can hold any entry point value. The assignment statement assigns to `g` the entry point name `a`. A subsequent call to `g` actually activates `a`.

The name `g` cannot be activated as a function or called with arguments. An attempt to use `g` in this way produces unpredictable results.

Any other attributes, including the `returns` attribute, that you specify in an entry variable declaration do not restrict the values that can be **assigned** to the variable. However, when you use the entry variable as the target of a `call` statement or in a function reference, the value that it currently holds must designate an entry point whose parameters and `returns` option match those specified in the declaration of the entry variable.

Entry Operations

You can perform the following operations on an entry value.

- Assign it to an entry variable.
- Compare it for equality or inequality with another entry value.
- Pass it as an argument.
- Return it from a function.
- Reference it in a `call` statement.
- Use it in a function reference.

You cannot perform calculations or conversions on entry values. Entry values cannot be transmitted in stream I/O.

Entry Values

In OpenVOS PL/I, an entry value consists of three parts. The *code pointer* points to an entry point of a procedure; the *display pointer* points to the stack frame of the block that immediately contains that procedure (that is, the stack frame from which the entry inherits automatic storage); the *static pointer* points, with a bias, to the static storage area for the program. Figure 4-2 is a graphic representation of an entry value.

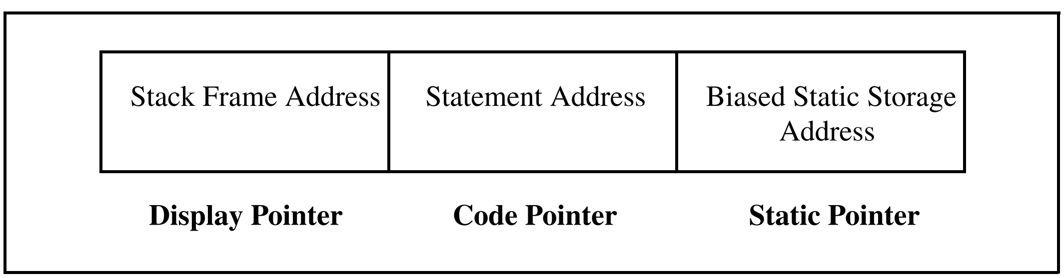

Figure 4-2. Format of an Entry Value

For the following discussion, only the display pointer and code pointer are significant.

If an entry value designates an entry point to an external procedure, the display pointer is null. Such an entry point is declared in an imaginary block that encompasses all external procedures in the compiled object module.

The display pointer of an entry value is significant only if you use entry variables, or if you use entry parameters and recursive procedures. The following paragraphs explain how the display pointer value is determined and used.

A procedure can reference automatic variables that are declared in an outer procedure. Because the variables are allocated in the outer procedure's stack frame, the inner procedure must be able to locate that stack frame.

Each stack frame for an internal block has a pointer to the stack frame of the containing block. That pointer value is supplied by the display pointer of the entry value used to activate the inner block. Unless the inner block is activated by a call involving an entry variable or an entry parameter, and unless recursion has been used, the display pointer indicates the one stack frame associated with the procedure that contains the newly activated procedure.

If the outer procedure has been activated recursively, more than one stack frame for that procedure exists (one for each activation). The display pointer of the entry value determines which stack frame is used when the inner procedure references an automatic variable declared in the outer procedure.

The following example illustrates the use of entry values in a recursive procedure.

```
a: procedure recursive;
%replace  TRUE           by '1'b;
%replace  FALSE          by '0'b;
declare   first_time     bit(1) aligned static initial(TRUE);
declare   x              fixed bin(15) automatic;
declare   e              entry variable static;
          .
          .
          .
     if first_time
     then e = b;
     else call f;
     call g;
          .
          .
          .
g: procedure;
     first_time = FALSE;
     call a;
        .
        .
        .
end g;

f: procedure;
     call e;
end f;

b: procedure;
       .
       .
       .
     x = 5;
     return;
end b;

end a;
```

In the preceding example, procedure `a` includes a call to procedure `g`. Procedure `g` calls procedure `a`. The second activation of procedure `a` then calls procedure `f` which, in turn, calls procedure `e`. Because procedure `b` has been assigned to procedure `e`, a call to `e` activates `b`.

Procedure `b` references the automatic variable `x`, which is declared in `a`. Because `a` has been activated twice, two existing stack frames contain an allocation of `x`.

The value `b` was assigned to `e` in the first activation of `a`. Therefore, the display pointer of the value of `e` designates the stack frame associated with that first activation of `a`. Consequently, when procedure `b` references an automatic variable declared in `a`, `b` refers to that first stack frame.

If procedure `b` is called directly from `a`, `b` uses the stack frame associated with the activation from which the call was made when referencing automatic variables declared in `a`. An older stack frame is used only when an entry name is assigned to an entry variable or passed as an argument to an entry parameter. When the entry name is thus passed or assigned, the stack frame associated with the current activation of the current block is assigned as the display pointer of the entry value.

File Data

A *file value* is the address of a file control block. A *file control block* is a 440-byte storage area used internally to associate an OpenVOS PL/I file with an OpenVOS port. See Chapter 14 for a discussion of I/O in PL/I.

A *file constant* is a named nonmember PL/I object declared with the `file` attribute and without the `variable` attribute. Every file constant has an associated file control block. A file constant is declared as shown in the following example.

```
declare   f    file;

     open file(f);
```

The reference to `f` in the `open` statement refers to the file control block associated with `f`.

The name of the file constant associated with a file control block is the *file ID* of that file control block.

In OpenVOS PL/I, all file constants have external scope. This means that each declaration of a file constant applies to all blocks in all source modules of a program. In a tasking environment, file constants are shared among all tasks, with the exception of the six file constants listed in the following table. The table also lists the predefined I/O port with which each of these file constants is associated.

File Constant	Corresponding Predefined I/O Port
`default_input`	`default_input`
`default_output`	`default_output`
`command_input`	`command_input`
`terminal_output`	`terminal_output`
`sysin`	`default_input`
`sysprint`	`default_output`

Each task sees a separate instance of these six file constants. For further information on these special file constants, see "Terminal I/O through Predefined I/O Ports" in Chapter 14. For further information on tasking, see the *OpenVOS PL/I Transaction Processing Facility Reference Manual* (R015).

The next two sections discuss the following topics.

- "File Variables"
- "File Operations"

File Variables

You can declare a file variable by specifying both the `file` and `variable` attributes in a `declare` statement. You can assign a file variable the value of a file constant; that is, the address of a file control block.

```
declare    f          file;
declare    (g,h)      file variable;
                .
                .
                .
          g = f;
          h = g;
```

In the preceding example, the names `g` and `h` are declared to be file variables. In the first assignment statement, `g` is assigned the value of the file constant `f`. After the assignment, an operation on `g` is equivalent to an operation on `f` because both `g` and `f` designate the same file control block. In the second assignment statement, the value of `f` is assigned to `h`.

File Operations

You can perform the following operations on a file value.

- Assign it to a file variable.
- Compare it for equality or inequality with another file value.
- Pass it as an argument.
- Return it from a function.
- Reference it in the `file` clause of an I/O statement.

You cannot perform calculations or conversions on file values. You cannot transmit file values in stream I/O.

Arrays

An *array* is an ordered set of objects having the same data type. Each individual value in the array is called an *element*.

You declare arrays with the `dimension` attribute, as shown in the following example.

```
declare    a     dimension(1:4) fixed bin(15);
```

If you specify the `dimension` attribute immediately after the variable name, you can omit the word `dimension`, as shown in the following example.

```
declare    a(1:4)     fixed bin(15);
```

The next three sections discuss the following topics.

- "Dimensions and Bounds"
- "Array Operations"
- "Array Elements"

Dimensions and Bounds

An array has from one to eight dimensions. Each dimension has a size. The number of elements in an array is the product of the sizes of its dimensions.

Each dimension has an integral upper bound and an integral lower bound. Each bound value must be in the following range.

-2,147,483,647 `<=` *`bound`* `<=` 2,147,483,647

The following restrictions also apply to the bounds of an array.

`lower_bound` `<=` *`upper_bound`* `<=` *`lower_bound`* + 2,147,483,646

The following formula calculates the size of a dimension.

`(`*`upper_bound`* - *`lower_bound`*`)` + 1

The following example declares a one-dimensional array.

```
declare   a(1:4)         fixed bin(15);
```

In the preceding example, the name `a` is declared to be a one-dimensional array. The lower bound is `1`, and the upper bound is `4`. The array contains four elements. Each element can hold a 2-byte fixed-point binary value. You can visualize the array `a` as a row of fixed-point values, as shown in Figure 4-3.

a(1)	a(2)	a(3)	a(4)

Figure 4-3. A One-Dimensional Array

The following example declares a two-dimensional array.

```
declare   b(-1:2,0:5)     float bin(53);
```

In the preceding example, the name `b` is declared to be a two-dimensional array. The first dimension has a lower bound of `-1` and an upper bound of `2`. The second dimension has a lower bound of `0` and an upper bound of `5`. The array contains 24 elements; each element can hold a floating-point binary value with a precision of 53.

You can visualize a two-dimensional array as a set of rows and columns. For example, `b` consists of four rows of six elements each, as shown in Figure 4-4.

`b(-1,0)`	`b(-1,1)`	`b(-1,2)`	`b(-1,3)`	`b(-1,4)`	`b(-1,5)`
`b(0,0)`	`b(0,1)`	`b(0,2)`	`b(0,3)`	`b(0,4)`	`b(0,5)`
`b(1,0)`	`b(1,1)`	`b(1,2)`	`b(1,3)`	`b(1,4)`	`b(1,5)`
`b(2,0)`	`b(2,1)`	`b(2,2)`	`b(2,3)`	`b(2,4)`	`b(2,5)`

Figure 4-4. A Two-Dimensional Array

The following example declares a three-dimensional array.

```
declare   c(25,4,2)      pointer;
```

In the preceding example, the name `c` is declared to be a three-dimensional array. Because no lower bounds are specified, they are assumed to be `1`. The array consists of 25 sets of four rows each, each row containing two elements, for a total of 200 elements. Each element can hold a pointer value.

The bounds of all dimensions of an array are collectively called the *extents* of the array. The allowable forms of array bounds in a declaration depend on the array's storage class. Storage classes are described in Chapter 6. The bounds of `static` arrays must be integer constants. The bounds of `automatic`, `defined`, and `based` arrays must be integer constants or integer-valued expressions. The bounds of parameter arrays must be either integer constants or an asterisk. If an asterisk is specified as the bounds of an array parameter, the bounds of the corresponding array argument are used as the bounds of the parameter, as shown in the following example.

```
declare   a(1:4)    fixed bin(15);
     call p(a);
        .
        .
        .
p: procedure(x);
declare   x(*)      fixed bin(15);
        .
        .
        .
end p;
```

In the preceding example, when the array `a` is passed by reference to procedure `p`, the parameter `x` is assigned the bounds of `a:(1:4)`. Any reference within `p` to `x` is a reference to `a`.

Array Operations

You can perform the following operations on an entire array.

- Assign scalar values to the array.
- Transmit the array in stream or record I/O.
- Pass the array by reference as an argument.
- Assign the array to another array of the same size and shape.

You cannot perform conversions or calculations on entire arrays. You cannot perform comparisons using relational operators, even for equality or inequality, on entire arrays. You can use references to individual scalar array elements in any context that permits a scalar variable reference.

In addition, you can use certain built-in functions on arrays. See Chapter 13 for a description of the built-in functions that can be used on arrays.

Array Elements

Array elements are referenced by their position within the array, using as many subscripts as the array has dimensions. The following example demonstrates how to reference array elements.

```
declare   c(25,4,2)      pointer;

     c(12,3,1) = null();
```

In the preceding example, element `c(12,3,1)` is assigned the null pointer.

See Chapter 8 for information on subscripted references.

A *subscript* must be an integer-valued expression. If you use either a fixed-point value with a fractional part or a floating-point value as a subscript, the compiler issues an error message. However, you can convert a nonintegral fixed-point value to an integer by using one of the following built-in functions: `trunc`, `ceil`, `floor`, or `round`. You can convert a floating-point value to an integer with the `fixed` built-in function. Built-in functions are described in Chapter 13.

Each subscript must lie within the range specified by the upper and lower bounds for the corresponding dimension. Unless you specify the `-check` compiler argument, the compiler does not produce code to check the range of subscript values. If you request checking, any subscript that falls outside the range is either diagnosed at compile time or signals the `error` condition at run time. See Chapter 15 for information on the `error` condition. See the *VOS PL/I User's Guide* (R145) for more information on compiler arguments.

Array elements are stored in *row-major order*. This means that if array elements are accessed in the order in which they are stored, the rightmost subscript varies most frequently and the leftmost subscript varies least frequently. For example, the first three elements of the array `c` from the preceding example are `c(1,1,1)`, `c(1,1,2)`, and `c(1,2,1)`. When accessing the elements of an array, it is most efficient to reference them in row-major order.

Structures

A *structure* is a hierarchically ordered set of values that can be of different data types. The immediate components of a structure are called *members* of the structure. A structure that is itself a member of another structure is called a *substructure*. A structure that is not a substructure is a called a *major structure*.

The next three sections discuss the following topics.

- "Level Numbers"
- "Structure Operations"
- "Structure Members"

Level Numbers

Each major structure and each member of a structure is declared with a *level number* to specify its place in the hierarchical organization. The following example illustrates the use of level numbers in a structure.

```
declare   1  s      ,
            2  a    fixed bin(15),
            2  b    float bin(24),
            2  t    ,
              3  p  pointer,
              3  q  char(10);
```

In the preceding example, `s` is a major, or level-one, structure. All major structures must have a level number of `1`. The members of `s` are `a`, `b`, and `t`. Member `t` is a substructure whose members are `p` and `q`.

You can visualize a structure organization as a tree. Figure 4-5 is a graphic representation of the structure `s` from the preceding example.

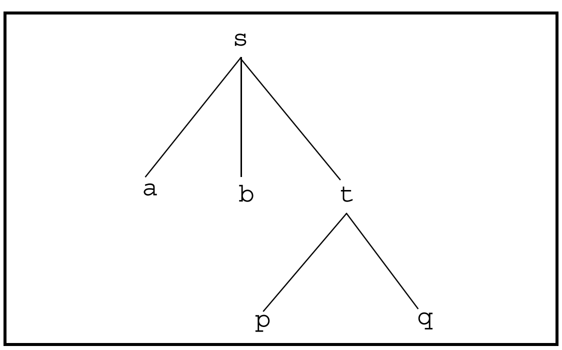

Figure 4-5. Tree Diagram of a Structure

A member can have any level number greater than the level number of the structure that contains it. Usually, each member is given a level number one greater than the level number of the containing structure.

The structure declared in the following example is equivalent to the structure declared in the previous example.

```
declare   1  s          ,
            20  a       fixed bin(15),
            20  b       float bin(24),
            20  t       ,
              30  p     pointer,
              30  q     char(10);
```

If a program contains two or more structures that contain a common construct, you can use the `like` attribute to simplify their declarations. The `like` attribute is described in Chapter 7.

Structure Operations

You can perform the following operations on an entire structure.

- Transmit it in stream or record I/O.
- Pass it as an argument.
- Assign it to another structure that has identical size and shape and corresponding members of identical data types.

You cannot perform conversions and calculations on entire structures. You cannot perform comparisons using relational operators, even for equality or inequality, on entire structures. References to individual scalar structure members can be used in any context that permits a scalar variable reference. You can use some built-in functions on structures. See Chapter 13 to determine which built-in functions can be used on structures.

Structure Members

You can reference structure members in any context that permits a reference to a variable of that data type. See Chapter 8 for information on how to reference structure members.

The name of a structure member must be unique within its immediately containing structure, but you can use the same name for a member of another structure or for a nonmember. You cannot reuse the names of major structures; only member's names can be reused.

All members of a structure, including any substructures, are considered to have the same storage class as the level-one structure. The storage class is specified for the level-one structure only. If no storage class is specified, the entire structure has the automatic storage class.

A structure declaration has a limit of 256 members per nesting level. For example, the following structure declaration is invalid because it contains 257 members at level two.

```
declare 1 s         ,
          2 member1      fixed bin(15),
          2 member2      fixed bin(15),
          2 member3      fixed bin(15),
               .
               .
               .
          2 member257    fixed bin(15); /* Too many members */
```

The preceding structure declaration causes a fatal error at compile time.

Storage classes are discussed in Chapter 6.

Arrays of Structures

Structures, like other variables, can be declared as arrays and can contain arrays as members. Structures declared as arrays are called *dimensioned structures*.

The following example shows a dimensioned structure declaration.

```
declare   1  s(5)        ,
             2  a        fixed bin(15),
             2  b(4)     float bin(24),
             2  t(3)     ,
                3  p     pointer,
                3  q(6)  char(10);
```

In the preceding example, `s` is an array that contains five elements. Each element of `s` is a structure. The third member of each of these structures is an array of three substructures. Each substructure contains a pointer variable and an array of 6 character-string variables of length 10.

The entire array, `s`, contains 5 occurrences of `a`, 20 occurrences of `b`, 15 occurrences of `t`, 15 occurrences of `p`, and 90 occurrences of `q`. Because each member occurs more than once, each member of an array of structures is also an array. For example, `a` is an array of five elements. If a member is an array in its own right (such as `b`, `t`, and `q` in the example), that array inherits the additional dimensions from the structures that contain it. For example, `b` is a two-dimensional array whose bounds are (5,4); `q` is a three-dimensional array whose bounds are (5,3,6).

You can use members of dimensioned structures only as arrays in stream I/O or as arguments to array parameters whose bounds have been specified as asterisks. They cannot be assigned, used in record I/O, used in the `addr` built-in function, used in the `defined` attribute, or passed to parameters with constant bounds.

You can use a subscripted reference to an individual member of a dimensioned structure in any context that permits a variable reference.

See Chapter 8 for information on referencing dimensioned structures and their members.

How the Compiler Determines Data Type

Every value referenced in a PL/I program must have a data type. This section discusses the following topics related to how data types are assigned to various values.

- "Variables and Function Results"
- "Expression Results"
- "Parameters"
- "Literal Constants"
- "Named Constants"
- "User-Defined Data Types"

Variables and Function Results

You must explicitly declare the data types of all function results. Generally, you must also explicitly declare the data types of all variables. If the data type of a name is not declared either explicitly or contextually, it is assumed to be a fixed-point 2-byte binary integer variable. Likewise, if a name is declared without a data type, it is assumed to be a fixed-point 2-byte binary integer.

See Chapter 7 for an explanation of how to specify the data types of variables and function results. Chapter 13 provides the data type returned by each built-in function.

Expression Results

The data type of the value produced by an expression depends upon the operators and operands used within the expression. Chapter 9 explains the type determination rules involved.

Parameters

All parameters to a procedure must be declared with a data type within that procedure. The data types of any parameters to a procedure in another compiled object module are described within the `entry` declaration for the entry point to that procedure.

Literal Constants

A *literal constant* is a constant whose value is derived from its form alone; literal constants are never declared. The data type of a literal constant is determined by the syntax of the constant value. For example, the following table shows some character-string constants and their data types.

Character-String Constant	Data Type
`'This is a character string.'`	`char(27)`
`'21 is a number.'`	`char(15)`
`''`	`char(0)`

The following table shows some bit-string constants and their data types.

Bit-String Constant	**Data Type**
`'1'b`	`bit(1)`
`'0101'b2`	`bit(8)`
`'011001'b3`	`bit(18)`
`'0111'b4`	`bit(16)`
`''b`	`bit(0)`

The following table shows some fixed-point arithmetic constants and their data types.

Fixed-Point Arithmetic Constant	**Data Type**
`25`	`fixed dec(2)`
`32700`	`fixed dec(5)`
`123456789`	`fixed dec(9)`
`4.75`	`fixed dec(3,2)`
`4100.01`	`fixed dec(6,2)`
`0`	`fixed dec(1)`
`0.`	`fixed dec(1,0)`
`9.834F+12`	`fixed dec(4,-9)`
`34894F-15`	`fixed dec(5,15)`

If a fixed-point constant contains a decimal point before its last digit, it is considered to be a scaled fixed-point value and is, therefore, stored and accessed like a noninteger fixed-point decimal variable. Fixed-point constants without a decimal point are integer constants and you can use them in operations with any other arithmetic data type—regardless of the other value's base and scale. For this reason, you should always write integer constants without a decimal point.

The following table shows some floating-point arithmetic constants and their data types.

Floating-Point Arithmetic Constant	**Data Type**
`5e+02`	`float dec(1)`
`4.5E1`	`float dec(2)`
`100e-04`	`float dec(3)`
`.001E-04`	`float dec(3)`
`0e0`	`float dec(1)`

The case of the letter `e` in a floating-point constant is insignificant.

Named Constants

A *named constant* is a file, label, or entry-point constant, or a format name. The data types of file constants and entry-point constants are explicitly declared.

User-Defined Data Types

The `type` attribute enables you to declare a user-defined data type by referencing the type of another member or structure. Data that is defined with the `type` attribute assumes the size and most characteristics of the referenced data. You can use the `type` attribute in any context in which data can be declared: variable declarations, parameter declarations, entry declarations, and the `returns` option of a `procedure` or `entry` statement. The `type` attribute is a nonstandard extension.

Consider the following declaration, which creates the type `integer`.

```
declare integer     based fixed bin(31);
```

In the following statement, the variables `a` and `b` are declared as the user-defined `integer` data type. They are processed as `fixed bin(31)`. The `based` attribute is a storage attribute; as discussed in the description of the `type` attribute in Chapter 7, storage attributes are not inherited.

```
declare (a,b)       type(integer);
```

In the following statement, the parameter `arg` is declared as the user-defined `integer` data type. It is considered `fixed bin(31)` and expects to be passed an argument that is declared as either the type `integer` or `fixed bin(31)`.

```
g: procedure(arg);
declare arg         type(integer);
```

In the next statement, the user-defined `integer` data type is referenced in the `returns` option of the `procedure` statement. The procedure `g` returns an item that is `fixed bin(31)`. The `type` attribute can also be used in the `returns` option of an `entry` statement.

```
g: procedure(arg)        returns(type(integer));
f: entry (arg1, arg2)    returns(type(integer));
```

In the following statement, the user-defined `integer` data type is referenced in the `returns` attribute of an `entry` declaration. The procedure `my_routine` returns an item that is `fixed bin(31)`.

```
declare my_routine entry returns(type(integer));
```

In the following statement, the procedure `my_routine` is explicitly declared with a parameter using the user-defined `integer` data type. When you call `my_routine`, you must pass it a parameter that is declared as either the type `integer` or `fixed bin(31)`.

```
declare my_routine entry (type(integer));
```

You can also use the `type` attribute in the declaration of an entry or file constant. In the following statement, the entry constant `e2` inherits the parameter and returns descriptors of the referenced item, `entry_template`.

```
declare entry_template   entry(fixed bin(15))
                         returns(fixed bin(15));
declare e2               type(entry_template);
```

If you declare a structure using the `type` attribute, that structure assumes the size and type name of each member of the referenced structure. In the following example, the structure `struct2` is declared as the user-defined type `struct_template`, indicating that it, too, is made up of two members, one `fixed bin(15)` element and one `char(30)` element. Again, the `based` attribute is not inherited.

```
declare 1 struct_template based,
          2 a             fixed bin(15),
          2 b             char(30);
declare struct2           type(struct_template);
```

See Chapter 7 for more information about the `type` attribute.

Data Alignment

OpenVOS PL/I allows you to specify one of two data alignment methods: shortmap alignment or longmap alignment. If you do not specify either alignment method, the alignment method is the system-wide default, which is site-settable by the system administrator. See the manual *OpenVOS System Administration: Administering and Customizing a System* (R281).

> **Note:** Do not confuse shortmap and longmap alignment with the `aligned` attribute. The `aligned` attribute refers only to the alignment of bit strings and character strings. See the description of the `aligned` attribute in Chapter 7 for more information.

For most programs, longmap alignment is available and is **much more efficient** than shortmap alignment. In general, the system-wide default should be set to longmap alignment if your system consists only of V Series modules.

If an existing data structure is defined using shortmap alignment rules, you will have to use shortmap alignment either implicitly or explicitly when accessing the data structure. In this case, longmap alignment could cause unpredictable results. For example, when certain structures are passed as arguments to some OpenVOS subroutines, shortmap alignment may be required for the structures. See the descriptions of specific subroutines in the *OpenVOS PL/I Subroutines Manual* (R005) for information on structures that require shortmap alignment.

You can specify an alignment method for an entire source module by using the `-mapping_rules` argument to the `pl1` command or by using the `%options` PL/I preprocessor statement. You can indicate the shortmap or longmap alignment rules in a specific declaration by using the `shortmap` or `longmap` attribute.

The compiler uses the following procedure to determine which alignment method to use.

1. If you specify the `shortmap` or `longmap` alignment attribute in a declaration, the compiler uses the specified alignment rules.

2. If you do **not** specify alignment rules by the preceding method, the compiler uses the alignment rules specified in the `%options` PL/I preprocessor statement.

3. If you do **not** specify alignment rules by either of the preceding methods, the compiler uses the alignment rules specified in the `pl1` command's `-mapping_rules` argument.

4. If you do **not** specify alignment rules by any of the preceding methods, the compiler uses the system-wide default alignment rules. The system-wide default alignment rules are site-settable by the system administrator.

Notice the order of precedence in the procedure the compiler uses to determine alignment rules. For example, the alignment indicated by the method in step 1 always overrides any alignment specifications indicated by the methods in steps 2 through 4.

The next four sections describe the following topics.

- "Specifying Alignment Rules for a Compilation Unit"
- "Specifying Alignment Rules for a Variable"
- "Shortmap Alignment Rules"
- "Longmap Alignment Rules"

Specifying Alignment Rules for a Compilation Unit

You can indicate the alignment rules (also called *mapping rules*) for a source module using either of the following methods.

- by using the `longmap` or `shortmap` argument to the `%options` PL/I preprocessor statement

- by specifying the `-mapping_rules` argument when you invoke the compiler

If you do not specify an alignment method for the source module using one of the preceding methods, the compiler uses the system-wide default alignment method.

If you use the `%options` statement or the `-mapping_rules` argument, the compiler uses the specified data alignment method when laying out storage for the source module. In addition, if you specify one of the `check` values available through both `%options` and `-mapping_rules`, the compiler diagnoses alignment padding in structures. Table 4-4 shows the available values.

Table 4-4. Values of the `-mapping_rules` Compiler Argument

Value	Description
`default`	Specifies the system-wide default alignment rules
`default/check`	Same as `default` except, in addition, the compiler diagnoses alignment padding between structure members
`shortmap`	Specifies the shortmap alignment rules
`shortmap/check`	Same as `shortmap` except, in addition, the compiler diagnoses alignment padding between structure members
`longmap`	Specifies the longmap alignment rules
`longmap/check`	Same as `longmap` except, in addition, the compiler diagnoses alignment padding between structure members

Note that the alignment values for the `%options` statement are the same as the values shown in Table 4-4 except that you use the `shortmap_check` and `longmap_check` values to check for alignment padding (that is, the `check` values contain underline characters instead of slant characters). Also, the `%options` statement has no equivalent to the `default` and `default/check` values.

Using an `%options` *`mapping_rule`* statement, such as `longmap`, to indicate an alignment method overrides the alignment method specified in the `-mapping_rules` compiler argument, but the compiler still diagnoses alignment padding within the structure if you have specified one of the `check` values in the `-mapping_rules` argument.

In addition to the `-mapping_rules` argument, you can use the `system_programming` option with the `%options` statement or the `-system_programming` compiler argument to diagnose alignment padding that appears in structures.

You must place the `%options` statement specifying alignment rules at the beginning of the source module **before** any data declarations or `procedure` statement. If you specify alignment rules in more than one `%options` statement, the compiler uses the last `%options` statement specified to determine the alignment method for the source module.

To specify shortmap alignment for a source module, do **one** of the following:

- Include either of the following statements in the source module.

  ```
  %options shortmap;
  %options shortmap_check;
  ```

- Compile the source module with **either** of the following arguments.

  ```
  -mapping_rules shortmap
  -mapping_rules shortmap/check
  ```

To specify longmap alignment for a source module, do **one** of the following:

- Include either of the following statements in the source module.

```
%options longmap;
%options longmap_check;
```

- Compile the source module with **either** of the following arguments.

```
-mapping_rules longmap
-mapping_rules longmap/check
```

Specifying Alignment Rules for a Variable

Both the `%options` statement and the `-mapping_rules` argument specify the alignment method for a compilation unit. To specify alignment for a specific variable declaration, use the `longmap` or `shortmap` attribute with that declaration. The `longmap` and `shortmap` attributes override the alignment specified for the rest of the compilation unit.

A structure using the `like` attribute inherits any alignment specifiers you have included with the original template, unless you specifically override them. Consider the following example.

```
declare 1 struct1   longmap,
          2 a       fixed bin(15),
          2 b       float dec(8),
          2 c       fixed bin(31);

declare 1 struct2 like struct1 shortmap;
```

In the preceding example, `struct1` is aligned under longmap rules. Since `struct2` is declared with the `like` attribute, it would also have longmap alignment, but that is overridden by the `shortmap` attribute included in the declaration.

User-defined data types, declared with the `type` attribute, inherit any alignment specifiers that are associated with the referenced variable; if you explicitly specify an alignment attribute that does not match that of the referenced variable, the compiler issues a warning.

In the following example, the user-defined type `TYPE1` is declared with longmap alignment. Variable `a`, which is of type `TYPE1`, also has longmap alignment, but variable `b` is declared with the `shortmap` attribute. The variable `b`'s declaration conflicts with the definition of `TYPE1`, so the compiler issues a warning.

```
declare TYPE1       fixed bin(15) longmap;

declare a           type(TYPE1);
declare b           type(TYPE1) shortmap; /* Alignment conflict */
```

If the typed entity (that is, the variable referenced by a variable declared with the `type` attribute) does not explicitly specify an alignment attribute, you can specify an alignment attribute for the corresponding variable. In this case, the variable's alignment attribute **overrides** the typed entity's default alignment. Since no conflict between the definitions exists, the compiler does not issue any warnings.

In the following example, the user-defined type `TYPE2` is declared without any explicit alignment attribute. Variable `c`, which is of type `TYPE2`, has longmap alignment, and variable `d`, which is also of type `TYPE2`, has shortmap alignment. The alignment attributes of variables `c` and `d` override the default alignment of `TYPE2`.

```
declare TYPE2        fixed bin(15);

declare c            type(TYPE2) longmap;
declare d            type(TYPE2) shortmap;
```

Shortmap Alignment Rules

When you use shortmap alignment, most nonstring data types are allocated so that they begin on an even-numbered byte boundary. With shortmap alignment, unaligned character data items are allocated so that they begin on a byte boundary. Unaligned bit strings are allocated so that they begin on the current bit. Table 4-5 shows the shortmap alignment rules.

Table 4-5. Shortmap Alignment Rules

Data Type	Restriction	Alignment	Size (in Bytes)
`fixed bin(p)`	*p* <= 15	mod2	2 bytes
`fixed bin(p)`	*p* > 15 but < 32	mod2	4 bytes
`fixed bin(p)`	*p* > 31 but < 64	mod2	8 bytes
`fixed dec(p,q)`	*p* <= 9	mod2	4 bytes
`fixed dec(p,q)`	*p* > 9	mod2	8 bytes
`float bin(p)`	*p* <= 24	mod2	4 bytes
`float bin(p)`	*p* > 24	mod2	8 bytes
`float dec(p)`	*p* <= 7	mod2	4 bytes
`float dec(p)`	*p* > 7	mod2	8 bytes
`char(n)`	N/A	Byte	*n* bytes
`char(n) aligned`	N/A	mod2	`2*ceil(n/2)` bytes
`char(n) varying`	N/A	mod2	`2*ceil(n/2) + 2` bytes
`bit(n)`	N/A	Bit	*n* bits
`bit(n) aligned`	N/A	mod2	`2*ceil(n/16)` bytes
`pointer`	N/A	mod2	4 bytes
`label`	N/A	mod2	8 bytes
`entry`	N/A	mod2	12 bytes
`file constant`	N/A	mod2	440 bytes
`file variable`	N/A	mod2	*n* bytes
`structure`	N/A	Maximum of members	†
`picture`	`picture` contains *n* non-*v* chars	Byte	*n* bytes

† The size of a structure is determined by the size and shape of its members.

Alignment boundaries are determined with the modulo operation, represented in Table 4-5 as mod*n*. The *n* specifies the number of bytes by which the boundary is divisible with no remainder. For example, mod2 alignment indicates that the compiler allocates a data item so that it begins on an even-numbered byte boundary.

A structure is aligned according to the maximum alignment required by its members.

A structure's size is equal to the sum of bytes allocated for all of the structure's members plus any additional bytes needed for alignment padding between members. Consider, as an example of shortmap alignment, the following structure.

```
declare    1 ex_struct shortmap,
             2 c      char(1),
             2 d      float dec(8),
             2 i      fixed bin(15);
```

When shortmap alignment rules apply, the `ex_struct` structure is allocated 12 bytes of storage. The `ex_struct` structure itself is aligned on a mod2 boundary because that alignment is the maximum alignment required by the structure's members. In this example, the first member, `ex_struct.c`, is aligned on a mod2 boundary because the structure is aligned on a mod2 boundary. Since the size of `ex_struct.c` is 1 byte, the compiler adds an additional byte of padding because the next member, `ex_struct.d`, must also begin on a mod2 boundary. Within the structure, the members are aligned as shown in the following table.

Element	Alignment	Byte Offset	Size (in Bytes)
`ex_struct`	mod2	0	12 bytes
`ex_struct.c`	mod1	0	1 byte
Alignment padding	N/A	1	1 byte
`ex_struct.d`	mod2	2	8 bytes
`ex_struct.i`	mod2	10	2 bytes

Figure 4-6 illustrates shortmap alignment within the `ex_struct` structure. In the figure, each square represents one byte of memory. A square containing an identifier, such as `c`, indicates a byte used for that member. A square containing an asterisk (`*`) indicates a byte used for alignment padding.

c	*	d	d	d	d	d	d	d	d	i	i

Figure 4-6. Shortmap Alignment in a Structure

You can use the `aligned` attribute with character-string data or bit-string data to ensure that the data is aligned on a mod2 boundary if the alignment is shortmap. The `aligned` attribute is used in the data-type specification of a variable and applies only to that variable. See Chapter 7 for more information about the `aligned` attribute.

Longmap Alignment Rules

When you use longmap alignment, **most** scalar data types are allocated so that they begin on an alignment boundary that is equal to the type's size. Table 4-6 shows the longmap alignment rules.

Table 4-6. Longmap Alignment Rules

Data Type	Restriction	Alignment	Size (in Bytes)
`fixed bin(p)`	*p* <= 15	mod2	2 bytes
`fixed bin(p)`	*p* > 15 but < 32	mod4	4 bytes
`fixed bin(p)`	*p* > 31 but < 64	mod8	8 bytes
`fixed dec(p,q)`	*p* <= 9	mod4	4 bytes
`fixed dec(p,q)`	*p* > 9	mod8	8 bytes
`float bin(p)`	*p* <= 24	mod4	4 bytes
`float bin(p)`	*p* > 24	mod8	8 bytes
`float dec(p)`	*p* <= 7	mod4	4 bytes
`float dec(p)`	*p* > 7	mod8	8 bytes
`char(n)`	N/A	Byte	*n* bytes
`char(n) aligned`	N/A	mod4	`4*ceil(n/4)` bytes
`char(n) varying`	N/A	mod2	`2*ceil(n/2) + 2` bytes
`bit(n)`	N/A	Bit	*n* bits
`bit(n) aligned`	N/A	mod4	`4*ceil(n/32)` bytes
`pointer`	N/A	mod4	4 bytes
`label`	N/A	mod8	8 bytes
`entry`	N/A	mod4	12 bytes
`file constant`	N/A	mod4	440 bytes
`file variable`	N/A	mod4	4 bytes
`structure`	N/A	Maximum of members	†
`picture`	`picture` contains *n* non-*v* chars	Byte	*n* bytes

† The size of a structure is determined by the size and shape of its members.

Alignment boundaries are determined with the modulo operation, represented in Table 4-6 as mod*n*. The *n* specifies the number of bytes by which the boundary is divisible with no remainder. For example, mod4 alignment indicates that the compiler allocates a data item so that it begins on a boundary that is divisible by 4.

A structure (and thus its first member) is aligned according to the maximum alignment required by its members. The data **within** the structure is aligned according to the rules shown in Table 4-6.

A structure's size is equal to the sum of bytes allocated for all of the structure's members plus any additional bytes needed for alignment padding between members. Consider, as an example of longmap alignment, the following structure.

```
declare    1 ex_struct longmap,
             2 c      char(1),
             2 d      float dec(8),
             2 i      fixed bin(15);
```

In the preceding example, when the longmap alignment rules apply, the `ex_struct` structure is allocated 18 bytes of storage. The `ex_struct` structure itself (and thus its first member) is aligned on a mod8 boundary because that alignment is the maximum alignment required by the structure's members. In the example, the `ex_struct.d` member requires mod8 alignment. Within the structure, the members are aligned as shown in the following table.

Element	Alignment	Byte Offset	Size (in Bytes)
`ex_struct`	mod8	0	18 bytes
`ex_struct.c`	mod1	0	1 byte
Alignment padding	N/A	1	7 bytes
`ex_struct.d`	mod8	8	8 bytes
`ex_struct.i`	mod2	16	2 bytes

Figure 4-7 illustrates longmap alignment within the `ex_struct` structure. In the figure, each square represents one byte of memory. A square containing an identifier, such as `c`, indicates a byte used for that member. A square containing an asterisk (`*`) indicates a byte used for alignment padding.

c	*	*	*	*	*	*	*	d	d	d	d	d	d	d	d	i	i

Figure 4-7. Longmap Alignment in a Structure

You can use the `aligned` attribute with character-string data or bit-string data to ensure that the data is aligned on a mod4 boundary if the alignment is longmap. The `aligned` attribute is used in the data-type specification of a variable and applies only to that variable. See Chapter 7 for more information about the `aligned` attribute.

Data-Type Compatibility

Data can be accessed by and passed between an OpenVOS PL/I program and programs written in other high-level languages to the extent that both languages define the particular data type.

Data can be passed to other programs through arguments. See Chapter 3 for additional information about passing arguments. See the *VOS PL/I User's Guide* (R145) for detailed information on how to call, from an OpenVOS PL/I program, a program written in BASIC, C, COBOL, FORTRAN, or Pascal.

Table 4-7 summarizes the type compatibility of the OpenVOS PL/I data types with the data types in other OpenVOS languages.

Table 4-7. Cross-Language Compatibility of Data Types

BASIC	C	COBOL	FORTRAN	Pascal	PL/I
name=7	float	comp-1	real*4	N/A	float bin(24)
name=15	double, long double	comp-2	real*8	real	float bin(53)
name%=15	short	comp-4	integer*2, logical*2	-32768..32767	fixed bin(15)
name%=31	long, int, enum	comp-5	integer*4, logical*4	integer	fixed bin(31)
N/A	long long int	N/A	N/A	N/A	fixed bin(63)
name=7	float	comp-1	real*4	N/A	float dec(7)
name=15	double, long double	comp-2	real*8	real	float dec(15)
name# = (*i*+*f*,*f*)	N/A	comp-6, binary, pic s9(*i*)v(*f*)	N/A	N/A	fixed dec(*i*+*f*,*f*)
name$=1	char	pic x display	character*1, logical*1	char	char(1)
name$=*n*	char *id*[*n*]	pic x(*n*) display	character**n*	array [1..*n*] of char	char(*n*)
name$<=*n*	char_varying(*n*)	pic x(*n*) display-2	string**n*	string (*n*)	char(*n*) varying
N/A	N/A	N/A	N/A	boolean	bit(1)
N/A	N/A	N/A	N/A	array[1..*n*] of boolean	bit(*n*)
N/A	N/A	N/A	N/A	set	bit aligned
N/A	(**id*)()	entry	external	procedure, function	entry variable
N/A	N/A	label	N/A	N/A	label
N/A	*type* **id*	pointer	N/A	^*type_name*	pointer

Notes:

1. In all OpenVOS high-level languages, data types must be aligned on the same type of boundary to be compatible.

2. Though the C data type `char id[n]` is allocated similarly to the data types shown in its row, it is essentially different from these data types. For example, unlike the other data types listed in this row, an array type in C, such as `char id[n]`, is treated as a pointer in almost all contexts.

3. In OpenVOS PL/I, the `fixed bin(15)` type does **not** support the value -32,768.

4. The precision of decimal data is described in terms of *`i`* (the number of places to the left of the decimal point) and *`f`* (the number of places to the right of the decimal point).

5. There is no equivalent of the OpenVOS COBOL data types `comp-3` (packed decimal) or `pic(9)` in the other high-level languages.

6. The C data type `char` and the PL/I data type `char(1)` are compatible for argument passing, but their function return values are incompatible.

Chapter 5:
Data-Type Conversions

This chapter discusses the following topics related to data-type conversions.

Overview

Arithmetic, pictured, or string values can be converted to other arithmetic, pictured, or string data types. No other types of data (such as label, file, pointer, and entry values) can take part in conversions.

PL/I implicitly converts some values that do not match the type required in a particular context. For example, the following can cause implicit data-type conversion.

- the assignment operator
- arithmetic operators
- relational operators
- the concatenate operator
- the `get` statement
- the `put` statement
- the `return` statement
- passing an argument by value to a subroutine
- passing an argument to a built-in function
- certain record-I/O statement options such as `key` and `keyfrom`

You can use conversion built-in functions to explicitly convert a value to a specific type. The conversion built-in functions follow.

```
binary
bit
character
convert
decimal
fixed
float
pointer
rel
```

A value to be converted is called a *source value*. The data type the value is to be converted to is called the *target data type*. The context in which the conversion occurs determines the target data type. For example, in an assignment statement, the data type of the variable on the left side of the assignment operator is the target data type for the source value on the right side of the assignment operator.

See Chapter 9 for the rules that determine the target data-type descriptions that result from arithmetic operators, relational operators, bit-string operators, and the concatenate operator.

In some cases, the PL/I compiler only explicitly gives a partial target data type. This can occur when you use a conversion built-in function or in other contexts, including certain arithmetic expressions. This chapter provides the rules for completing target data types.

Arithmetic Conversions

This section explains the rules governing the conversion of arithmetic values to various data types. It discusses the following topics.

- "Arithmetic to Arithmetic Conversion"
- "Arithmetic to Bit-String Conversion"
- "Arithmetic to Character-String Conversion"

Arithmetic to Arithmetic Conversion

Conversion from one arithmetic data type to another most frequently occurs when a program uses arithmetic or relational operators. The base (binary or decimal) and scale (fixed-point or floating-point) of the target must be known.

If the source value is floating-point and the target is fixed-point, the precision and scaling factor of the target must also be known. In all other cases, if the precision and scaling factor of the target are not explicit, you can determine default precision and scaling factor based on the data type of the source value and the target's base and scale.

If the base of the source value and the target are the same, the default is not to change the precision. If the bases are different, the following general rules determine the default target precision.

```
binary_precision = ceil(decimal_precision * 3.32)
decimal_precision = ceil(binary_precision / 3.32)
```

Notes:

1. The `ceil` built-in function rounds a nonintegral result up to the next highest integer.
2. If the target is fixed-point, 1 is added to the converted precision.
3. If the resulting precision is greater than the maximum precision allowed for the target base and scale, the maximum is used.

Table 5-1 summarizes the rules for converting precisions and scaling factors.

Table 5-1. Precision Conversion Rules for Arithmetic to Arithmetic Conversion

Partial Target Data Type *r* **= precision** *s* **= scaling factor**	**Source Data Type** *N* **= maximum fixed-binary precision** *p* **= precision** *q* **= scaling factor**			
	`fixed binary`	`fixed decimal`	`float binary`	`float decimal`
`fixed binary`	*r* = *p*	*r* = `min`† `(ceil (`*p*`*3.32)+1,`*N*`)`‡	§	§
`fixed decimal`	*r* = `min (ceil(`*p*`/3.32) +1,18)`	*r* = *p* *s* = *q*	§	§
`float binary`	*r* = `min(`*p*`,53)`	*r* = `min(ceil (`*p*`*3.32),53)`	*r* = *p*	*r* = `ceil (`*p*`*3.32)`
`float decimal`	*r* = `min (ceil(`*p*`/3.32) ,15)`	*r* = `min(`*p*`,15)`	*r* = `min(ceil (`*p*`/3.32),15)`	*r* = *p*

† The `min` built-in function returns the lesser of two arguments. See Chapter 13 for explanations of the `ceil` and `min` built-in functions.
‡ The value of *N* is 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.
§ Conversions from floating-point data to fixed-point data arise only in situations where the target precision is explicit.

The following conversions produce approximate values.

- converting a nonintegral fixed-point decimal value to a floating-point value
- converting a floating-point value to a fixed-point decimal value

If a nonintegral fixed-point decimal value is converted to a fixed-point integer, or to a fixed-point decimal value with a smaller scaling factor, excess fractional digits are truncated; no rounding occurs. To control truncation or rounding, use a built-in function: `round`, `ceil`, `floor`, or `trunc`.

Table 5-2 lists the target precisions for some common source and target data types.

Table 5-2. Target Precisions for Some Common Arithmetic to Arithmetic Conversions

Source Data Type	Target Data Type
`fixed bin(15)`	`fixed dec(6,0)`
`fixed bin(31)`	`fixed dec(11,0)`
`fixed bin(63)`	`fixed dec(18,0)`
`float bin(24)`	`float dec(8)`
`float bin(53)`	`float dec(15)`
`fixed dec(4)`	`fixed bin(15)`
`fixed dec(7)`	`fixed bin(25)`
`float dec(16)`	`float bin(20)`
`float dec(15)`	`float bin(50)`

You can use the `decimal` and `binary` built-in functions to explicitly convert the base of an arithmetic value. The `fixed` and `float` built-in functions explicitly convert the scale of an arithmetic value. Built-in functions are explained in Chapter 13.

Arithmetic to Bit-String Conversion

Converting an arithmetic value to a bit-string value involves the following steps.

1. The arithmetic value is converted to a fixed-point binary integer under the rules for arithmetic to arithmetic conversion explained earlier in this chapter.

2. The absolute value of the binary representation of that integer is interpreted as a bit-string value.

The length of the bit string is equivalent to the precision of the binary integer. The precision of the integer is determined by the source data type, as summarized in Table 5-3.

Table 5-3. Bit-String Lengths from Arithmetic to Bit-String Conversions

Source Data Type	Length of Bit String (Precision of Binary Integer)
`fixed dec(p,q)`	`min(N†, ceil((p-q)*3.32))`
`float dec(p)`	`min(N, ceil(p*3.32))`
`fixed bin(p)`	`p`
`float bin(p)`	`min(N,p)`

† The value of *N* (the maximum fixed binary precision) is 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.

If the target of the conversion supplies a longer bit-string length, the resultant bit-string value is padded on the right with zero bits to make it the length of the target. If the target supplies a shorter bit-string length, the rightmost bits of the string are truncated to make it the length of the target.

The following example illustrates arithmetic to bit-string conversion.

```
declare   a    fixed bin(15);
declare   b    bit(16) aligned;
declare   c    bit(7) aligned;

a = -5;   /* Absolute value bit representation: '000000000000101'b */
b = a;    /* b = '0000000000001010'b                               */
c = a;    /* c = '0000000'b                                        */
```

In the preceding example, the bit-string value is right-padded with one zero bit when assigned to `b`; the string is right-truncated to a length of 7 when assigned to `c`.

Under the following conditions, changing the maximum precision can change a program's results or expected behavior:

- A conversion from `fixed decimal`, `float decimal`, or `float binary` to `bit` whose result size is greater than 31 bits
- A conversion from `fixed binary` to `bit` or `char` whose result size is greater than 31 bits but that has no `fixed binary` operands whose precision is greater than 31
- A conversion to `fixed binary` from `bit` or `char` that does not specify a precision, has a result precision greater than 31, and whose result is input to the `bit` built-in function
- A conversion from a bit string that is longer than 31 bits to `fixed decimal`, `float decimal`, or `float binary`

When the compiler encounters any of the preceding conditions and you have specified the `-system_programming` command-line argument, the compiler returns a warning.

Consider the following example:

```
declare fixeddec18   fixed decimal(18);
declare fixedbin59   fixed bin(59);
declare fixedbin63   fixed bin(63);
declare bit64        bit(64);
declare bit59        bit(59);
declare bit63        bit(63);

fixeddec18 = 123456789123456789;
bit64 = bit(fixeddec18);
fixedbin59 = 123456789123456789;
bit59 = bit(fixedbin59);
fixedbin63 = 1234567891234567890;
bit63 = bit(fixedbin63);
```

When you compile the preceding program with the `-max_fixed_bin 31` argument, the compilation fails because `fixedbin59` and `fixedbin63` exceed the limit for the maximum precision. However, compiling the program with the `-max_fixed_bin 63` argument results in the following:

- `bit64` has the value
  `'000110110110100110110100101110101100110100000101111100010101000 00'b`

- `bit59` has the value
  `'00110110110100110110100101110101100110100000101111100010101'b`

- `bit63` has the value
  `'00100010010001000010000111101001100000000100011101101101101001 0'b`

Consider another example:

```
declare fixeddec18   fixed decimal(18);
declare bit64        bit(64);

fixeddec18 = 123456123456123456;
bit64 = bit(fixeddec18);
```

When you compile this program with the `-max_fixed_bin 31` argument, the compilation and bind succeed, but during the execution, the program returns the following error, and the execution fails:

```
The default error handler has been invoked.
A fixed-point value is too large to fit in the target variable.
```

However, if you compile the same program with the `-max_fixed_bin 63` argument, `bit64` has the value
`'0001101101101001101010110000101011111111001011110010010000000000'b`.

This behavior occurs because of the intermediate conversion to the `fixed decimal` type during the conversion to the `bit` type.

Implicitly converting an arithmetic value to a bit-string value might result in a compiler warning message. To perform the conversion explicitly and suppress the compiler warning, use the `bit` built-in function as explained in Chapter 13.

Arithmetic to Character-String Conversion

The arithmetic to character-string conversion most commonly occurs when an arithmetic value is involved in one of the following operations.

- list-directed stream output
- concatenation
- assignment to a character-string variable

The result of the conversion is the character representation of the arithmetic value, usually with several leading space characters.

The arithmetic to character-string conversion is a two-step process.

1. The arithmetic value is converted to a decimal value; the scale is unchanged. The formulae provided in Table 5-1 determine the precision of the decimal value.

2. The decimal value is converted to a character string. The formula used to derive the character-string value depends on the type of the decimal value.

The following four cases are possible.

- floating-point: `float dec(`*p*`)`
- integer: `fixed dec(`*p*`)` or `fixed dec(`*p*`,0)`
- nonintegral fixed-point: `fixed dec(`*p*`,`*q*`)`, where `0 < `*q*` <= `*p*
- nonintegral fixed-point: `fixed dec(`*p*`,`*q*`)`, where *q*` < 0` or *q*` > `*p*

The following sections describe these four cases.

- "Case 1: Floating-Point"
- "Case 2: Integer"
- "Case 3: Nonintegral Fixed-Point with `0 < q <= p`"
- "Case 4: Nonintegral Fixed-Point with `q < 0` or `q > p`"

Case 1: Floating-Point

If the decimal value has floating-point scale and precision *p*, the length of the resulting character string is *p*` + 6`. If the decimal value is negative, the first character of the string is a minus sign; otherwise, the first character is a space. The second character is the most significant digit of the mantissa. This character is followed by a decimal point and the remaining *p*` - 1` digits of the mantissa. The character `E` and then the sign of the exponent and a two-character exponent value follow the mantissa.

If the exponent requires three digits, the mantissa contains only *p*` - 1` digits.

If the floating-point value is infinity (positive or negative), the first character of the resultant string is a space or a minus sign; the next eight characters are `infinity`. The rest of the string is filled with space characters. If the precision is less than 3, the rightmost characters from the word `infinity` are truncated. An infinite value with decimal precision 1 converts to `' infini'`.

For example, if a `float bin(24)` value is being converted to a character string, it is first converted to the `float dec(8)` type. The length of the resultant character string is 8 + 6, or 14. The following table shows some sample conversion results.

`float bin(24)` **Source Value**	**Character-String Target Value**
`0.000000E+00`	`' 0.0000000E+00'`
`-7.531000E+02`	`'-7.5310000E+02'`
`5.499990E-06`	`' 5.4999900E-06'`
Infinity	`' infinity     '`
Negative infinity	`'-infinity     '`

Case 2: Integer

If the decimal value has fixed-point scale, precision *p*, and a scaling factor of zero, the length of the resulting character string is `p + 3`. The character-string value consists of the *p* digits of the decimal value with no leading zeroes, preceded by a minus sign if the value is negative, preceded by sufficient spaces to fill the string. The value zero has one integral zero digit.

For example, if a `fixed bin(15)` value is converted to a character string, it is first converted to the `fixed dec(6)` type. The length of the character string is 6 + 3, or 9. The following table shows some sample conversion results.

`fixed bin(15)` **Source Value**	**Character-String Target Value**
`0`	`'        0'`
`52`	`'       52'`
`-31043`	`'   -31043'`

If the precision of a fixed-binary value is 60 or greater, and if its magnitude is greater than 999,999,999,999,999,999, a special case occurs for OpenVOS PL/I in order to allow the conversion of the value to a character string. When the fixed-binary value is converted to the intermediate `fixed decimal(18)` value, the compiler behaves as though the intermediate value can contain 19 digits. (This is possible because in OpenVOS, a `fixed decimal(18)` value is actually represented as a 64-bit binary value.) The compiler then converts the intermediate value to a `character(21)` value, taking an extra digit position from the left.

Consider another example:

```
declare fixeddec18   fixed decimal(18);
declare fixedbin59   fixed bin(59);
declare fixedbin63   fixed bin(63);

fixeddec18 = 123456789123456789;
fixedbin59 = 123456789123456789;
fixedbin63 = 1234567891234567890;
```

If you compile this program with the `-max_fixed_bin 31` argument, the compilation fails because `fixedbin59` and `fixedbin63` exceed the limit for the maximum precision. However, if you compile this program with the `-max_fixed_bin 63` argument, the results are as shown in the following table.

Variable Name	Source Value	Character-String Target Value
`fixeddec18`	`123456789123456789`	`'      123456789123456789'`
`fixedbin59`	`123456789123456789`	`'      123456789123456789'`
`fixedbin63`	`1234567891234567890`	`'     1234567891234567890'`

Case 3: Nonintegral Fixed-Point with `0 < q <= p`

If the decimal value has fixed-point scale, precision *p*, and scaling factor *q*, where `0 < q <= p`, the length of the resulting character-string value is `p + 3`. The value consists of *q* fractional digits, preceded by a decimal point, preceded by `p - q` integral digits with no leading zeroes, preceded by a minus sign if the decimal value is negative, preceded by sufficient spaces to fill the string. If the integral part is zero, a single integral zero character is included in the result.

For example, if a `fixed dec(5,2)` value is converted to a character string, the string length is 5 + 3, or 8. The following table shows some sample conversion results.

`fixed dec(5,2)` Source Value	Character-String Target Value
`0.00`	`'    0.00'`
`-50.00`	`'  -50.00'`
`27.42`	`'   27.42'`
`0.05`	`'    0.05'`
`-0.01`	`'    0.01'`

Case 4: Nonintegral Fixed-Point with `q < 0` or `q > p`

If the decimal value has fixed-point scale, precision *p*, and scaling factor *q*, where `q < 0` or `q > p`, the length of the resulting character-string value is `p + 4` if `q < 10` and `p + 5` if `q >= 10`. The value consists of a one- or two-digit scaling factor preceded by the sign of the scaling factor, preceded by the letter `F`, preceded by up to *p* significant digits of the value with no decimal point and no leading zeroes, preceded by a minus sign if the value is negative. This value is preceded by as many space characters as are needed to fill out the string.

For example, if a `fixed dec(2,5)` value is converted to a character string, the string length is 2 + 4, or 6. The following table shows some sample conversion results.

`fixed dec(2,5)` Source Value	Character-String Target Value
`0.00012`	`' 12F-5'`
`-0.00003`	`' -3F-5'`
`84F05`	`' 84F-5'`
`-.32F3`	`'-32F-5'`

The three characters added to the precision in Case 2 and Case 3 are needed when $p = q$. In such cases, characters are needed to hold a sign, an integral zero, a decimal point, and p digits.

Because the results of an arithmetic to character-string conversion are not strictly intuitive, the compiler warns you of any such implicit conversions if you compile with the `-system_programming` argument. To suppress these warnings, make the conversions explicit by using the `character` built-in function, as shown in the following example.

```
declare    astr      char(14) varying;
declare    num       fixed bin(15);
           .
           .
           .
     astr = character(num);
```

To remove leading spaces from the result of an arithmetic to character-string conversion, use the `ltrim` built-in function.

```
astr = ltrim(char(num));
```

Built-in functions are discussed in Chapter 13.

Character-String Conversions

This section explains the rules governing the conversion of character-string values to various data types. It discusses the following topics.

- "Character-String to Character-String Conversion"
- "Character-String to Arithmetic Conversion"
- "Character-String to Bit-String Conversion"

Character-String to Character-String Conversion

You can convert a character-string value to a character-string value of a different length. If the target length is not explicit, it defaults to the length of the source value.

If the length of the target is shorter than the source value, the value is truncated to that length. If the length of the target is longer than the source value, the value is right-padded with spaces to equal the length of the target.

If the target is a varying-length character string with a maximum length greater than or equal to the length of the source value, the entire value is assigned to the target. The current length of the target is set to the length of the source value. Note that space characters in the source string are treated the same as any other characters and are transferred to the target.

If the target is a varying-length character string with a maximum length less than the length of the source value, the source value is truncated to the maximum length of the target. The current length of the target is set to the maximum length.

The following example illustrates character-string to character-string conversion.

```
declare   astr      char(8);
declare   bstr      char(12);
declare   cstr      char(3);
declare   vstr      char(24) varying;

     astr = 'abcdef  ';
     bstr = astr;        /* bstr = 'abcdef      ' */
     cstr = astr;        /* cstr = 'abc'          */
     vstr = astr;        /* vstr = 'abcdef  '     */
```

The variable `astr` is assigned a value of length 8. The character-string value is padded on the right with four space characters when assigned to `bstr`. The value is truncated to three characters when assigned to `cstr`. The value is assigned to `vstr` without any change; the current length of `vstr` is set to 8.

You can use the `character` built-in function to explicitly perform a character-string to character-string conversion; see Chapter 13.

Character-String to Arithmetic Conversion

You can convert a character string to an arithmetic value only if all nonspace characters in the string together comprise a contiguous valid literal arithmetic constant.

If the context does not supply a target base and scale, decimal and fixed-point are the defaults. If a target precision is not supplied, the maximum precision allowed for the base and scale is used. If no target scaling factor is explicitly specified, it is assumed to be zero.

If the source value is the null string or if it contains all space characters, the resultant arithmetic value is zero. Otherwise, the value of the constant represented by the contents of the string is converted to the data type of the target, using the rules for arithmetic to arithmetic conversion.

The following table shows some examples of character-string to fixed-point integer conversions.

Character-String Source Value	**`fixed dec(18)` Target Value**
`' 5e+0 '`	`5`
`' -7'`	`-7`
`'-4.7 '`	`-4`
`'18'`	`18`
`' .05 '`	`0`
`' '`	`0`
`''`	`0`
`'1 1'`	Invalid
`'a'`	Invalid

If the source value is invalid for conversion to an arithmetic value, the `error` condition is signaled.

You can explicitly convert a character string to an arithmetic value by using the `binary`, `decimal`, `fixed`, or `float` built-in function. These functions are explained in Chapter 13.

Character-String to Bit-String Conversion

You can convert a character-string value that contains only `'0'` and `'1'` characters to a bit-string value. If you attempt to convert a character-string value that contains any other character, including the space character, the `error` condition is signaled.

If the length of the target is not specified, it is the same as the length of the source value. Each character in the source value is converted to a `'0'` or `'1'` bit.

A null character string converts to a null bit string.

If the context provides a length for the target, the resultant string is truncated or right-padded with zero bits to make it the proper length.

The following table shows some examples of character-string to bit-string conversion.

Character-String Source Value	**Bit-String Target Value**
`''`	`''b`
`'010'`	`'010'b`
`' '`	Invalid
`' 1'`	Invalid
`'1 '`	Invalid
`'2'`	Invalid

If the source value is invalid for conversion to a bit-string value, the `error` condition is signaled.

You can use the `bit` built-in function to explicitly convert a character-string value to a bit-string value; see Chapter 13.

Bit-String Conversions

This section explains the rules that govern the conversion of bit-string values to various data types. It discusses the following topics.

- "Bit-String to Bit-String Conversion"
- "Bit-String to Arithmetic Conversion"
- "Bit-String to Character-String Conversion"

Bit-String to Bit-String Conversion

A bit-string to bit-string conversion can occur in bit-string expressions or when a bit-string value is assigned to a bit-string variable of a different length.

If the target variable length, *n*, is shorter than the source value, the leftmost *n* bits of the value are assigned to the target. The other bits are truncated.

If the target variable is longer than the source value, the value is right-padded with zero bits to equal the length of the target.

The following example illustrates a bit-string to bit-string conversion.

```
declare   a    bit(5) aligned;
declare   b    bit(10) aligned;
declare   c    bit(3) aligned;

     a = '01110'b;
     b = a;         /* b = '0111000000'b */
     c = a;         /* b = '011'         */
```

The bit-string value is right-padded to a length of 10 when it is assigned to `b`. The string is truncated to a length of 3 when assigned to `c`.

You can explicitly perform bit-string to bit-string conversion with the `bit` built-in function as explained in Chapter 13.

Bit-String to Arithmetic Conversion

Bit-string values can be implicitly converted to nonnegative integers.

If no target base or scale is supplied for the target, the defaults are binary and fixed-point. If the target precision is not explicit, the default is the maximum precision allowed for the base and scale.

The bit-string to arithmetic conversion is invalid if the length of the bit-string value is greater than either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.

The bits of the bit-string value are interpreted as a base 2 integer. The rightmost bit of the string is considered to be the unit's position of the integer; the length of the bit string is the precision of the integer. (The precision of the integer is always the value of `$MAX_FIXED_BIN` unless it is explicitly specified.)

The rules for arithmetic to arithmetic conversion are used to convert the integer value to the data type of the target.

A null bit-string value converts to zero.

The following table provides some examples of bit-string to arithmetic conversions.

Bit-String Source Value	Arithmetic Target Value
`'1101'b`	`13`
`''b`	`0`
`'00000'b`	`0`

Because the conversion of bit strings to integer values produces reasonable, predictable results, small bit strings can be used to hold small nonnegative integers. However, arithmetic operators require arithmetic operands. Therefore, use the `binary` or `decimal` built-in functions to explicitly convert bit-string values to binary or decimal integer operands. See Chapter 13 for descriptions of built-in functions.

Bit-String to Character-String Conversion

A bit-string value is converted to a character-string value by converting each bit to a `'0'` or `'1'` character. This produces a character-string value with the same length as the source bit-string value.

A null bit string converts to a null character string.

If the context supplies a length for the target character string, the string is either truncated or right-padded with space characters to conform to the given length.

The following table shows some sample bit-string to character-string conversions.

Bit-String Source Value	Character-String Target Value
`'0'b`	`'0'`
`''b`	`''`
`'1011'b`	`'1011'`

To explicitly convert a bit-string value to a character string, use the `character` built-in function as described in Chapter 13.

Pictured Data Conversions

This section explains the rules governing conversions to and from pictured data. It discusses the following topics.

- "Conversions to Pictured Data Types"
- "Pictured to Arithmetic Conversion"
- "Pictured to Bit-String Conversion"
- "Pictured to Character-String Conversion"

Conversions to Pictured Data Types

A conversion of arithmetic, string, or pictured data to a pictured data type is a two-step process.

1. The source value is converted to a fixed-point decimal value as described by the target picture. See Chapter 4 for more information on this conversion.

2. The fixed-point decimal value is converted to a character string under control of the picture characters described in Chapter 4.

If the fixed-point decimal value described by the picture is not sufficient to retain all digits to the left of the decimal point in the converted source value, the program is in error and produces unpredictable results. Excess fractional digits are truncated.

The length of the resultant character string is the number of characters in the target picture, excluding any `v` picture characters.

If the fixed-point decimal value is zero and the picture does not contain at least one `9`, `t`, `i`, `r`, or `y` character, the resultant character string contains all zero-suppression characters. Zero-suppression characters are space characters, except where asterisks are specifically requested.

Negative values cannot be converted to a pictured type unless the picture contains at least one sign character.

The following table shows some examples of conversions to pictured values.

Source Value	Target Picture	Result
`5.2`	`zzzvzz`	`'  520'`
`0.01`	`zzzvzz`	`'   01'`
`0`	`zzz`	`'   '`
`1234`	`zzzzv`	`'1234'`
`12345`	`99999`	`'12345'`
`123`	`99999`	`'00123'`
`-105.02`	`$**,***.99`	Invalid
`-105.02`	`$**,***.99cr`	`'$***105.02cr'`
`105.02`	`$**,***.99cr`	`'$***105.02  '`
`-75`	`----v--`	`' -7500'`
`75`	`----v--`	`'  7500'`
`-20`	`-999`	`'-020'`
`20`	`-999`	`' 020'`
`-275.03`	`$$$$$v.99-`	`' $275.03-'`
`25.01`	`$$$$$v.99-`	`'  $25.01 '`
`-7.5`	`$$,$$$v.99db`	`'     $7.50db'`
`0`	`-****v.**`	`'********'`
`5`	`-****v.**`	`' ***5.00'`
`-75`	`-****v.**`	`'-**75.00'`
`.75`	`z.vzz`	`'  75'`
`.75`	`zv.zz`	`' .75'`
`0`	`zz$`	`'   '`

Pictured to Arithmetic Conversion

The first step in a pictured to arithmetic conversion is to convert the source pictured value to a fixed-point decimal value. The precision of the fixed-point decimal value is determined by the picture as described in Chapter 4.

If the context specifies a different arithmetic data type, the fixed-point decimal value is converted to the proper type under the rules for arithmetic to arithmetic conversion discussed earlier in this chapter.

In some cases, the source pictured value cannot be converted to an arithmetic value. Attempting to use such a pictured value in a context that expects an arithmetic value produces unpredictable results. The `valid` built-in function, described in Chapter 13, tests whether a pictured value can be converted to an arithmetic value.

Pictured to Bit-String Conversion

Conversion of a pictured value to a bit string is a two-step process.

1. The pictured value is converted to a fixed-point decimal value as described in Chapter 4.
2. The fixed-point decimal value is converted to a bit string under the rules for arithmetic to bit-string conversion as explained in "Arithmetic Conversions" earlier in this chapter.

Pictured to Character-String Conversion

Pictured data is stored as character strings. The data is not converted when used in a context that expects a character-string value.

Chapter 6:
Storage Classes

This chapter discusses the following topics related to storage classes.

- "Overview"
- "Automatic Storage"
- "Static Storage"
- "Based Storage"
- "Defined Storage"
- "Parameters"
- "Storage Sharing"

Overview

Every PL/I variable has a storage class. The *storage class* determines two characteristics of the variable.

- when and how its storage is allocated
- the valid forms of its extent expressions

You can declare any variable, except a parameter, to have one of the following storage classes.

- `automatic`
- `based`
- `defined`
- `static`

If you do not specify a storage class for a nonparameter variable, the compiler supplies `automatic` by default.

Any variable that appears in the parameter list of a `procedure` or `entry` statement is recognized as having the parameter storage class.

The declared length of a string variable and the bounds of an array are called *extents*. The extents determine the size of the variable's storage. Extents are evaluated when storage is allocated for the variable. The permitted form of extents depends on the storage class.

Automatic Storage

The automatic storage class is the default for all nonparameter variables. Because it is the default, the `automatic` keyword is usually not specified. In the following example, the variable `x` has the automatic storage class.

```
a:   procedure;

declare   x    fixed bin(15);
```

The next two sections discuss the following topics.

- "Allocation of Automatic Variables"
- "Extents of Automatic Variables"

Allocation of Automatic Variables

Storage for an automatic variable is allocated when the procedure or begin block containing the variable declaration is activated. Each time the procedure or begin block is reactivated, new storage is allocated within the stack frame associated with that block activation. Therefore, if a block is activated recursively, each stack frame for that block contains an instance of each automatic variable declared in that block.

Depending on the value specified in the `-processor` argument of the `pl1` command, there are different limits on the maximum number of bytes available for a procedure or begin block's initial stack frame. Therefore, the value specified in the `-processor` argument affects the amount of nondynamic automatic storage available for a block. If the value specified in the `-processor` argument indicates the IA-32 processor, the maximum number of bytes available for each block's initial stack frame is 2,147,483,584 bytes.

The amount of automatic storage you can actually declare is somewhat less than these limits because temporary variables generated by the compiler also count toward the limit. Note that although the OpenVOS PL/I compiler supports extremely large values (such as 2,147,483,646), the system does not support them.

When a block activation terminates and its stack frame is popped from the stack, the storage of all automatic variables within the stack frame is freed. A block activation terminates when any of the following situations occurs.

- A procedure returns to its caller.
- A begin block executes its `end` statement.
- A `goto` statement transfers control to a previous block activation.

In the last case, all block activations between the block containing the `goto` statement and the block to which control is transferred are popped from the stack. The storage of all automatic variables declared in those block activations is freed.

Extents of Automatic Variables

The extents of automatic variables must be integer-valued expressions. These expressions cannot contain references to other automatic or defined variables declared in the same block.

The extent expression of an automatic variable is evaluated each time the containing block is activated. The value of the expression is saved in the stack frame, effectively fixing the size of the variable for that block activation. Any subsequent assignment to a variable used in the extent expression does not affect the variable size.

The following example illustrates the evaluation of extent expressions for automatic variables.

```
outer:     procedure;

declare    n    fixed bin(15);
           .
           .
           .
inner:     procedure;

declare    a(n) fixed bin(15);
declare    m    fixed bin(15);
           .
           .
           .
      n = 10;
      m = hbound(a,1);
```

In the preceding example, the size of `a` is determined by evaluating `n` when procedure `inner` is activated and storing the value in the stack frame associated with `inner`. A subsequent assignment to `n` does not change the size of `a`. When `inner` is activated, `n` must have a current integer value; the name `n` cannot be redeclared within `inner` as an automatic or defined variable. The `hbound` function returns the upper bound of an array dimension. The reference to `hbound` in the second assignment statement returns the value that `n` had at the time `inner` was activated. For an explanation of the `hbound` function, see Chapter 13.

Static Storage

Static storage is used for variables that must retain their values between calls to a procedure. You can assign an initial value to a static variable by specifying the `initial` attribute in the variable declaration.

```
declare    z    fixed bin(15) static initial(10);
```

For information on the `initial` attribute, see Chapter 7.

The next three sections discuss the following topics.

- "Allocation of Static Variables"
- "Extents of Static Variables"
- "Scope of Static Variables"

Allocation of Static Variables

Storage for all static variables is allocated within the program module prior to program execution and remains allocated throughout program execution.

Only one instance of the storage for each static variable is allocated. This holds true even if the procedure is activated recursively. A value assigned to a static variable is retained between calls to the procedure.

The following example illustrates one use for static variables.

```
a:   procedure;

     call incrementer;
     call incrementer;
          .
          .
          .
incrementer: procedure;

declare   count     fixed bin(15) static initial(0);

     count = count + 1;

     if count > 15
     then stop;

end incrementer;

end a;
```

In the preceding example, the storage for `count` is allocated before the program executes and its value is initialized to 0. The first call to `incrementer` sets `count` to 1. The second call references the same storage for `count` and changes the value to 2. All subsequent calls to `incrementer` continue to reference the same storage for `count`.

Extents of Static Variables

Because the extents of static variables are evaluated before program execution, they must be integer constants. Variable extents are not allowed.

Scope of Static Variables

Most PL/I variables have internal scope. However, you can specify external scope for static variables.

Each static variable has either the `internal` or `external` scope attribute. If neither is specified, `internal` is assumed.

Internal static variables are known in the block in which they are declared and all blocks contained within that block, except those blocks in which the same name is redeclared.

External static variables follow the same scope rules as internal static variables, with one difference: all declarations of the same name that include the `external` attribute refer to the same storage. The attributes specified in all such declarations must be equivalent after defaults are added. If an `initial` attribute appears in one external declaration it must appear in all declarations of that external name. If a different block contains a declaration of the same name without the `external` attribute, that declaration refers to a separate object with its own storage.

Note that the compiler does not produce initialization information for unreferenced external static variables unless you specify the `-table` argument of the `pl1` command. See the *VOS PL/I User's Guide* (R145) for more information on the `-table` argument.

The following example uses static variables with external scope and with internal scope.

```
a:   procedure;
declare   x    char(5) static external;
declare   y    fixed bin(15) static internal;
     .
     .
     .
     b:   procedure;
     declare   x    char(5) static external;
     declare   y    fixed bin(15) static internal;
          .
          .
          .
     end b;

end a;

c:   procedure;
declare   x    char(5) static external;
declare   y    fixed bin(15) static internal;
     .
     .
     .
end c;
```

In the preceding example, the three procedures `a`, `b`, and `c` all share the same instance of the variable `x`. Each of the procedures has its own instance of the variable `y`.

PL/I external variables are known and shared by all blocks in all source modules of a program, in a manner similar to the common variables of FORTRAN, except that each external static variable is shared independently of others. In a tasking environment, each task sees a separate instance of each external variable unless you specify otherwise at bind time, or you use the `shared` attribute described in Chapter 7.

Like internal static variables, you can initialize external variables with the `initial` attribute. If the name of an external static variable is identical to a message name in the current message file, and the declaration does **not** include the `initial` attribute, the binder will initialize that variable to the message code that corresponds to that message name. For example, if you are using the standard message file, the variable `e$end_of_file` in the following example is initialized to `1025`.

```
declare   e$end_of_file       fixed bin(15) static external;
```

For information on message files, see the manual *OpenVOS System Administration: Administering and Customizing a System* (R281). See also the description of the `use_message_file` command in the *OpenVOS Commands Reference Manual* (R098).

All external static variables with names beginning with `e$`, `m$`, `q$`, or `r$` are shared in a tasking environment. For information on tasking, see the *OpenVOS PL/I Transaction Processing Facility Reference Manual* (R015).

Based Storage

Storage is not allocated for based variables unless and until explicitly requested. Based variables serve as templates or descriptions of storage.

The next four sections discuss the following topics.

- "Based Variables and Pointers"
- "Allocation of Based Variables"
- "Freeing Based Storage"
- "Extents of Based Variables"

Based Variables and Pointers

To reference storage, a program needs two pieces of information.

- the address of the storage
- a data-type description of the storage

This information enables the program to locate and interpret storage.

A pointer value is a storage address. A based variable is a data-type description. Therefore, you can use a pointer and a based variable together to reference storage. You use the locator qualifier symbol (`->`) to associate a pointer with a based variable, as shown in the following example.

```
declare   x    char(3) based;
declare   p    pointer;
          .
          .
          .
     p -> x = 'abc';
```

In the preceding example, assume that an address or pointer value has been assigned to `p`. The reference `p -> x` is a pointer-qualified reference that enables you to assign a value to the storage addressed by `p` as if that storage contained a character-string variable with the attributes of `x`.

You can obtain pointer values in two ways.

- The `addr` built-in function returns the address of a variable's storage.
- The `allocate` statement dynamically allocates storage described by a based variable and sets a pointer to the address of that storage.

In the following example, the `addr` built-in function is used to assign a value to a pointer.

```
declare   x(5) float bin(24);
declare   y    float bin(24) based;
          .
          .
          .
     p = addr(x(k));
     p -> y = 1.00E+02;
```

In the preceding example, the `addr` built-in function returns the address of `x(k)` and assigns it to the pointer `p`. The second assignment statement assigns the value `1.00E+02` to the storage addressed by `p` and described by `y`. The variables `x(k)` and `y` share this storage. The effect is to assign the value `1.00E+02` to `x(k)`.

The preceding example would be incorrect if `y` had been declared with a different data type. For information on how based variables share storage, see ‘‘Storage Sharing’’ later in this chapter.

Using a based variable to access the storage of another variable has drawbacks. Readers of the program might have difficulty keeping track of what storage a pointer addresses. Also, if a based variable does not accurately describe the storage addressed by the pointer, the results are unpredictable. See the *VOS PL/I User’s Guide* (R145) for information about using based variables in optimized code.

Based variables and pointers are often used in applications where the number of instances of a record are not known when the block is activated and must be determined as the program executes. List structures consisting of based structures linked together via pointers provide dynamic data structures for use in these situations.

The following example copies a list structure.

```
a:     procedure;
declare old                  pointer;
declare new                  pointer;

declare   1  record           based,
            2  field1         float bin(24),
            2  field2         float bin(24),
            2  son            pointer,
            2  daughter       pointer;

     allocate record set (old);
          .
          .
          .
     new = copy(old);

copy:     procedure(rec_in) returns(pointer) recursive;
declare   (rec_in, rec_out)   pointer;

declare   null                builtin;

     if rec_in = null()
          then return(null());

     /* Else copy record */

     allocate record set(rec_out);

     rec_out -> record.field1 = rec_in -> record.field1;
     rec_out -> record.field2 = rec_in -> record.field2;
     rec_out -> record.son = copy(rec_in -> record.son);
     rec_out -> record.daughter = copy(rec_in -> record.daughter);

     return(rec_out);

end copy;
end a;
```

If all, or nearly all, references to a based variable use the same pointer, you can include a reference to that pointer in the `based` attribute. A reference to that based variable that is not explicitly qualified by a pointer is assumed to be qualified by the pointer specified in the `based` attribute. The use of such references is known as *implicit pointer qualification*.

The following example illustrates implicit pointer qualification.

```
declare   x         float bin(24) based(p);
declare   y         float bin(24);
declare   z         float bin(24);
declare   (p, q)    pointer;

          p = addr(y);
          q = addr(z);

          x = 7.308E+12;
          q -> x = 3.49E-01;
```

In the preceding example, `p` is designated in the declaration of `x` as an implicit pointer qualifier. Therefore, the unqualified reference `x` is equivalent to `p -> x`. Because `p` has been assigned the address of `y`, the unqualified reference to `x` refers to `y`. The reference `q -> x` is unaffected by the implicit pointer qualification; it refers to `z`.

Allocation of Based Variables

No storage is allocated for a based variable unless you code the `allocate` statement. The `allocate` statement allocates storage described by a based variable and assigns the address of that storage to a specified pointer.

Storage for based variables is allocated on a heap. A *heap* is a mass of randomly accessible storage that is associated with a process and available for allocation.

In the following example, the based variable `x` is a template for two areas of dynamically allocated storage.

```
declare   x(10)     float bin(24) based;
declare   (p, q)    pointer;
          .
          .
          .
          allocate x set(p);
          allocate x set(q);
```

The first `allocate` statement in the preceding example allocates storage of sufficient size to hold an array of 10 4-byte floating-point values and assigns the address of that storage to the pointer `p`. A subsequent reference to `p -> x` references that storage as an array of 10 floating-point values.

The second `allocate` statement in the example allocates a second similar area of storage and assigns the address of that storage to the pointer `q`. Subsequent references to `q -> x` refer to that second area of storage; `p -> x` continues to reference the first area of storage.

Note that OpenVOS PL/I requires the `set` clause of the `allocate` statement. See Chapter 12 for more information on the `allocate` statement.

In most contexts, a based variable reference must have an explicit or implicit pointer qualifier. The only exceptions are the `allocate` statement, and the `size` and `bytesize` built-in functions. In these contexts, the based variable describes an amount of storage.

Once storage has been allocated for a based variable, that storage remains allocated on the heap throughout program execution unless it is explicitly freed with the `free` statement.

Freeing Based Storage

An area of storage previously allocated by an `allocate` statement can be freed by a `free` statement. In the following example, an area of storage is allocated and then freed.

```
allocate x set(p);
free p -> x;
```

Once storage has been freed, you can no longer access that storage. An attempt to use a pointer that points to freed storage, such as `p` in the preceding example, produces unpredictable results. You can subsequently reuse a pointer to point to another storage address.

Extents of Based Variables

You can declare an array of based variables and use subscripts to reference individual elements of the array. To refer to an element of a based array, you can write references such as `p -> x(k)` or `q -> x(5)`.

The extents of a based variable must be integer-valued expressions. The extents are evaluated for each reference to the based variable; they are **not** frozen when storage is allocated. The programmer must ensure that any extents accurately describe the storage being referenced. The following example illustrates this situation.

```
declare   x(n) char(1) based;

     n = 10;
     allocate x set(p);
     n = m;
     p -> x(5) = 'a';
```

The `allocate` statement in the preceding example allocates heap space for an array of 10 1-byte character strings. Subsequently, the value of `n` is changed. This changes the extents of the array `x`. The reference `p -> x(5)` in the last assignment statement is valid only if `5 <= m <= 10`.

Defined Storage

A variable declared with the `defined` attribute is an alternative description of another variable; therefore, the two variables share storage. The following example illustrates how a defined variable is used.

```
declare   c(5) char(1);
declare   x    char(5) defined(c);

     x = 'abcde';
```

In the preceding example, `x` is a character-string variable whose value has a length of 5. It shares storage with, and is an alternate description of, the array `c`. The assignment statement assigns the character `'a'` to `c(1)`, the character `'b'` to `c(2)`, and so on.

For information on the rules for specifying alternative descriptions of shared storage, see "Storage Sharing" later in this chapter.

The next two sections discuss the following topics.

- "Allocation of Defined Variables"
- "Extents of Defined Variables"

Allocation of Defined Variables

Separate storage is not allocated for a defined variable; instead, it shares the storage of the variable on which it is defined. The allocation of storage for a variable is not affected by having another variable defined on it.

Extents of Defined Variables

The extents of a defined variable must be integer-valued expressions. These expressions are evaluated when the block is activated, and the results are stored in the stack frame, as are the extents of automatic variables. The extent expressions of a defined variable must not contain a reference to any automatic or defined variable declared in the same block.

Parameters

If a name appears in the parameter list of a `procedure` or `entry` statement, it has the parameter storage class.

See "Parameters and Arguments" in Chapter 3 for information on how arguments are passed to parameters.

The next two sections discuss the following topics.

- "Allocation of Parameters"
- "Extents of Parameters"

Allocation of Parameters

A parameter has no storage of its own. It shares storage with its corresponding argument.

In the following example, `x` and `y` share storage during the call to `p`.

```
a:     procedure;
declare   x    float bin(24);

     call p(x);
          .
          .
          .
     p:      procedure(y);
     declare   y    float bin(24);
               .
               .
               .
     end p;
end a;
```

During the call to `p` in the preceding example, `x` and `y` describe the same storage. When this occurs, the argument is said to have been passed *by reference*.

If an argument is passed to a parameter by value, its storage is copied to a temporary area of storage in the stack frame of the calling procedure. The corresponding parameter refers to that temporary storage area.

Extents of Parameters

The extents of a parameter can be either integer constants or asterisks.

If a parameter has constant extents, any corresponding argument passed by reference must have identical extents; if the corresponding argument is passed by value, it is converted to have the same extents as the parameter.

> **Note:** Array arguments **must** be passed by reference.

If the extents of a parameter are asterisks, its corresponding argument can have extents of any value; the parameter takes on the extents of the corresponding argument. The extents of the parameter remain fixed for that activation.

Note that only string lengths and array bounds of parameters can be asterisks. Arithmetic precisions must always be constants.

The following example demonstrates the use of asterisk extents.

```
declare   a         char(24) varying;
declare   b         char(12) varying;
          .
          .
          .
     call p(a);
          .
          .
          .
     call p(b);
          .
          .
          .
p: procedure(x);

declare   x         char(*) varying;
```

During the first call in the preceding example, the maximum length of `x` is 24. During the second call, the maximum length is 12.

Storage Sharing

PL/I provides two storage classes designed explicitly to permit storage to be shared by more than one variable: `based` and `defined`. Storage can be shared only if one of the following conditions is true.

- The data types of all variables sharing the storage are identical.
- All variables sharing the storage are unaligned bit strings.
- All variables sharing the storage are nonvarying unaligned character strings or pictured variables.

The last two cases in the preceding list are called *string overlays*.

The next three sections discuss the following topics.

- "String Overlays"
- "Data-Type Matching"
- "Untyped Storage Sharing"

String Overlays

In the case of a string overlay, the sharing variables must be all nonvarying unaligned character strings or all unaligned bit strings. The variables need not be scalar; you can use any of the following in string-overlay storage sharing.

- arrays having any extents and any number of dimensions
- structures containing only the required type of string data
- scalar string variables

These dissimilar variables can share storage because their storage is known to contain no gaps or extraneous information; nonvarying unaligned character strings contain only characters, and unaligned bit strings contain only bits.

In the following example, variables with different data types share storage.

```
declare    x(4,4)    char(1);
declare    y(16)     char(1) defined(x);
declare    z         char(5) defined(x);
declare    1  s      based,
             2  a    char(8),
             2  b    char(8);
declare    p         pointer;

        p = addr(x);
```

In the preceding example, `x` has a storage area large enough to contain 16 characters. This storage is entirely shared by `y`. The first 5 characters of storage are shared by `z`. Furthermore, a reference to `p -> s` references all 16 characters of `x`. The reference `p -> s.a` references the first 8 characters, and `p -> s.b` references the last 8 characters.

You could not use a based character string of more than 16 characters to access the storage of `x`. However, you could use any based character string of less than 16 characters to access part of the storage of `x`.

A based or defined string overlay must not be larger than that which is overlaid.

Data-Type Matching

Variables that are unsuitable for string-overlay storage sharing can share storage only if their data types match exactly. This section discusses the following topics related to which data types match.

- "Arithmetic Data"
- "String Data"
- "Pictured Data"
- "Arrays"
- "Structures"

Arithmetic Data

The base, scale, and precision of arithmetic variables must match if they are to share storage.

String Data

The `aligned` and `varying` attributes as well as the string lengths are considered to be a part of the data type. Therefore, if strings are to share storage, these characteristics must match, unless the strings qualify for string overlay sharing.

Pictured Data

A pictured variable can share storage only with other pictured variables that have identical pictures.

Arrays

Arrays can share storage with other arrays only if they share the following:

- the same data type
- the same number of dimensions
- the same number of elements in each dimension

The upper and lower bounds of each dimension need not match, but the number of elements per dimension must be the same. The following example shows an array sharing storage with another array.

```
declare   x(2:10)   fixed bin(15);
declare   y(8:16)   fixed bin(15) defined(x);
```

In the preceding example, the element `y(8)` shares storage with the element `x(2)`, `y(9)` shares with `x(3)`, and so forth.

A single element of an array can share storage with a nonarray variable of the same data type.

Structures

Two structures can share storage only if they are left-to-right equivalent. *Left-to-right equivalence* means that a sharing structure must be a valid description of the left part of the storage being shared. This means the structures must have identical members up to and including all members contained anywhere within the last substructure being shared. If any part of a substructure is shared, the entire substructure must be shared. In the following example, three structures are declared.

```
declare  1  s       ,
           2  a     fixed bin(15),
           2  b     ,
             3  c   char(1),
             3  d   bit(5) aligned,
           2  e     float bin(24);

declare  1  t       based,
           2  a     fixed bin(15),
           2  b     ,
             3  c   char(1),
             3  d   bit(5) aligned;
declare  1  u       based,
           2  a     fixed bin(15),
           2  b     ,
             3  c   char(1);
```

In the preceding example, the structure `t` could legitimately share storage with the structure `s`. However, the structure `u` cannot share the storage of `s` because the declaration of `u` does not describe all of the level-two item `s.b`.

You can use the `like` attribute to simplify the declarations of structures that are left-to-right equivalent. The `like` attribute is described in Chapter 7.

Untyped Storage Sharing

In OpenVOS PL/I, defined variables can share storage with variables of dissimilar type, provided that the actual content of storage is valid for the type of variable used to retrieve it. However, this sort of storage sharing makes the program dependent on the way variables are stored. Storage methods vary, depending on your module's processor family. Furthermore, untyped storage sharing is not standard and might not be allowed in other PL/I implementations, regardless of storage methods.

> **Note:** When sharing storage, a defined variable must **not** be larger than its base reference variable.

In the following example, a varying-length character-string variable shares storage with a structure.

```
declare    1  str_struct   ,
             2  length     fixed bin(15),
             2  fix_str    char(10);

declare    var_str         char(10) varying defined(str_struct);
```

If you use untyped data sharing, all data must be properly aligned. Unaligned nonvarying character strings and pictured data must be byte-aligned; all other data, excluding unaligned bit strings, must be aligned according to the mapping rules that are in effect. See Chapter 4 for a list of the alignment requirements of each data type.

For information about data alignment, see Chapter 4. For information about how values are stored and aligned in OpenVOS PL/I, see Appendix B.

Chapter 7:
Declarations and Attributes

This chapter discusses the following topics related to declarations and attributes.

- "Overview"
- "Label Prefixes"
- "The `declare` Statement"
- "Attributes"

The final section, "Attribute Reference Guide," discusses each of the OpenVOS PL/I attributes.

Overview

All names used in a PL/I program, except the names of built-in functions, must be declared. You can declare a name in two ways.

- explicitly, by including it in a `declare` statement
- contextually, by using it as a label prefix

You can declare a name only once, explicitly or contextually, within a block unless it is the name of a structure member. You can redeclare the names of structure members within a block provided that you follow two rules.

- No two immediate members of the same structure can have the same name.
- A name cannot be declared more than once as a nonmember within a block.

Each declaration of a name has a scope. A *scope* is a region of the program in which a reference to a name is associated with a particular declaration. A declaration can have either internal scope or external scope.

If a declaration has *internal scope*, its scope includes the block in which it is declared and all blocks contained within that block, except those blocks in which the same name is redeclared.

A declaration with *external scope* has the same scope as an internal declaration except that all declarations in the program of that name that include the `external` attribute refer to the same storage. Only file constants, static variables, and external entry points can have external scope. File constants and external entry points acquire external scope by default. Static variables have internal scope unless you explicitly specify the `external` attribute in the variable declaration.

See Chapter 6 for a discussion of storage classes. See Chapter 3 for a discussion of block structure and scope.

Label Prefixes

A name is contextually declared when it appears as the label prefix of a PL/I language statement. Label prefixes have the following form.

```
label_prefix:  statement
```

Often the label prefix is written on the line before the statement, as shown in the following example.

```
label_prefix:
     statement
```

The following types of names are declared contextually.

- procedure names
- entry point names
- format names
- statement labels

The type of a contextually declared name is determined by the statement to which the label prefix is affixed. The following statements are four examples of contextual declarations.

```
list_titles:    procedure;

second:         entry(rate,total) returns(bit(1) aligned);

record_form:    format(e(4), x(3), a);

TOP:
     read file(f) into(emp_record);
```

A name contextually declared within a block **cannot** also be explicitly declared in a `declare` statement within that same block—except that the name can be reused as the name of a structure member. Every nonmember name is explicitly or contextually declared exactly once.

The next four sections discuss the following topics.

- "Procedure Names"
- "Entry Point Names"
- "Format Names"
- "Statement Labels"

Procedure Names

A name used as a label prefix on a `procedure` statement is contextually declared to be a procedure name. The name is declared within the block that **contains** the `procedure` statement, **not** the block initiated by the `procedure` statement. Procedure names cannot be subscripted.

Every `procedure` statement must have a label prefix.

The contextual declaration includes a description of each parameter referenced by the `procedure` statement and, if the procedure is a function, a description of the data returned by the procedure.

The following example contains two contextual declarations of procedure names.

```
a:     procedure;
            .
            .
            .
     get_address:   procedure(a,b) returns(pointer);
     declare    a     fixed bin(15);
     declare    b     float dec(7);
            .
            .
            .
     end get_address;
end a;
```

In the preceding example, `get_address` is contextually declared as a procedure name within procedure `a`. The attributes of `get_address` are as follows:

- `internal`
- `entry(fixed bin(15), float dec(7))`
- `returns(pointer)`

An *imaginary block* encompasses all external procedures in a source module. The name of an external procedure, such as `a` in the previous example, is declared within this imaginary block. Within a source module, no two external procedures can have the same name.

> **Note:** Names of all external procedures that are part of other source modules must be declared, with the `entry` attribute, in a `declare` statement if they are to be referenced in the current source module. Only **external** procedure names from other source modules can be declared in `declare` statements. Therefore, the `external` attribute need not be specified.

In the following example, the names of two OpenVOS-supplied procedures are declared.

```
a:     procedure;

declare    hash entry (char(*) varying, fixed bin(15))
                          returns (fixed bin(15));
declare    unhex entry (char(*) varying, fixed bin(31),
                          fixed bin(15));
```

See Chapter 13 for information on OpenVOS-supplied functions. The `declare` statement is discussed later in this chapter.

Entry Point Names

A name used as a label prefix on an `entry` statement is contextually declared to be an entry point constant. The name is not declared within the procedure that contains the `entry` statement, but within the block that contains that procedure. Entry-point constant names cannot be subscripted.

Every `entry` statement must have a label prefix.

Entry point constants are similar to procedure names; in fact, a procedure name is a special type of entry point constant. The contextual declaration of an entry point constant includes a description of each parameter referenced by the `entry` statement. If the procedure containing the entry point is a function, the contextual declaration also includes a description of the data to be returned by the procedure.

The following example contains an internal procedure with two entry points.

```
a:     procedure;
declare   error_code     fixed bin(15);
declare   error_message  char(256) varying;

     if error_code = 0
     then error_message = get_default_message();
     else error_message = get_message(error_code);

     get_message: procedure(p_code) returns(char(256) varying);
     declare   p_code    fixed bin(15);
     declare   message   char(256) varying;
               .
               .
               .
     get_default_message: entry returns(char(256) varying);
               .
               .
               .
     return(message);
     end get_message;

end a;
```

In the preceding example, `get_default_message` is contextually declared as an entry point to the procedure `get_message`. The entry point name is declared within procedure `a`. The attributes of `get_default_message` are `entry` and `returns(char(256) varying)`.

If **any** entry point to a procedure has the `returns` option, **all** entry points must have the `returns` option. The data types specified in the `returns` options need not be identical. However, the value returned by each `return` statement immediately contained within the procedure must be convertible to the data type specified in the `returns` option of each entry point.

All entry points to external procedures are declared within the imaginary block that encompasses all external procedures in the source module. Within a source module, no two entry points to external procedures can have the same name.

> **Note:** Names of all external entry points that are part of other source modules must be declared in `declare` statements if they are to be referenced in the current source module. Only **external** entry points from other source modules and entry variables can be declared in `declare` statements. Therefore, the `external` attribute need not be specified.

The following example includes a declaration of an entry point from another source module and a declaration of an internal entry variable.

```
a:     procedure;

declare   s$write         entry (char(*) varying);
declare   next_entry      entry variable;
```

Entry data is discussed in Chapter 4. The `declare` statement is discussed later in this chapter.

Format Names

A name used as the label prefix on a `format` statement is contextually declared as a format name. The name is declared in the block that contains the `format` statement. Format names cannot be subscripted.

Every `format` statement must have a label prefix. In the following example, the name `f` is contextually declared as a format name.

```
a:     procedure;

f:     format(e(12), x(4), f(7,2), skip, a);
      .
      .
      .
end a;
```

A format name is not a statement label and cannot be referenced in a `goto` statement. It can only be referenced in an `r` format item in the format list of a `get` or `put` statement.

```
rec_format:   format(a, e(7,2), f(6));
          .
          .
          .
     get edit(rec_number, in_record)(f(4), r(rec_format));
```

Chapter 14 describes the use of formats.

Statement Labels

A name used as a label prefix on any statement other than a `procedure`, `entry`, or `format` statement is contextually declared as a statement label constant. The declaration is established in the block that contains the statement to which the label prefix is attached.

In the following example, the names `START_LOOP` and `CLEAN_UP` are contextually declared as statement label constants within procedure `a`.

```
a:    procedure;

     on reenter
          goto START_LOOP;

START_LOOP:
     do while(request ^= 'stop');
                    .
                    .
                    .
          if error_code ^= 0
          then goto CLEAN_UP;
end; /* End of do-loop */

CLEAN_UP:
     call cleaner;
     return;

end a;
```

Statement labels are often typed in uppercase for visibility.

Statement labels cannot be attached to `declare` statements; a `declare` statement with a label prefix is not valid. Similarly, statement labels cannot be attached to PL/I preprocessor statements such as the `%include` statement.

A statement label attached to a `begin` statement is declared within the block that **contains** the `begin` statement, **not** within the begin block initiated by the `begin` statement.

> **Note:** Statement labels are **not** the same as entry points. A reference to a statement label can refer only to a statement within a stack frame that is already on the stack. An invocation of an entry point pushes a new stack frame onto the stack. See the discussions of label and entry data in Chapter 4 for more information.

Statement labels can be subscripted by a single, optionally signed, integer constant *k*, where `-32767 <= k <= 32767`. If a label is subscripted, all occurrences of that label name within the block must be subscripted. All such label prefixes collectively constitute a declaration of the name as an array of statement label constants.

The following example includes a contextual declaration of an array of statement label constants.

```
          goto CASE(k);

     CASE(1):
               .
               .
               .
     CASE(2):
               .
               .
               .
     CASE(3):
               .
               .
               .
     CASE(*):            /* Default label */
               .
               .
               .
```

In the preceding example, `CASE` is contextually declared as an array of statement label constants. The bounds of the array are `(1:3)`.

An array of statement label constants cannot be referenced as an array value. You can use an element of an array of statement label constants in any context that permits a statement label.

See Chapter 4 for more information on label data.

The `declare` Statement

The `declare` statement explicitly declares one or more names and specifies the attributes of the objects identified by those names. The names are declared in the block that contains the `declare` statement.

The `declare` statement is not an executable statement and cannot have a label prefix. A `declare` statement can appear anywhere within a procedure or begin block, except as the `then` clause or `else` clause of an `if` statement or as the on-unit of an `on` statement. A variable's declaration need not precede its first reference in the program.

Commonly, all `declare` statements in a block appear at the beginning of that block. Mixing `declare` statements with executable statements, while not incorrect, can make the program difficult to read.

The `declare` statement can take many forms. The following section, "Common Forms of the `declare` Statement," discusses three simple forms of the `declare` statement. Usually, only these three forms are needed. A subsequent section, "General Form of the `declare` Statement," discusses the other forms of the `declare` statement.

Common Forms of the declare Statement

Three simple forms of the declare statement are commonly used. These three forms allow you to do the following:

- declare a single name
- declare several names with identical attributes
- declare a structure

You can use any of the three forms to declare arrays.

The next three sections discuss the following topics.

- "Declaring a Single Name"
- "Declaring Multiple Names"
- "Declaring Structures"

Declaring a Single Name

The following format declares a single name with its list of attributes.

```
declare name [attribute] ...;
```

In the preceding format:

- *name* is the name being declared.
- *attribute* is an attribute of the object identified by *name*.

Each of the following declare statements declares a single name.

```
declare   a    fixed bin(15);
declare   b    char(10) varying static initial('abc');
declare   c    dimension(5) float dec(7);
```

The last of the preceding examples declares an array. You could also declare an array as shown in the following example.

```
declare   c(5) float dec(7);
```

The bounds of the array must be the first attribute in the attribute list if they appear without the keyword dimension.

Declaring Multiple Names

The following format declares several names to have the same attributes.

```
declare (name, name ...)[attribute] ...;
```

In the preceding format:

- *name* is a name being declared.
- *attribute* is an attribute to be assigned to each *name*.

Each of the following `declare` statements declares more than one name.

```
declare   (a,b,c)   fixed bin(15);
declare   (p,q)     pointer static initial(null());
declare   (x,y)(5)  float dec(7);
```

In the last of the preceding examples, `x` and `y` are each declared to be one-dimensional arrays. Each array consists of five floating-point decimal values. You could also declare this example as shown in the following example.

```
declare   (x,y)     dimension(5) float dec(7);
```

Declaring Structures

The following format declares a structure.

```
declare 1  structure_name [attribute] ...,
     [level_number name [attribute] ...,] ...
     level_number name [attribute] ...;
```

In the preceding format:

- *structure_name* is the name of the structure being declared.
- *attribute* is an attribute specified for a structure or structure member. Structure attributes are limited to storage class and the `dimension` attribute.
- *level_number* is the level number of the structure member. Each member must have a level number that is greater than the level number of its containing structure.
- *name* is the name of a structure member. A structure member can itself be a structure.

The following `declare` statement declares a structure.

```
declare   1  s          static,
            2  a(5)      float dec(7),
            2  b         fixed bin(15),
            2  c         ,
              3  d       pointer,
              3  e       char(10) initial('abc');
```

In the preceding example, `s` is declared to be a static structure with members `a`, `b`, and `c`. Member `c` is itself a substructure with members `d` and `e`.

The storage class of a structure applies to each member of the structure. You can specify the `initial` attribute for an elementary member of a static structure, but not for a structure itself, as the preceding example illustrates.

The following example declares an array of three structures.

```
declare   1  struct(3)     ,
            2  code        fixed bin(15),
            2  text        char(256) varying;
```

You can use the `like` attribute to simplify declarations of two or more structures containing similar constructs. The `like` attribute is described in "Attribute Reference Guide" later in this chapter.

General Form of the `declare` Statement

The `declare` statement has the following general form.

```
declare decl_string [,decl_string] ...;
```

Each *decl_string* has the following syntax.

```
              ⎧                   name                     ⎫
[level_number]⎨                                             ⎬
              ⎩ (decl_string [,decl_string]...)             ⎭
[attribute] ...
```

In the preceding format:

- *level_number* is the level number of the name; this element is rarely used except in structure declarations.
- *name* is a name being declared.
- *attribute* is an attribute specified for the name.

The following examples are complex forms of the `declare` statement.

```
declare   ((a fixed, b float) decimal, c bit) static;
declare   1 s static, 2 (d fixed, e float) initial(0);
```

A `declare` statement that contains a list of names in parentheses is called a *factorized declaration*. You can convert a factorized declaration to a defactorized declaration by the following process.

1. Copy the level number and attribute list assigned to the innermost set of parentheses onto each name contained within that set of parentheses. (For example, in the first of the preceding examples, the `decimal` attribute is assigned to both `a` and `b`.)

2. Remove the innermost set of parentheses.

3. Repeat the process for the next innermost set of parentheses.

Defactorizing the previous examples produces the following code.

```
declare   a       fixed decimal static;
declare   b       float decimal static;
declare   c       bit static;

declare 1  s      static,
          2  d    fixed initial(0),
          2  e    float initial(0);
```

If defactorization produces more than one level number for a name—even if the level numbers are equal—the declaration is invalid. The compiler detects such situations and issues an error message.

The specification of duplicate attributes containing more than a simple keyword is also invalid, even if the attributes match exactly.

With the exception of the format described in "Declaring Multiple Names," earlier in this chapter, factorized declarations are usually difficult to read. For this reason, you should limit their use.

Attributes

Attributes are used in `declare`, `procedure`, `entry`, and `open` statements to describe the properties of PL/I objects.

- You can specify attributes in a `declare` statement to specify the data type, storage class, and scope of a declared name. Attributes can also specify whether the named object is a variable or a constant.
- You can specify attributes in a function's `procedure` and `entry` statements to describe the data type of the returned value.
- You can specify file attributes in an `open` statement. Such attributes are assigned to the file control block being opened. You can also specify file attributes in the declaration of a file constant. See Chapter 14 for information on file attributes.

The next two sections discuss the following topics.

- "Default Attributes"
- "Attribute Consistency"

Default Attributes

If a list of specified attributes is incomplete, the compiler adds default attributes. Table 7-1 lists some of the rules the compiler uses for supplying default attributes.

Table 7-1. Default Attributes

Attribute Specified by User	Attribute Not Specified	Default Attribute
`binary` or `decimal`	`fixed` or `float`	`fixed`
`fixed` or `float`	`binary` or `decimal`	`binary`
`static`	`internal` or `external`	`internal`
`external`	Storage class	`static` †
Nonparameter, nonmember variable	Storage class	`automatic`
`bit` or `character`	Length	Length of 1
`file` or `entry` with storage class	N/A	`variable`
`file` or `entry` with array bounds	N/A	`variable`
`file` or `entry` with level number	N/A	`variable`
`file` or `entry`	`variable`, storage class, array bounds, or level number	`constant`

† This attribute applies only to variables, not to entry or file constants.

If no data type is specified in a declaration, that declaration is incomplete. In such a case, the compiler issues a warning message and supplies the attributes `fixed bin(15)`.

If an arithmetic value is declared with no precision, a default precision is supplied. Table 7-2 lists the default precisions for arithmetic data types.

Table 7-2. Default Arithmetic Precisions

Data Type	Default Precision
`fixed binary`	15
`fixed decimal`	9
`float binary`	24
`float decimal`	7

If no scaling factor is specified for a fixed-point decimal value, the default is zero.

Attribute Consistency

After supplying any default attributes, the compiler checks each declaration for consistency and completeness.

- If more than one data type or more than one storage class is specified in a declaration, that declaration is inconsistent and invalid.
- A name declared with the `builtin` attribute cannot have any other attributes.
- A name declared with the `file` or `entry` attribute and without the `variable` attribute is a named constant, not a variable. Its scope is external. If a storage class, array bound, or member's level number is specified, or if the name is a parameter, the attribute `variable` is supplied by default.
- Any declaration that violates any of the restrictions for each attribute as described later in this chapter is also inconsistent and invalid.

Table 7-3 lists the valid data types.

Table 7-3. Valid Data Types

```
fixed binary(precision)

fixed decimal(number_of_digits [,scaling_factor])

float binary(precision [, scaling_factor])

float decimal(precision)

picture

character(length) [varying]
                  [aligned]

bit(length)[aligned]

pointer

label

entry [returns(data_type)] [variable]

file [           variable                ]
     [[file_description_attribute]...    ]

builtin

Structure
```

Table 7-4 lists the valid storage classes.

Table 7-4. Valid Storage Classes

`automatic`
`based` [(*pointer_reference*)]
`static`
`defined` (*reference*)
Parameter
Structure member

Attribute Reference Guide

This section describes each of the attributes that are permitted in a `declare`, `procedure`, `entry`, or `open` statement. The attributes are listed in alphabetical order for easy reference.

These discussions assume that all attribute lists have been made complete by the application of defaults as described in "Default Attributes" earlier in this chapter.

▶ `aligned`

The `aligned` attribute ensures that data begins on a boundary that is appropriate for the alignment rules in effect. An aligned string is always stored in the fewest even number of bytes necessary to hold the declared string length. The `aligned` attribute is an optional part of the data-type specification of a bit string or character string.

When passing a string value by reference to a parameter, you must ensure that **both** the argument and the parameter have the `aligned` attribute or that **neither** has the `aligned` attribute.

Appendix B discusses internal storage. Chapter 4 and Appendix B discuss data alignment. Chapter 4 discusses bit-string data and character-string data. Chapter 3 discusses parameters and arguments.

▶ `automatic`

The `automatic` attribute specifies that a variable has the automatic storage class. Storage for automatic variables is allocated when the containing procedure or begin block is activated, and is freed when the activation terminates. Separate storage is allocated for each call to the procedure.

If no storage class is specified for a variable, the compiler supplies `automatic` by default. Chapter 6 discusses storage classes.

You can abbreviate `automatic` to `auto`.

▶ `based` [(*pointer_reference*)]

The `based` attribute specifies that a variable has the based storage class. You can include a reference to a pointer variable or a reference to a pointer-valued function in the `based` attribute; this reference serves as the default or implicit pointer qualifier for unqualified references to the based variable name. Chapter 6 discusses based storage.

▶ `binary` [`(`*precision* [`,` *scaling_factor*] `)`]

In an arithmetic data-type description, the `binary` attribute specifies that the base is binary. You can state the precision in this attribute or in the `fixed` or `float` attribute, but you cannot specify it more than once. If you do not specify a precision, a default is supplied: 15 for a fixed-point value or 24 for a floating-point value; the default scaling factor is 0. For fixed-point values, the compiler only accepts a scaling factor of 0. You cannot specify a scaling factor for floating-point values.

If specified, the precision must be in the following ranges.

```
fixed binary: 0 < precision <= $MAX_FIXED_BIN
float binary: 0 < precision <= 53
```

The maximum precision that you can specify for `fixed binary` is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see “Fixed-Point Binary Data” in Chapter 4 for more information.

If you specify `binary` without `fixed` or `float`, the default is `fixed`. If you specify either `fixed` or `float` without `binary` or `decimal`, the default is `binary`.

Chapter 4 discusses arithmetic data.

You can abbreviate `binary` to `bin`.

▶ `bit` [`(`*length*`)`]

The `bit` attribute describes a bit-string value. The permissible form of the extent expression, *length*, depends on the storage class of the declared name. If specified, *length* must be in the following range.

```
0 <= length <= 32767
```

The default length is 1.

If the `bit` attribute describes a parameter, you can specify an asterisk (`*`) for *length*. The parameter then takes on the length of the associated argument.

Chapter 6 discusses storage classes. Chapter 4 discusses bit-string data.

▶ `builtin`

The `builtin` attribute declares the name of a PL/I built-in function. Built-in functions need to be declared only in the following cases.

- when the built-in function is referenced with no argument list
- when the built-in function name has been declared as the name of another object in an outer block

The following example illustrates the use of the `builtin` attribute.

```
a:   procedure;
declare    substr    char(12) varying;
declare    date      builtin;

     substr = date;

b:   procedure;
declare    substr    builtin;

     y = substr(x,3,2);

end b;
end a;
```

In the preceding example, any reference to `substr` within procedure `b` refers to the `substr` built-in function. A reference to `substr` within procedure `a`, outside of procedure `b`, refers to a varying-length character string.

Chapter 13 describes the OpenVOS PL/I built-in functions.

▶ `character` [`(`*length*`)`]

The `character` attribute describes a character-string value. The permissible forms of the extent expression, *length*, depend on the storage class of the string. If specified, *length* must be in the following range.

```
0 <= length <= 32767
```

The default length is 1.

See the description of the `varying` attribute, later in this chapter, for more information on how you can use it with the `character` attribute.

If the `character` attribute describes a parameter, you can specify an asterisk `(*)` for *length*. The parameter then assumes the length of the associated argument.

> **Note:** You cannot pass nonconstant (asterisk) extents to C or COBOL programs. In addition, if **any** `string` or `character` argument of a called FORTRAN subprogram takes nonconstant extents, the PL/I calling program must declare **all** `char(`*n*`)` [`varying`] arguments to take nonconstant extents.

Chapter 6 discusses storage classes. Chapter 4 discusses character-string data.

You can abbreviate `character` to `char`.

▶ `condition`

The `condition` attribute is used in a `declare` statement to describe a programmer-defined condition name, as shown in the following example.

```
declare   negative_result      condition;

          on condition(negative_result)
               begin;
                 .
                 .
                 .
               end;
```

Programmer-defined condition names can have external scope. Programmer-defined condition names cannot be arrays or structure members.

Programmer-defined conditions are OpenVOS extensions. See "Programmer-Defined Conditions" in Chapter 15 for more information on programmer-defined conditions.

You can abbreviate `condition` to `cond`.

▶ `decimal` [`(`*`precision`* [`,` *`scaling_factor`*] `)`]

In the description of an arithmetic value, the `decimal` attribute specifies that the base is decimal. You can specify the precision of the value (and, for fixed-point values, *`scaling_factor`*) in the `decimal` attribute or in a `fixed` or `float` attribute. You cannot specify the precision more than once. If specified, *`precision`* must be in the following ranges.

`fixed decimal: 0 <` *`precision`* `<= 18; -18 <=` *`scaling_factor`* `<= 18`
`float decimal: 0 <` *`precision`* `<= 15;` *`scaling_factor`* does not apply

If you omit the precision in the description, the following defaults are supplied.

`fixed decimal:` *`precision`* `= 9;` *`scaling_factor`* `= 0`
`float decimal:` *`precision`* `= 7;` *`scaling_factor`* does not apply

If you specify `decimal` without `fixed` or `float`, the default is `fixed`. If you specify `fixed` or `float` without `decimal` or `binary`, the default is `binary`.

Chapter 4 discusses arithmetic data.

You can abbreviate `decimal` to `dec`.

▶ `defined(`*`variable_reference`*`)`

In a variable declaration, the `defined` attribute specifies that the variable has the defined storage class. Storage for the declared name is shared with the variable referenced in the `defined` attribute. Chapter 6 discusses defined storage and storage sharing.

You can abbreviate `defined` to `def`.

▶ `dimension(`*bounds* [*,bounds*] ...`)`

The `dimension` attribute describes an array. When the `dimension` attribute appears immediately after the array name, you can omit the word `dimension`.

You **cannot** apply the `dimension` attribute to `file` or `entry` constants or to condition names; it can only be specified for variables.

Each extent expression, *bounds*, in the `dimension` attribute specifies the bounds of a dimension of the array. You can specify up to eight *bounds* expressions with the `dimension` attribute (an eight-dimensional array). Each *bounds* expression can take any of the following forms.

- *lbound*:*hbound*
- *hbound*
- `*`

The *lbound* value specifies the lower bound of the dimension. The *hbound* value specifies the upper bound of the dimension. If *hbound* only is specified, the lower bound is assumed to be 1. The asterisk (`*`) can only be used in parameter descriptions. It signifies that **both** the upper and lower bounds of the dimension are to be taken from the corresponding array argument.

The permissible forms of *lbound* and *hbound* depend on the array's storage class. The extents of an automatic, based, or defined array are limited only to integer-valued expressions; the extents of a static array must be integer constants or constant-valued expressions; the extents of an array parameter must be either integer constants, constant-valued expressions, or asterisks. Chapter 6 describes storage classes.

The following example contains two array declarations.

```
declare    x          dimension(5) float bin(53);
declare    y(1:5)     float bin(53) static;
```

In the preceding example, both `x` and `y` are one-dimensional arrays of five floating-point values.

Chapter 4 discusses array data.

You can abbreviate `dimension` to `dim`.

▶ `direct`

When applied to a file opening, the `direct` file attribute specifies that records in the file are accessed by their ordinal position in the file.

You can specify the `direct` file attribute in the declaration of a file constant or in an `open` statement; you **cannot** specify file attributes in the declaration of a file variable. If you specify the `direct` file attribute in the declaration of a file constant, it applies to all openings of the file control block associated with that file constant.

If you specify the `direct` file attribute, the `record` and `keyed` file attributes are supplied automatically. See "Attributes" earlier in this chapter, and Chapter 14 for more information on file attributes.

► `entry [(parameter_descriptor [,parameter_descriptor] ...)]`

If the `entry` attribute appears in a declaration without the `variable` attribute, the declared name is an entry point to an external procedure from another source module. In that case, each *`parameter_descriptor`* must be a set of attributes that is identical to the set of attributes specified for the corresponding parameter at that entry point. Entry points are `entry` statements or `procedure` statements.

If you specify either the `variable` attribute or a storage-class attribute with the `entry` attribute in a `declare` statement, the declared name is an entry variable.

If the `entry` attribute appears in a parameter description, `returns` option, or `returns` attribute, it specifies that a procedure requires or returns an entry value.

If a name declared with the `entry` attribute is a parameter, the parameter storage class is implied; thus, the `variable` attribute is implied.

If one or more of the parameters in a `procedure` or `entry` statement are structures, the `entry` attribute used in declaring the entry point name must have a parameter descriptor for each member of each structure, including all substructures. You can use the `like` attribute to simplify structure descriptions.

Any string lengths or array bounds specified in an `entry` attribute must be identical to those specified in the parameters of the corresponding `procedure` or `entry` statement. Programs that violate this rule produce unpredictable results.

> **Note:** The parameter descriptors in an `entry` attribute can contain only data description attributes. The following attributes are **not** allowed within a parameter descriptor: `external`, `internal`, `builtin`, `condition`, `variable`, and all storage-class attributes.

Chapter 4 discusses entry data.

► `external`

The `external` attribute specifies that the declared name has external scope. If the same name is declared more than once within a program, all declarations of the name that contain the `external` attribute identify the same object.

You can use the `external` attribute only in the declarations of static variables or `file` or `entry` constants. If you specify `external` for a variable without specifying a storage class, `static` is supplied by default.

All `external` declarations of a particular name within a program must have exactly the same attributes, including any `initial` attribute.

You can abbreviate `external` to `ext`.

▶ `file`

The `file` attribute specifies that the declared name is either a file constant or a file variable.

If you specify the `file` attribute without the `variable` attribute, the declared name is a file constant. File constants have external scope.

Each file constant has an associated file control block. You can use a file control block to perform input and output on files and devices. The declaration of a file constant can include the `external` attribute and the file attributes: `record`, `stream`, `input`, `output`, `update`, `keyed`, `direct`, `sequential`, and `print`. When the file is opened, any file attributes specified in the declaration are merged with attributes supplied by the `open` statement or with attributes implied by the I/O statement that implicitly opens the file control block.

If you specify the `variable` attribute with the `file` attribute in a `declare` statement, the declared name is a file variable. Likewise, if you specify a storage-class attribute with the `file` attribute in a `declare` statement, the declared name is a file variable. A file variable can be assigned file values. A file value is the address of a file control block.

You **cannot** supply file attributes in the declaration of a file variable. File attributes are associated with file control blocks; they cannot be applied to a file variable.

If the `file` attribute appears in a parameter description, `returns` option, or `returns` attribute, it specifies that a procedure requires or returns a file value.

If a name declared with the `file` attribute is a parameter, the parameter storage class is implied; thus, the `variable` attribute is implied.

Chapter 14 discusses PL/I input and output.

▶ `fixed [(`*`number_of_digits`*` [,`*`scaling_factor`*`])]`

The `fixed` attribute specifies that the scale of an arithmetic value is fixed-point.

You can specify the precision (*`number_of_digits`* and, if the base is decimal, *`scaling_factor`*) in the `fixed` attribute or in a `binary` or `decimal` attribute, but you cannot specify the precision more than once. If you do not specify a precision, one of the following default values is supplied.

```
fixed decimal:  (9,0)
fixed binary:   (15)
```

If you specify `fixed` without `binary` or `decimal`, the default is `binary`. If you specify `binary` or `decimal` without `fixed` or `float`, the default is `fixed`.

Chapter 4 discusses arithmetic data.

▶ `float` [`(`*precision*`)`]

The `float` attribute specifies that the scale of an arithmetic value is floating-point. You can specify the precision in the `float` attribute or in a `binary` or `decimal` attribute, but you cannot specify the precision more than once. If you do not specify the precision, one of the following default values is supplied.

```
float decimal: (7)
float binary:  (24)
```

If you specify `float` without `binary` or `decimal`, the default is `binary`. If you specify `binary` or `decimal` without `fixed` or `float`, the default is `fixed`.

Chapter 4 discusses arithmetic data.

▶ `in`

The `in` attribute specifies that the parameter is input-only. You can specify the `in` attribute as part of a parameter description in an `entry` attribute or in a parameter declaration. If a called procedure, or any of its descendants, attempts to store a value in the storage associated with the parameter, a run-time error might result.

The following example illustrates a use of the `in` attribute.

```
declare s$error entry(fixed bin(15) in, char(*) varying in,
                      char(*) varying in);
```

If you include the `in` attribute in the parameter declarations in a called procedure, the compiler detects any attempt to modify the parameter. If the compiler detects such an attempt, it issues a severity-2 error message.

The following example declares an input-only parameter.

```
declare   p_in      char(*) varying in;
```

When you use the `in` attribute, the storage of the corresponding argument is **not** copied when passed by value. This saves code and stack space when you use constant arguments (particularly string constants).

> **Note:** The `in` attribute is an OpenVOS extension; using it makes your program implementation-dependent.

▶ `initial(`[`(`*factor*`)`] *value* [`,`[`(`*factor*`)`] *value*]`...)`

The `initial` attribute assigns an initial value to a variable or an array of variables. You can specify the `initial` attribute only for arithmetic, pictured, string, or pointer variables that have the static storage class or are members of a static structure.

An initial value can take any of the following forms.

- an optionally signed arithmetic constant
- a character-string constant
- a bit-string constant
- a reference to the `null` built-in function

Each factor is an integer constant indicating the number of times the value following it is to be used. Note that factors are always enclosed in parentheses. If a factor is used with a character-string value or bit-string value, the string value must also be enclosed in parentheses.

The following example demonstrates several uses of the `initial` attribute.

```
declare sum         fixed bin(15) static initial(0);

declare   x(8)      fixed dec(7,2) static
                    initial(1, 10, 5.2, (5)0);

declare   str(3)    char(2) static initial('al',(2)('nl'));
declare   z         pointer static initial(null());

declare 1  struct(2) static,
          2  average  float bin(24) initial(.302, .304),
          2  name     char(7) varying initial('Mays', 'Ott');
```

You can specify initial values for all members of each element of an array of structures. In the preceding example:

- the value `.302` is assigned to `struct(1).average`
- the value `.304` is assigned to `struct(2).average`
- the value `'Mays'` is assigned to `struct(1).name`
- the value `'Ott'` is assigned to `struct(2).name`

You can specify more than one initial value only if the declared name is an array. In that case, you must specify exactly one value for each element in the array. The values are assigned to the array in row-major order. Note that the following declaration is **not** allowed because it does not provide an initial value for each element in the array.

```
declare switch(8)     bit(1) aligned static initial('0'b);
```

Chapter 4 discusses arrays and row-major order.

The initial values must be constants that can be converted to the data type of the associated variable. The initial values are converted and assigned to the variables prior to program execution. Each converted value must not exceed 2048 bytes.

You can specify the `null` built-in function as an initial value for pointer variables. If you use the `null` built-in function in this way, the name `null` must not have been declared as anything other than the `null` built-in function in the current block or any outer block.

You can abbreviate `initial` to `init`.

▶ `input`

The `input` file attribute specifies that the file is read-only.

You can specify the `input` file attribute in the declaration of a file constant or in an `open` statement; you **cannot** specify the `input` file attribute in the declaration of a file variable. Any file attributes specified in the declaration of a file constant apply to all openings of the file control block associated with that file constant.

Every time a file control block is opened, it must have either the `input`, `output`, or `update` file attribute; the default is `input`. A file control block opened with the `input` file attribute is open for reading only. See "Attributes" earlier in this chapter, and Chapter 14 for more information on file attributes.

- ▶ `internal`

The `internal` attribute specifies that the declared name has internal scope.

You can use the `internal` attribute in the declarations of variables with any storage class or in the declaration of parameters. Because all variables and parameters acquire internal scope by default, the `internal` attribute is usually omitted.

If you specify the `static` attribute without specifying the `internal` or `external` attribute, the default is `internal`.

You can abbreviate `internal` to `int`.

- ▶ `keyed`

The `keyed` file attribute specifies a file's access.

You can specify the `keyed` file attribute in the declaration of a file constant or in an `open` statement; you **cannot** specify the `keyed` file attribute in the declaration of a file variable. If you specify a file attribute, such as `keyed`, in a file constant declaration, the attribute applies to all openings of the file control block associated with that file constant.

If a file control block is opened with the `keyed` and `direct` file attributes, the file is open for direct access; if the `direct` file attribute is not specified, the file is open for keyed sequential access.

Whenever you specify the `keyed` file attribute, the compiler supplies the `record` file attribute automatically. See "Attributes" earlier in this chapter, and Chapter 14 for more information on file attributes.

- ▶ `label`

The `label` attribute specifies that the declared name is a label. The `label` attribute can appear in the following contexts.

- in a `declare` statement, to describe a label variable
- in an `entry` attribute, to describe a label-valued parameter
- in a `returns` attribute or `returns` option, to describe a returned label value

Chapter 4 discusses label data.

- ▶ `like` *`unsubscripted_structure_reference`*

The `like` attribute declares a structure whose members are identical to the members of another structure.

The reference within the `like` attribute must be a simple reference to the name of a structure declared in the current block or in a containing block. This referenced structure acts as a template. The logical structure of the template is copied into the current declaration; any storage class or other level-one attributes of the template structure are ignored. If you do not specify the storage class in addition to the `like` attribute, the default is automatic.

The following example illustrates a simple use of the `like` attribute.

```
declare 1 common_a       based,
          2 first        fixed bin(15),
          2 second       ,
            3 a          fixed bin(31),
            3 b          char(24) varying;

declare 1 struct1        ,
          2 record1      like common_a,
          2 size         fixed bin(15);

declare 1 struct2        static,
          2 rate         float bin(24),
          2 long_rec     ,
            3 size       fixed bin(31),
            3 stand_rec  like common_a;
```

The declaration of `struct1` from the preceding example is expanded as shown in the following example.

```
declare   1 struct1          ,
            2 record1        ,
              3 first        fixed bin(15),
              3 second       ,
                4  a         fixed bin(31),
                4  b         char(24) varying,
            2 size           fixed bin(15);
```

You **cannot** declare additional members of the structure at levels defined by the template substructure. For example, the following declaration is not allowed.

```
declare   1 struct3          ,
            2 record         like common_a,
              3 flags        bit(8) aligned;
```

You could, of course, append additional level-two items to the structure.

The declaration of the template structure can itself contain the `like` attribute, as shown in the following example.

```
declare 1 temp_structure     ,
          2 a                fixed bin(15),
          2 b                ,
            3  one           char(5),
            3  two           picture '-zzz9.99',
          2 c                bit(8);

declare 1 f_record           static,
          2 label            like temp_structure,
          2 data             char(64) varying;

declare 1 g_record           ,
          2 parta            ,
            3 x              like temp_structure,
            3 y              like f_record,
          2 partb            char(64) varying;
```

In the preceding example, because the `like` attribute copies the declarations of the members of the template structure into the current declaration, the members of `f_record` and `g_record.parta.y` are identical in name and data type.

Unless explicitly declared, a structure declared with the `like` attribute inherits the data alignment attributes of the original template. In the following example, the structure `template` is declared with the `longmap` attribute. The structure `object1` has two members that are of the same type as `length` and `size`, but the declaration of `object1` specifies the `shortmap` attribute, which overrides the `longmap` attribute specified in `template`. The last structure, `object2`, is identical to `template` in the size, type, and alignment of its members.

```
declare   1 template         longmap,
            2 length         fixed bin(15),
            2 size           fixed bin(31);

declare   1  object1         shortmap like template;

declare   1  object2         like template;
```

You **cannot** use the `like` attribute in two structures to refer to each other. Therefore, the following example is invalid.

```
declare   1  type_a      ,
            2  x         fixed bin(15),
            2  record_a  like type_b;

declare   1  type_b      ,
            2  record_b  like type_a;
```

Such mutually recursive structures cause a compile-time error.

You can use the `like` attribute to describe a structure parameter in the declaration of an entry point to an external procedure. The following example declares an entry point that takes two arguments: a structure and a 2-byte integer.

```
declare struct_count entry(1 like struct1, fixed bin(15));
```

All operations allowed on structures are supported for structures declared using the `like` attribute.

You can pass arguments declared using the `like` attribute to routines written in other OpenVOS languages provided you can legally pass the equivalent structure without the `like` attribute.

> **Note:** The `like` attribute is part of full PL/I, but not part of PL/I Subset G. Therefore, using the `like` attribute makes a program implementation-dependent. Furthermore, the OpenVOS PL/I implementation expands the standard `like` attribute. Using `like` to declare parameters or to describe parameters within entry declarations is nonstandard. Using a template structure that is itself declared with the `like` attribute is also nonstandard.

▶ `longmap`

The `longmap` attribute specifies the longmap data alignment rules for a PL/I structure or any scalar type. A structure is aligned according to the maximum alignment required by its members. The following example shows a structure that is aligned using longmap alignment rules.

```
declare 1 struct    longmap,
          2 type    fixed bin(15),
          2 value   float dec(8),
          2 size    fixed bin(31);
```

In the preceding example, the structure is aligned on a mod8 boundary because the largest member, `struct.value`, requires a mod8 boundary. Each member is aligned on the boundary required for its data type. The size of the entire structure is 20 bytes; there are 6 bytes of padding between `struct.type` and `struct.value`. The members are stored as shown in the following table.

Element	**Alignment**	**Byte Offset**	**Size (in Bytes)**
`struct`	mod2	0	20 bytes
`struct.type`	mod2	0	2 bytes
Alignment padding	N/A	2	6 bytes
`struct.value`	mod8	8	8 bytes
`struct.size`	mod4	16	4 bytes

The `longmap` attribute is an OpenVOS extension. For additional information about data alignment, see Chapter 4.

▶ `options(c)`

The `options(c)` attribute declares entry points to C language functions and subroutines, which expect argument values to be pushed on the stack rather than being passed by reference.

You can specify this attribute only in the declaration of an external entry point. The effect of the `options(c)` attribute is that arithmetic, varying-length character-string, and structure arguments are passed by value; array, bit-string, and nonvarying character-string arguments are passed by reference.

See the *VOS PL/I User's Guide* (R145) for more information on calling C routines from OpenVOS PL/I programs.

▶ `output`

The `output` file attribute specifies that a file is write-only.

You can specify the `output` file attribute in the declaration of a file constant or in an `open` statement; you **cannot** specify the `output` file attribute in the declaration of a file variable.

Any file attributes specified in the declaration of a file constant apply to all openings of the file control block associated with that file constant.

Every time a file control block is opened, it must have either the `input`, `output`, or `update` file attribute; the default is `input`. A file control block opened with the `output` file attribute is opened for writing only.

See "Attributes" earlier in this chapter, and Chapter 14 for more information on file attributes.

▶ `picture '`*`picture_description`*`'`

The `picture` attribute describes pictured data. The picture description serves as a template for how a value is to be edited before being assigned to the variable. The picture description also governs conversion of the pictured value to a fixed-point decimal value.

Pictured values are stored as character strings, but you can operate on them as if they were fixed-point decimal values.

Chapter 4 discusses pictured data.

You can abbreviate `picture` to `pic`.

▶ `pointer`

The `pointer` attribute describes pointer data. A pointer value is a storage address.

Chapter 4 discusses pointer data. Chapter 6 describes how to use pointers with based variables.

You can abbreviate `pointer` to `ptr`.

- `print`

  The `print` file attribute creates a special kind of stream output file that is formatted so that it can be spooled to a printer.

  You can specify the `print` file attribute in the declaration of a file constant or in an `open` statement; you **cannot** specify the `print` file attribute in the declaration of a file variable.

  Any file attributes specified in the declaration of a file constant apply to all openings of the file control block associated with that file constant.

  A file control block opened with the `print` file attribute is automatically given the `stream` and `output` file attributes. Chapter 14 discusses the file attributes and print files.

- `record`

  The `record` file attribute opens a file control block for record I/O.

  You can specify the `record` file attribute in the declaration of a file constant or in an `open` statement; you **cannot** specify the `record` file attribute in the declaration of a file variable.

  Any file attributes specified in the declaration of a file constant apply to all openings of the file control block associated with that file constant.

  Every time a file control block is opened, it must have either the `record` or `stream` file attribute. The default is `stream`. See "Attributes" earlier in this chapter, and Chapter 14 for more information on file attributes.

- `returns(`*`attribute_list`*`)`

  The `returns` attribute designates the data type of the value returned by a function.

  The `returns` attribute is part of the entry data-type specification of a function entry point.

  The attribute list of the `returns` attribute can include only data-type attributes. Any string length specified in a `returns` attribute must be an integer constant. Because functions cannot return arrays or structures, you cannot specify level numbers or the `dimension` attribute in the `returns` attribute.

- `sequential`

  The `sequential` file attribute specifies that records in the file being opened will be accessed sequentially.

  You can specify the `sequential` file attribute in the declaration of a file constant or in an `open` statement; you **cannot** specify the `sequential` file attribute in the declaration of a file variable.

  Any file attributes specified in the declaration of a file constant apply to all openings of the file control block associated with that file constant.

Every time you open a file control block with the `record` file attribute, that control block must have either the `sequential` or `direct` file attribute. The default is `sequential`.

If the `sequential` file attribute is specified, the compiler supplies the `record` file attribute automatically. See "Attributes" earlier in this chapter, and Chapter 14 for more information on file attributes.

You can abbreviate `sequential` to `seql`.

▶ `shared`

The `shared` attribute specifies that the declared variable is to be shared among all tasks in a tasking environment.

You can specify the `shared` attribute only in the declaration of an external static variable.

Any variable declared without the `shared` attribute is allocated on a per-task basis, which means that each task sees a separate instance of each external variable.

For information on OpenVOS PL/I tasking, see the *OpenVOS PL/I Transaction Processing Facility Reference Manual* (R015).

> **Note:** Because the `shared` attribute is an OpenVOS extension, it makes your program implementation-dependent.

▶ `shortmap`

The `shortmap` attribute specifies the shortmap data alignment rules for a PL/I structure or any scalar data type. Shortmap alignment means that most data items, except characters and bit strings, begin on an even-byte boundary. The following example shows a structure that is aligned using shortmap alignment rules.

```
declare 1 struct    shortmap,
          2 type    fixed bin(15),
          2 value   float dec(8),
          2 size    fixed bin(31);
```

The structure is aligned on a mod2 boundary. The size of the entire structure is 14 bytes. The members are stored as shown in the following table.

Element	Alignment	Byte Offset	Size (in Bytes)
`struct`	mod2	0	14 bytes
`struct.type`	mod2	0	2 bytes
`struct.value`	mod2	2	8 bytes
`struct.size`	mod2	10	4 bytes

The `shortmap` attribute is an OpenVOS extension. For additional information about data alignment, see Chapter 4.

▶ `static`

The `static` attribute specifies that the declared name has the static storage class. Space for static objects is allocated within the program module prior to program execution.

If no storage class is specified for a nonparameter variable, the default is `automatic`. If the `static` attribute is specified without a scope attribute (`internal` or `external`), the default is `internal`.

Chapter 6 discusses storage classes.

▶ `stream`

The `stream` file attribute opens a file control block for stream I/O.

You can specify the `stream` file attribute in the declaration of a file constant or in an `open` statement; you **cannot** specify the `stream` file attribute in the declaration of a file variable.

Any file attributes specified in the declaration of a file constant apply to all openings of the file control block associated with that file constant.

Every time a file control block is opened, it must have either the `stream` or `record` file attribute. The default is `stream`. A file control block opened with the `stream` file attribute is opened for stream I/O. See "Attributes" earlier in this chapter, and Chapter 14 for more information on file attributes.

▶ `type (`*`reference`*`)`

The `type` attribute enables a declaration to reuse the declaration of another variable. The referenced variable cannot have an argument list, a subscript list, or a pointer-qualification, nor can it reference itself, either directly or indirectly.

The `type` attribute is an OpenVOS extension.

A variable declared with the `type` attribute inherits the size and most of the attributes of the referenced variable. Array dimensions and parameter descriptors are also inherited. Note that the `internal`, `external`, `builtin`, `automatic`, `static`, `based`, and `defined` attributes are not inherited.

If a typed variable is declared with a different mapping attribute than the referenced variable, the mapping attribute declared for the typed variable overrides the one declared for the referenced variable, and the compiler issues a warning.

Consider the following declarations using the `type` attribute. The variable `a` inherits the size and the `fixed` attribute from the variable `INTEGER`, but `a` does **not** inherit the `based` attribute.

```
declare INTEGER          based fixed bin(31);
declare a                type (INTEGER);
```

In the following examples, the variables `b` and `c` both inherit the size (`bin(15)`) and the `fixed` attribute of the variable `SHORT_INT`. Variable `b` also inherits the `longmap` attribute; the declaration of variable `c` specifies the `shortmap` attribute, which overrides the `longmap` specification in the declaration of `SHORT_INT`. In this case, the compiler issues a warning that the mapping attributes of `c` and `SHORT_INT` do not match.

```
declare SHORT_INT         fixed bin(15) longmap;
declare b                 type (SHORT_INT);
declare c                 type (SHORT_INT) shortmap;
```

Chapter 4 describes the use of the `type` attribute.

- ▶ `update`

  The `update` file attribute specifies that records in the file being opened can be read, written, rewritten, or deleted.

  You can specify the `update` file attribute in the declaration of a file constant or in an `open` statement; you **cannot** specify the `update` file attribute in the declaration of a file variable.

  Any file attributes specified in the declaration of a file constant apply to all openings of the file control block associated with that file constant.

  Every time a file control block is opened it must have the `input`, `output`, or `update` file attribute. The default is `input`.

  If you specify the `update` file attribute, the `record` file attribute is supplied automatically. See "Attributes" earlier in this chapter, and Chapter 14 for more information on file attributes.

- ▶ `variable`

  The `variable` attribute specifies that the declared name is a file variable or entry variable rather than a named constant.

  You can specify the `variable` attribute in the declarations of most data types. The `variable` attribute is especially useful in file or entry declarations because these data types default to constants, while other data types default to variables.

  Chapter 4 discusses file and entry data.

- ▶ `varying`

  You specify the `varying` attribute with the `character` attribute to describe varying-length character-string data. Varying-length character strings can accept values having any length that does not exceed the length specified in the `character` attribute.

  Chapter 4 discusses character-string data.

  You can abbreviate `varying` to `var`.

▶ `volatile`

The `volatile` attribute addresses the possibility that the value of a variable might change in a way that the compiler cannot detect. This situation can occur when a program asynchronously modifies the value of a variable, such as when an on-unit for the `break` condition (which is asynchronously signaled) changes a variable, or when multiple processes share and modify the contents of a region of virtual memory. If the variable is not declared as `volatile` and has been asynchronously modified, the compiler does not guarantee that code referencing that variable uses its current value; instead, the compiler may remove redundant references to the variable when it optimizes the code.

If you declare a variable with the `volatile` attribute, the compiler will not optimize expressions containing references to that variable, and it will not keep the values of such variables in registers between references. If a structure is given the `volatile` attribute, that attribute applies to all members of the structure.

The use of the `volatile` attribute does not guarantee access to a shared variable. See the *VOS Reference Manual* (R002) and the *OpenVOS PL/I Subroutines Manual* (R005) for more information on shared virtual memory.

The `volatile` attribute is an OpenVOS extension.

Chapter 8: References

This chapter discusses the following topics related to references.

Overview

A *reference* is the use of a name within any statement other than the declaration of that name. A reference includes any subscripts, pointer-qualifiers, or structure names necessary to uniquely identify a particular object. A reference to an entry point can include an argument list.

The following examples are references.

```
x
y(5,k)
p->s.a(k)
f(z*5+b, sqrt(z))
q.nextp->node.field1
```

Every reference is associated with a declaration. The process of determining which declaration a reference is associated with is called *reference resolution*. All references are resolved during the compilation of the program. The rules for resolving references are described later in this chapter.

Variable Reference Contexts

A variable reference can occur in the following contexts.

- any context where a variable is assigned a value
- any context that expects the address of a variable
- any context that expects a value from a variable

If a value is expected, the variable must have been previously assigned a value, must be a static variable declared with the `initial` attribute, or must be an external static variable with a name that matches a message name in the current message file. If you attempt to reference the value of a variable that has no value, the results are unpredictable.

You can transmit a value to a variable in several ways. For example, you can use the assignment statement or the `get` statement. A variable that appears on the left side of an assignment statement is assigned the value of the expression on the right side of the statement. A variable that appears in the list of a `get` statement receives an input value.

The following example illustrates two ways variables can receive values.

```
declare   a    fixed bin(15) static initial(0);
declare   b    fixed bin(15);
declare   c    fixed bin(15);

     b = a;
     get edit(c)(f(5));
```

Variables can also receive values in other contexts. For example, a variable might receive a value when it is used in any of the following ways.

- passed as an output argument to a subroutine
- used in the `into` or `keyto` option of the `read` statement
- used as the index of a `do` statement

The following built-in functions require a variable reference, but do not require that the variable have a value: `addr`, `bytesize`, `dimension`, `hbound`, `lbound`, `maxlength`, and `size`. The argument of the `length` built-in function can be a fixed-length string variable that has not been assigned a value. If a character-string variable is declared with the `varying` attribute, you must assign it a value before referencing it in the `length` function.

If you pass a variable by reference as an output-only argument, it need not have a value.

Because the record I/O statements copy the storage of a variable, the variable you reference in a `from` or `keyfrom` clause need not have a value. However, if you use a variable with no definite value in the `from` or `keyfrom` clause of a `write` or `rewrite` statement, the output is unpredictable.

Simple References

A *simple reference* is a name without subscripts, without pointer-qualifiers, and without structure-qualifiers.

The following example includes several simple references.

```
declare   total          fixed bin(31);
declare   str            char(10) varying;
declare   board(8,8)     bit(1) aligned;

          str = 'abc';
          total = length(str);
          board = '1'b;
```

In the preceding example, each reference to the name `total` or `str` in the assignment statements is a simple reference; no subscripts or qualifiers are used. The reference to `board` in the last assignment statement is a reference to an entire array; the effect is to assign the value `'1'b` to each element of the array.

For information on array data, see Chapter 4.

To assign a value to a specific array element, you must use a subscripted reference, as described in the following section.

Subscripted References

A *subscripted reference* is a name followed by a parenthesized list of subscript expressions. The name must have been declared as an array. The number of subscript expressions must equal the number of dimensions declared for the array. Each subscript expression must produce a fixed-point integer value within the range defined by the lower and upper bounds declared for that dimension of the array. Subscript expressions are separated by commas.

The following example illustrates how subscripted references are used.

```
declare   grid(5,5)      float bin(24);
declare   element(10)    float bin(24);

          grid(k*2,3) = element(1);
```

In the preceding example, both `grid(k*2,3)` and `element(1)` are subscripted references. (The reference to `k` is an example of a simple reference.)

You can use asterisks as subscript expressions in an array reference. If you use asterisks as subscripts, **all** subscripts in that reference must be asterisks. OpenVOS PL/I does not support cross-section references. For example, a reference to `grid(*,3)` is illegal. A reference with asterisk subscripts, as shown in the following example, refers to every element of the array.

```
declare   a(5,5)    float bin(24);

     a(*,*) = 0;            /* Equivalent to  a = 0;            */
          .
          .
          .
     put skip list(a(*,*)); /* Equivalent to  put skip list(a); */
```

The assignment statement in the preceding example assigns the value `0` to each element in the array `a`. The `put` statement outputs all of the elements of `a`.

Structure-Qualified References

You can redeclare the names of structure members within the same block. This means that the scope of two identical names can overlap.

The following example declares a structure.

```
declare  1 s        ,
           2  a     fixed bin(15),
           2  b     ,
              3  a  fixed dec(7,2),
              3  c  float bin(53);
```

Based on the preceding declaration, a reference to `a` is ambiguous because the scope of the level-two `a` (a fixed-point binary integer) overlaps the scope of the level-three `a` (a floating-point decimal). See Chapter 7 for information on how structures are declared.

To avoid ambiguity, if a member's name is redeclared within the same block, you must qualify any reference to that member with the name of the containing structure. If the containing structure is itself a member of another structure, you can also redeclare its name within the same block. When this situation occurs, you must also qualify the structure name with the name of the structure that contains it. Repeated application of this rule ultimately creates an unambiguous structure-qualified reference to each structure member.

A *structure-qualified reference* is a sequence of names written left to right in order of increasing level numbers. The names are separated by periods. Spaces surrounding the periods are permissible but not necessary. The reference need not begin with the name of the major structure, and need not include the names of all substructures. You can skip intermediate levels within the reference as long as the result is unambiguous. You must include sufficient names to make the reference unambiguous.

The following example illustrates how the names of structure members are qualified.

```
declare  1  s         ,
            2  a      fixed bin(15),
            2  b      ,
               3  a   fixed dec(7,2),
               3  c   float bin(53);

     s.a = 12;
     s.b.a = 12876.99;
          .
          .
          .
     b.a = 10.43;
```

In the preceding example, the structure-qualified reference `s.a` refers unambiguously to the fixed-point level-two `a`. The reference `s.b.a` refers to the floating-point level-three `a`. Assuming that the name `b` is not redeclared within the current block, the reference `b.a` is equivalent to `s.b.a`. The variable `c` can be referenced as `s.b.c`, `b.c`, `s.c`, or `c`, provided the name `c` is not redeclared in the same block.

A structure-qualified reference that includes the name of the major structure and each substructure down to the member is a *fully qualified reference*. If the names of one or more of the containing structures are omitted from the structure-qualification, it is a *partially qualified reference*.

Many programmers omit the names of intermediate structures in long references, but it is considered a good practice to include the name of the major structure in all references to structure members. This practice makes programs easier to read and maintain.

In the preceding example, `s.b.a` is a fully qualified reference; `b.a` is a partially qualified reference. The following examples illustrate fully qualified references.

```
s.a
s.b
s.b.c
```

If a structure contains an array or is itself an array element, you can use subscript qualifiers anywhere within a structure-qualified reference. Most programmers place subscripts immediately following the array name that the subscripts qualify.

The following example declares a dimensioned structure.

```
declare   1  s(10)       ,
             2  a        fixed bin(15),
             2  b(3)     float bin(24),
             2  c(3)     ,
                3  d     pointer;
```

The reference `s(k).a` is equivalent to the reference `s.a(k)`, but the former is more common. The following examples are common forms of other subscripted references.

```
s(k).b(j)
s(k).c(j).d
```

A reference to `s.c.d(k)(j)` or `s.c.d(k,j)` is equivalent to `s(k).c(j).d`.

Pointer-Qualified References

In most contexts, a reference to a based variable must be qualified, implicitly or explicitly, by a pointer variable. Such a qualified reference is a *pointer-qualified reference*. You can set up implicit qualification when declaring the based variable. For information on implicit pointer qualification, or the use of based variables and pointers in general, see Chapter 6.

To explicitly qualify a based variable, you must use a pointer-qualified reference consisting of a reference to a pointer, followed by the locator qualifier symbol (`->`), followed by the name of the based variable.

The following example includes a pointer-qualified reference.

```
declare    a          float bin(24) based;
declare    p          pointer;
           .
           .
           .
     p->a = -1.23E+03;
```

In the preceding example, the reference `p->a` is a pointer-qualified reference to the based variable `a`.

Because a pointer variable can itself be based, multiple pointer-qualification is possible. The following example illustrates multiple pointer-qualification.

```
declare    1 node        based,
             2  nextp    pointer,
             2  value    float bin(53);

declare    headp         pointer;
           .
           .
           .
     headp->node.nextp->node.value = 8.53E-03;
```

In the preceding example, `headp->node.nextp->node.value` is a pointer-qualified reference to `node.value`. The reference to `node.value` is qualified by a reference to the pointer `node.nextp`. The reference to `node.nextp` is, in turn, qualified by a reference to the pointer `headp`.

The pointer value used to qualify a based variable need not be a pointer variable. You can also use pointer-valued functions and pointer-valued built-in functions, as shown in the following example.

```
declare   next_node    entry returns(pointer);

     next_node()->node.value = 1.23E+03;
```

In the preceding example, `next_node()->node.value` is a pointer-qualified reference. The pointer qualifier, `next_node()`, is a reference to a pointer-valued function.

Entry References

An *entry reference* is an entry-point name optionally followed by a parenthesized argument list. An argument list can contain from 0 to 127 arguments. Each argument in the list can be a reference or an expression. If a list contains more than one argument, the arguments are separated by commas.

Every entry point must be declared either contextually or explicitly. An entry point is declared contextually when it appears as a label prefix on a `procedure` or `entry` statement within the current procedure. An entry point is declared explicitly when it is given the `entry` attribute in a `declare` statement. For information on contextual and explicit entry declarations, see Chapter 7.

You can activate a subroutine by referencing an entry point to that subroutine within a `call` statement. Such an invocation is a *subroutine reference*.

You can activate a function by referencing an entry point to that function within an expression. Such an invocation is a *function reference*.

If an entry reference appears in any other context, such as in an assignment to an entry variable or in an argument list, it is an *entry-value reference*.

For information on block activation, see Chapter 3.

The next three sections discuss the following topics.

- "Function References"
- "Subroutine References"
- "Entry-Value References"

Function References

You can declare a function entry point in two ways.

- explicitly, in a `declare` statement with the `returns` attribute and without the `variable` attribute
- contextually, in a `procedure` or `entry` statement that includes the `returns` option

An entry point declared in either of the preceding ways must always be referenced as a function. A function reference always returns a value.

A function is activated whenever you reference one of its entry points with an argument list. If the function has no parameters, you must reference it with the empty argument list, `()`.

The following example includes two function references.

```
declare   location  entry returns(pointer);
declare   floater   entry(fixed bin(15)) returns(float bin(24));
declare   p         pointer;
declare   int       fixed bin(15);
declare   fract     float bin(24);
          .
          .
          .
     p = location();
     fract = floater(int) + 1.8e-03;
```

In the preceding example, the reference `location()` calls the function entry point `location`, which returns a pointer value. The reference `floater(int)` calls the function entry point `floater`, passing to it the value of `int` and receiving a floating-point value in return.

> **Note:** Built-in function references differ in several respects from other function references. See "Built-In Function References," later in this chapter, for information about built-in function references.

Subroutine References

An entry point declared without the `returns` attribute or the `returns` option and without the `variable` attribute must always be referenced as a subroutine. Subroutines are activated by `call` statements.

The following example illustrates a call to a subroutine.

```
declare   subr      entry(fixed dec(7,2), pointer);
declare   salary    fixed dec(7,2);
declare   next      pointer;
          .
          .
          .
     call subr(salary, next);
```

In the preceding example, `subr(salary, next)` is a subroutine reference that includes two arguments.

If a subroutine entry point is declared without arguments, you can call it without an argument list or with an empty argument list, as shown in the following example.

```
declare   s$continue_to_signal     entry;
               .
               .
               .
     call s$continue_to_signal;
               .
               .
               .
     call s$continue_to_signal();
```

Entry-Value References

A reference to an entry point is a reference to the entry value itself if the entry-point reference meets all of the following conditions.

- It is written without an argument list.
- It is not the subroutine reference of a `call` statement.
- It is not a reference to a built-in function.

An entry-point reference is **not** a function reference to the entry and does not activate a new block.

For information on entry values, see Chapter 4.

You can pass an entry value to an `entry` parameter or assign an entry value to an `entry` variable.

The following example illustrates the use of an entry value.

```
declare   e         entry returns(pointer);
declare   v         entry variable returns(pointer);
declare   eplace    entry(entry, pointer);
declare   p         pointer;
              .
              .
              .
          v = e;
          p = v();
          call eplace(e,p);
```

In the preceding example, the reference to `e` in the first assignment statement is a reference to the entry value of `e`. The effect of the assignment is to make a subsequent reference to `v` equivalent to a reference to `e`. Therefore, the second assignment statement activates the function that contains the entry point `e` and assigns the returned pointer value to `p`. The `call` statement invokes a subroutine whose first argument is an entry value. In this case, the entry value of `e` is used. If `v` had been referenced in the argument list instead of `e`, the effect would have been the same.

You can reference members of an array of entry variables with subscripts and an argument list. In such a case, the argument list follows the subscript list.

The following example shows how elements of entry arrays are referenced.

```
declare   e(5)      entry(fixed bin(31)) variable returns(pointer);
declare   finder    entry(fixed bin(31)) returns(pointer);
declare   long_int  fixed bin(31);
declare   p         pointer;
          .
          .
          .
          e(k) = finder;
          p = e(k)(long_int);
          e(k+1) = e(k);
```

In the preceding example, `e` is an array of five entry values. The reference `e(k)` is a reference to one of these entry values. The reference `e(k)(long_int)` is a pointer-valued function reference that calls the entry point `e(k)` and passes it the fixed-point argument `long_int`. If `e` had been declared without parameters, an empty argument list, `()`, would be used in place of `(long_int)`.

Built-In Function References

References to built-in functions differ from references to other functions in the following respects.

- Whenever a reference to a built-in function is encountered, the function is activated, regardless of the context in which the reference occurs.
- You **cannot** assign a built-in function to an `entry` variable or pass a built-in function as an argument to an `entry` parameter.
- You need not declare built-in functions that are referenced with an argument list.
- If you declare, with the `builtin` attribute, the name of a built-in function that takes no arguments, you can reference that function without an argument list.

The `oncode` built-in function is one example of a built-in function that does not take arguments. In the following example, `oncode` is declared with the `builtin` attribute and referenced without an argument list.

```
declare   oncode         builtin;

declare  e$abort_output  static external;
           .
           .
           .
     if oncode = e$abort_output
     then stop;
```

If you do not declare the name of a built-in function that takes no arguments, you must include an empty argument list, `()`, in each reference to that function.

Chapter 13 describes the OpenVOS PL/I built-in functions.

Reference Resolution

The compiler resolves references by finding the innermost containing block that contains any applicable declaration. Applicability is determined as follows:

- A simple or subscripted reference to a name is applicable to any declaration of that name.
- A fully qualified or partially qualified structure reference is applicable to any structure declaration that includes the same hierarchy of names as is used in the structure reference.

Once a block containing an applicable declaration is found, other containing blocks are not searched in the attempt to resolve the reference.

If the block has only one applicable declaration, the reference is resolved to that declaration. If the block has more than one applicable declaration, the reference must be a fully qualified reference to exactly one declaration in that block; otherwise, the reference is ambiguous and invalid.

The presence of subscripts, an argument list, or a pointer-qualifier do not affect the resolution of a reference and cannot make an ambiguous reference unique.

A simple or subscripted reference to a name is considered to be a fully qualified reference to a nonmember declaration of that name. This means that if a member and a nonmember are declared to have the same name in the same block, a simple or subscripted reference to the name is resolved to the nonmember. In such a case, all references to the member must be structure-qualified.

The following example illustrates how structure-qualified references are resolved.

```
declare   x        fixed bin(31);
declare 1 struct   ,
          2  x     char(1),
          2  y     float dec(18),
          2  z     bit(1) aligned;

     get list(x);
     get skip list(struct.x);
```

In the preceding example, the unqualified reference to `x` in the first `get` statement is resolved to the nonmember `x`. The second `get` statement contains a reference to the structure member `x`.

Chapter 9:
Expressions and Operators

This chapter discusses the following topics related to expressions and operators.

- "Overview"
- "Arithmetic Operators"
- "Relational Operators"
- "Bit-String Operators"
- "The Concatenate Operator"
- "Order of Operand Evaluation"

Overview

An *operator* performs an operation (evaluation) on one or more operands. An *operand* is a variable, constant, expression, or value upon which an operator or statement works. An *expression* is a series of one or more operands and zero or more operators that can be evaluated. Table 9-1 lists the PL/I operators, along with their name, type, and a brief description. Note that `a` and `b` represent operands.

Table 9-1. PL/I Operators

Operator	Name	Type	Description
`+`	Plus	Arithmetic prefix	Results in the value of *a*
`-`	Minus	Arithmetic prefix	Negates *a*
`+`	Addition	Arithmetic infix	Adds *a* to *b*
`-`	Subtraction	Arithmetic infix	Subtracts *a* from *b*
`*`	Multiplication	Arithmetic infix	Multiplies *a* by *b*
`/`	Division	Arithmetic infix	Divides *a* by *b*
`**`	Exponentiation	Arithmetic infix	Raises *a* to the power of *b*
`=`	Equal to	Relational	Tests whether *a* equals *b*
`^=`	Not equal to	Relational	Tests whether *a* does not equal *b*
`>`	Greater than	Relational	Tests whether *a* is greater than *b*
`<`	Less than	Relational	Tests whether *a* is less than *b*
`>=`	Greater than or equal to	Relational	Tests whether *a* is greater than or equal to *b*
`<=`	Less than or equal to	Relational	Tests whether *a* is less than or equal to *b*
`^>`	Not greater than (equivalent to `<=`)	Relational	Tests whether *a* is less than or equal to *b*
`^<`	Not less than (equivalent to `>=`)	Relational	Tests whether *a* is greater than or equal to *b*
`&`	And	Bit-string infix	Forms the bitwise AND of *a* and *b*
| or `!`	Or	Bit-string infix	Forms the bitwise inclusive OR of *a* and *b*
`^`	Not (complement)	Bit-string prefix	Forms the bitwise complement of *a*
|| or `!!`	Concatenate	Concatenate	Concatenates *a* and *b*

An *infix* operator is written between two operands. For example, `+` is an infix operator in the expression `a + b`. A *prefix* operator is written in front of an operand. For example, `-` is a prefix operator in the expression `-b`. Each operand is explained later in this section.

The following examples are PL/I expressions.

```
a + b
a * b
a * -b
a > b
a | b
^a
a || b
a
```

Arithmetic Operators

This section discusses the PL/I arithmetic operators, which can only take arithmetic operands and always yield arithmetic values. Table 9-2 lists the PL/I arithmetic operators.

Table 9-2. PL/I Arithmetic Operators

Operator	Description
`+` (prefix)	Plus
`-` (prefix)	Minus
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`**`	Exponentiation

Arithmetic operators require arithmetic operands. If an operand has the type `picture`, it is converted to a fixed-point decimal value. Other nonarithmetic operands must be explicitly converted to arithmetic values by one of the following conversion built-in functions: `binary`, `decimal`, `fixed`, or `float`.

The operands of the exponentiation operator need not have the same data type. If the data types of the two operands of any other arithmetic infix operator differ in base or scale, the operands are converted to a common arithmetic data type. The common data type is determined according to the rules in Table 9-3.

Table 9-3. Rules for Determining the Common Data Type of Arithmetic Operands

Operands	Common Data Type
One `fixed`, one `float`	`float`
One `binary`, one `decimal`	`binary` †

† If one operand is a fixed-point binary value and the other is a noninteger fixed-point decimal value, the common data type is `fixed decimal`; this case is nonstandard and produces a compiler error message.

A difference in the precision of the operands does **not** cause operand conversion.

The data type of the result produced by an arithmetic operator is determined by the converted data types of the operands and the rules governing precision discussed in the following sections.

The next two sections discuss the following topics.

- "Floating-Point Operations"
- "Fixed-Point Operations"

Floating-Point Operations

Except for exponentiation, the precision of a floating-point result of an arithmetic operation is always the maximum precision of the converted operands.

Information on exponentiation appears later in this chapter.

Fixed-Point Operations

The precision of a fixed-point result is effectively derived by aligning the decimal points of the two operands. The number of digits in the result is always limited by the maximum number of digits allowed for the result base (the value of `$MAX_FIXED_BIN` for binary, and 18 for decimal). However, with the exception of division, results preserve all fractional digits.

> **Note:** The maximum number of digits that you can specify for binary is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.

In division, all integer quotient digits are preserved along with as many fractional digits as possible within the limits for the base. Because fixed-point binary data does not support any fractional digits, the compiler issues a warning when you use the division operator with fixed-point binary operands.

This section discusses the following topics.

- "Prefix Operators"

- "Addition and Subtraction"
- "Multiplication"
- "Division"
- "Exponentiation"

Prefix Operators

The result of the plus and minus prefix operators has the data type of the converted operand.

Addition and Subtraction

The precision of the result depends on the common base of the operands.

This section discusses the following topics.

- "Decimal Addition and Subtraction"
- "Binary Addition and Subtraction"

Decimal Addition and Subtraction

The number of digits in the result of fixed-point decimal addition or subtraction is the maximum integral precision of the two operands, plus the maximum scaling factor of the two operands, plus 1, provided this value does not exceed the maximum precision of 18. The scaling factor of the result is the larger of the scaling factors of the two operands. If the precisions of the operands are `(p,q)` and `(r,s)`, the full precision of the result can be stated as shown in the following formula.

```
(min(18, max(p-q,r-s) + max(q,s) + 1), max(q,s))
```

If both operands are integers, the preceding formula can be simplified to the following formula.

```
(min(18, max(p,r) + 1))
```

For example, to add a `fixed dec(7,2)` value and a `fixed dec(5,2)` value, you calculate the precision of the result as shown in the following example.

```
fixed dec(7,2)          /* p = 7; q = 2 */
fixed dec(5,2)          /* r = 5; s = 2 */

precision = (min(18, max(7-2, 5-2) + max(2,2) + 1), max(2,2))
          = (min(18, 5 + 2 + 1), 2)
          = fixed dec(8,2)
```

If the values you were adding were constants, you would use the same formula, but you would use the precision and scaling factors of the actual values. For example, if you want to add 25.3 and 37.5, both values have a precision of 3 and a scaling factor of 1. The formula for this calculation follows.

```
precision = (min(18, max(3-1, 3-1) + max(1,1) + 1), max(1,1))
          = (min(18, 2 + 1 + 1), 1)
          = fixed dec(4,1)
```

If both operands were integers defined as fixed-point decimal values (for example, 47 and 73), use the following formula to calculate the precision.

```
precision = (min(18, max(2,2) + 1))
          = (min(18, 2 + 1))
          = fixed dec(3)
```

For information on the `min` and `max` functions, see Chapter 13.

Binary Addition and Subtraction

The precision of fixed-point binary addition or subtraction is the maximum precision of the operands, plus one. The maximum precision is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see “Fixed-Point Binary Data” in Chapter 4 for more information.

If this value exceeds the value of `$MAX_FIXED_BIN`, the precision is the value of `$MAX_FIXED_BIN`.

If the maximum precision is `N`, and the precisions of the operands are `(p)` and `(r)`, the precision of the result can be stated as shown in the following formula.

```
(min(N,max(p,r) + 1))
```

For example, to add two variables defined as `fixed bin(15)`, the precision of the result can be stated as shown in the following formula.

```
precision = (min(N, max(15, 15) + 1))
          = (min(N, 15 + 1))
          = fixed bin(16)
```

For information on the `min` and `max` functions, see Chapter 13.

Multiplication

The precision of the result depends on the common base of the operands.

This section discusses the following topics.

- “Decimal Multiplication”
- “Binary Multiplication”

Decimal Multiplication

The number of digits in the result of fixed-point decimal multiplication is the sum of the number of digits in the operands, plus one. If this value exceeds the maximum allowable precision, 18, the maximum is used. The scaling factor of the result is the sum of the scaling factors of the operands. If the precisions of the operators are `(p,q)` and `(r,s)`, the full precision of the result follows.

```
(min(18,p + r + 1), q + s)
```

Note that one more unit of precision is added than is needed. This rule is derived from full PL/I, which uses the same formula for complex as well as real fixed-point numbers.

For example, to multiply a `fixed dec(7,2)` value and a `fixed dec(5,2)` value, you calculate the precision of the result as shown in the following formula.

```
precision = (min(18, 7 + 5 + 1), 2 + 2)
          = (min(18, 13), 4)
          = fixed dec(13,4)
```

To multiply two fixed-point decimal constants, such as 43.25 and 11.0, you calculate the precision using the precision and scaling factors of the actual values, as shown in the following formula.

```
precision = (min(18, 4 + 3 + 1), 2 + 1)
          = (min(18, 8), 3)
          = fixed dec(8,3)
```

The `min` function is explained in Chapter 13.

Binary Multiplication

The resultant precision of fixed-point binary multiplication is the sum of the precisions of the operands, plus one. The maximum precision is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see “Fixed-Point Binary Data” in Chapter 4 for more information.

If the maximum precision is `N`, and the precisions of the operands are `(p)` and `(r)`, you can use the following formula to calculate the precision of the result.

```
(min(N,p + r + 1))
```

For example, to multiply two values that are declared as `fixed bin(15)`, you calculate the precision as shown in the following formula.

```
precision = (min(N, 15 + 15 + 1))
          = (min(N, 31))
          = fixed bin(31)
```

The `min` function is explained in Chapter 13.

Division

The division operator can operate on fixed-point values only if both operands are fixed-point decimal. The number of digits in the resultant fixed-point decimal value is 18. The scaling factor is 18, minus the number of digits in the first operand, plus the scaling factor of the first operand, minus the scaling factor of the second operand. If the precision of the first operand is `(p,q)` and the precision of the second operand is `(r,s)`, you can calculate the precision of the result using the following formula.

```
(18, 18 - p + q - s)
```

For integer operands, the formula can be simplified to the following formula.

```
(18,18 - p)
```

Because the result from the division operator has a large fraction, it generally cannot be added to or subtracted from another value. Alignment of the decimal points usually produces a value too large to be supported.

You can use the `divide` built-in function to divide fixed-point values and control the precision of the result. See Chapter 13 for further information about the `divide` function.

For example, to divide a `fixed dec(7,3)` value and a `fixed dec(4,2)` value, you calculate the precision of the result as shown in the following formula.

```
precision = (18, 18 - 7 + 3 - 2)
          = fixed dec(18, 12)
```

If both values are integers, such as 48 and 6, you calculate the precision of the result as shown in the following formula.

```
precision = (18, 18 - 2)
          = fixed dec(18, 16)
```

Improper use of the / operator can cause unexpected results. Consider the following example, which produces a run-time error.

```
declare   (two, six) fixed dec(1) volatile;
declare   ninetyseven fixed dec(2) volatile;
declare   result fixed dec(3);

     two = 2;
     six = 6;
     ninetyseven = 97;

     result = ninetyseven + six/two; /* INVALID OPERATION */

     if result = 100 then
          .
          .
          .
```

In the preceding example, the result of `six/two` is a `fixed dec(18,17)` value. The value of `ninetyseven` is scaled to fit in a `fixed dec(18,17)` in order to be added to the result of `six/two`. Since 97 with a scaling factor of 17 cannot fit in 64 bits, an error results.

> **Note:** The `volatile` attribute was used in the preceding example in order to avoid the compile-time optimizations of constant folding and constant propagation. See the *VOS PL/I User's Guide* (R145) for more information on these optimizations and on the use of the `volatile` attribute during optimization. See also the description of the `volatile` attribute in Chapter 7 of this manual.

Exponentiation

The result of exponentiation is usually the first operand, *x*, raised to the power of the second operand, *y*. This general rule has the following exceptions.

- If `x = 0` and `y <= 0`, `x**y` causes the `error` condition to be signaled.

- If `x < 0` and Case 1, as described later in this section, does not apply, the `error` condition is signaled.

Also, note the following special cases.

- If `x = 0` and `y > 0`, the result of `x**y` is `0`.
- If `x ^= 0` and `y = 0`, the result of `x**y` is `1`.

The data type of the result of exponentiation depends on the operands involved. The following three cases are possible.

Case 1: The first operand is a fixed-point value with precision `(p)` or `(p,q)` such that `(p + 1) * y - 1` does not exceed the maximum precision for the base, and the second operand is a positive integer constant.

The conditions are as follows:

- The first operand is a fixed-point value with precision `(p)` or `(p,q)`.
- The second operand is a positive integer constant, `y`.
- If the first operand is a decimal value, the following must be true.

  ```
  (p + 1) * y - 1 <= 18
  ```

- If the first operand is a binary value, the following must be true.

  (`p` + 1) * `y` - 1 <= $MAX_FIXED_BIN

  The maximum precision is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.

If the preceding conditions are met, the result is a fixed-point value with the base of the first operand and the following precision.

```
((p + 1) * y - 1, q * y)
```

If the first operand has no scaling factor, or a scaling factor of 0, the precision of the result can be stated more simply.

```
((p + 1) * y - 1)
```

For example, if the first operand is a `fixed dec(5,2)` value and the second operand is the integer constant 2, you calculate the precision of the result as shown in the following formula.

```
precision = ((5 + 1) * 2 - 1, 2 * 2)
          = (6 * 2 - 1, 4)
          = fixed dec(11, 4)
```

If the first operand has no scaling factor, you calculate the precision as shown in the following formula.

```
precision = ((5 + 1) * 2 - 1)
          = (6 * 2 - 1)
          = fixed dec(11)
```

Case 2: The second operand is a fixed-point integer value, but Case 1 does not apply.

The conditions are as follows:

- The first operand has precision `(p)`.
- The second operand is a fixed-point integer value.
- Case 1 does not apply.

If the preceding conditions are met, the result is a floating-point value with the base of the first operand. The resultant precision is the precision of the first operand, provided this value does not exceed the maximum precision for the resultant base and scale.

If the first operand is a decimal value, you can calculate the precision of the result using the following formula.

```
min(15,p)
```

If the first operand is a binary value, you can calculate the precision of the result using the following formula.

```
min(53,p)
```

The `min` function is explained in Chapter 13.

For example, if the first operand is a `fixed dec(5)` value and the second operand is a `fixed dec(2)` value, you calculate the precision of the result with the following formula.

```
precision = min(15,5)
          = float dec(5)
```

If the first operand is a `fixed bin(15)` value and the second operand is a `fixed dec(2)` value, you calculate the precision of the result with the following formula.

```
precision = min(53,15)
          = float bin(15)
```

Case 3: The second operand is a noninteger or floating-point value.

In this case, the result is a floating-point value with the base as shown in Table 9-3. The precision of the result is the larger of the precisions of the operands, provided this precision does not exceed the maximum precision for floating-point values of the resultant base.

For example, assume that *p* represents the converted precision of the first operand and *r* represents the converted precision of the second operand. If the base is decimal, you can calculate the precision of the result using the following formula.

```
min(15,max(p,r))
```

If the base is binary, you can calculate the precision of the result using the following formula.

```
min(53,max(p,r))
```

For information on the `min` and `max` functions, see Chapter 13.

For example, to calculate the precision of the result of an exponentiation operation where the first operand is a fixed-point decimal value and the second is a noninteger decimal value, consider the case of `11.00 ** 0.5 (fixed dec(4,2) ** fixed dec(2,1))`.

```
precision = min(15, max(4,2))
          = float dec(4)
```

If the first operand is a `fixed bin(15)` value and the second operand is a `float bin(24)` value, use the following formula.

```
precision = min(53, max(15,24))
          = float bin(24)
```

Relational Operators

This section describes the relational operators, which compare two values. Table 9-4 lists the PL/I relational operators.

Table 9-4. PL/I Relational Operators

Operator	Description
`=`	Equal to
`^=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`^<`	Not less than (equivalent to `>=`)
`^>`	Not greater than (equivalent to `<=`)

If either operand of a relational operator is an arithmetic or pictured value, the operands are converted to a common arithmetic data type using the same rules that are used for converting the operands of the addition operator. In all other cases, the data types of the two operands must be equivalent. For the purposes of determining data-type equivalence for relational operands, the following attributes are ignored.

- `aligned`
- `varying`
- `returns`
- `variable`
- string-length

Label, entry, and file data can be compared for equality or inequality only. Arithmetic, string, and pointer data can be compared using any of the relational operators. However, relational operators take only scalar operands. Arrays and structures cannot be compared using relational operators.

The result of a relational operator is always a bit string of length `1` representing a Boolean value. The bit-string value is `'1'b` if the relationship is true, and `'0'b` if the relationship is false.

The next six sections discuss the following topics.

- "Arithmetic and Pictured Value Comparisons"
- "Character-String Value Comparisons"
- "Bit-String Value Comparisons"
- "Label and Entry Value Comparisons"
- "Pointer Value Comparisons"
- "File Value Comparisons"

Arithmetic and Pictured Value Comparisons

Arithmetic and pictured values are compared algebraically after conversion to a common type.

> **Note:** Because floating-point numbers are approximations, if one or both operands of the equality operator (`=`) have floating-point scale, the result is unreliable. However, if an integer having no more than 15 decimal digits is converted to a floating-point number, the converted value always compares equal to the original integer value.

Character-String Value Comparisons

If two character-string values of unequal length are compared using the relational operators, the shorter value is treated as though it were padded on the right with space characters to make the lengths equal.

Character-string values are compared from left to right, one character at a time, until an inequality is found. Relative values are determined using the ASCII collating sequence. For example, the character `'A'` is less than the character `'B'`; therefore, the string `'ABABA'` is less than the string `'ABABB'`.

See Appendix D for a list of the characters in the ASCII character code set.

Bit-String Value Comparisons

If two bit-string values of unequal length are compared using the relational operators, the shorter value is treated as though padded on the right with zero bits to equal the length of the longer value.

Bit-string values are compared from left to right, one bit at a time, until an inequality is found. A one bit, `'1'b`, is greater than a zero bit, `'0'b`. For example, the string `'011'b` is greater than the string `'010'b`.

Label and Entry Value Comparisons

Label values and entry values can only be compared for equality or inequality. A label value cannot be compared with an entry value. Two label values or two entry values are equal only if they designate the same statement and the same stack frame. For example, if two label variables designate the same statement but different stack frames, they are not equal.

See Chapter 4 for information on label values and entry values.

Pointer Value Comparisons

Pointer values can be compared using any of the relational operators. Pointer values are equal only if they designate the same storage location.

File Value Comparisons

File values can only be compared for equality or inequality. File values are equal only if they designate the same file control block.

Bit-String Operators

This section describes bit-string operators, which compare bit strings. Table 9-5 lists the PL/I bit-string operators.

Table 9-5. PL/I Bit-String Operators

Operator	Description
`&`	And
| or `!`	Or (inclusive)
`^`	Not (complement)

Bit-string operators require bit-string operands. Operands of other data types must be converted to bit-string values using the `bit` built-in function. See Chapter 13 for further information about the `bit` function.

If the two operands specified for a bit-string infix operator, `&` or | or `!`, are of differing lengths, the shorter value is treated as though it were padded on the right with zero bits to equal the length of the longer value.

The result of the `^` operand is a bit string whose bits are the complements of the bits in the operand: each zero bit becomes a one bit, and each one bit becomes a zero bit. The complement of the null string is the null string.

The result of the `&` or | or `!` operator is a bit string whose length is that of the longer operand. Each bit of the result is determined as shown in the following table.

a	**b**	**a & b**	**a|b**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Consider the following example.

```
declare   x    bit(5) aligned;
declare   y    bit(5) aligned;

     x = '01011'b
     y = '11001'b
```

The following table illustrates the results of the bit-string operators from the preceding example.

<table>
<tr><th>Expression</th><th>Result</th></tr>
<tr><td><code>^x</code></td><td><code>'10100'b</code></td></tr>
<tr><td><code>x & y</code></td><td><code>'01001'b</code></td></tr>
<tr><td><code>x | y</code></td><td><code>'11-11'b</code></td></tr>
<tr><td><code>x &'11'b</code></td><td><code>'01000'b</code></td></tr>
</table>

The Concatenate Operator

The concatenate operator, `||` or `!!`, concatenates two strings; that is, it joins two strings together to form one string.

The operands can be character strings or bit strings. If both operands are bit strings, the result of concatenation is a bit string; otherwise, both operands are converted to character strings and the result is a character string. Chapter 5 discusses the rules for data-type conversion.

The length of the result string is the sum of the lengths of the converted operands.

Consider the following example.

```
a = 'abc';
b = 'wxyz';
c = '010110'b;
```

The following table illustrates the results of the concatenate operator from the preceding example.

<table>
<tr><th>Expression</th><th>Result</th></tr>
<tr><td><code>a||b</code></td><td><code>'abcwxyz'</code></td></tr>
<tr><td><code>b||c</code></td><td><code>'wxyz010110'</code></td></tr>
<tr><td><code>c||c</code></td><td><code>'010110010110'b</code></td></tr>
<tr><td><code>a||5</code></td><td><code>'abc   5'</code></td></tr>
</table>

In the preceding table, to produce the final result, `'abc    5'`, the fixed-point constant `5` is first converted to the character-string value `'    5'`. See Chapter 5 for an explanation of arithmetic to character-string conversion.

Order of Operand Evaluation

A single expression might contain a number of operands. The order in which the operands are evaluated is determined by two considerations.

- the priority of the operators
- the use of parentheses

The next three sections discuss the following topics.

- "Operator Priority"
- "Parentheses in Expressions"
- "Unevaluated Operands"

Operator Priority

The operators are divided into seven levels of priority. Operations involving operators with the highest priority are performed before those involving operators with lower priority.

If two operators within an expression have the same priority, the order of evaluation is determined by their order of appearance. In most cases, operators of equal priority are evaluated from left to right; however, the `**`, `^`, and prefix operators are evaluated from right to left. Table 9-6 lists the seven priority levels in order of decreasing priority and lists the order of evaluation for each.

Table 9-6. Order of Operator Priority

Priority	Operators	Order of Evaluation
1	`**` `^` `+` (prefix) `-` (prefix)	Right to left
2	`*` `/`	Left to right
3	`+` (infix) `-` (infix)	Left to right
4	`||` or `!!`	Left to right
5	`=` `^=` `>` `<` `>=` `<=` `^<` `^>`	Left to right
6	`&`	Left to right
7	`|` or `!`	Left to right

The following expressions contain multiple operators.

```
a ** 2 + b < c
a + b + c ** -d
```

In the first expression of the preceding example, `a ** 2` is evaluated first. That result is then added to `b`. Finally, that result is compared to `c`. In the second expression, `-d` is evaluated first, followed by the `**` operator. Next, `a + b` is evaluated. The second `+` operator is evaluated last.

Parentheses in Expressions

You can use parentheses to control the order of evaluation of an expression. The innermost set of parentheses is always evaluated first. Therefore, the following two expressions are **not** equivalent.

```
a + b * c
(a + b) * c
```

In evaluating the first expression of the preceding example, `b * c` is evaluated first and the result is then added to `a`. In the second expression, `a + b` is evaluated first because it appears in parentheses; that result is then multiplied by `c`.

Unevaluated Operands

If the result of an expression can be determined without evaluating all of the operands, the insignificant operands might not be evaluated. A program that depends on all operands being evaluated is invalid. The results produced by such a program might change when it is compiled with optimization enabled or when it is moved to another implementation of PL/I. Likewise, a program that depends on certain operands not being evaluated is invalid.

For example, the following statement is invalid.

```
if a ^= 0 & b / a = 5
then goto TOP;
```

In the preceding example, if the current value of `a` is `0`, the expression `b / a` might still be evaluated. This would cause the `zerodivide` condition to be signaled. See Chapter 15 for a discussion of exception conditions. To avoid an error, rewrite the preceding statement as shown in the following example.

```
if a ^= 0
then if b / a = 5
     then goto TOP;
```

Chapter 10: OpenVOS Preprocessor Statements

This chapter discusses the following topics related to the OpenVOS preprocessor statements.

Overview

When you compile an OpenVOS PL/I source module, the compiler invokes preprocessors to process preprocessor statements in the source module before the compiler translates the source code into object code. The OpenVOS PL/I compiler invokes two preprocessors: the OpenVOS preprocessor and the PL/I preprocessor. The OpenVOS preprocessor is the same preprocessor invoked by the other OpenVOS compilers (except OpenVOS C and OpenVOS Standard C) during compilation. The PL/I preprocessor is specific to the OpenVOS PL/I compiler. This chapter describes the OpenVOS preprocessor statements, and Chapter 11 describes the PL/I preprocessor statements.

For more information on the OpenVOS and PL/I preprocessors, see the *VOS PL/I User's Guide* (R145).

Using OpenVOS Preprocessor Statements

The OpenVOS preprocessor statements allow you to conditionally compile a source module. *Conditional compilation* enables you to switch on or off various statements in a source module. This is useful, for example, if you want a program to compile different lines of source code on different processors.

Each OpenVOS preprocessor statement begins with the dollar-sign (`$`) character (for example, `$define`). The dollar-sign character and the preprocessor statement cannot be separated by spaces. All other tokens must be separated from each other by at least one space.

OpenVOS preprocessor statements **must begin in the first column** of the source module. Indentation of nested `$if` statements is, therefore, not allowed.

An OpenVOS preprocessor statement must be contained on a single line. A line containing an OpenVOS preprocessor statement cannot contain comments or parts of the source language. (An exception is the `$endif` statement, which ignores any text following it on the same line, thus allowing you to comment on the source code.)

The OpenVOS Preprocessor Statements

Table 10-1 summarizes the OpenVOS preprocessor statements.

Table 10-1. OpenVOS Preprocessor Statements

Statement	Description
`$define`	Defines a preprocessor variable inside a source module or binder control file.
`$else`	Processes the lines up to the next `$endif` statement.
`$elseif`	Evaluates an expression as true or false. If the expression is true, the compiler executes the source code up to the next `$elseif` or `$endif` statement. If the expression is false, the compiler ignores this source code.
`$endif`	Closes the most recent `$if` statement.
`$if`	Evaluates an expression as true or false. If the expression is true, the compiler executes the source code up to the next `$else`, `$elseif`, or `$endif` statement. If the expression is false, the compiler ignores this source code.
`$undefine`	Undefines a preprocessor variable.

The following sections describe each preprocessor statement in greater detail.

- "The `$define` Statement"
- "The `$else` Statement"
- "The `$elseif` Statement"
- "The `$endif` Statement"
- "The `$if` Statement"
- "The `$undefine` Statement"

The `$define` Statement

The `$define` statement defines a preprocessor variable inside a source module. A *preprocessor variable* is a sequence of 1 to 256 alphabetic characters, digits, and underline (_) characters, in any position, used by the preprocessor. The preprocessor distinguishes between uppercase and lowercase alphabetic characters.

The `$define` statement has the following syntax.

```
$define identifier
```

Once a `$define` statement defines *`identifier`* as a defined variable, you can use *`identifier`* as the argument of the `defined` function. (See "The `$if` Statement," later in this chapter, for more information on the `defined` function.) The *`identifier`* remains defined until it is undefined by the `$undefine` statement.

In the following example, `var_a` is a defined variable; thus, the value of the `defined` function will be true, and the statements between the `$if` statement and the `$endif` statement will be compiled.

```
$define var_a
     .
     .
     .
$if defined(var_a)
     count = count + 1;
     .
     .
     .
$endif
```

No more than 100 preprocessor variables (including predefined preprocessor variables) can be defined for a source module. See the *VOS PL/I User's Guide* (R145) for a list of preprocessor variables that are predefined by the compiler.

The `$else` Statement

If the expression in the immediately preceding `$if` or `$elseif` statement is false, the `$else` statement processes the lines up to the next `$endif` statement. The `$else` statement has the following syntax.

```
$else
```

An `$if` or `$elseif` statement must precede an `$else` statement.

The `$elseif` Statement

If the expression in the immediately preceding `$if` or `$elseif` statement is false, the `$elseif` statement evaluates the expression contained in the `$elseif` clause as true or false.

- If the expression is true, the compiler expands the source code up to the next `$elseif` or `$endif` statement.
- If the expression is false, the compiler ignores this source code.

The `$elseif` statement has the following syntax.

```
     $elseif defined expression
```

An `$if` statement or another `$elseif` statement must precede an `$elseif` statement.

"The `$if` Statement," later in this chapter, describes the use of expressions in `$if` statements. This information also applies to the use of expressions in `$elseif` statements.

The `$endif` Statement

The `$endif` statement closes the most recent `$if` statement. The `$endif` statement has the following syntax.

```
$endif [ignored_text]
```

An `$endif` statement is required for each `$if` statement specified.

You can optionally place text on the same line after the `$endif` statement to comment the source code. The preprocessor ignores the text if it is preceded by a space.

The `$if` Statement

The `$if` statement evaluates an expression as true or false. If the expression is true, the compiler executes the source code up to the next `$else`, `$elseif`, or `$endif` statement. If the expression is false, the compiler ignores this source code. The `$if` statement and the accompanying `defined` function have the following syntax.

```
$if defined expression
```

At minimum, *`expression`* consists of an identifier enclosed in parentheses (for example, `(VER_13)`). In addition, you can use parentheses and the `&` (and), `|` (or), and `^` (not) operators to form more complex expressions. The order of operator precedence, from highest to lowest precedence, is `^`, `&`, and `|`.

In an expression, preprocessor variables that are defined evaluate to true; undefined preprocessor variables evaluate to false.

- If the expression is true, the compiler expands all statements up to the next `$else`, `$elseif`, or `$endif` statement, unless processing is explicitly disabled in that region of text by another preprocessor statement.

- If the expression is false, the compiler ignores all statements until it encounters another `$else`, `$elseif`, or `$endif` statement.

  – If the compiler encounters the `$else` statement, the compiler expands the statements up to the next `$endif` statement.

  – If the compiler encounters the `$elseif` statement, the expression in the `$elseif` is evaluated, and subsequent statements are conditionally executed up to the next `$else`, `$elseif`, or `$endif`.

  – If the compiler encounters the `$endif` statement, the immediately preceding `$if` statement ends, and the compiler goes on to the next statement in the program.

A source module can contain no more than 32 nested `$if` and `$elseif` statements.

In the following example, the compiler assigns a value of 15 to the variable `x` if either the `__I80860__` preprocessor variable **or** the `__MC68000__` preprocessor variable is defined.

```
$if defined (__I80860__) | defined (__MC68000__)
     x = 15;
```

The `$undefine` Statement

The `$undefine` statement undefines a preprocessor variable. The `$undefine` statement has the following syntax.

```
$undefine identifier
```

Example Using OpenVOS Preprocessor Statements

The following example illustrates the use of OpenVOS preprocessor statements in a source module.

```
check_revs:
     procedure options(main);

declare   x         fixed bin(15);

$define rev_3

$if defined (rev_0)

     put skip list('Original version');

$elseif defined (rev_1) | defined (rev_2)

     put skip list('Updated version');

$else
     put skip list('Undefined version');
$endif

end check_revs;
```

If the preceding program is compiled with the `-list` argument of the `pl1` command, the command creates an object module and a compilation listing. The following compilation listing illustrates the effects of preprocessing on compilation. Note that only the relevant portion of the compilation listing is shown.

```
 1   check_revs:
 2        procedure options(main);
 3
 4   declare   x         fixed bin(15);
 5
 6   +++$define rev_3
 7
 8   +++$if defined (rev_0)
 9   +++
10   +++     put skip list('Original version');
11   +++
12   +++$elseif defined (rev_1) | defined (rev_2)
13   +++
14   +++     put skip list('Updated version');
15   +++
16   +++$else
17        put skip list('Undefined version');
18
19   +++$endif
20
21   end check_revs;
```

As shown in the preceding compilation listing, the compiler inserts three plus signs (`+++`) in front of each line of the compilation listing that, after preprocessing, will not be compiled.

Since the `$define` preprocessor statement defines the preprocessor variable `rev_3`, only the following controlling expression (from line 17) evaluates to true.

```
17        put skip list('Undefined version');
```

Thus, after preprocessing, the only line of conditional code to be compiled is line 17. The code on lines 10 and 14 is not compiled.

For additional examples using the OpenVOS preprocessor statements, see the *VOS PL/I User's Guide* (R145).

Chapter 11: PL/I Preprocessor Statements

This chapter discusses the following topics related to the PL/I preprocessor statements.

- "Overview"
- "Using PL/I Preprocessor Statements"
- "The PL/I Preprocessor Statements"

Overview

As described in Chapter 10, the compiler invokes two preprocessors: the OpenVOS preprocessor and the PL/I preprocessor. These preprocessors process special statements, called preprocessor statements, in the source module before the compiler translates the source code into object code. The PL/I preprocessor is specific to the OpenVOS PL/I compiler, while the OpenVOS preprocessor is the same preprocessor that is invoked by the other OpenVOS compilers (except OpenVOS C and OpenVOS Standard C).

This chapter describes the PL/I preprocessor statements. Chapter 10 describes the OpenVOS preprocessor statements. For additional information on the PL/I and OpenVOS preprocessors, see the *VOS PL/I User's Guide* (R145).

Using PL/I Preprocessor Statements

The PL/I preprocessor statements (also known as compile-time statements) allow you to do the following:

- conditionally compile portions of a source module
- alter program text
- control the generation of a compilation listing
- enable compiler options

Each PL/I preprocessor statement begins with the percent-sign (`%`) character (for example, `%do`). The percent-sign character and the preprocessor statement cannot be separated by spaces. All other tokens must be separated by at least one space.

The PL/I Preprocessor Statements

Table 11-1 summarizes the function of each PL/I preprocessor statement.

Table 11-1. PL/I Preprocessor Statements

Statement	Description
`%` (null)	Performs no operation.
`%do`	Introduces a `%do`-group, which contains one or more PL/I language statements and preprocessor statements.
`%end`	Closes the most recent `%do`-group.
`%if`	Evaluates an expression as true or false and controls subsequent compilation. An `%if` statement contains a `%then` clause and, optionally, an `%else` clause.
`%then` clause	Enables compilation if the expression in the associated `%if` statement evaluated to true.
`%else` clause	Enables compilation if the expression in the associated `%if` statement evaluated to false.
`%include`	Inserts a text file into the program text.
`%list`	Re-enables the compiler's listing facility.
`%nolist`	Disables the compiler's listing facility.
`%options`	Specifies compiler options, many of which correspond to command-line arguments. The options are `default_char_set`, `default_mapping`, `no_default_mapping`, `longmap`, `longmap_check`, `mapcase`, `no_mapcase`, `max_fixed_bin`, `processor`, `shortmap`, `shortmap_check`, `system_programming`, `no_system_programming`, and `untyped_storage_sharing`. See the *VOS PL/I User's Guide* (R145) for more information on each option.
`%page`	Starts a new page in the compilation listing.
`%replace`	Creates a synonym for a literal constant or declared name.

The following sections describe each of the PL/I preprocessor statements in greater detail.

- "The `%` (Null) Statement"
- "The `%do` and `%end` Statements"
- "The `%if` Statement"
- "The `%include` Statement"
- "The `%list` Statement"
- "The `%nolist` Statement"
- "The `%options` Statement"
- "The `%page` Statement"
- "The `%replace` Statement"

The `%` (Null) Statement

The `%` (null) preprocessor statement is a no-operation statement. It contains no text. The `%` (null) statement has the following syntax.

```
% ;
```

The space before the semicolon is optional.

Use the `%` (null) statement when you want no action to take place. For example, if you want to test for a certain condition without performing an action, use the `%` (null) statement within an `%if`-`%then`-`%else` construct, as shown in the following example.

```
%if OS_REV = 12
     %then
        % ;
      %else
        call error('Check version number.');
```

The `%do` and `%end` Statements

The `%do` preprocessor statement introduces a `%do`-group; the `%end` preprocessor statement ends a `%do`-group. A `%do`-group contains one or more PL/I language statements (including null statements) and preprocessor statements that are evaluated during the first phase of compilation. A `%do`-group has the following syntax.

```
%do;
  .
  .
  .
%end;
```

The `%do` statement is followed by a group of language or preprocessor statements that can be used in any context in which a single statement is expected by the preprocessor; the `%do`-group is often the object of the `%then` or `%else` clause of an `%if` statement. In a construct that contains nested `%do`-groups, an `%end` statement terminates the group that begins with the last unpaired `%do` statement. A `%do`-group can be nested up to 64 levels.

Note that each `%do` statement must have a corresponding `%end` statement.

The following example shows a `%do`-group that contains two other preprocessor statements. These statements are executed if `OS_REV` is equal to 12.

```
%if OS_REV = 12
     %then
     %do;
        %include 'r12_constants';
        %include 'r12_reg_files';
     %end;
```

See the *VOS PL/I User's Guide* (R145) for more examples using the `%do` and `%end` preprocessor statements.

The `%if` Statement

The `%if` preprocessor statement evaluates an expression as true or false and controls subsequent compilation. The `%if` statement has the following syntax.

```
%if expression
%then statement
[%else statement]
```

The compiler evaluates *`expression`* to a scalar bit value. If *`expression`* is true (that is, the value is 1), the `%then` clause is compiled; if *`expression`* is false (the value is 0), compilation resumes with the `%else` clause or the next statement after the construct if no `%else` clause is specified.

In the following example, if the condition is true, the variable `i` is assigned a value of 3; if the condition is false, `i` is assigned a value of 4.

```
%if rev = 3.2
     %then
         i = 3;
     %else
         i = 4;
```

Within *`expression`*, the operands can be decimal constants, bit constants, or character constants. If a `%replace` synonym is used in place of a constant, the synonym must have been previously defined within the compilation unit. (See "The `%replace` Statement," later in this chapter, for more information on `%replace`.)

The following rules govern operand conversion in an `%if` expression.

- For arithmetic operations, each operand is converted to `fixed dec(5,0)` before the operation is performed, and the result is converted to `fixed dec(5,0)`. The null string is assigned the value 0 if it is converted to integer.
- Any character value being converted to an arithmetic value must be an optionally signed integer.
- Converting a fixed-point value to a bit value results in a bit string of length 17.
- Converting a fixed-point decimal value to a character value results in a character string of length 8; leading zeros are replaced by spaces, and the rightmost space is replaced by a minus sign if the value is negative.

Table 11-2 illustrates data-type conversion rules for operands in `%if` expressions. The table lists the operators involved in data-type conversions and shows the rules that govern the conversions for each operator.

Table 11-2. Data-Type Conversion Rules for `%if` Expressions

Operator	Data-Type Conversion Rule
`+ - * / **` (arithmetic infix)	Operands are converted to `fixed dec(5,0)`, if necessary.
`+ -` (arithmetic prefix)	The operand is converted to `fixed dec(5,0)`.

`> < = ^= <=`
`>= ^< ^>`
(relational)
	Operands of the same type are not converted. If the operands are of different types and one is arithmetic, both are converted to `fixed dec(5,0)`; otherwise, both are converted to character types.

| or `!` `&` (bit-string infix)	Operands are converted to the `bit` data type, if necessary.
`^` (bit-string prefix)	The operand is converted to the `bit` data type, if necessary.
|| or `!!` (concatenate)	If both operands are bit data types, they are not converted; otherwise, the operands are converted to character types.

> **Note:** In Table 11-2, an *infix operator* is an operator written between two operands. A *prefix operator* is an operator written in front of an operand.

Like other PL/I conditional expressions, a PL/I-preprocessor conditional expression can use any arithmetic, relational, bit-string, or concatenate operator, with the exception of the exponentiation operator. The conditional expression cannot contain any function references, including references to built-in functions.

The statement following the `%then` or `%else` clause can be a preprocessor statement, a `%do`-group, or another PL/I statement that is suitable for use with a `then` or `else` PL/I language statement.

The compiler checks a statement's syntax even if the statement is not compiled (for example, parentheses must balance, all statements must end with a semicolon, and so on). The debugger can list the lines in noncompiled units, but no generated code is associated with these lines.

In the following example, the `%if` statement contains the expression `OS_REV = '10.1'`, which checks a condition and returns a true or false value. If the value is true, the `%do`-group in the `%then` clause is compiled; if the value is false, the `%else` clause is compiled.

```
%if OS_REV = '10.1'
     %then
     %do;
          put skip list ('block a');
          if func_a(block) = -1
          then call error ('Error in block.');
     %end;
     %else
     %if OS_REV = '11.0'
          %then
          %do;
               put skip list ('block a');
               if func_a(block) = 0
               then call error ('Error in block.');
          %end;
          %else
          if func_a(block) < 0
          then call error ('Error in block.');
     .
     .
     .
end;
```

In the preceding example, notice that the `%else` clause contains another `%if-%then-%else` group. Each `%else` clause is associated with the most recent unpaired `%then` clause at the same nesting level; likewise, each `%then` clause is associated with the most recent unpaired `%if` statement at the same nesting level.

See the *VOS PL/I User's Guide* (R145) for more examples using the `%if` preprocessor statement.

The `%include` Statement

The `%include` preprocessor statement inserts the contents of a text file into the program in place of the `%include` statement.

The `%include` statement has the following syntax.

```
%include 'file_name';
```

The file name can be the full or relative path name of a text file but is usually a simple file name. This file is called an *include file*.

The file-name portion of the path name can be an extended name. For example:

```
%include 'new file';
```

For more information about extended names, see *Using OpenVOS Extended Names* (R631).

The name of a PL/I include file must have the suffix `.incl.pl1`. You are not required to specify the suffix when you specify *`file_name`*, however. If you do not specify the suffix, the compiler searches for the file *`file_name`*`.incl.pl1`.

If you specify a simple file name, the compiler searches for the file in your include libraries. An *include library* is a directory that contains include files. Usually, your current directory is the first include library on the search list. You can use the `list_library_paths` command to list the directories the compiler searches, in the order in which they are searched. To change the search list, use the `add_library_path`, `delete_library_path`, or `set_library_paths` command. See the *OpenVOS Commands Reference Manual* (R098) for more information about these commands.

You can specify the `%include` statement in place of an identifier, literal constant, or punctuation symbol. The include file can contain any PL/I program text: executable statements, declarations, other preprocessor statements, and so forth.

The `%include` statement is often used to include structure member declarations or `%replace` preprocessor statements that are common to more than one source module. In the following example, the `%include` statement declares a commonly used structure.

```
declare  1  dev_user_info,
%include 'dev_user_info_inner';
```

The `declare` statement in the preceding example declares a structure named `dev_user_info`. The include file `dev_user_info_inner.incl.pl1` contains the following structure member declarations for the `dev_user_info` structure.

```
2 person_name       char(32) varying,
2 process_id        fixed bin(31),
2 time_assigned     fixed bin(31);
```

Often, an include file containing members of a structure does not contain the terminating semicolon; this practice allows you to add additional structure members. In this case, you must terminate the `%include` statement in your program with two semicolons: one to end the `%include` statement and one to end the `declare` statement. See Chapter 7 for the syntax of the `declare` statement.

A source module can contain up to 999 include files. However, a source module and all of its include files, combined, cannot have more than 32,767 source lines.

The text of all include files specified in a source module will appear in a compilation listing unless you use the `%nolist` preprocessor statement. See "The `%nolist` Statement," later in this chapter, for more information.

The `%list` Statement

The `%list` preprocessor statement re-enables the compiler's listing option after it has been disabled by the `%nolist` preprocessor statement. The `%list` statement has the following syntax.

```
%list;
```

All text following the `%list` statement appears in the compilation listing unless another `%nolist` statement is encountered.

If the program is compiled without the `-list` command-line argument, the preprocessor ignores the `%list` statement.

If you specify more than one `%list` or `%nolist` statement, the source module must contain an equal number of `%list` and `%nolist` statements.

See the next section, "The `%nolist` Statement," for an example illustrating the use of `%list`.

The `%nolist` Statement

The `%nolist` preprocessor statement disables the compiler's listing option. The `%nolist` statement has the following syntax.

```
%nolist;
```

The text following a `%nolist` statement does not appear in the compilation listing. However, if the compiler encounters a subsequent `%list` statement, the text following that statement does appear in the listing. The `%nolist` and `%list` combination is often used to suppress the listing of include files, as shown in the following example.

```
declare   total     fixed bin(15);
%nolist;
%include 'accounting_constants';
%list;
```

If the program is compiled without the `-list` command-line argument, the preprocessor ignores any `%nolist` statements.

The `%options` Statement

The `%options` preprocessor statement specifies certain compiler options within the source module. See the next section, "Specifying Compiler Options with the `%options` Statement," for descriptions of these options.

The `%options` statement has the following syntax.

%options *option* [,*option*] . . . ;

Compiler options specified with the `%options` statement take precedence over compiler options specified as arguments of the `pl1` command.

Compiler options must be specified using either all uppercase characters or all lowercase characters. Option names must **not** be preceded by hyphens.

The `%options` statement must appear before any executable statement in the compilation unit. Note that you can place the `%options` statement directly after the `procedure` statement, since `procedure` is not an executable statement.

The following example illustrates the use of the `%options` statement.

```
opt_demo:
     procedure options(main);
%options mapcase;
     call a;
a: procedure;
      .
      .
      .
end a;
end opt_demo;
```

See the *VOS PL/I User's Guide* (R145) for more examples using the `%options` preprocessor statement.

Specifying Compiler Options with the `%options` Statement

You can specify the following compiler options in the `%options` statement.

- `default_char_set`
- `mapcase` or `no_mapcase`
- `max_fixed_bin`
- `processor`
- `system_programming` or `no_system_programming`
- `untyped_storage_sharing`
- `default_mapping` or `no_default_mapping`
- the following alignment options: `longmap`, `shortmap`, `longmap_check`, and `shortmap_check`

The compiler options are described as follows:

▶ `default_char_set` $\left[=\right] \left\{ \begin{array}{c} \texttt{none} \\ \texttt{latin\_1} \end{array} \right\}$

Specifies the default right graphic set of character-string constants to be used in a compilation unit. If you specify `none`, every character-string constant `'const'` in the source code is treated as though it were written `shift('const')`.

If you do not specify a default character set with the `default_char_set` option, the default character set is `latin_1`, which causes the compiler to check that all string constants are valid National Language Support (NLS) strings.

See Chapter 13 for a description of the `shift` built-in function. See the *National Language Support User's Guide* (R212) for information about NLS.

The `default_char_set` option has no corresponding argument in the `pl1` command.

- ▶ `default_mapping`
- ▶ `no_default_mapping`

The `no_default_mapping` option specifies that the compiler diagnose the following:

- level-one structures that do not explicitly specify a mapping attribute
- aligned bit and character strings and arrays of aligned strings greater than or equal to eight bytes that do not explicitly specify a mapping attribute and that are not members of a structure

If you specify `default_mapping`, the compiler does not perform this check.

- ▶ `longmap`

Specifies the longmap alignment rules. *Longmap alignment* causes most scalar data types to be allocated so that they begin on a boundary that is equal to the type's size. A structure is allocated so that it begins on a boundary that is equivalent to the strictest boundary requirement.

For additional information about longmap alignment, see "Data Alignment" in Chapter 4 and "Alignment Compatibility" in Chapter 3.

Note that you can specify only **one** of the following alignment options: `longmap`, `longmap_check`, `shortmap`, or `shortmap_check`. The `longmap` option corresponds to the `longmap` option of the `pl1` command's `-mapping_rules` argument.

- ▶ `longmap_check`

Specifies the longmap alignment rules, and in addition, instructs the compiler to diagnose alignment padding within structures.

For additional information about longmap alignment, see "Data Alignment" in Chapter 4 and "Alignment Compatibility" in Chapter 3.

Note that you can specify only **one** of the following alignment options: `longmap`, `longmap_check`, `shortmap`, or `shortmap_check`. The `longmap_check` option corresponds to the `longmap/check` option of the `pl1` command's `-mapping_rules` argument.

- ▶ `mapcase`
- ▶ `no_mapcase`

The `mapcase` option specifies that the compiler interpret all uppercase letters as their lowercase counterparts, except those in character-string constants, rendering the compiler case-insensitive. The `no_mapcase` option specifies that the compiler interpret all characters as they are written, rendering the compiler case-sensitive. By default, the compiler is case-sensitive.

If you specify the `mapcase` option, place the `%options` statement **before** the first `procedure` statement in the source module. This ensures that mapcasing is applied to that `procedure` statement, as well as to the remainder of the source module.

Note that when you compile a source module with the `mapcase` option, and the module contains an external variable name or entry name with one or more uppercase letters, you may be unable to bind the resulting object module with other programs that define the same external variable and that have not specified the `mapcase` option. If the binder encounters a reference to the original name (for example, in a binder control file), it will not recognize the original name and its lowercase version as the same file name.

The `mapcase` option corresponds to the `-mapcase` argument of the `pl1` command.

▶ `max_fixed_bin` [=] *`max_precision`*

Specifies the maximum precision for `fixed bin` values in a PL/I compilation unit. The *`max_precision`* value is `31` (the default) or `63`.

This option is useful when working with very large `fixed bin` values. For example, if you specify the value `63` for the `max_fixed_bin` option (or for the `-max_fixed_bin` command-line argument), the following code fragment produces the expected result (the value of `fb31` multiplied by itself):

```
declare fd18        fixed dec(18);
declare fb31        fixed bin(31);

fd18 = fb31*fb31;
```

However, if you do **not** set the maximum precision to 63, the result of the preceding example is undefined if the magnitude of `fb31` is large enough so that the result of `fb31*fb31` does not fit in a `fixed bin(31)`.

This option, along with the `-max_fixed_bin` command-line argument, sets the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol.

▶ `processor` [=] *`processor_value`*

Specifies a target processor; that is, the processor on which the compiled and bound code will run. If you do not specify a target processor, the code is compiled for the system-wide default processor.

If your module is an ftServer V Series module, you can specify the `pentium4` value for *`processor_value`*.

The `processor` option differs from the `pl1` command's `-processor` argument in that you **cannot** use the `processor` option to generate code for a different processor family. For example, if you attempt to compile a program containing the following line on a module using the pa7100 processor, the compiler returns an error.

```
%options processor pentium4
```

See the *VOS PL/I User's Guide* (R145) for additional information about specifying a processor.

▶ `shortmap`

Specifies the shortmap alignment rules. *Shortmap alignment* causes most nonstring data types to be allocated so that they begin on an even-numbered byte boundary.

Unaligned character data items are allocated so that they begin on a byte boundary. Unaligned bit strings are allocated so that they begin on the current bit.

For additional information about shortmap alignment, see "Data Alignment" in Chapter 4 and "Alignment Compatibility" in Chapter 3.

Note that you can specify only **one** of the following alignment options: `longmap`, `longmap_check`, `shortmap`, or `shortmap_check`. The `shortmap` option corresponds to the `shortmap` option of the `pl1` command's `-mapping_rules` argument.

- `shortmap_check`

  Specifies the shortmap alignment rules, and in addition, instructs the compiler to diagnose alignment padding within structures.

  For additional information about shortmap alignment, see "Data Alignment" in Chapter 4 and "Alignment Compatibility" in Chapter 3.

  Note that you can specify only **one** of the following alignment options: `longmap`, `longmap_check`, `shortmap`, or `shortmap_check`. The `shortmap_check` option corresponds to the `shortmap/check` option of the `pl1` command's `-mapping_rules` argument.

- `system_programming`
- `no_system_programming`

  The `system_programming` option causes the compiler to check for cases of implicit data-type conversion, alignment padding, and references to structures without the level-one structure name. In addition, the compiler checks for sparse label arrays if the default case is not specified.

  If you specify `no_system_programming`, the compiler does not perform these checks. The default option is `no_system_programming`.

  The `system_programming` option corresponds to the `pl1` command's `-system_programming` argument. See the *VOS PL/I User's Guide* (R145) for additional information about the `-system_programming` argument.

- `untyped_storage_sharing`

  Loosens the restrictions on how variables, especially based variables, share storage. If you specify this option, variables of any combination of data types can share the same storage as long as the variables do not violate the alignment constraints of the target processor.

  Specify the `untyped_storage_sharing` option once, at the beginning of the compilation unit.

Consider the following example.

```
storage_unshared:
     procedure options(main);

declare based_15          fixed bin(15) based;
declare based_char2       char(2) based aligned shortmap;
declare (p,q)             pointer;
declare v                 pointer volatile;
declare (a,b,c)           fixed bin(15);

     c = 4;
     p = addr(c);
     v = p;
     q = v;
     a = p -> based_15;
     q -> based_char2 = 'ab';
     b = p -> based_15;

     if a = b
          then put skip list('a = b');
     else
          put skip list('a ^= b');

     put skip list('a = ', ltrim(a));
     put skip list('b = ', ltrim(b));

end storage_unshared;
```

The preceding program produces the following output.

```
a = b
a =  4
b =  4
```

The OpenVOS PL/I compiler would consider the preceding program to be **invalid** because it attempts to overlay a character string (`based_char2`) on a binary integer (`based_15`). Although they are the same size, these two variables would not normally share storage, because they are different data types. Note, however, that the compiler does **not** detect this error. Any output from such a program would be unpredictable.

If you had specified the `untyped_storage_sharing` option in the preceding program, `based_char2` and `based_15` could share storage with no problem. In this case, the program would have the following output.

```
a ^= b
a =  4
b =  24930
```

(Note that `24930` is the binary equivalent of '`ab`'.)

You should use the `untyped_storage_sharing` option only when there is a specific compatibility goal because it tends to degrade object-code quality for based, external, and parameter variables. Stratus strongly recommends adherence to OpenVOS PL/I's storage sharing rules to avoid this code degeneration.

The `%page` Statement

The `%page` preprocessor statement starts a new page in the compilation listing. The `%page` statement has the following syntax.

```
%page;
```

When the compiler encounters a `%page` statement, it advances the listing file to the next page.

The `%replace` Statement

The `%replace` preprocessor statement creates a synonym for a literal constant or declared name within a program. The `%replace` statement has the following syntax.

$$\texttt{\%replace}\ \textit{synonym}\ \texttt{by} \left\{ \begin{array}{l} \textit{literal\_constant} \\ \textit{declared\_name} \end{array} \right\} ;$$

The compiler replaces each occurrence of *synonym* that follows the `%replace` statement with the declared name or literal constant specified. The `%replace` statement is most often used to specify table sizes or to give names to special literal constants whose significance would not otherwise be obvious.

The following example demonstrates the use of the `%replace` statement.

```
/* Literal constant synonyms */

%replace  TRUE                by '1'b;
%replace  FALSE               by '0'b;
%replace  TABLE_SIZE          by 400;
%replace  MOTOR_POOL          by 5;

/* Declared name synonym */

%replace  RESULT_TABLE        by x;

/* Variables */

declare   x(TABLE_SIZE)       bit(1) static;
declare   department_number   fixed bin(15);
declare   k                   fixed bin(15);

/* Execution */
          .
          .
          .
     do k = 1 to TABLE_SIZE;
          if department_number = MOTOR_POOL
          then RESULT_TABLE(k) = TRUE;
     end;
```

To differentiate between declared names and synonyms, a common practice is to type synonyms for constants in uppercase, and declared names in lowercase.

The `%replace` statement operates on program text without regard to the meaning of the text. Therefore, the synonym could be the same as a keyword such as `stop` or `read`. The use of such synonyms might create program statements that are unrecognizable to the compiler and result in compiler error messages. You can avoid this situation by using unique names.

A synonym cannot be replaced by more than one value in a program. For example, the following sequence is not allowed.

```
%replace  SIZE      by 100;
     .
     .
     .
%replace  SIZE      by 50;
```

Two identical `%replace` statements are allowable, as shown in the following example.

```
%replace SIZE       by 100;
     .
     .
     .
%replace SIZE       by 100;
```

The %replace statement operates without regard to block structure. All occurrences of the specified synonym that appear in the program text after the %replace statement are replaced even if they are in other blocks. The following example illustrates this concept.

```
a: procedure;

declare   y    fixed bin(15);
%replace  x    by y;

     x = 5;

b: procedure;

declare   y    char(5);

     x = 'abcde';

end b;

end a;
```

In the previous example, any reference to x within the program text is replaced by y. If the reference occurs within procedure b, it refers to a character string; if the reference occurs outside of b, within a, it refers to a fixed-point binary number. If a reference to x occurs after the end of a in a block where y is not redeclared, the reference cannot be resolved. Declaring x at any point after the %replace statement has the same effect as redeclaring y.

For more information about block structure, see Chapter 3.

Chapter 12: Statements

This chapter describes the OpenVOS PL/I language statements in alphabetical order. Table 12-1 summarizes the OpenVOS PL/I statements.

Table 12-1. Summary of OpenVOS PL/I Statements

Statement	Description
`allocate`	Allocates an area of storage
Assignment	Sets the value of a variable
`begin`	Marks the start of a begin block
`call`	Activates a subroutine
`close`	Closes a file control block
`declare`	Designates the attributes of variables, entry points, and files
`delete`	Deletes a record from a keyed update file
`do`	Introduces a do-group
`end`	Closes a do-group, begin block, or procedure
`entry`	Defines a secondary entry point to a procedure
`format`	Defines a format list for use with `get` and `put` statements
`free`	Frees an area of storage
`get`	Reads arithmetic, pictured, or string values from a file, device, or string variable
`goto`	Transfers control to a specified statement
`if`	Sets up a condition that determines whether a statement is executed, or determines which of two statements is executed
Null	Provides null `then` or `else` clauses in an `if` statement, null on-units in an `on` statement, or multiple label prefixes on a single statement
`on`	Establishes an on-unit
`open`	Opens a file control block

Table 12-1. Summary of OpenVOS PL/I Statements *(Continued)*

Statement	Description
`procedure`	Marks the beginning of a procedure
`put`	Writes arithmetic, pictured, and string values to a file, device, or character-string variable
`read`	Transmits a file record into a program variable or a buffer
`return`	Terminates activation of the current procedure and transfers control back to the calling block
`revert`	Reverts an on-unit established within the current block activation
`rewrite`	Overwrites a record in a keyed update file
`signal`	Forces the execution of an on-unit
`stop`	Terminates program execution
`write`	Writes a record to a file

See the Preface for an explanation of the syntax notation used in the statement descriptions.

The allocate Statement

Purpose

The `allocate` statement allocates an area of storage of sufficient size to hold values described by a based variable.

Syntax

```
⎧allocate⎫
⎨        ⎬ based_variable set (pointer);
⎩ alloc  ⎭
```

Operands

▶ *`based_variable`*
A based, nonmember variable.

▶ *`pointer`*
A pointer variable.

Explanation

When an `allocate` statement is executed, an area of storage described by *`based_variable`* is allocated in the user heap. The address of that storage is assigned to *`pointer`*.

The storage remains allocated until freed by a `free` statement. If the storage is not explicitly freed, it remains allocated until the end of program execution.

If lack of space prevents the run time from allocating storage for the variable, the `allocate` statement signals the `error` condition with the oncode `e$storage`.

Chapter 6 discusses based storage and pointers.

Examples

In the following example, the `allocate` statement allocates 40 bytes of storage, which is the area needed to store an array of 10 4-byte floating-point values. The pointer `p` is set to the address of the allocated storage.

```
declare table(10) float bin(24) based;
declare p pointer;
          .
          .
          .
     allocate table set(p);
```

The Assignment Statement

Purpose

The assignment statement sets the value of a variable.

Syntax

```
target_variable = expression;
```

Operands

- `target_variable`
  A variable or pseudovariable.

- `expression`
  An expression that yields a value with a data type that can be converted to the data type of `target_variable`.

Explanation

When an assignment statement is executed, `target_variable` and `expression` are evaluated in an unspecified order. The value of the expression is converted to the data type of the target variable, and that converted value is assigned to the target variable. Chapter 5 discusses data-type conversions.

The target variable can be a reference to an entire array or structure **only** if one of the following is true.

- The expression is a reference to an entire structure that has the same size, hierarchic organization, and member data types as the target. In this case, the contents of the referenced structure are copied into the storage of the target.

- The expression is a reference to an entire connected array that has the same number of dimensions, the same dimension sizes, and the same component data type as the connected target. (Such an array can have asterisk extents.) In this case, the contents of the referenced array are copied into the storage of the target.

- The target is a connected array (that is, it is not a member of a dimensioned structure), and the expression is a scalar value. If this is the case, the expression is converted to the data type of the array elements and is assigned to each element of the array.

If the target is a character-string variable and the expression value is also a character string, the target and expression variables must not partially overlap in storage so that the target begins to the right of the source. This restriction becomes important when either or both the target or expression are references to the `substr` pseudovariable. The PL/I pseudovariables are described in Chapter 13.

The following table shows some valid and invalid assignments.

Valid Assignment	Invalid Assignment

`a = a;`
`a = a||b;`
`a = b||a;`
`substr(a,2,3);`
	`substr(a,2,3) = a;`

Because the order in which the target and expression are evaluated is unspecified, functions or on-units activated as part of an assignment evaluation must not assign values to subscripts or pointers used within the target reference. Likewise, the storage of the target must not be freed by the execution of a function or on-unit called during evaluation of the expression. Programs that depend on the order in which the target and expression are evaluated produce unpredictable results.

Examples

Examples of the assignment statement follow.

```
a = b + c;

x(k) = 5;

p->node.value = sqrt(x(j));

substr(s,i,3) = 'abc';

struct.code = 0;
```

The begin Statement

Purpose

The `begin` statement marks the start of a begin block.

Syntax

```
begin;
```

Explanation

Each `begin` statement must have a corresponding `end` statement. These statements delimit a begin block. When the `begin` statement is executed, the begin block is activated. Block activation is terminated when the corresponding `end` statement, a `goto` statement (out of the block), or a `return` statement is executed.

> **Note:** A `return` statement within a begin block returns to the **caller** of the procedure that contains the begin block.

The `begin` statement is an executable statement. It can appear as a `then` or `else` clause of an `if` statement, as an on-unit of an `on` statement, or as a simple statement anywhere in a procedure.

The execution of a begin block consumes more CPU time than the execution of a simple do-group. Therefore, a begin block is normally used only in the following contexts.

- as an on-unit
- to limit the scope of certain declarations
- to allocate new automatic variables over a small region of a procedure

See Chapter 3 for a discussion of blocks and block activation.

Examples

The following example shows one begin block as a statement in a program and another as an on-unit of the on statement.

```
begin;
   declare   code      fixed bin(15);
   declare   meaning   fixed bin(31);

     code = rel(p);
     call check_code(code, meaning);
     if meaning = ERROR
     then return;    /* Return to caller of current procedure */
end;
on warning
     begin;
          error_code = oncode();
          if error_code = ABORT_OUTPUT
          then call clean_up;
          else call s$error(error_code, caller, message);
          stop;
     end;
```

The call Statement

Purpose

The `call` statement activates a subroutine (that is, a procedure that does not return a value).

Syntax

`call` *`entry_reference`*`;`

Operands

- ▶ *`entry_reference`*

  A reference to a subroutine name, subroutine entry-point name, or entry-valued function, or an entry variable that has been assigned an entry value. The *`entry_reference`* can include a parenthesized argument list containing from 0 to 127 arguments, including those with dynamic extents.

Explanation

When a `call` statement is executed, the subroutine designated by *`entry_reference`* is activated. Any arguments specified in the entry reference are passed to the called procedure. The called procedure must not have a `returns` option and must have the same number and type of parameters as the entry reference has arguments. You can call a procedure with no arguments using an entry reference with an empty argument list, `()`, or no argument list.

Each argument specified in the entry reference is evaluated and passed to the subroutine. If possible, each argument is passed by reference; otherwise, each argument is passed by value. The order in which these arguments and other components of the entry reference are evaluated is undefined. Chapter 3 discusses block activation and argument passing.

Examples

In the following example, the `call` statement activates the procedure `e`.

```
declare   x              fixed bin(15);
declare   name_string    char(10);
declare   error_code     fixed bin(15);

     call e(name_string, 5 + x, error_code);
               .
               .
               .
e:   procedure(p_str,p_num,p_code);
declare   p_str     char(10);
declare   p_num     fixed bin(15);
declare   p_code    fixed bin(15);
               .
               .
               .
```

The close Statement

Purpose

The `close` statement closes a file control block.

Syntax

```
close file(file_reference);
```

Operands

- *file_reference*
  A reference to a file constant or file-valued function, or a file variable that has been assigned a file constant value.

Explanation

Execution of a `close` statement closes the file control block associated with the file reference. If the file control block is currently closed, the `close` statement has no effect and is not an error.

Once a file control block is closed, you can reopen it and give it different file attributes or associate it with a different file or device.

The `open` statement is described later in this chapter. PL/I input and output is discussed in Chapter 14.

Examples

In the following example, the `close` statement closes the file `f`, which had been opened for input, so that it can be opened for output.

```
open file(f) title('data_file') stream input;
     .
     .
     .
close file(f);

open file(f) title('error_file') sequential output;
```

The declare Statement

Purpose

The `declare` statement designates the attributes of variables, entry points, and files.

Syntax

```
{ declare } decl_string [,decl_string] ...;
{ dcl     }
```

Each *decl_string* has the following components.

```
                { name                                 }
[level_number]  { (decl_string [, decl_string]...)     }  [attribute] ...
```

Operands

- *level_number*
  An integer indicating the level of *name* within a structure.

- *name*
  The name of the object being declared.

- *attribute*
  An attribute to be assigned to *name*.

Explanation

The `declare` statement is not executable. It cannot be used as a `then` or `else` clause of an `if` statement or as the on-unit of an `on` statement. Aside from these restrictions, the `declare` statement can appear anywhere within a block.

A `declare` statement cannot have a label prefix.

Chapter 7 describes how names are declared and explains the PL/I attributes. Chapter 3 discusses the scope of declarations.

Examples

The following examples illustrate the `declare` statement.

```
declare   f              file;

declare   array(12)      float bin(24);

declare   (a,b,c)        fixed bin(15);

declare 1  structure     ,
          2  first       fixed bin(15),
          2  second      ,
            3  seconda   char(10),
            3  secondb   char(10),
          2  third       fixed bin(15);

declare   s$write        entry (char(*) varying);
```

The delete Statement

Purpose

The `delete` statement deletes a record from a keyed update file.

Syntax

`delete file(`*`file_reference`*`) [key(`*`key_expression`*`)];`

You can specify the `file` and `key` clauses in any order.

Operands

- *`file_reference`*
  A reference to a file value with an associated file control block. If the file control block is open, it must have the `keyed` and `update` attributes.

- *`key_expression`*
  An expression whose value can be converted to a varying-length character string with a maximum length of 64. A null key value, while technically invalid in standard PL/I, is not diagnosed as an error by OpenVOS PL/I.

Explanation

If the file control block associated with *`file_reference`* is open, it must have the `keyed` and `update` attributes. If the file is not open, it is implicitly opened with the `record`, `update`, and `sequential` attributes. If the file is not declared to have the `keyed` attribute, this implicit opening signals the `error` condition.

When a `delete` statement is executed, the key expression, if specified, is evaluated and converted to a varying-length character-string key value. If the file is open for keyed sequential access, the `delete` statement operates on the record with that index-key value. The current position of the file is changed to the deleted record, unless the key value is the null string, `''`, in which case the current position does not change.

If the file is open for direct access, the key value is further converted to an integer value; the `delete` statement operates on the record with that ordinal position in the file.

If a record in the file has the key value, that record is deleted. If a record with that key value does not exist, the `key` condition for the file is signaled. If control returns from a `key` condition on-unit, execution resumes with the statement following the `delete` statement.

If the file is open for keyed sequential access, you can omit the `key` clause from the `delete` statement. This action causes the current record to be deleted. The current record is usually the record most recently read.

PL/I input and output is discussed in Chapter 14. The `open` statement is discussed later in this chapter.

Examples

The following examples illustrate the `delete` statement.

```
delete file(f) key(25);

delete key(c||'.old') file(g);

delete file(h);
```

The do Statement

Purpose

The do statement introduces a do-group.

Syntax

```
do [ index = start [ [to finish] [by increment] ] ]
                   [      repeat next           ]

[while(test_expression)];
```

Operands

▶ *index*

A variable reference.

▶ *start*

An expression specifying a value to be assigned to *index* when the do-group is entered.

▶ *next*

An expression specifying a value to be assigned to *index* on each subsequent execution of the do-group. This expression is re-evaluated for each subsequent pass through the loop.

▶ *finish*

An expression specifying a value to be compared with the value *index* after each pass through the loop.

▶ *increment*

An expression specifying a value to be added to *index* after each pass through the loop. The increment expression must not evaluate to zero. If *increment* is **positive**, the do-group is executed if *index* is less than or equal to *finish*. If *increment* is **negative**, the do-group is executed if *index* is greater than or equal to *finish*.

▶ *test_expression*

A Boolean expression evaluated before each pass through the loop. The do-group is not executed if this expression is false.

Explanation

The `do` statement must have a corresponding `end` statement. The `do` and `end` statements delimit a group of statements called a *do-group*. Depending on the form of the `do` statement, the do-group is executed zero, one, or more times.

A `do` statement cannot be used as an on-unit, but can appear anywhere else within a procedure or begin block. For example, the `do` statement can appear in the `then` or `else` clause of an `if` statement.

Control can never be transferred into a non-simple do-group except at the `do` statement. When control is transferred to a `do` statement, any index is reset to its start value and other control expressions are re-evaluated.

The flow of control within a do-group can be altered by `if`, `return`, or `goto` statements. Do-groups can also contain other do-groups. For this discussion, assume that control is not transferred out of the do-group and that no statements within the do-group are ever skipped.

As shown in the preceding syntax section, the `do` statement takes the following four forms.

- simple-do
- do-while
- do-repeat
- iterative-do

The following sections describe these forms.

The Simple-Do

When a *simple-do* is executed, the do-group is executed exactly once.

Syntax

```
do;
```

Explanation

The simple-do is most commonly used within an `if` statement to allow more than one statement to appear in a `then` or `else` clause.

Examples

The following example shows the simple-do executing two statements if the condition is true.

```
if count < 20
then do;
          call short_order;
          call check_error;
     end;
```

The Do-While

When a *do-while* is executed, the do-group is executed repeatedly as long as the test expression is true.

Syntax

```
do while(test_expression);
```

Explanation

When the `do` statement is first executed, the test expression is evaluated. If the expression is false, the do-group is not executed; control is transferred to the statement following the group's `end` statement. If the test expression is true, the do-group is executed. When execution of the do-group completes, the test expression is re-evaluated. If the expression is still true, the do-group is executed again. The do-group is repeatedly executed until the test expression becomes false.

Examples

In the following example, the do-group is executed as long as the expression is true. When `key_value` equals `LAST_KEY`, execution continues with the statement following the `end` statement.

```
key_value = START_READING;
do while(key_value ^= LAST_KEY);
     read file(f) into(data_record) keyto(key_value);
     call process_record(data_record);
end;
```

The Do-Repeat

A *do-repeat* is similar to a do-while in that the do-group is executed as long as the test expression is true. However, in a do-repeat, an index variable is set to a specified value for the first pass through the do-group, and to another specified value for all subsequent passes through the do-group.

Syntax

```
do index = start repeat next [while(test_expression)];
```

Explanation

When the `do` statement is first executed, the expression *`start`* is evaluated and its value is assigned to the variable *`index`*. If you specify the `while` clause, *`test_expression`* is evaluated. If the test expression is false, the do-group is not executed and control is transferred to the statement following the do-group's `end` statement. If the test expression is true, or you omit the `while` clause, the do-group is executed.

When control reaches the end of the do-group, the expression *`next`* is evaluated and its value is assigned to *`index`*. If you specify the `while` clause, the test expression is re-evaluated. If the test expression is false, control is transferred to the statement following the do-group's `end` statement; otherwise, the do-group is executed again. Note that the value of the index variable is updated before the test expression is re-evaluated.

If you omit the `while` clause, the loop repeats indefinitely.

The variable *index* **cannot** be an array or a structure, but it can be an element of an array or a member of a structure. The data type of *index* must make it a suitable target for assignment from both *start* and *next*. No other restrictions apply to *start* or *next*.

The reference to *index* is not completely re-evaluated on each pass through the loop. The values of any subscripts, pointer qualifiers, or string lengths remain unchanged.

Examples

In the following example, the do-group is executed as long as the error code is not equal to 0. If the value of `str` is `first`, the procedure `get_records` is called. If the value of `str` is `subsequent`, that line is skipped.

```
do str = 'first' repeat 'subsequent' while(error_code ^= 0);
     if str = 'first'
     then call get_records;
     put skip list(message(str));
     call scroll_records;
     call process_top_record(error_code);
end;
```

The Iterative-Do

An *iterative-do* is similar to a do-repeat, except that the index variable is incremented for **each** pass through the do-group.

Syntax

`do` *index* `=` *start* [`to` *finish*] [`by` *increment*] [`while(`*test_expression*`)`]`;`

Explanation

When the `do` statement is first executed, the *start*, *finish*, and *increment* expressions, as well as the *index* reference, are evaluated in an unspecified order. The *finish* and *increment* expressions are never re-evaluated unless control is transferred from the do-group and then back to the `do` statement again.

The value of the *start* expression is converted, if necessary, to the data type of *index* and that value is assigned to *index*.

The do-group is **not** executed if any of the following conditions are true.

- You specify a `while` clause and *test_expression* is false.
- The *index* value is greater than *finish*, and *increment* is either positive or omitted.
- The *increment* value is negative, and *index* is less than *finish*.

In each of the preceding cases, control transfers to the statement following the do-group's `end` statement.

If none of the preceding conditions caused control to transfer elsewhere, the do-group is executed. After execution occurs, the following checks occur.

- The *increment* value is added to *index*, and *index* is tested. If *index* is greater than *finish*, and *increment* is either positive or omitted, the do-group is not executed again. If *index* is less than *finish*, and *increment* is negative, the do-group is not executed again.

- The *test_expression* is evaluated. If it is false, the do-group is not executed again.

Note that the value of the index variable is incremented before the test expression is evaluated.

You can specify the `to` and `by` clauses in any order; if you use a `while` clause, it must be specified last. You can omit either the `to` or `by` clause. If you omit the `by` clause, the default increment is 1. If you omit the `to` clause, *index* is not compared with *finish* and the do-group repeats indefinitely unless stopped by the `while` clause. If you omit both the `to` and `while` clauses, the do-group repeats indefinitely.

If you specify an increment expression that evaluates to zero, the effect is unpredictable.

If you omit both the `to` and `by` clauses, the do-group is executed once; it is not repeated.

If *start* equals *finish*, the do-group is executed exactly once.

Examples

In the following example, the index is initialized to zero and incremented by 1; the do-group is repeated until the index exceeds `limit`, as long as `i*i` is not equal to `p_int`.

```
int_square_root: procedure(p_int) returns(fixed bin(15));

/* Find the integral non-negative square root, if any, of p_int */

declare   p_int          fixed bin(15);
declare   i              fixed bin(15);
declare   limit          fixed bin(15);

     limit = divide(p_int,2,15) + 1;

     do i = 0 to limit by 1 while(i*i ^= p_int);
     end;

     if i <= limit
     then return(i);   /* i is square root of p_int */
     else return(-1);  /* Root not found */

end int_square_root;
```

The end Statement

Purpose

The `end` statement closes a do-group, begin block, or procedure.

Syntax

`end [name];`

Operands

- *name*

  The name that appears in the label prefix of the `do` statement, `begin` statement, or `procedure` statement that corresponds to the `end` statement.

Explanation

An `end` statement cannot appear as part of a `then` or `else` clause of an `if` statement, or as an on-unit.

An `end` statement can have a label prefix that can be referenced by `goto` statements. Such `goto` statements can be contained within the same do-group or block that is closed by the `end` statement.

The effect of executing an `end` statement depends on where the `end` statement appears. The following three sections explain the effect of executing an `end` statement in a do-group, a begin-block, and a procedure, respectively.

Executing `end` in a Do-Group

When the `end` statement that closes a do-group is executed, the do-group might repeat depending on the `do` statement that heads the group. If the do-group does not repeat, execution continues with the statement following the `end` statement.

Executing `end` in a Begin Block

Execution of the `end` statement that closes a begin block terminates the activation of that begin block. The previous block activation becomes current again and execution resumes with the statement following the `end` statement.

If the begin block is an on-unit, control returns to the source of the signal, if possible. If return is not possible, the result of executing the `end` statement is unpredictable.

Chapter 15 contains more information on on-units. Chapter 3 discusses PL/I procedures and blocks.

Executing `end` in a Procedure

When the `end` statement that closes a procedure is executed, the current block activation is terminated. Control returns to the statement following the `call` statement that invoked the procedure. If the procedure is the main procedure of the program, the `end` statement terminates program execution.

If the `procedure` statement that heads the procedure contains a `returns` option, the procedure is a function and execution of the `end` statement is invalid. Block activation of a function must be terminated by a `return` statement before control reaches the `end` statement.

Examples

In the following example, `end` statements close a begin-block, a do-group, and a procedure. Each `end` statement is aligned with the statement that opened the group.

```
reader: procedure;
          .
          .
          .
     on endfile(f)
          begin;
               close file(f);
               put skip list('End of file');
               call clean_up;
               stop;
          end; /* End of begin block       */

     do k = 1 to 10;
      .
      .
      .
     end;      /* End of do-group          */

end reader;    /* End of procedure         */
```

The entry Statement

Purpose

The `entry` statement defines a secondary entry point to a procedure.

Syntax

```
name: entry [ (parameter[,parameter]...) ] [returns(attribute_list)];
```

Operands

- ▶ *name*

  The entry-point name.

- ▶ *parameter*

  A parameter describing an argument to be passed to the procedure when it is activated at this entry point. All parameters must be declared within the procedure that contains the `entry` statement.

- ▶ *attribute_list*

  A set of data-type attributes describing the value to be returned if the procedure is a function.

Explanation

An `entry` statement executed as the consequence of normal program flow from the previous statement has no effect; execution resumes with the statement following the `entry` statement.

An `entry` statement **cannot** be immediately contained within a begin block or do-group.

Each parameter in the `entry` statement must be declared within the immediately containing procedure. Parameters always have the parameter storage class. A parameter cannot be a member of a structure nor can it be declared with a storage-class attribute.

The list of parameters specified in an `entry` or `procedure` statement need not be the same as those specified in another entry point to the same procedure. Different entry points can specify different parameters, the same parameters in a different order, or even a different number of parameters. A program is in error if it references a parameter that is not specified at the entry point used to invoke the procedure.

If any entry point includes a `returns` option, **all** entries to that procedure must specify a `returns` option. While the attributes specified in these `returns` options need not be identical, any `return` statement immediately contained within the procedure must return a value that can be converted to each of the data types specified in the `returns` options.

Each entry statement must have a label prefix that provides the name of the entry point. The declaration of this name is established in the block **containing** the procedure in which the entry statement appears. This makes the name known in that block and in all contained blocks.

Every call to a procedure must be made with an argument list that contains one argument for each parameter in the parameter list for the specified entry point. An argument list can have up to 127 arguments. Each argument must be capable of being passed to the corresponding parameter.

If you do not specify the returns option in the entry statement, the entry point must always be accessed by a call statement and the procedure must not contain any return statements that specify a return value.

If you specify the returns option, the set of specified data-type attributes must describe a scalar value. When the procedure is invoked at this entry point, all values returned by the procedure are converted to that type before being returned as the function value of the procedure. All return statements in a function must specify a return value and the procedure must not execute its own end statement. All activations of such procedures must result from the evaluation of a function reference.

Chapter 3 discusses block activations and argument passing.

Examples

The following example shows a procedure that has three entry points: mouse_trap, east, and west.

```
mouse_trap:    procedure(p_rcode, p_message);
declare   p_rcode        fixed bin(15);
declare   p_message      char(*) varying;
          .
          .
          .
east: entry;
          .
          .
          .
west: entry(p_codep, p_str, p_message, p_rcode);
declare   p_codep        pointer;
declare   p_str          char(*);
          .
          .
          .
     return;
end mouse_trap;
```

The format Statement

Purpose

A `format` statement defines a format list that can be used by `get` and `put` statements.

Syntax

```
name: format( [k]format_item [,[k]format_item]...);
```

Operands

- *name*

  A nonsubscripted label prefix that serves as the format name.

- *k*

  A constant specifying the number of times a format item is to be repeated. The constant must be in the range 0 to 255, inclusive.

- *format_item*

  Any of the data or control format items described in Chapter 14.

Explanation

A format list is used during the execution of a `get` or `put` statement to control the transmission of data to or from a stream I/O file.

A format name is not a statement label and cannot be referenced in a `goto` statement.

Execution of a `format` statement has no effect unless it occurs as the consequence of the evaluation of an `r` format from the format list of a `get` or `put` statement. The `r` format includes the name of a `format` statement.

Each time control passes to a format list as part of the execution of a `get` or `put` statement, all format items between the last used format item and the next data format item are evaluated. The next data format item is then used to control the conversion of the next piece of data being transmitted to or from the stream file.

If control reaches the end of the format list in a `format` statement, control returns to the `r` format that transferred control to the `format` statement.

If control reaches the end of a format list in a `get` or `put` statement, and one or more values remain to be transmitted in the statement's I/O list, control transfers to the beginning of the format list.

Chapter 14 discusses PL/I input and output.

Examples

In the following example, the `format` statement defines a format list labeled `str_form`. In the subsequent `put` statement, `r(str_form)` instructs the compiler to substitute the format list defined in `str_form` in place of the label `str_form`.

```
str_form:    format(a,x(3));
          .
          .
          .
     put edit(p,q)(r(str_form),e(14,3));
```

The free Statement

Purpose

The `free` statement frees an area of storage that has been allocated by an `allocate` statement.

Syntax

`free` *`based_reference`*`;`

Operands

- *`based_reference`*
  A pointer-qualified, nonsubscripted reference to a nonmember based variable. The pointer qualification can be either explicit or implicit, but the reference must be to an area of storage allocated by an `allocate` statement.

Explanation

The value of the pointer that qualifies the based reference determines the location of the storage to be freed. The pointer must not be null.

The data type of the based variable must be the same as was used when the storage was allocated. If the variable is an array, the number of dimensions, and the size of each dimension, as well as the component data type, must be the same as was used at the time of allocation. If the variable is a structure, the hierarchic organization and member data types must be the same.

Once an area of storage is freed, it must not be referenced. Any pointers that address the storage are invalid and their values must not be used. If you violate these rules, the results are unpredictable.

Chapter 6 discusses based storage and pointers.

Examples

The following example shows an area of 400 bytes of storage being allocated for a table, then freed. The pointer `p` points to the beginning address of the storage area.

```
declare table(100)  float bin(24) based;
declare p           pointer;

     allocate table set(p);   /* Allocate 400 bytes of storage */
          .
          .
          .
     free p->table;           /* Free the 400 bytes of storage */
```

The get Statement

Purpose

The `get` statement reads arithmetic, pictured, or string values from a file, device, or character-string variable.

Syntax

```
get [ file(file_reference) [skip[(skips)]] ]
    [         string(string)               ]

          list(input_item [,input_item]...)
  edit(input_item[,input_item]...)(format_item[,format_item]...)
```

You can specify the `file`, `skip`, and `list` options, or the `file`, `skip`, and `edit` options, in any order. The format list is part of the `edit` option and must immediately follow the input list.

Operands

- *file_reference*

  A reference to a file constant or file-valued function, or a file variable that has been assigned a file value. If the associated file control block is open, it must have been opened as a stream input file. If the file control block is closed, the `get` statement opens it and assigns it the stream input attributes.

- *skips*

  A reference to a positive fixed-point integer specifying the number of line boundaries to skip in the file before beginning to receive input. The default value is 1. Whenever the `skip` option is used, the current column is reset to 1.

- *string*

  A character-string valued expression. The `get` statement uses the string as if it were a line from a file. When the `string` option is used, the `get` statement must not attempt to read a new line.

- *input_item*

  A reference to a variable in which to store input, or a comma-list of input items followed by an iterative-do, which are all contained in parentheses.

  ```
  (input_item [,input_item] ... iterative_do)
  ```

No comma separates an iterative-do from its associated list of input items. Note that if an input list contains an iterative-do, it must have two sets of parentheses: one enclosing the list of input items, and one as part of the iterative-do.

- *`format_item`*
  A data or control format item describing how the input is formatted.

Explanation

If you specify the `file` option, *`file_reference`* represents the file control block from which input is to be read. If the file control block is open, it must have the `stream` and `input` attributes. If the file control block is closed, the `get` statement opens it and gives it the `stream` and `input` attributes.

If you specify the `string` option, the `get` statement cannot read more than one line, and either the `list` or `edit` option is required. If you specify the `edit` option with `string`, the format list must not include any `column` or `skip` formats.

If you omit both the `string` and `file` options, the `sysin` file is used by default. The `sysin` file is always associated with the `default_input` port.

The number of lines read by a `get` statement is determined by the size of the list, the `skip` option, and any control formats specified in the format list. Unless control items or a `skip` option force new lines to be read, transmission begins with the current position of the current line. As many lines are read as are necessary to satisfy the input list.

If you specify the `skip` option, it is evaluated before the input list and format list. The input list and format list are evaluated together. Each list is evaluated from left to right.

Each *`input_item`* can be an array reference, a structure reference, or a scalar variable reference. A scalar variable reference causes one value to be transmitted from the input stream; if `edit` is specified, a scalar variable uses one data format. A reference to an array of length *`n`* causes *`n`* values to be transmitted, one for each element in the array; if `edit` is specified, *`n`* data formats are used. Values are transmitted to the array in row-major order as described in the discussion of arrays in Chapter 4. If a reference to a structure variable appears in the input list, all members of the structure, and members of all contained substructures, receive a value. The values are transmitted in left-to-right order. If `edit` is specified, each value requires one data format.

Chapter 14 discusses PL/I input and output.

If a parenthesized input list contains an iterative-do, values are transmitted under control of that iterative-do as if it were a do-group. The following example includes a use of the `get` statement with an iterative-do.

Examples

In the following example, the first `get` statement transmits 10 values to the array `a`, then transmits a value to `b`, and finally transmits a value to `c`.

The second `get` statement transmits a value to the `k`th element of `a`, transmits a value to `k`, and then, using the new value of `k` as a subscript, transmits a value to `b(k)`. All three values are transmitted using the `e` format.

The last `get` statement transmits single values, in order, to `b`, `a(1)`, `a(2)`, `a(3)`, `a(4)`, `a(5)`, and `c`.

```
declare   a(10)     float bin(24);
declare   (b,c)     float bin(24);
           .
           .
           .
     get file(f) list(a,b,c);
     get file(f) edit(a(k),k,b(k))(3 e(14,6));
     get file(f) list(b,(a(k) do k = 1 to 5),c);
```

The goto Statement

Purpose

The `goto` statement transfers control to a specified statement.

Syntax

```
⎧ goto  ⎫
⎨       ⎬ label;
⎩ go to ⎭
```

Operands

- ▶ *`label`*

  A reference to a statement label or label-valued function, or a label variable that has been assigned a label value.

Explanation

Execution of a `goto` statement transfers control to the statement designated by the label reference.

The value of *`label`* must designate a statement in the current block or in another active block.

If a `goto` statement transfers control to a statement outside the current block, the current block activation is terminated. All previous block activations back to the block activation containing the statement are also terminated. The block activation for the block containing the statement is made current and control is transferred to the labeled statement.

Chapter 4 describes label data. Chapter 7 discusses label declarations. Chapter 3 explains block activation.

Examples

The following examples illustrate the `goto` statement.

```
goto l;

goto CASE(k);

go to TOP;
```

The if Statement

Purpose

An `if` statement sets up a condition that determines whether a statement is executed, or determines which of two statements is executed.

Syntax

`if` *`expression`* `then` *`then_clause`* `;` [`else` *`else_clause`* `;`]

Operands

- *`expression`*
  A Boolean-valued expression.

- *`then_clause`* and *`else_clause`*
  A begin block, do-group, or any PL/I statement other than `end`, `procedure`, `declare`, `entry`, or `format`. Both the *`then_clause`* and *`else_clause`* must end with a semicolon.

Explanation

When an `if` statement is executed, *`expression`* is evaluated; the result must be a bit-string value of length 1. If *`expression`* is true (`'1'b`), *`then_clause`* is executed; if *`expression`* is false (`'0'b`), *`then_clause`* is skipped.

If you specify an `else` clause and *`expression`* is false, *`else_clause`* is executed. If *`expression`* is true, *`else_clause`* is skipped.

When execution of the `then` or `else` clause is complete, control is transferred to the statement following the `if` statement. Control also transfers to the statement following the `if` statement when *`expression`* is false and no `else` clause is specified.

Either *`then_clause`* or *`else_clause`*, or both, can be do-groups or begin blocks. Typically, do-groups are used rather than begin blocks.

Neither *`then_clause`* nor *`else_clause`* can have a label prefix. However, a begin block or do-group used as a *`then_clause`* or *`else_clause`* can contain labeled statements.

Either *`then_clause`* or *`else_clause`*, or both, can be another `if` statement. When this is the case, the first `else` clause is matched with the nearest preceding `then` clause. This closes the `then` clause. Each subsequent `else` clause is matched with the nearest preceding unclosed `then` clause. Note that *`else_clause`* can be the null statement.

Examples

In the following example, the `if` statement introduces a simple condition with a single `then` clause.

```
if a > b
then b = b + 1;
```

In the following example, the `if` statement introduces a do-group as the object of the `then` clause, and has an `else` clause.

```
if a = b
then do;
     b = b + 1;
     a = a - 1;
end;
else stop;
```

In the following example, the `if` statement includes a nested `if` statement as the object of the `then` clause. The first `else` clause contains the null statement.

```
if a < b
then if c > d
     then x = 5;
     else;
else x = 10;
```

The Null Statement

Purpose

The null statement is used in `if` statements to provide null `then` or `else` clauses, in `on` statements to provide null on-units, or to effectively provide multiple label prefixes on a single statement.

Syntax

```
;
```

Explanation

Execution of the null statement has no effect.

Examples

In the following example, the `on` statement contains a null on-unit.

```
     on endpage(f);  /* If endpage(f) is signaled, continue */

A:;
B:;
C:
     if code ^= 0
     then if count = TOTAL_RECORDS
          then signal endfile(f);
          else;
     else count = count + 1;
```

The first `if` statement is preceded by two labeled null statements, effectively giving that `if` statement three labels. Because the three labels technically designate different statements, they do not compare as equal.

The second (nested) `if` statement has a null statement in its `else` clause. The inclusion of this `else` clause causes the subsequent `else` clause to be associated with the first `then` clause.

The on Statement

Purpose

The on statement establishes an on-unit to be executed when a specific condition occurs.

Syntax

on *condition_name on_unit*;

Operands

▶ *condition_name*

The name of a computational, file, or system condition, or a reference to a programmer-defined condition, as shown in the following format.

condition(*condition_name*)

▶ *on_unit*

A begin block, the word system, or any PL/I statement other than the following: declare, do, end, entry, format, if, procedure, or return.

Explanation

Execution of an on statement establishes the on-unit as if it were a procedure to be called each time the specified condition is signaled; it does not execute the on-unit.

The on statement can have a label prefix and that label can be referenced in goto statements; however, an on-unit **cannot** have a label prefix.

If an on-unit for the specified condition has already been established in the current block activation, this new on-unit replaces it. An on-unit remains established until one of the following situations occurs.

- It is replaced by another on-unit for the same condition.
- It is reverted by a revert statement.
- The block activation in which it was established is terminated.

If system is specified as the *on_unit*, the default on-unit is used. This technique is often used inside on-units to prevent recursion.

You can establish an on-unit for each block or let the caller's on-unit handle the condition. Any on-unit you establish for a block is reverted when the block returns to its caller or is otherwise terminated.

When a condition is signaled, the on-unit established for that condition is called just as if it were a procedure with no parameters. The block activation that results from this call is terminated when control reaches the end of the on-unit or a `goto` statement transfers control out of the on-unit. When control reaches the end of an on-unit, control returns to the source of the signal. On-units for some conditions **cannot** return to the source of the signal. An attempt to do so produces unpredictable results.

If a `goto` statement transfers control out of the on-unit, the block activations are terminated for the on-unit and for all blocks back to, but not including, the block to which control is transferred.

The `signal` statement forces the execution of an on-unit. The `revert` statement reverts an on-unit. See Chapter 15 for information about conditions, and how a signal is resolved to an on-unit.

Examples

The following example defines on-units for the system conditions `endfile`, `break`, and `warning`, and the programmer-defined condition `tired`.

```
declare  tired     condition;
declare  s$error   entry (fixed bin(15), char(*) var, char(*) var);

     on endfile(sysin)
          begin;
               call cleanup;
               stop;
          end;

     on break stop;

     on warning
          begin;
               if oncode() = e$abort_output
               then do;
                    call processor;
                    call finisher;
                    stop;
               end;
               else call s$error(oncode(), caller, error_message);
          end;

     on condition(tired)
          begin;
               put skip list('This is taking too much time.');
               put skip(2) list('Program terminating.');
               stop;
          end;
```

The open Statement

Purpose

The `open` statement opens a file control block.

Syntax

```
open file(file_reference)[title(title_string) ]

[[stream][linesize(line_size) ][                          input                    ]]
[                             [ [output][print][pagesize(page_size) ]              ]]
[                                                                                  ]] ;
[               [record] [ input  ] [sequential] [keyed]                           ]
[                        [ output ] [  direct  ]                                   ]
[                        [ update ]                                                ]
```

You can specify the options and attributes in any order.

Operands

- *file_reference*

  A reference to a file constant or file-valued function, or a file variable that has been assigned a file value.

- *title_string*

  An expression that produces a character-string value. The character string is used to identify, and specify attributes for, a file or device. For further information, see the Explanation.

- *line_size*

  A reference to a fixed-point positive integer that specifies the number of characters per line of a stream file. The default is the line size of the device to which the file is attached. If the file is not attached to a device with a line size, the default is 80. You can specify the `linesize` option when opening a file for stream I/O only.

- *page_size*

  A reference to a fixed-point positive integer that specifies the number of lines per page for a print file. The default value is 60. You can specify the `pagesize` option when opening a print file only.

Explanation

If the file control block identified by the file reference is already open, the `open` statement is ignored, even if its attributes disagree with those of the current opening of the file control block.

If the `title` option is omitted, the file control block is connected to a file using the file ID as the title. The file ID of a file control block is the name of the file constant that owns the control block.

File Attributes

When an `open` statement is executed, the file control block associated with *`file_reference`* is opened with the attributes specified in the `open` statement and any attributes specified in the declaration of the file constant associated with the file control block.

The complete set of attributes to be assigned to the file control block being opened is derived through the following process.

1. Any attributes specified in the declaration of the file constant are combined with any attributes specified in the `open` statement.

2. Implied attributes are added.

3. Default attributes are added.

See Chapter 14 for information about implied and default attributes.

If the `file` option specifies a file variable, the same process occurs using the file-description attributes declared with the file constant whose value is held by the file variable.

The final set of attributes must be one of the following consistent sets.

```
stream input

stream output [print]

record input sequential [keyed]

record input direct keyed

record output sequential [keyed]

record output direct keyed

record update sequential [keyed]

record update direct keyed
```

The `title` Option

The `title` option of the `open` statement specifies the path name of the actual file being opened and certain characteristics of that file. The `title` option has the following form.

```
title(title_string)
```

The syntax of *`title_string`* is as follows. Note that *`title_string`* is enclosed in apostrophes.

```
            ┌       -sequential          ┐
            │         -stream            │
'path_name  │ -relative [record_size]    │  [-index [name]]
            │   -fixed [record_size]     │
            └                            ┘

[-keyis n m] [-duplicatekeys] [ -append   ] [-delete] [-noblank]
                              [ -truncate ]
 ┌   -nolock      ┐
 │ -recordlock    │
 │    -wait       │ [-dirtyinput] [-fileid file_id]
 │   -nowait      │
 │ -implicitlock  │
 └                ┘
[-volumeid volume_id] [-ownerid owner_id]'
```

The following list describes the elements of *`title_string`*.

▶ *`path_name`*

The value *`path_name`* is a full or relative path name of the file or device being opened. If you omit the `title` option, the default path name is the file ID of the file control block associated with the file reference specified in the `file` clause of the `open` statement, except in the following cases.

- The file ID is either `sysin` or `sysprint`. These control blocks are always associated with the terminal.
- A port with the file ID name is already attached to a file or device. In this case the existing attachment is used; any path name you specify in the `title` option is ignored.

▶
```
┌       -sequential          ┐
│         -stream            │
│ -relative [record_size]    │
│   -fixed [record_size]     │
└                            ┘
```

The `-sequential`, `-stream`, `-relative`, and `-fixed` options describe the organization of the file.

If an existing file is opened for `input` or `update`, or for `output` with the `-append` or `-truncate` option, the file organization is retrieved from the operating system.

If an existing file is opened for `output` with neither the `-append` option nor the `-truncate` option, the file is deleted and re-created.

Whenever an output file is created or re-created and a file organization is not specified, the file organization is obtained according to the following set of rules.

- If the file is open for stream output, the default organization is `-sequential`.
- If the file is open for sequential record output, the default organization is `-sequential`. If the file is opened with the `keyed` option, `-index primary` is also the default.
- If the file is open for direct record output, the default organization is `-relative 1024`.
- If you specify `-relative` or `-fixed` without a record size, the default is `1024`.

The file organization, whether obtained from OpenVOS or specified in the `title` option, must be consistent with the PL/I file attributes, as described in the following table.

PL/I File Attributes	**OpenVOS File Organization**
`stream`	Sequential, stream, relative, or fixed
`record sequential`	Sequential, stream, relative, or fixed
`record keyed sequential`	Sequential, relative, or fixed (an index is required; if there is no index, one is created)
`record direct`	Relative or fixed

▶ `-index` [*name*]

You can specify the `-index` option only if you are opening the file for `keyed sequential` access or for `sequential` input. If you do not specify an index name with the `-index` option, the default is `primary`. For output files, the index is created as records are written to the file. For update files, the index is updated as records are written, rewritten, or deleted. For input files, the index determines the sequential order of the file.

▶ `-keyis` *n m*

If you use an embedded-key index, specify the `-keyis` option. The embedded keys must begin in column *n* (byte *n*) and continue for *m* columns. Both *n* and *m* must be unsigned integers and must define a field within the records. The `key`, `keyfrom`, or `keyto` option is still used in `read`, `write`, `rewrite`, and `delete` statements that operate on the file. Both COBOL and BASIC programs require embedded keys; therefore, a PL/I program that processes a file that is also processed by a COBOL or BASIC program should use embedded keys. PL/I verifies that the key specified in the `key` or `keyfrom` option is actually embedded in the record in the specified field.

If you specify the `-keyis` option, you must specify the **byte number** at which the key should begin, followed by the length of the key. For example, if you are using records comprised of varying-length character strings and you want the index key to begin on the first character of the record and extend for five characters, specify `-keyis 3 5`. Specifying `3` instead of `1` allows for the two initial bytes indicating the length of the character-varying string.

► `-duplicatekeys`

If you specify the `-duplicatekeys` option, the existence of duplicate key values in the file does not signal the `key` condition. The `read` statement always accesses the first record having a specified key. You can specify the `-duplicatekeys` option only if the file is being opened for keyed sequential access.

► [`-append` | `-truncate`]

You can specify the `-append` option only if the file is open for keyed sequential output or sequential output. The `-append` option appends output to the end of the existing file, if one exists.

You can specify the `-truncate` option only if you are opening the file for output. The `-truncate` option deletes records in the existing file and empties all indexes. The access control list, file organization, maximum record size, and all indexes are retained. When you write to the file, new records are created, along with new index entries for all existing indexes.

If you specify neither `-append` nor `-truncate` for an output file, the existing file is deleted and a new file is created. In this case, the access control list, file organization, maximum record size, and indexes are not retained. If the file does not exist, the operating system creates one. The new file has the default file attributes that are discussed in the `create_file` command description in the *OpenVOS Commands Reference Manual* (R098).

► `-delete`

The `-delete` option deletes the file when it is closed.

► `-noblank`

Unless you specify the `-noblank` option, OpenVOS PL/I implicitly appends a space character to the end of each input line of a stream input file processed by the `get` statement with the `list` option specified. This space character prevents input fields from breaking over lines. Some implementations of PL/I do not do this. To provide compatibility with such implementations, specify the `-noblank` option.

► [`-nolock` | `-recordlock` | `-wait` | `-nowait` | `-implicitlock`]

These options allow you to override the default locking modes. The default locking rules allow one process to write a file or multiple processes to read a file, but prevent any process from reading a file while another process is writing to it.

If you do not explicitly specify a locking mode in the `open` statement, set-lock-don't-wait mode is the default.

Table 12-2 explains the effect of each locking option.

Table 12-2. Locking Mode Options

Option	Locking Mode	I/O Types	Description
`-nowait`	Set-lock-don't-wait	All	The operating system tries to lock the file. If it cannot, the `undefinedfile` condition is signaled with the oncode set to `e$file_in_use` (`1084`).
`-wait`	Wait-for-lock	`input`, `update`, `output -append`	The operating system tries to lock the file. If it cannot, the program is suspended until the lock becomes available.
		`output` without `-append`	Same as `-nowait`.
`-nolock`	Don't-set-lock	`input`, `update`, `output -append`	The operating system does not lock the file. Before performing I/O on the file, you must lock it using OpenVOS service subroutines. See the *OpenVOS PL/I Subroutines Manual* (R005) for more information.
		`output` without `-append`	Same as `-nowait`.
`-implicitlock`	Implicit-locking	All	Each time you perform I/O on the file, the operating system locks the file for the duration of that I/O and then releases it.

Table 12-2. Locking Mode Options

Option	Locking Mode	I/O Types	Description
-recordlock	Record-locking	update	Each individual record is locked when accessed by a record I/O statement.
		input	No locking occurs. You can read any records in a file that another process has **not** opened for update in the record-locking mode.

▶ -dirtyinput

If you are opening a file for record input, the -dirtyinput option allows you to read from the file even though another user might be modifying it. If you choose this option, there is no guarantee that you will see a consistent view of the file data.

▶ -fileid *file_id*

You can specify a file ID only if the path name you specify in the title option identifies a tape drive. The *file_id* value is a string of 17 or fewer letters, digits, or both; *file_id* cannot contain any space characters.

If the file control block is open for output, the -fileid option specifies the tape file ID to be written into the tape label. If the file is open for input, the file ID you provide is compared to the file ID in the label; if they are not equal, the error condition is signaled.

Note: Do not confuse the tape file ID with the file ID of the file control block.

See the *OpenVOS Commands User's Guide* (R089) for a discussion of tape processing.

▶ -volumeid *volume_id*

You can specify a volume ID only if the path name you specify in the title option identifies a tape drive. The *volume_id* value is a string of six or fewer letters, digits, or both; *volume_id* cannot contain any space characters.

If the file control block is open for output, the -volumeid option specifies the tape volume ID to be written into the tape label. If the file is open for input, the volume ID you provide is compared to the volume ID in the label; if they are not equal, the error condition is signaled.

See the *OpenVOS Commands User's Guide* (R089) for a discussion of tape processing.

▶ -ownerid *owner_id*

You can specify an owner ID only if the path name you specify in the title option identifies a tape drive. The *owner_id* value is a string of 14 or fewer letters, digits, or both; the owner ID must not contain any space characters. If processing an IBM tape, *owner_id* must be 10 or fewer letters, digits, or both.

If the file control block is open for output, the `-ownerid` option specifies the tape owner ID to be written into the tape label. If the file is open for input, the owner ID you provide is compared to the owner ID in the label; if they are not equal, the `error` condition is signaled.

See the *OpenVOS Commands User's Guide* (R089) for a discussion of tape processing.

Examples

The following examples illustrate the `open` statement.

```
open file(f) stream input;

open file(g) title('data_file.94-12-10') print linesize(80)
     pagesize(60);

open file(f) update direct;

open file(f) title('%s1#d02>system>error_file -append') output;
```

The procedure Statement

Purpose

The `procedure` statement marks the beginning, and main entry point, of a procedure.

Syntax

```
name:{procedure} [(parameter [,parameter] ...)]
     {proc     }

[returns(attribute_list)] [recursive] [inline]

[options {            (main)                   }] ;
[        { (max_optimization_level(number))    }]
```

You can specify the `returns`, `recursive`, `inline`, and `options` options in any order, but any parameters must precede them all.

Operands

- *`name`*

  A label prefix that designates the name of the procedure.

- *`parameter`*

  A parameter of the procedure. Each parameter must be declared within the procedure, without a storage-class attribute. A parameter cannot be declared as a structure member. A parameter list for a procedure can contain from 0 to 127 parameters.

- *`attribute_list`*

  A set of data-type attributes describing the scalar value returned by a function.

- *`number`*

  An integer representing the maximum optimization level for a block.

Explanation

A *procedure* is a block of statements initiated by a `procedure` statement and terminated by an `end` statement. The `end` statement that terminates the procedure can reference the same name as the corresponding `procedure` statement. Procedures can contain other procedures, begin blocks, and any PL/I statements.

The `procedure` statement establishes the declaration of the procedure name in the block **containing** the `procedure` statement. Consequently, the name is known in that block and in all contained blocks.

Execution of a `procedure` statement as a consequence of normal program flow from the previous statement has no effect; execution resumes with the statement that follows the procedure's `end` statement. The statements within the procedure are executed only when the procedure is activated by a `call` statement or function reference.

The `procedure` statement establishes the primary entry point for a procedure. Every activation of the procedure at that entry point must be made with an argument list containing one argument for each parameter in the parameter list. Each argument must be capable of being passed to its corresponding parameter when it is invoked.

If you specify the `inline` option, the procedure itself is substituted for the call, and the actual arguments are substituted for the formal parameters. Procedures that use the `inline` option have no stack frame and execute more quickly than procedures that do not use the `inline` option. In order to take advantage of the `inline` option, you must compile the program with optimization level 3 or 4. See Chapter 3 for additional information about `inline` procedures.

If you specify the `returns` option, the procedure is a function. The set of data-type attributes within the `returns` option must describe a scalar value. When the function is invoked at the primary entry point, the value returned by the procedure is converted to that type before being returned as the function value. All `return` statements in the function must specify a return value that can be converted to the specified data type. A function must not execute its own `end` statement. A function cannot be activated by a `call` statement; it can only be activated as the result of a function reference.

If you do not specify the `returns` option, and the procedure is not the main procedure of a program, the procedure can be activated only by a `call` statement. Any `return` statements within such a procedure must not specify a return value.

The `max_optimization_level` option enables you to specify an optimization level for a particular block that is lower than the optimization level you have specified for the compilation unit. If you specify the `max_optimization_level` option for a procedure, the compiler determines which optimization level—that of the compilation unit or the one you specified for the procedure—is lower and compiles the procedure at the lower optimization level. The `max_optimization_level` option applies to all blocks within that block, unless you explicitly specify a lower optimization level for a contained block.

For example, if you compile the program containing the following code fragment at optimization level 3 or 4, the procedure `get_input` will be compiled at optimization level 3. However, if you compile the program at optimization level 2, `get_input` will also be compiled at optimization level 2.

```
get_input: procedure (a,b) options(max_optimization_level(3));
           .
           .
           .
end get_input;
```

See the *VOS PL/I User's Guide* (R145) for information about optimization levels and the arguments to the `pl1` compiler command.

If you call a procedure recursively, you must specify the `recursive` option.

The `options(main)` option designates which external procedure of the program is to receive control when execution begins. If no procedure has the `options(main)` option, the first external procedure receives control, unless you specify a different initial entry point in the `bind` command.

Chapter 3 discusses block activation and argument passing.

Examples

The following program fragment defines the main procedure `first`, which calls two procedures, `p` and `q`. Procedure `p` is a function that takes two arguments and returns a value. Procedure `q` is recursive.

```
first:    procedure;

declare   salary          float bin(24);
declare   string          char(24);
declare   total           fixed bin(15);

          .
          .
          .
     total = p(salary, 40);
     call q(string, 0);
          .
          .
          .
p:   procedure(p_rate, p_num) returns(fixed bin(15));

declare   (p_rate, p_num) float bin(24);
declare   earnings        float bin(24);

     earnings = p_rate * p_num;
     return(earnings);

end p;

q:   procedure(x, loop_count) recursive;

declare   x          char(*);
declare   loop_count  fixed bin(15);

     loop_count = loop_count + 1;
     do while (loop_count <= 3);
         call q(x || x, loop_count);
     end;

end q;
end first;
```

The put Statement

Purpose

The `put` statement writes arithmetic, pictured, and string values to a file, device, or string variable.

Syntax

```
put  ⎡ [file(file_reference)] ⎡ skip [(skips)]      ⎤ [page] ⎤
     ⎢                        ⎣ line(line_number)   ⎦        ⎥
     ⎢                                                       ⎥
     ⎣              string(string_reference)                 ⎦

⎡                  list(output_item [,output_item]...)                           ⎤
⎢ edit(output_item [,output_item]...)(format_item [,format_item]...)            ⎥
⎣                                                                                ⎦
```

You can specify the options in any order, but the format list is part of the `edit` option and **must** immediately follow the output list.

Operands

- *file_reference*
  A reference to a file constant or file-valued function, or a file variable that has been assigned a file value.

- *skips*
  An expression that produces a fixed-point integer value specifying the number of lines, including the current line, to be skipped in the file before output begins. If you specify the `skip` option without a *skips* value, the compiler supplies a default of 1.

- *line_number*
  An expression that produces a fixed-point integer value specifying the line of a print file on which to begin output. The `line` option is evaluated as if it were a `line` format item. Line numbers are determined relative to the top of the page.

- *string_reference*
  A reference to a string variable. If you specify the `string` option, the output from the `put` statement is assigned to this variable instead of being written to a file.

► *`output_item`*

An array reference, structure reference, or scalar-valued expression, or a parenthesized list of output items containing an iterative-do, as shown in the following syntax.

```
(output_item [,output_item] ... iterative_do)
```

No comma separates an iterative-do from its associated list of output items. Note that an output list containing an iterative-do must have two sets of parentheses, one enclosing the list of output items and one enclosing the iterative-do.

► *`format_item`*

A data or control format item describing how the output is to be formatted.

Explanation

If you specify the `file` option, *`file_reference`* represents the file control block that is to receive output. If the file control block is open, it must have the `stream` and `output` attributes. If the file control block is closed, the `put` statement opens it and gives it the `stream` and `output` attributes.

If you specify the `string` option, the `put` statement cannot write more than one line, and either the `list` or `edit` option is required. If you specify the `edit` option with `string`, the format list must not include any `column`, `line`, `page`, `skip`, or `tab` formats.

If you omit both the `file` option and the `string` option, `file(sysprint)` is the default. When `sysprint` is opened, it acquires the `print` attribute by default.

If you specify the `page` option, a page break is inserted and output begins on the next page. If you specify both the `page` and `line` options, the `page` option is evaluated first.

The `skip`, `page`, and `line` options are always evaluated **prior** to writing any output produced by the statement.

A single instance of the `put` statement can be used repeatedly to produce output to a file. If you want to start each new set of output on a new line, you can perform either of the following actions.

- Include the `skip` option in the `put` statement, as shown in the following example.

  ```
  put skip list(a, b, c);
  ```

  This approach has the possible disadvantage that it produces an empty line at the beginning of the output.

- Use two `put` statements, including the newline instruction in the second, as shown in the following example.

  ```
  put file(f) list(a,b,c); put file(f) skip;
  ```

  Because it has no `list` or `edit` option, the second `put` statement only puts a linemark into the stream; the effect is that a line is skipped before the next `put` statement begins its output.

After any skip, page, or line options have been evaluated, the list of output items is evaluated together with any format list. The lists are evaluated from left to right.

A scalar value in the list of output items causes one value to be transmitted to the output stream. If edit is specified, a scalar value uses one data format. If an array variable of *n* elements appears in the output list, *n* values are transmitted and, if edit is specified, *n* data formats are used. Values are transmitted from the array in row-major order. If a structure variable appears in the output list, the values of all members of the structure, and members of all contained substructures, are transmitted. The values are transmitted in left-to-right order. If edit is specified, each value requires one data format.

The number of lines written by a put statement is determined by the following:

- the number and converted size of output items specified
- the skip, line, and page options
- any control formats specified in a format list

Unless control items or options force new lines to be written, transmission begins with the current position of the current line; enough lines are used to receive the entire list of output items.

Formatting PL/I input and output is discussed in Chapter 14.

If a parenthesized output list contains an iterative-do, values are transmitted under control of that iterative-do as if it were a do-group.

Examples

In the following example, the first put statement writes the 10 values of the array a, followed by the value of b, followed by the value of c.

The second put statement writes the value of a(k), and the value of c. Both values are written using the same e format.

The last put statement writes the values of a(1), a(2), a(3), a(4), a(5), and c, in that order. This statement also illustrates the use of the put statement with an iterative-do.

```
declare   a(10)     float bin(24);
declare   (b,c)     float bin(24);
declare   k         fixed bin(15);
          .
          .
          .
     put file(f) list(a,b,c);
     put file(f) edit(a(k),c)(e(14,6));
     put file(f) list((a(k) do k = 1 to 5), c);
```

The read Statement

Purpose

The `read` statement transmits a file record into a program variable or a buffer.

Syntax

```
read file(file_reference) { into(variable_reference) }
                          { set(pointer_reference)   }

[ key(key_value)        ]
[ keyto(key_variable)   ] ;
```

Operands

▶ *`file_reference`*

A reference to a file constant or file-valued function, or a file variable that has been assigned a file value. If the file control block associated with *`file_reference`* has been opened, it must have been given either the `input` or `update` attribute. If the file is closed, the `read` statement opens it and gives it the `record`, `input`, and `sequential` attributes.

▶ *`variable_reference`*

A reference to a variable that receives a copy of the file record. The reference must be to an arithmetic, string, or pictured variable or to an array or structure of such variables. The reference must not be to an array or structure consisting entirely of unaligned bit strings.

▶ *`pointer_reference`*

A reference to a pointer variable that receives the buffer address of a copy of the file record.

▶ *`key_value`*

An expression that can be converted to a varying-length character-string value with a maximum length of 64. The character-string value must represent the key value of a record in the file referenced by *`file_reference`*. A null key value `('')`, while technically invalid in standard PL/I, is not diagnosed in OpenVOS PL/I.

▶ *`key_variable`*

A varying-length character-string variable that receives the key value of the file record. The maximum length of a key value is 64 characters.

Explanation

If you specify the `into` option, the `read` statement copies a record from the record file associated with *`file_reference`* into the storage of the variable referred to by *variable_reference*. The record being read is a copy of storage and must have been produced by a `write` or `rewrite` statement. The `from` option of the `write` or `rewrite` statement and the `into` option of the `read` statement must identify variables with identical data types. If arrays are referenced, the number of dimensions, dimension sizes, and component data types must match. If structures are referenced, the hierarchic organization and member data types must be identical. Furthermore, these variables must not be unaligned bit strings or structures that consist entirely of unaligned bit strings. If you violate these rules, the results are unpredictable.

If you specify the `set` option, the `read` statement copies a record from the file associated with *file_reference* into a buffer associated with the file control block of *file_reference*. The pointer referred to by *pointer_reference* is set to the address of the record. The buffer remains allocated until another `read` operation is performed on the file control block, or until the file is closed. To access the data in the buffer, you must use a based variable that correctly describes the buffered data.

The `key` option is required for direct access, optional for keyed sequential access, and is not allowed for nonkeyed sequential access.

If you specify the `key` option, *key_value* is converted to a character string of up to 64 characters. If the file is open for keyed sequential access, the file is positioned to read the next record whose index-key value is *key_value*. If the file is open for direct access, the key value is further converted to an integer. The file is then positioned to the record having that ordinal position.

If no record satisfies the `key` option, the `key` condition is signaled.

If you do not specify the `key` option, the `read` statement reads the next record following the current record in a sequential or keyed sequential access file. The record read becomes the current record. The only exception to this is when the `just_positioned` switch of the port is set and the current record is not deleted, in which case the current record is read and the position in the file does not change. The `just_positioned` switch is set only by the following operating system subroutines: `s$seq_position`, `s$rel_position`, `s$keyed_position`, and `s$seq_lock_record`.

The `keyto` clause is always optional. If you specify the `keyto` clause, the key value of the record read is returned to *key_variable*.

If you specify the `key` or `keyto` option, the file must have been previously opened with the `keyed` attribute. If the file is closed, the presence of the `key` or `keyto` option does **not** cause implicit file opening with the `keyed` attribute.

OpenVOS PL/I allows you to use a simple form of the `read` statement to read a line from a stream file. When used this way, the `read` statement has the following syntax.

```
read file(file_reference) into(variable_reference);
```

In this case, the target of *variable_reference* must be a varying-length character string. This form of the `read` statement allows you to read a line with a length of up to 80 characters from a stream file. If the line read from the file is longer than *variable_reference*, the `error` condition is signaled. Because this usage is not standard, it makes your program implementation-dependent.

Chapter 14 discusses PL/I input and output. Chapter 6 discusses pointers and based storage.

Examples

The following are examples of the `read` statement.

```
read file(f) into(x);

read file(g) into(y) key(n+1);

read file(f) set(p) keyto(emp_num);
```

The `return` Statement

Purpose

The `return` statement terminates activation of the current procedure and transfers control back to the calling block.

Syntax

return [(*returned_value*)];

Operands

- *returned_value*

  An expression whose value can be converted to the data type specified in the `returns` option at each entry point of the containing procedure. The converted value is returned as the value of the function. Only functions can return a value.

Explanation

If you specify a returned value, the containing procedure must be a function; a `returns` option must appear in the `procedure` statement and all `entry` statements in the procedure. The procedure must be activated by a function reference.

If you do not specify a returned value, the containing procedure cannot be a function; it cannot have a `returns` option in the `procedure` statement or any `entry` statements in the procedure. The procedure must be activated by a `call` statement.

If a `return` statement is executed within a begin block, the block activations of both the begin block and the containing procedure are terminated. The activations of all intervening begin blocks are also terminated. If a begin block is an on-unit, it cannot contain a `return` statement.

Block activation is discussed in Chapter 3.

Examples

The following example uses the `return` statement to terminate the current procedure and transfer control back to the calling procedure.

```
a: procedure;
                 .
                 .
                 .
          result = b();
                 .
                 .
                 .
          return;              /* Return to caller of a */

          b: procedure returns(bit(1) aligned);
                          .
                          .
                          .
                   return('1'b);  /* Return to caller of b */
          end b;

end a;
```

The revert Statement

Purpose

The `revert` statement reverts an on-unit established within the current block activation.

Syntax

```
revert condition_name;
```

Operands

- *`condition_name`*
  The name of a computational, file, or system condition, or a reference to a programmer-defined condition. The reference has the following form.

  ```
  condition(condition_name)
  ```

Explanation

The execution of a `revert` statement cancels any on-unit that has been previously established for the specified condition in the current block activation. If no on-unit has been established for the specified condition in the current block activation, the `revert` statement has no effect and is not an error.

If the condition includes a file reference, the condition is qualified by the file control block associated with that file reference. This means that if `f` and `g` designate different file control blocks, `revert endpage(f)` and `revert endpage(g)` revert different on-units; if `f` and `g` designate the same file control block, the two statements are equivalent.

The `on` statement establishes an on-unit for an exceptional condition. The `signal` statement forces the execution of the on-unit for a particular condition.

Exception handling is discussed in Chapter 15.

Examples

The following example uses the `revert` statement to transfer control back to an on-unit that was defined in a calling procedure elsewhere in the program.

```
declare   not_found      condition;

     on error
          begin;
            .
            .
            .
          end;

     on key(f) stop;

     on condition(not_found)
          call try_next_value;

     revert error;
     revert key(f);
     revert condition(not_found);
```

The rewrite Statement

Purpose

The `rewrite` statement overwrites a record in a keyed update file.

Syntax

```
rewrite file(file_reference) from(variable_reference)

[key(key_value)];
```

You can specify the `file`, `from`, and `key` options in any order.

Operands

- *`file_reference`*
  A reference to a keyed update file.

- *`variable_reference`*
  A reference to a variable whose storage is to replace the current value of the file record.

- *`key_value`*
  An expression whose value can be converted to a varying-length character string of up to 64 characters. The character-string value is the key value of the record that is to be rewritten. Null key values, although technically invalid in standard PL/I, are **not** diagnosed in OpenVOS PL/I.

Explanation

When the `rewrite` statement is executed, the storage of *`variable_reference`* is copied as the new record value.

If the file control block associated with *`file_reference`* is open, it must have the `keyed` and `update` attributes. If the file is not open, it is implicitly opened with the attributes `record`, `update`, and `sequential`. If the file is not declared to have the `keyed` attribute, this implicit opening causes the `error` condition to be signaled.

The `key` option is required for direct access and optional for keyed sequential access.

If you omit the `key` option, the current record is rewritten. Chapter 14 discusses the current position of a file.

If you specify the `key` option, *`key_value`* is converted to a varying-length character-string value with a maximum length of 64. If the file is open for keyed sequential access, the `rewrite` statement operates on the first record with that index-key value. The current position is set to the rewritten record. If the key value is the null string (`''`), the current position of the file does not change.

If the file is open for direct access, *`key_value`* is further converted to an integer value indicating the ordinal position of a file record. The `rewrite` statement operates on that record.

Examples

The following are examples of the `rewrite` statement.

```
rewrite file(f) from(x) key(n+1);

rewrite file(g) from(y(k));
```

The signal Statement

Purpose

The `signal` statement forces the execution of an on-unit.

Syntax

```
signal condition_name;
```

Operands

- *condition_name*

  The name of a computational, file, or system condition, or a reference to a programmer-defined condition. The reference has the following form.

  ```
  condition(condition_name)
  ```

Explanation

When the `signal` statement is executed, the specified condition is signaled. When a condition is signaled, the most recently established on-unit for that condition is executed. See the description of the `on` statement, earlier in this chapter, for information on establishing on-units.

The `signal` statement has two primary uses.

- to signal programmer-defined conditions
- to test on-units during the debugging process

The `on` statement establishes an on-unit for a specific condition. The `revert` statement reverts the on-unit for a specific condition.

Exception handling is discussed in Chapter 15.

Examples

The following example uses the `signal` statement to signal the programmer-defined condition `too_small`.

```
declare   too_small      condition;

     on condition(too_small)
          begin;
               put skip(2) list('The value is too small.');
               signal error;
          end;
          .
          .
          .
     signal condition(too_small);
          .
          .
          .
     signal endpage(f);
```

The stop Statement

Purpose

The `stop` statement terminates program execution.

Syntax

```
stop;
```

Explanation

When a `stop` statement is encountered, program execution—not just the current block activation—is terminated. Any open files are closed just prior to the end of execution.

The `stop` statement is often used to terminate a program if an error or other unwanted condition arises. For this reason, the `stop` statement frequently occurs within on-units.

Examples

The following example uses the `stop` statement to terminate program execution if an error occurs.

```
on error
     begin;
          call s$error(oncode(), MY_NAME, message);
          stop;
     end;
```

The write Statement

Purpose

The `write` statement writes a record to a file.

Syntax

```
write file(file_reference) from(variable_reference)

[keyfrom(key_value)];
```

You can specify the `file`, `from`, and `keyfrom` options in any order.

Operands

- *file_reference*
  A reference to a file constant, file-valued function, or file variable that has been assigned a file value.

- *variable_reference*
  A reference to a variable whose storage is to be copied into the file record. The target of the reference must be an arithmetic, string, or pictured variable or an array or structure of such variables. The target must not be an unaligned bit string or a structure consisting entirely of unaligned bit strings.

- *key_value*
  An expression whose value can be converted to a varying-length character string of up to 64 characters. The character-string value serves as the key value of the new record. A null key value (`''`), while technically invalid, is not diagnosed in OpenVOS PL/I.

Explanation

Execution of a `write` statement writes a record into the file identified by *file_reference*. The new record contains a copy of the storage of the variable identified by *variable_reference*.

If the file control block associated with *file_reference* is open, it must have been opened with either the `stream` or `record` attribute **and** either the `output` or `update` attribute. If the file control block is closed, the `write` statement opens it and gives it the attributes `output`, `record`, and `sequential`.

If the file control block associated with *file_reference* was opened with the `keyed` and `sequential` attributes, you can specify the `keyfrom` clause. If the file was opened with the `direct` attribute, the `keyfrom` clause is required. If the file control block is open for nonkeyed sequential access, the `keyfrom` clause is not allowed.

If you specify the `keyfrom` clause, *`key_value`* is converted to a varying-length character string with a maximum length of 64. If the file is open for keyed sequential access, this value is used as an index key for the written record. The current position of the file is set to that record. If *`key_value`* is the null string (`''`), the current position of the file does not change.

If the file is open for direct access, the key value is further converted to an integer representing an ordinal record position in the file; the new record is written to that position.

If you specify the `keyfrom` option, and a record with the specified key value already exists, the `key` condition is signaled, unless you specified `-duplicatekeys` in the `title` option of the `open` statement.

If you omit the `keyfrom` option, the record is appended to the end of the file and the current position of the file does not change.

If you specify a non-null value in the `keyfrom` option, the current position of the file is reset to the written record; otherwise, the current position of the file does not change.

You can specify the `keyfrom` option only if the file has already been opened with the `keyed` attribute. If the file is closed, the presence of the `keyfrom` option does **not** cause implicit file opening with the `keyed` attribute.

OpenVOS PL/I allows you to use a simple form of the `write` statement to write a line of text to a stream file. When used this way, the `write` statement has the following syntax.

```
write file(file_reference) from(variable_reference);
```

In this case, the target of *`variable_reference`* must be a varying-length character string. This form of the `write` statement allows you to write a line with a length of up to 80 characters to a stream file. Because this usage is not standard, it makes your program implementation-dependent.

PL/I input and output is discussed in Chapter 14.

Examples

The following are examples of the `write` statement.

```
write file(f) from(x);

write file(g) from(y) keyfrom(n+1);
```

Chapter 13: Functions

This chapter discusses the following topics related to PL/I functions.

- "Built-In Functions"
- "OpenVOS-Supplied Functions"

Built-In Functions

OpenVOS PL/I has many built-in functions. Built-in functions differ from user-defined functions in the following ways.

- Whenever a reference to a built-in function is encountered, the function is activated regardless of the context in which the reference occurs.

- You cannot assign a built-in function to an entry variable or pass a built-in function as an argument to an entry parameter.

- You need not declare entry points for built-in functions that take arguments.

- If you declare, with the `builtin` attribute, the name of a built-in function that takes no arguments, you can reference that function without an argument list. See Chapter 7 for more information about the `builtin` attribute.

Some PL/I built-in functions have abbreviations. For a complete list of abbreviations, see Appendix A.

The next three sections discuss the following topics.

- "Pseudovariables"
- "Summary of Built-In Functions"
- "Built-In Function Descriptions"

Pseudovariables

Some of the PL/I built-in functions are also known as pseudovariables. *Pseudovariables* are expressions that appear on the left side of an assignment statement. OpenVOS PL/I supports the following pseudovariables.

- `pageno`
- `string`
- `substr`
- `unspec`

Note that each of these pseudovariables is also the name of an OpenVOS PL/I built-in function. A pseudovariable and the like-named built-in function are related in this way: assuming that nothing intervenes to change the value, the converted value assigned to a pseudovariable is returned by a subsequent reference to the like-named built-in function.

An assignment to the `pageno` pseudovariable has no effect other than changing the value returned by the `pageno` built-in function. An assignment to any other pseudovariable affects the value of a variable in storage.

You cannot use a pseudovariable if the pseudovariable name has been declared to be anything other than a built-in function. To use a pseudovariable in an inner block and the pseudovariable name has been declared as something other than a built-in function in an outer block, you must redeclare the name with the `builtin` attribute in the current block, as illustrated in the following example.

```
a:     procedure;
declare   pageno     fixed bin(31);
          .
          .
          .
     b:    procedure;
     declare   pageno     builtin;
     declare   f          file;

          pageno(f) = 1;

     end b;
end a;
```

The pseudovariables `pageno`, `string`, `substr`, and `unspec` are described with the built-in functions of the same name.

In addition to the PL/I built-in functions and pseudovariables, the operating system supports a number of other functions. The PL/I version of these functions are described in "OpenVOS-Supplied Functions" later in this chapter. The OpenVOS-supplied functions differ from the OpenVOS PL/I built-in functions in that they must be declared with the `entry` statement.

Summary of Built-In Functions

Table 13-2 through Table 13-10 summarize the built-in functions that are described in this chapter. These tables use the symbols in Table 13-1 to describe arguments and results.

Table 13-1. Symbols Representing Data Types of Arguments and Results

Symbol	Description
A	Array
B	Bit string
C	Character string
D	Any named object
E	Entry value
F	File value
I	2-byte integer
J	Any integer
K	Integer constant
L	4-byte integer
N	Any numeric value
P	Pointer value
Q	Fixed-point decimal with nonzero scaling factor
R	Floating-point value
S	Any string: character or bit
U	Fixed-length character string
V	Varying-length character string
V(*n*)	Varying-length character string with maximum length
X	Any arithmetic or string value
Z	Pictured value

Note: When 8-byte integer operands are passed to built-in functions that accept only 2- or 4-byte integers (for example, the *p* argument of `addrel`), the compiler converts the 8-byte integers appropriately, with no overflow detection.

Table 13-2 summarizes the arithmetic and mathematical built-in functions.

Table 13-2. Arithmetic and Mathematical Built-In Functions

Function	Arguments	Result	Description
`abs`	*N1*	*N*	Returns absolute value of *N1*
`ceil`	N1	*N*	Rounds *N1* up to integer
`divide`	*N1*, *N2*, *K1* [, *K2*]	*N*	Divides *N1* by *N2*
`exp`	*R1*	*R*	Returns *e* raised to the *R1* power
`floor`	*N1*	*N*	Rounds *N1* down to integer
`log`	*R1*	*R*	Returns natural logarithm of *R1*
`log10`	*R1*	*R*	Returns base-10 logarithm of *R1*
`log2`	*R1*	*R*	Returns base-2 logarithm of *R1*
`max`	*N1*, *N2*	*N*	Returns larger of *N1* and *N2*
`min`	*N1*, *N2*	*N*	Returns lesser of *N1* and *N2*
`mod`	*N1*, *N2*	*N*	Returns remainder of *N1*/*N2*
`round`	*Q1*, *K1*	*N*	Rounds *Q1* to *K1* decimal digits
`sign`	*N1*	*I*	Returns sign of *N1*: -1, 0, or 1
`sqrt`	*R1*	*R*	Returns square root of *R1*
`trunc`	*N1*	*N*	Returns integer part of *N1*

Table 13-3 summarizes the trigonometric built-in functions.

Table 13-3. Trigonometric Built-In Functions

Function	Arguments	Result	Description
`acos`	*`R1`*	*`R`*	Returns arc cosine in radians of *`R1`*
`asin`	*`R1`*	*`R`*	Returns arc sine in radians of *`R1`*
`atan`	*`R1`* [,*`R2`*]	*`R`*	Returns arc tangent in radians of *`R1`* or *`R1/R2`*
`atand`	*`R1`* [,*`R2`*]	*`R`*	Returns arc tangent in degrees of *`R1`* or *`R1/R2`*
`atanh`	*`R1`*	*`R`*	Returns hyperbolic arc tangent of *`R1`*
`cos`	*`R1`*	*`R`*	Returns cosine of *`R1`* (*`R1`* in radians)
`cosd`	*`R1`*	*`R`*	Returns cosine of *`R1`* (*`R1`* in degrees)
`cosh`	*`R1`*	*`R`*	Returns hyperbolic cosine of *`R1`* (*`R1`* in radians)
`sin`	*`R1`*	*`R`*	Returns sine of *`R1`* (*`R1`* in radians)
`sind`	*`R1`*	*`R`*	Returns sine of *`R1`* (*`R1`* in degrees)
`sinh`	*`R1`*	*`R`*	Returns hyperbolic sine of *`R1`* (*`R1`* in radians)
`tan`	*`R1`*	*`R`*	Returns tangent of *`R1`* (*`R1`* in radians)
`tand`	*`R1`*	*`R`*	Returns tangent of *`R1`* (*`R1`* in degrees)
`tanh`	*`R1`*	*`R`*	Returns hyperbolic tangent of *`R1`* (*`R1`* in radians)

Table 13-4 summarizes the string built-in functions.

Table 13-4. String Built-In Functions

Function	Arguments	Result	Description
bool	*B1, B2, B3*	*B*	Performs Boolean operation on *B1* and *B2*
collate	None	*C*	Returns ASCII collating sequence
copy	*S1,J1*	*S*	Concatenates *J1* occurrences of *S1*
index	*S1,S2*	*I*	Returns position of *S2* in *S1*
length	*S1*	*I*	Returns length of *S1*
ltrim	*C1* [,*C2*]	*C*	Returns *C1* with leftmost *C2* characters removed
maxlength	*S1*	*I*	Returns maximum length of *S1*
rtrim	*C1* [,*C2*]	*C*	Returns *C1* with rightmost *C2* characters removed
scaneq	*C1* [,*C2*]	*I*	Returns length of *C1* before *C2* characters
scanne	*C1* [,*C2*]	*I*	Returns length of *C1* before non-*C2* characters
search	*C1* [,*C2*]	*I*	Returns position of leftmost *C1* character found in *C2*
string	*X1*†	*S*	Converts *X1* to string value
substr	*S1,J1* [,*J2*]	*S*	Returns substring of *S1* beginning at *J1*
translate	*C1,C2* [,*C3*]	*C*	Performs translation of *C1*
trim	*C1, C2, C3*	*C*	Returns *C1* with leftmost *C2* and rightmost *C3* removed
valid	*Z1*	*B*	Checks validity of pictured value *Z1*
verify	*C1,C2*	*I*	Returns position of leftmost *C1* character not found in *C2*

† This argument can also be an array or structure suitable for string-overlay storage sharing.

Table 13-5 summarizes the conversion built-in functions.

Table 13-5. Conversion Built-In Functions

Function	Arguments	Result	Description
`binary`	*X1* [,*K1*]	*N*	Converts *X1* to binary number
`bit`	*X1* [,*J1*]	*B*	Converts *X1* to bit string
`byte`	*J1*	*C*	Returns character whose rank is *J1*
`character`	*X1* [,*J1*]	*C*	Converts *X1* to character string
`convert`	*D1*, *X1*	*D*	Converts *X1* to the data type, precision, and scale of *D1*
`decimal`	*X1* [,*K1* [,*K2*]]	*N*	Converts *X1* to decimal number
`fixed`	*X1*,*K1* [,*K2*]	*N*	Converts *X1* to fixed-point value
`float`	*X1*,*K1*	*N*	Converts *X1* to floating-point value
`rank`	*C1*	*I*	Returns position of *C1* in ASCII sequence

Table 13-6 summarizes the condition built-in functions.

Table 13-6. Condition Built-In Functions

Function	Arguments	Result	Description
`oncode`	None	*I*	Returns status code for current condition
`onfile`	None	*C*	Returns file ID for file condition
`onkey`	None	*C*	Returns key value for `key` condition
`onloc`	None	*C*	Returns block where condition was signaled

Table 13-7 summarizes the pointer built-in functions.

Table 13-7. Pointer Built-In Functions

Function	Arguments	Result	Description
`addr`	*D1*	*P*	Returns address of *D1*
`addrel`	*P1, J1*	*P*	Increments pointer *P1* by *J1* bytes
`entryinfo`	None	*P*	Returns pointer to `entry_info` structure
`null`	None	*P*	Returns the null pointer value
`paramptr`	*I1, P1*	*P*	Returns pointer to *n*th parameter of the routine pointed to by *P1*
`pointer`	*L1*	*P*	Returns pointer to byte *L1*
`rel`	*P1*	*L*	Returns ordinal byte addressed by *P1*

Table 13-8 summarizes the array built-in functions.

Table 13-8. Array Built-In Functions

Function	Arguments	Result	Description
`dimension`	*A1, K1*	*L*	Returns extent of *K1* dimension of *A1*
`hbound`	*A1, K1*	*L*	Returns upper bound of *K1* dimension of *A1*
`lbound`	*A1, K1*	*L*	Returns lower bound of *K1* dimension of *A1*

Table 13-9 summarizes the National Language Support (NLS) built-in functions.

Table 13-9. NLS Built-In Functions

Function	Arguments	Result	Description
`charcode`	*C*	*I*	Returns character code of *C*
`charwidth`	*N*	*I*	Returns size of characters in set
`collateascii`	None	*C*	Returns string of ASCII characters
`iclen`	*C* [, *N*]	*I*	Returns length of *C* in bytes
`lockingcharcode`	*C*	*I*	Performs encoding of *C*
`lockingshiftintroducer`	None	*C*	Returns right control character
`lockingshiftselector`	*N*	*C*	Returns right control character of *N*
`shift`	*C* [, *N*]	*V*	Returns canonical NLS string
`singleshiftchar`	*N*	*C*	Returns right control character of *N*
`unshift`	*V(n)* [, *n*]	*V*	Returns ambiguous character string in default character set

Table 13-10 summarizes the remaining built-in functions.

Table 13-10. Miscellaneous Built-In Functions

Function	Arguments	Result	Description
`bytesize`	*D1*	*L*	Returns number of bytes required to store *D1*
`date`	None	*C*	Returns current date
`datetime`	None	*C*	Returns current date-time
`lineno`	*F1*	*I*	Returns current line number of *F1*
`pageno`	*F1*	*I*	Returns current page number of *F1*
`size`	*D1*	*L*	Returns number of bits to store *D1*
`time`	None	*C*	Returns current time
`unspec`	*D1*	*B*	Returns storage of *D1*

Built-In Function Descriptions

This section describes, in alphabetical order, the built-in functions supported by OpenVOS PL/I.

▶ `abs(x)`
The `abs` function returns the absolute value of a number.

The value *x* must be arithmetic. The result is the absolute value of *x*. The result has the same data type as *x*.

The following table shows some sample results of the `abs` function.

x	**abs(x)**
12.4	12.4
-1.9E-03	1.9E-03
0	0

▶ `acos(x)`
The `acos` function returns the arc cosine of a floating-point number.

The value *x* must be a floating-point number within the following range.

```
-1 <= x <= 1
```

The result is the arc cosine of *x* expressed in radians, in the range 0 to π. The result has the same data type as *x*.

For example, the result of `acos(0.0000E+00)` is `1.5708E+00`.

▶ `addr(x)`
The `addr` function returns a pointer value to an area of storage.

The value *x* can be any named object. The result is a pointer value to the storage referenced by *x*.

The target of the reference *x* must not be an unaligned bit string or a structure or array consisting entirely of unaligned bit strings. If *x* is a reference to a parameter, the argument corresponding to the parameter must not have been passed by value and the argument must not be an array that is a member of a dimensioned structure. The storage of such an array is fragmented and cannot be accessed by a pointer and a based variable.

For example, the statement `p = addr(code)` sets the pointer `p` to point at the storage of `code`.

▶ `addrel(p,n)`
The `addrel` function adds a specified number of bytes to a pointer address.

The value *p* must be a pointer, and *n* must be a fixed-point integer value.

The result is the sum of the value of the pointer *p* and the integer *n*. The result has the pointer data type.

The following example illustrates the `addrel` function.

```
declare   headp           pointer;
declare   1  struct       ,
             2  first     fixed bin(15),
             2  second    fixed bin(31);

          headp = addr(struct.first);
          headp = addrel(headp,2);
```

The first assignment statement sets the pointer `headp` to the address of `struct.first`. The second assignment statement changes the value of `headp` to the address of `struct.second`.

The `addrel` built-in function is an OpenVOS extension.

▶ `asin(`*x*`)`
The `asin` function returns the arc sine of a floating-point number.

The value of *x* must be a floating-point number within the following range.

```
-1 <= x <= 1
```

The result is the arc sine of *x*, expressed in radians, in the range -π to π. The result has the same data type as *x*. For example, the result of `asin(1.00E+00)` is `1.57E+00`.

▶ `atan(`*x* [`,`*y*]`)`
The `atan` function returns the arc tangent, in radians, of a floating-point value.

Both *x* and *y*, if specified, must be floating-point values. If you specify both *x* and *y*, they cannot both be zero.

If you specify *y*, the result is the arc tangent, in radians, of *x*`/`*y*. If you omit *y*, the result is the arc tangent in radians of *x*.

The result *r* is in the following range.

```
-π <= r <= π
```

If you specify *y*, the result has the common data type and the maximum precision of *x* and *y*. If you omit *y*, the result has the same data type as *x*. Chapter 9 explains how common data types are determined.

For example, the result of `atan(1.000000E+00)` is `7.853982E-01`; the result of `atan(3.0E+00,4.0E+00)` is `6.4E-01`.

▶ `atand(`*x* [`,`*y*]`)`
The `atand` function returns the arc tangent, in degrees, of a floating-point number.

Both *x* and, if specified, *y* must be floating-point values. If you specify both *x* and *y*, they cannot both be zero.

If you specify *y*, the result is the arc tangent, in degrees, of `x/y`. If you omit *y*, the result is the arc tangent, in degrees, of *x*.

The result, *r*, is in the following range.

```
-90 <= r <= 90
```

If you specify *y*, the result has the common data type and the maximum precision of *x* and *y*. If you omit *y*, the result has the data type of *x*. Chapter 9 explains how common data types are determined.

For example, the result of `atand(1.0000E+00)` is `4.5000E+01`; the result of `atand(3.00E+00,4.00E+00)` is `3.69E+01`.

- `atanh(x)`

The `atanh` function returns the hyperbolic arc tangent of a floating-point number.

The value of *x* must be a floating-point number within the following range.

```
-1 < x < 1
```

The result is the hyperbolic arc tangent of *x*. The result has the same data type as *x*.

For example, the result of `atanh(0.000000E+00)` is `0.000000E+00`; the result of `atanh(-5.00E+01)` is `-5.49E-01`.

- $\left\{\begin{array}{c}\texttt{binary}\\\texttt{bin}\end{array}\right\}$ `(x [,p])`

The `binary` function converts a number or a string to a binary value.

The reference *x* must represent either an arithmetic or string value. If *x* is a fixed-point decimal value with a nonzero scaling factor, then *p*—an integer constant specifying the precision of the result—is required.

The result is the value *x* converted to the binary base. If *x* is a floating-point value, the result has the floating-point binary data type; otherwise, the result has the fixed-point binary data type. If you specify *p*, it must be a positive integer constant; the result then has precision `(p)`; otherwise, the precision is determined by the standard rules for data-type conversion, as explained in Chapter 5.

The *p* argument cannot go beyond the maximum precision allowed for the result type. The maximum precision for the floating-point binary data type is 53, and the maximum precision for the fixed-point binary data type is the value of `$MAX_FIXED_BIN`.

For example, the result of `binary('090')` is the 2-byte fixed-point binary value 90.

- `bit(s [,l])`

The `bit` function converts a number or a string to a bit string.

The reference *s* must represent an arithmetic or string value. If you specify a value for *l*, the length of the resultant string, the value must be a non-negative fixed-point integer. If you do not specify `l`, the length is determined by the standard rules for data-type conversions as explained in Chapter 5.

The result is the value of *s* converted to a bit string.

For example, the result of `bit('010')` is `'010'b`; the result of `bit(32,2)` is `'01'b` (the constant `32` converts to the bit string `'0100000'b`).

▶ `bool(x,y,z)`
The `bool` function performs a Boolean operation on two bit strings and returns the resultant bit string.

Both *x* and *y* must be bit-string values, and *z* must be a bit-string constant of length 4. The result is a bit string whose length is the maximum of the lengths of *x* and *y*. The value of each *i*th bit of the resultant string is determined by the corresponding bit in *x* and *y*, as follows:

x(i)	***y(i)***	**Result(*i*)**
0	0	First bit of *z*
0	1	Second bit of *z*
1	0	Third bit of *z*
1	1	Fourth bit of *z*

If *x* and *y* are null strings, the result is a null string. If *x* and *y* are different lengths, the shorter of the two is padded on the right with zero bits to make the lengths equal.

The following example illustrates the `bool` function.

```
declare    (x,r)      bit(7) aligned;
declare    y          bit(4) aligned;
%replace   AND        by '0001'b;
%replace   OR         by '0111'b;
%replace   EX_OR      by '0110'b;
%replace   IMPLY      by '1101'b;
%replace   IFF        by '1001'b;

     x = '1100110'b;
     y = '0101'b;
     r = bool(x,y,EX_OR);  /* r = x or y, exclusive */
```

The resultant value of `r` is `'1001110'b`.

The `bool` built-in function is an OpenVOS extension.

▶ `byte(x)`
The `byte` function returns the ASCII character corresponding to the specified value.

The value of *x* must be a fixed-point integer. The result is a character string of length 1 containing the character whose rank in the ASCII collating sequence is the rightmost 8 bits of *x*. (Note that only the characters whose rank is between 33 and 127 are printing characters. The rest are nonprinting characters.) Appendix D lists the ASCII character set.

The following table shows some sample values produced by the `byte` function.

x	**bit(x)**	**byte(x)**
38	'0100110'b	'&'
65	'1000001'b	'A'
90	'1011010'b	'Z'
109	'1101101'b	'm'

The `byte` built-in function is an OpenVOS extension.

▶ `bytesize(`*x*`)`
The `bytesize` function returns the number of bytes of storage needed to allocate the specified variable.

The value of *x* must be a nonsubscripted reference to a nonmember variable. The result is the `fixed bin(31)` value indicating the number of bytes of storage necessary to allocate the variable *x*.

The following example illustrates the `bytesize` function.

```
declare  fx              fixed bin(15) based;
declare  fx_bsize        fixed bin(15);
declare  struct_bsize    fixed bin(15);
declare  1  struct     ,
           2  first    char(4),
           2  second   fixed bin(31),
           2  third    float dec(7);

     fx_bsize = bytesize(fx);
     struct_bsize = bytesize(struct);
```

The `bytesize` built-in function is an OpenVOS extension.

▶ `ceil(`*x*`)`
The `ceil` function returns the smallest integer greater than or equal to the specified value.

The value of *x* must be arithmetic. The result is the smallest integer greater than or equal to *x*, with the same data type as *x*. However, if *x* is a fixed-point value, the scaling factor of the result is zero and the precision is determined as follows:

Data Type of *x*	**Data Type of `ceil(`*x*`)`**
`fixed bin(`*p*`)`	`fixed bin(`*p*`+1)`
`fixed dec(`*p*`)`	`fixed dec(`*p*`+1)`
`fixed dec(`*p*`,`*q*`)`	`fixed dec(`*p*`-`*q*`+1)`

Note: If the calculated precision exceeds the maximum allowed for the given base and scale, the maximum is used; if the calculated precision is less than 1, then 1 is used.

The complete precision formulae for fixed-point results are as follows:

Base and Scale	Result Precision
`fixed bin`	`(min(`*N*`, `*p*`+1))`
`fixed dec`	`(min(18, max(`*p*`-`*q*`+1, 1)))`

In the preceding table, the maximum precision (*N*) is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.

The following table shows some sample results produced by the `ceil` function.

x	`ceil(`*x*`)`
`3.1`	`4`
`-3.1`	`-3`
`0`	`0`

▶ { `character` | `char` } `(`*s* [`,`*l*]`)`

The `character` function converts a number or a string to a character string.

The reference *s* must represent an arithmetic or string value. The result is the value of *s* converted to a character string. If you specify *l*, the length of the result, the value must be a non-negative fixed-point integer.

If you specify *l*, the result is a character string of length *l*; if you do not specify *l*, the length of the result is determined by the rules for data-type conversion provided in Chapter 5.

The following example illustrates the `character` function.

```
declare   sh   fixed bin(15);
declare   lg   fixed bin(31);
declare   str  char(15) varying;
declare   str2 char(15) varying;

     sh = 24;
     lg = 50000;
     str = char(sh);
     str2 = char(lg);
```

The current length of `str` is `9` and its value is `'       24'`; the current length of `str2` is `14` and its value is `'         50000'`.

- `charcode(`*`c`*`)`

  The `charcode` function returns a value representing the NLS encoding of the specified character.

  The value of *c* is a single 1-byte character. The result is a `fixed bin(15)` integer value representing the NLS encoding. If *c* is a single-shift character, `charcode` returns the character-set number associated with the character. If *c* is not a single-shift character, `charcode` returns a value signifying the NLS character category to which *c* belongs.

  The following table lists the possible values for *c* and the corresponding values that `charcode` returns. The table also contains the name and meaning associated with *c*.

Value of *c* (in Hex.)	**Return Value**	**Name: Description**
00 to 0E	-1	`C0`: Left (ASCII) control character
0F	0	`SS0`: Single-shift character for character set 0
10 to 1F	-1	`C0`: Left (ASCII) control character
20 to 7E	-2	`G0`: Left (ASCII) graphic character
7F	-6	`DEL`: Delete character
80	1	`SS1`: Single-shift character for character set 1
81	4	`SS4`: Single-shift character for character set 4
82	5	`SS5`: Single-shift character for character set 5
83	6	`SS6`: Single-shift character for character set 6
84	7	`SS7`: Single-shift character for character set 7
85	8	`SS8`: Single-shift character for character set 8
86	9	`SS9`: Single-shift character for character set 9
87	10	`SS10`: Single-shift character for character set 10
88	11	`SS11`: Single-shift character for character set 11
89	12	`SS12`: Single-shift character for character set 12
8A	13	`SS13`: Single-shift character for character set 13
8B	14	`SS14`: Single-shift character for character set 14
8C	15	`SS15`: Single-shift character for character set 15
8D	-3	`C1`: Right control character
8E	2	`SS2`: Single-shift character for character set 2
8F	3	`SS3`: Single-shift character for character set 3
90	-5	`LSI`: Locking-shift introducer character
91 to 9F	-3	`C1`: Right control character (except LSI, SS*n*)
A0 to FF	-4	`G1`: Right graphic character

See the *National Language Support User's Guide* (R212) more information about NLS.

The `charcode` built-in function is an OpenVOS extension.

▶ `charwidth(`*`n`*`)`
The `charwidth` function returns the number of bytes occupied by a character in the specified character set.

The value of *`n`* represents a character set number. The result is a `fixed bin(15)` integer value corresponding to the number of bytes occupied by a character in the character set represented by *`n`*.

The following table lists the value that `charwidth` returns for each character set.

Character Set Number	**Character-Set Defined Constant**	**Return Value**
0	`ASCII_CHAR_SET`	1
1	`LATIN_1_CHAR_SET`	1
2	`KANJI_CHAR_SET`	2
3	`KATAKANA_CHAR_SET`	1
4	`HANGUL_CHAR_SET`	2
5	`SIMPLIFIED_CHINESE_CHAR_SET`	2
6	`CHINESE1_CHAR_SET`	2
7	`CHINESE2_CHAR_SET`	2
8	`USER_DOUBLE_BYTE_CHAR_SET`	2

See the *National Language Support User's Guide* (R212) for more information about NLS.

The `charwidth` built-in function is an OpenVOS extension.

▶ `collate [()]`
The `collate` function returns a string representing the computer's internal character code set. This function takes no arguments.

The result is a 256-byte character string representing the character set of the computer in ascending order, from 00x to FFx. The first 128 bytes contain the 128 ASCII characters in ascending order; the remaining 128 bytes contain the supplementary default character set. The system-wide default character set is Latin alphabet No. 1. Your system administrator can change the default.

Note that the character set must be one of the following single-byte character sets: ASCII, Latin alphabet No. 1, or katakana. The `collate` function is not supported for double-byte character sets such as kanji, hangul, or graphics characters. Appendix D lists the ASCII character set.

You must specify the empty argument list (`()`), unless you declare the name `collate` with the `builtin` attribute, as shown in the following example.

```
declare   collate   builtin;
declare   long_str  char(256);

     long_str = collate;
```

▶ `collateascii [()]`

The `collateascii` function returns a string representing the ASCII character set. This function takes no arguments.

The result is a 128-byte character string representing the ASCII character set in ascending order; that is, from 00x to 7Fx.

The `collateascii` built-in function is similar to the `collate` built-in function, which returns a 256-byte character string containing values from 00x to FFx in sequence. Appendix D lists the ASCII character set.

You must specify the empty argument list (`()`), unless you declare the name `collateascii` with the `builtin` attribute, as shown in the following example.

```
declare   collateascii  builtin;
declare   short_str  char(128);

     short_str = collateascii;
```

The `collateascii` built-in function is an OpenVOS extension.

▶ `convert(`*`t,e`*`)`

The `convert` function converts an expression to the data type, precision, and scale of the specified variable.

The *`t`* must be a reference to a scalar variable, and *`e`* must be an arithmetic or string value. The result is the expression *`e`* converted to the data type, precision, and scale of *`t`*.

The following example illustrates the `convert` function.

```
declare a      float bin(24);
declare b      fixed bin(15);

     b = 23;
     put skip list (b, ' , ', convert (a,b));
```

The output of the preceding example is `23    ,     2.3000000E+01`.

The `convert` function is an OpenVOS PL/I extension.

▶ `copy(`*`s,n`*`)`

The `copy` function concatenates a specified number of copies of a string in order to create a longer string.

The value of *s* must be a character string or a bit string. The value of *n* must be a positive fixed-point integer. The result is a character string created by concatenating *n* occurrences of *s*.

If *s* is a bit-string value, the result is a bit string. If *s* is a character-string value, the result is a character string. The length of the resultant string is the product of *n* and the length of *s*.

The following example illustrates the `copy` function.

```
declare   str       char(8);
declare   bstr      bit(8);

     str = copy('abcd',2);     /* str = 'abcdabcd'    */
     bstr = copy('01'b,4);     /* bstr = '01010101'b  */
     str = copy('0',6);        /* str = '000000  '    */
     bstr = copy('1'b,3);      /* bstr = '11100000'b  */
```

▶ `cos(`*x*`)`
The `cos` function returns the cosine, in radians, of an angle.

The value of *x* must be a floating-point number representing the radian measure of an angle. The result is the cosine of angle *x*. The result has the same data type as *x*.

For example, the result of `cos(1.57E+00)` is `7.96E-04`; the result of `cos(0.000000E+00)` is `1.000000E+00`.

▶ `cosd(`*x*`)`
The `cosd` function returns the cosine, in degrees, of an angle.

The value of *x* must be a floating-point number representing the measure of an angle in degrees. The result is the cosine of angle *x*. The result has the same data type as *x*.

For example, the result of `cosd(4.500000E+01)` is `7.071068E-01`; the result of `cosd(3.686333E+01)` is `8.000687E-01`.

▶ `cosh(`*x*`)`
The `cosh` function returns the hyperbolic cosine of an angle.

The value of *x* must be a floating-point number representing the radian measure of an angle. The result is the hyperbolic cosine of angle *x*. The result has the same data type as *x*.

For example, the result of `cosh(1.57E+00)` is `2.51E+00`; the result of `cosh(0.000000E+00)` is `1.000000E+00`.

▶ `date [()]`
The `date` function returns a string representing the system date. This function takes no arguments.

The result is a character string of length 6 that represents the system date. The string has the following form: *yymmdd*. The term *yy* represents the year and is in the range 00 to 99. The term *mm* represents the month and is in the range 01 to 12. The term *dd* represents the day of the month and is in the range 01 to 31.

You must specify the empty argument list (`()`), unless you declare the name `date` with the `builtin` attribute, as in the following example.

```
declare   str       char(6);
declare   date      builtin;

          str = date;
```

▶ `datetime [()]`

The `datetime` function returns a string representing the system date-time. This function takes no arguments.

The result is a character string of length 14 that represents the system date-time. The string has the following form: *`YYYYMMDDhhmmss`*.

The following table lists the components of the 14-character string.

Component	Definition	Values
`YYYY`	Year	`0001` through `9999`
`MM`	Month	`01` through `12`
`DD`	Day	`01` through `31`
`hh`	Hour	`00` through `23`
`mm`	Minutes	`00` through `59`
`ss`	Seconds	`00` through `59`

For example, if the current date and time is August 29, 1997, 10:15 a.m., the `datetime` function returns the following string.

```
19970829101500
```

The `datetime` built-in function is an OpenVOS extension.

▶ `{decimal | dec}(x [,p [,q]])`

The `decimal` function converts a number or a string to a decimal value.

The reference *x* must represent an arithmetic or string value. If specified, *p* (the precision of the result) must be a positive integer constant. Similarly, *q* (the scaling factor of the result), if specified, must be an integer constant. The result is the value *x* converted to a decimal value.

If *x* is a floating-point value, the result has the floating-point decimal data type; otherwise, the result has the fixed-point decimal data type. If you omit *p*, the precision of the result is determined by the rules for data-type conversions, as explained in Chapter 5. If the result is a floating-point value, you **cannot** specify *q*. If the result is a fixed-point value, *q* represents the scaling factor of the result. If you specify *p* and omit *q*, a scaling factor of zero is assumed.

For example, the result of `decimal('10',5,2)` is the decimal value 010.00; the result of `decimal(1.2E-01,4)` is the decimal value 1.200E-01.

▶ { `dimension` | `dim` } `(`*x*`,`*n*`)`

The `dimension` function returns the number of elements in the specified dimension of an array.

The value of *x* must be a reference to an array. The reference *n* must be a positive integer constant. The array referenced by *x* must have at least *n* dimensions. The result is a 4-byte fixed-point binary integer indicating the number of elements in the *n*th dimension of the array referenced by *x*.

The following example illustrates the `dimension` function.

```
declare   matrix(12,2:10)     char(1);
declare   rows                fixed bin(31);
declare   columns             fixed bin(31);

     rows = dim(matrix,1);         /* rows = 12   */
     columns = dim(matrix,2);      /* columns = 9 */
```

▶ `divide(`*x*`,` *y*`,` *p* [`,`*q*]`)`

The `divide` function divides a number by another number.

Both *x* and *y* must represent arithmetic values. The value *p* must be a positive integer constant. The value *q*, if specified, can be zero or it can be a positive or negative integer constant. The result is the value of *x* divided by the value of *y*.

The *p* argument cannot go beyond the maximum precision allowed for the result type.

The result has the common data type of *x* and *y*, and precision *p*. If specified, *q* is the scaling factor of the result. Chapter 9 explains how common data types are determined.

You must **not** specify *q* if the result is a floating-point value.

If the value of *y* is zero, the program is incorrect and the results of continued execution are unpredictable.

The following example illustrates the `divide` function.

```
declare   x    fixed bin(15) static initial(8);
declare   y    fixed bin(31) static initial(4);
declare   z    fixed bin(15);

     z = divide(x,y,15);       /* z = 2 */
```

▶ `entryinfo` [`()`]

The `entryinfo` function returns a pointer to a structure that contains information about the current stack frame. This function takes no arguments.

The result is a pointer to a structure containing information about the current stack frame. The entry's information structure has the following format.

```
declare   1 entry_info,
            2 version    fixed bin(15),
            2 info       fixed bin(15),
            2 frameptr   pointer;
```

In the preceding declaration:

- `entry_info.version` is a version number.
- `entry_info.info` contains the low-order bit of `info` if the currently executing routine is a function whose return value cannot be passed in a register. Otherwise, `entry_info.info` contains the low-order bit. All other unused bits are reset. Note that `entry_info.info` is not used for programs running on ftServer V Series modules.
- `entry_info.frameptr` contains the stack-frame pointer of the currently executing routine.

You must specify the empty argument list (`()`) unless you declare the name `entryinfo` with the `builtin` attribute, as shown in the following example.

```
declare   entryinfo     builtin;
declare   ei_ptr        pointer;

     ei_ptr = entryinfo;
```

The `entryinfo` function is intended to be used in conjunction with the `paramptr` function. See the description of the `paramptr` built-in function, later in this chapter, for more information.

Note that you cannot call the `entryinfo` function from within an inline procedure.

The `entryinfo` built-in function is an OpenVOS extension.

▶ `exp(`*x*`)`
The `exp` function returns the base of *e* raised to the power of the specified value.

The value *x* must be a floating-point number. The function returns the base of the natural logarithm, *e*, raised to the power of *x*; e^x. The value of *e* is approximately 2.718.

The resultant data type is the same as the data type of *x*. For example, the result of `exp(2.000E+00)` is `7.389E+00`.

▶ `fixed(`*x*`,`*p* [`,`*q*]`)`
The `fixed` function converts a number or a string to a fixed-point value.

The value of *x* must be an arithmetic or string value. The value *p* (the precision of the result) must be a positive integer constant. The value of *q* (the scaling factor of the result), if specified, must also be an integer constant. A binary-based value cannot have a nonzero scaling factor.

The *p* argument cannot go beyond the maximum precision allowed for the result type.

The result is *x* converted to a fixed-point arithmetic value according to the data-type conversion rules provided in Chapter 5. If *x* is a decimal, character-string, or picture value, the result has the `decimal` data type; otherwise, the result has the `binary` data type. For example, the result of `fixed(1.23E+02,18)` is the 8-byte decimal integer `123`; the result of `fixed('495',3)` is the 4-byte decimal integer `495`.

- `float(`*x*`,`*p*`)`

The `float` function converts a number or a string to a floating-point value.

The reference *x* must represent an arithmetic or string value. The value *p* (the precision of the result) must be a positive integer constant. The result is the value of *x* converted to a floating-point value according to the rules for data-type conversions provided in Chapter 5.

If *x* is a decimal, character-string, or picture value, the result has the `decimal` data type; otherwise, the result has the `binary` data type.

The following example illustrates the `float` function.

```
declare   dfix      fixed dec(9,2);
declare   dfloat    float dec(7);
declare   cstr      char(13);

     dfix = 12.50;
     dfloat = float(dfix,7);    /* dfloat = 1.250000E+01 */
     cstr = '-8.983369E+03';
     dfloat = float(cstr,7);    /* dfloat = -8.983369E+03 */
```

- `floor(`*x*`)`

The `floor` function returns the largest integer less than or equal to the specified value.

The value of *x* must be arithmetic. The result is the largest integer that is less than or equal to *x*.

The result has the same data type as *x*. However, if the base of *x* is fixed-point, the precision of the result is shown in the following table.

Data Type of *x*	Data Type of `floor(x)`
`fixed bin(p)`	`fixed bin(p+1)`
`fixed dec(p)`	`fixed dec(p+1)`
`fixed dec(p,q)`	`fixed dec(p-q+1)`

Note: If the calculated precision exceeds the maximum allowed for the given base and scale, the maximum is used; if the calculated precision is less than 1, then 1 is used.

The complete precision formulae for fixed-point results are as follows:

Base and Scale	Result Precision
`fixed bin`	`(min(`*N*`,`*p*`+1))`
`fixed dec`	`(min(18,max(`*p*`-`*q*`+1,1)))`

In the preceding table, the maximum precision (*N*) is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.

The following table shows some sample results produced by the `floor` function.

x	**floor(x)**
13.125	13
-3.125	-4
0	0
1.827E+02	1.820E+02

▶ `hbound(`*x*`,`*n*`)`
The `hbound` function returns the upper bound of the specified dimension of an array.

The reference *x* must represent an array variable having at least *n* dimensions. The value *n* must be an integer constant greater than zero. The result is a 4-byte fixed-point binary integer indicating the upper bound of the *n*th dimension of the array *x*.

The following example illustrates the results of the `hbound` function.

```
declare   r                     fixed bin(31);
declare   a(3:5,2,-10:10,4:7)  fixed bin(15);

     r = hbound(a,1);     /* r = 5 */
                   .
                   .
                   .
     r = hbound(a,2);      /* r = 2 */
```

▶ `iclen(`*c* [`,`*n*]`)`
The `iclen` function returns the length, in bytes, of an NLS string.

The value of *c* can be any canonical or common NLS string, or the first portion of such a string. The value of *n* indicates the number of NLS characters in the initial substring of the character string for which `iclen` will calculate the length. The result is a 2-byte integer value indicating the length of *c*, in bytes.

If *n* is specified, and if *c* does not contain at least that many NLS characters, the result is zero. Otherwise, the result is the number of bytes occupied by the first *n* NLS characters.

If *n* is omitted, the result is the number of bytes occupied by all NLS characters in *c*. If *c* is an entire canonical or common NLS string, `iclen(c)` is always identical to `length(c)`; if *c* is the first non-null portion of such a string, `iclen(c)` is between 1 and `length(c)`, inclusive.

The following example illustrates three instances in which `iclen` computes the length of an NLS string.

```
a: procedure;

%options default_char_set = none;

declare   string      char(10);
declare   cv_string1  char(20) varying;
declare   cv_string2  char(20) varying;
declare   length      fixed bin(15);

     string = 'côté';
     cv_string1 = 'côté';

     length = iclen(string, 4);
     put skip list('The length of ', string, 'in string = ', length);

     length = iclen(cv_string1);
     put skip list('The length of ', cv_string1,
                   'in cv_string1 = ', length);

     cv_string2 = unshift(cv_string1, 1);

     length = iclen(cv_string2);
     put skip list('After unshift, the length of ', cv_string2,
                   'in cv_string2 = ', length);

end a;
```

The preceding example produces the following output.

```
The length of  côté in          string =            6
The length of  côté in     cv_string1 =             6
After unshift, the length of  côté in cv_string2 =              4
```

In the preceding example, the `default_char_set` option causes the compiler to store all strings as canonical strings (that is, with a single shift before each right graphic character, even those from Latin alphabet No. 1). Also, the `unshift` function removes single-shift characters from all right graphic characters from Latin alphabet No. 1. See Chapter 11 for more information on the `default_char_set` option. See the description of the `unshift` function, later in this chapter, for more information on `unshift`.

The `iclen` built-in function is an OpenVOS extension.

See the *National Language Support User's Guide* (R212) for more information about NLS.

▶ `index(s,c)`
The `index` function returns the position of the specified substring within a string.

Both *s* and *c* must be character strings, or both must be bit strings. The result is a 2-byte binary integer indicating the position of the substring *c* within the string *s*. The integer indicates the position within *s* of the leftmost character or bit of the substring *c*.

If either *c* or *s* is a null string, the result is zero. If substring *c* is not contained in *s*, the result is zero.

For example, the result of `index('ddedef','def')` is `4`; the result of `index('abc','123')` is `0`.

▶ `lbound(x,n)`
The `lbound` function returns the lower bound of the specified dimension in an array.

The value of *x* must be an array variable having at least *n* dimensions. The value *n* must be an integer constant greater than zero. The result is a 4-byte binary integer indicating the lower bound of the *n*th dimension of the array *x*.

The following example illustrates the `lbound` function.

```
declare   r                   fixed bin(31);
declare   a(3:5,2,-10:10,4:7) fixed bin(15);

     r = lbound(a,1);    /* r = 3 */
               .
               .
               .
     r = lbound(a,2);     /* r = 1 */
```

▶ `length(s)`
The `length` function returns the number of characters or bits in a string.

The value of *s* must be a character string or a bit string. The result is a 2-byte binary integer indicating the number of characters or bits in the current value of string *s*. The null string has a length of zero.

For example, the result of `length('abc')` is `3`; the result of `length('01100'b)` is `5`.

▶ `lineno(x)`
The `lineno` function returns the current line number of the file control block associated with the specified value.

The target of the reference *x* must be a file value associated with a file control block that has been opened for stream output with the `print` attribute. The result is a 2-byte binary integer indicating the current line number of the file control block associated with *x*. See Chapter 4 for additional information about file control blocks.

▶ `lockingcharcode (c)`
The `lockingcharcode` function returns the character-set number associated with the specified locking-shift character.

The value of *c* is a single 1-byte character. The result is a 2-byte value representing the character-set number associated with the locking-shift character specified by *c*.

If *c* is a valid locking-shift character, `lockingcharcode` returns the character-set number associated with the character. If *c* is not a valid locking-shift character, `lockingcharcode` returns the value `-1`.

The following table lists the possible values for *c* and the corresponding values that `lockingcharcode` returns. The table also contains the name and meaning associated with each *c* value.

Value of *c* (in Hex.)	**Return Value**	**Name: Description**
00 to 9F	-1	Invalid value for the *c* argument
A0	1	`LS1`: Locking-shift character for character set 1
A1	4	`LS4`: Locking-shift character for character set 4
A2	5	`LS5`: Locking-shift character for character set 5
A3	6	`LS6`: Locking-shift character for character set 6
A4	7	`LS7`: Locking-shift character for character set 7
A5	8	`LS8`: Locking-shift character for character set 8
A6	9	`LS9`: Locking-shift character for character set 9
A7	10	`LS10`: Locking-shift character for character set 10
A8	11	`LS11`: Locking-shift character for character set 11
A9	12	`LS12`: Locking-shift character for character set 12
AA	13	`LS13`: Locking-shift character for character set 13
AB	14	`LS14`: Locking-shift character for character set 14
AC	15	`LS15`: Locking-shift character for character set 15
AD	-1	Invalid value for the *c* argument
AE	2	`LS2`: Locking-shift character for character set 2
AF	3	`LS3`: Locking-shift character for character set 3
B0 to FF	-1	Invalid value for the *c* argument

See the *National Language Support User's Guide* (R212) for more information about NLS.

The `lockingcharcode` built-in function is an OpenVOS extension.

▶ { lockingshiftintroducer / lsi } ()

The `lockingshiftintroducer` function returns the right control character that corresponds to the locking-shift introducer character.

This function takes no arguments. The result is a single character containing the right control character that corresponds to the locking-shift introducer character. The return value is 90x.

You must specify the empty argument list (`()`), unless you declare the name `lockingshiftintroducer` with the `builtin` attribute, as shown in the following example.

```
declare lsi_char                 char(1);
declare lockingshiftintroducer   builtin;

     lsi_char = lockingshiftintroducer;
```

The `lockingshiftintroducer` built-in function is an OpenVOS extension.

▶ { lockingshiftselector / lss } (*n*)

The `lockingshiftselector` function returns the right control character corresponding to the locking-shift selector for the specified character set.

The value of *n* represents a character set number. The result is a single character containing the right control character corresponding to the locking-shift selector for the character set represented by *n*. Each right graphic character set has a different locking-shift character associated with it.

The following table lists the return value of `lockingshiftselector` for each character set.

Character Set Number	Character-Set Defined Constant	Return Value (in Hex.)
1	LATIN_1_CHAR_SET	A0
2	KANJI_CHAR_SET	AE
3	KATAKANA_CHAR_SET	AF
4	HANGUL_CHAR_SET	A1
5	SIMPLIFIED_CHINESE_CHAR_SET	A2
6	CHINESE1_CHAR_SET	A3
7	CHINESE2_CHAR_SET	A4
8	USER_DOUBLE_BYTE_CHAR_SET	A5

See the *National Language Support User's Guide* (R212) for more information about NLS.

The `lockingshiftselector` built-in function is an OpenVOS extension.

▶ `log(x)`
The `log` function returns the logarithm to the base `e` of a positive floating-point value.

The value of *x* must be a floating-point value greater than zero. The result is the logarithm to the base `e` of *x* (the natural, Naperian, or hyperbolic logarithm of *x*). The result has the same data type as *x*.

The following table shows some sample results produced by the `log` function.

x	**log(x)**
1.000000E+00	0.000000E+00
1.200000E+01	2.484907E+00
1.000000E+02	4.605170E+00

▶ `log10(x)`
The `log10` function returns the logarithm to the base 10 of a positive floating-point value.

The value of *x* must be a floating-point number greater than zero. The result is the logarithm to the base 10 of *x* (that is, the common or Briggsian logarithm of *x*). The result has the same data type as *x*.

The following table shows some sample results produced by the `log10` function.

x	**log10(x)**
1.000000E+00	0.000000E+00
1.200000E+01	1.079181E+00
1.000000E+02	2.000000E+00

▶ `log2(x)`
The `log2` function returns the logarithm to the base 2 of a positive floating-point value.

The value of *x* must be a floating-point number greater than zero. The result is the logarithm to the base 2 of *x*. The result has the same data type as *x*.

The following table shows some sample results produced by the `log2` function.

x	**log2(x)**
1.000000E+00	0.000000E+00
1.200000E+01	3.584963E+00
1.000000E+02	6.643856E+00

▶ `ltrim(s [,c])`
The `ltrim` function removes the specified characters from the left side of a string.

Both *s* and *c* must be of types that can be converted to character strings. If you omit *c*, the value of the second operand defaults to a single space character: (`' '`). The result is the character-string value of *s* with all occurrences of any of the characters in string *c* removed from the left. If the leftmost character in string *s* does not occur in *c*, the result is the character-string value of *s*.

The `ltrim` function is often applied after the `character` function to remove leading space characters from the result of an arithmetic to character-string conversion.

The following table shows some sample results produced by the `ltrim` function.

s	***c***	**`ltrim(s,c)`**
`'abc'`	`'ab'`	`'c'`
`'abc'`	`'ba'`	`'c'`
`'abc'`	`'b'`	`'abc'`
`'aaabc'`	`'a'`	`'bc'`
`'abc'`	Omitted	`'abc'`
`' abc'`	Omitted	`'abc'`
`char(123)`	Omitted	`'123'`

The `ltrim` built-in function is an OpenVOS extension.

► `max(x,y)`
The `max` function converts two values to a common data type and then returns the maximum of the two converted values.

Both *x* and *y* must be arithmetic values. The result has the common data type of *x* and *y*. Chapter 9 explains how common data types are determined.

For example, the result of `max(12,10)` is `12`; the result of `max(1.230000E-02, 27)` is `2.700000E+01`.

► `maxlength(s)`
The `maxlength` function returns the maximum length of a string.

The target of the reference *s* must be either a character-string variable or a bit-string variable. The result is a 2-byte fixed-point integer indicating the maximum number of characters or bits in the string *s*. Any current value of the string *s* has no effect on the value of the `maxlength(s)` function.

The following example illustrates the `maxlength` function.

```
declare   flen      fixed bin(15);
declare   vlen      fixed bin(15);
declare   str       char(10);
declare   vstr      char(24) varying;
declare   bstr      bit(7) aligned;

     vstr = 'abc';
     vlen = maxlength(vstr);  /* vlen = 24 */
     flen = maxlength(str);   /* flen = 10 */
     flen = maxlength(bstr);  /* flen = 7  */
```

The `maxlength` built-in function is an OpenVOS extension.

- `min(`*x*`,`*y*`)`
  The `min` function converts two values to a common data type and then returns the smaller of the two converted values.

  Both *x* and *y* must be arithmetic values. The result has the common data type of *x* and *y*. Chapter 9 explains how common data types are determined.

  For example, the result of `min(37,8)` is `8`; the result of `min(-1,-3.5)` is `-3.5`.

- `mod(`*x*`,`*y*`)`
  The `mod` function divides one value by another value and returns the truncated remainder.

  Both *x* and *y* must be arithmetic values. The result is the truncated remainder produced by dividing *x* by *y*. The result, *n*, has the sign of *y* and the smallest magnitude for which *x*`-`*n* is a multiple of *y*.

  If *y* is zero, the result is *x*; otherwise, the following formula determines the result.

```
x - (y * floor(x / y))
```

The result has the common data type of *x* and *y*. Chapter 9 explains how common data types are determined.

Assume that after conversion to a common type, the precision of *x* is `(`*p*`,`*q*`)` and the precision of *y* is `(`*r*`,`*s*`)`. The following formulae are used to determine the precision of the result.

Common Type	Result Precision
Fixed-point binary	`(min(`*N*`,`*r*`))`
Fixed-point decimal	`(min(18,`*r*`-`*s*`+max(`*q*`,`*s*`)), max(`*q*`,`*s*`))`
Floating-point	`(max(`*p*`,`*r*`))`

In the preceding table, the maximum precision (*N*) is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.

Assume that `x` and `y` are declared as shown in the following example.

```
declare    x    fixed dec(9,3);
declare    y    fixed dec(6,2);
```

The following table shows some sample results of `mod(x,y)`.

x	**y**	**mod(x,y)**
7.000	4.00	3.000
-10.000	-3.00	-1.000
8.000	-3.00	-1.000
-14.000	6.00	4.000
7.510	0.75	0.010
10.059	3.33	0.069
3.125	0.00	3.125
0.000	1.73	0.000

▶ `null [()]`

The `null` function returns the null pointer value. This function takes no arguments.

The result is the null pointer value.

You must specify the empty argument list (`()`), unless you declare the name `null` with the `builtin` attribute, as shown in the following example.

```
declare    p         pointer;
declare    q         pointer static initial(null());
declare    null      builtin;

      p = null;
```

▶ `oncode [()]`

The `oncode` function returns a status code. This function takes no arguments.

The result is a 2-byte binary integer status code indicating the reason the current condition was signaled.

> **Note:** If a break causes a `break` condition, **or** if you signal a condition with the `signal` statement, the result of the oncode is zero.

The `oncode` function is most commonly used within an on-unit. See Chapter 15 for information on conditions and on-units.

You must specify the empty argument list (`()`), unless you declare the name `oncode` with the `builtin` attribute, as shown in the following example.

```
declare   oncode                 builtin;
declare   s$continue_to_signal   entry;
declare   e$abort_output         fixed bin(15) static external;

on warning
     begin;
          if oncode = e$abort_output
          then stop;
          else call s$continue_to_signal;
     end;
```

▶ `onfile [ () ]`

The `onfile` function returns a file ID of the file control block for which the most recent `endfile`, `endpage`, or `key` condition was signaled. This function takes no arguments.

The result is a character string representing the file ID of the file control block for which the most recent `endfile`, `endpage`, or `key` condition was signaled. The file ID is the name of the file constant that owns the file control block.

The `onfile` function is most commonly used within an on-unit. See Chapter 15 for information on PL/I file conditions and on-units. See Chapter 4 and Chapter 14 for information on file control blocks.

You must specify the empty argument list (`()`), unless you declare the name `onfile` with the `builtin` attribute, as shown in the following example.

```
declare   f      file;
declare   g      file;
declare   h      file variable;
declare   onfile builtin;

     on endfile(h)
          begin;
               if onfile = 'f'
               then h = g;
               else goto FINISH;
          end;
```

▶ `onkey [ () ]`

The `onkey` function returns the key value for which the most recent `key` condition was signaled. This function takes no arguments.

The result is a character string containing the key value for which the most recent `key` condition was signaled.

The `onkey` function is most commonly used within an on-unit. See Chapter 15 for information on PL/I file conditions and on-units.

You must specify the empty argument list (`()`), unless you declare the name `onkey` with the `builtin` attribute, as shown in the following example.

```
declare g                  file;
declare onkey              builtin;
declare duplicate_key      entry;

onkey
     begin;
          if onkey = 'last'
          then goto CLEAN_UP;
          else call duplicate_key;
     end;
```

▶ `onloc [()]`
The `onloc` function returns the entry name of the block in which the current condition was signaled. This function takes no arguments.

The result is a character string containing the entry name of the block in which the current condition was signaled. The entry name is the name of the entry point at which the block was entered. If the current block is a begin block, the name has the following format.

```
begin.line_number
```

The value of *`line_number`* is the source module line number on which the `begin` statement that initiated the begin block appears.

See Chapter 15 for information on condition handlers. See Chapter 3 for information on blocks.

You must specify the empty argument list (`()`), unless you declare the name `onloc` with the `builtin` attribute, as shown in the following example.

```
declare   onloc     builtin;

     on error
          begin;
               put skip list('Error ', oncode());
               put skip list('Occurred in block ', onloc);
               stop;
          end;
```

▶ `pageno(x)`
The `pageno` function returns the current page number.

The target of the reference *x* must be a file value associated with a file control block that has been opened for stream output with the `print` attribute. The result is a 2-byte binary integer indicating the current page number in the file control block associated with *x*. If the `pageno` pseudovariable has been set to the value *n*, the `pageno` built-in function returns *n*. See Chapter 4 and Chapter 14 for additional information on file control blocks.

The following example illustrates the `pageno` function.

```
declare   f         file;
declare   thumb     fixed bin(15);

     open file f print;
     pageno(f) = 5;
     thumb = pageno(f);  /* thumb = 5 */
```

The `pageno` pseudovariable is used to alter the value returned by a subsequent reference to the `pageno` built-in function. The assignment statement takes the following form.

`pageno(`*x*`) = `*expression*`;`

The file control block associated with the file reference must be open and must have the attributes `stream output print`.

The expression on the right side of the assignment statement is evaluated and converted to a 2-byte binary integer value. This value is assigned as the current page number of the file control block identified by *x*.

Note that using the `pageno` pseudovariable in an assignment statement has no effect on file output. Assignment to the `pageno` pseudovariable only changes the current page number that serves as the value of the `pageno` built-in function.

The following example illustrates the `pageno` pseudovariable.

```
declare   f              file;
declare   book_mark      fixed bin(15);
          .
          .
          .
     open file(f) title('(master_disk)>data>data_file.(date)')
          stream output print;
     pageno(f) = 10;
     put file(f) list(a,b,c);
     book_mark = pageno(f);
```

In the preceding example, the `put` statement writes to the current page of the file, which is 1. The final assignment statement sets `book_mark` to 10.

▶ `paramptr(`*n*`,`*p*`)`
The `paramptr` function returns a pointer to the specified parameter of a routine.

The result is a pointer to the *n*th parameter of the routine producing the `entry_info` structure. The value of *p* is set by a previous call to the `entryinfo` built-in function.

The following example illustrates the `paramptr` function.

```
declare n             fixed bin(15);
declare p             pointer;
declare ei_ptr        pointer;

     n = 1;                   /* Initializes n to first parameter
*/
     ei_ptr = entryinfo();    /* Returns ptr to entry_info
                                 structure */
     p = paramptr(n, ei_ptr); /* Returns ptr to first parameter of
                                 routine producing entry info
                                 structure */
```

Note that you cannot call the `paramptr` function from within an inline procedure.

The `paramptr` built-in function is an OpenVOS extension.

▶ { `pointer` | `ptr` } `(`*x*`)`

The `pointer` function returns a pointer to the byte with the specified address.

The expression *x* is converted to an integer indicating an absolute byte position. The result is a pointer to the byte with that absolute address.

The `pointer` built-in function is an OpenVOS extension.

▶ `rank(`*x*`)`

The `rank` function returns a character's decimal rank in the ASCII collating sequence.

The reference *x* must represent a nonvarying character string of length 1. The result is a 2-byte binary integer indicating the position (the decimal rank) of the character *x* in the ASCII collating sequence. Appendix D lists the ASCII collating sequence.

For example, the result of `rank('A')` is `65`; the result of `rank('1')` is `49`.

The `rank` built-in function is an OpenVOS extension.

▶ `rel(`*p*`)`

The `rel` function returns the absolute byte number addressed by the specified pointer.

The value of *p* must be a pointer value. The result is a 4-byte integer indicating the absolute byte number addressed by the pointer *p*.

If the result of this function is assigned to a `fixed bin(63)` value, the result is sign-extended.

The result of `rel(null())` is `1`.

The `rel` built-in function is an OpenVOS extension.

- `round(x,n)`

  The `round` function rounds a number.

  The expression *x* must represent a fixed-point decimal value with a nonzero scaling factor. The value *n* must be an integer constant, optionally signed. The result is the value of *x* rounded so that the *n*th decimal position is expressed to its nearest integer.

  The result has the fixed-point decimal data type. If the precision of *x* is `(p,q)`, the precision of the result is as shown in the following formula.

  ```
  (max(1,min(p-q+1+n,18)), n)
  ```

  The following table shows some sample results produced by the `round` function.

x	*n*	**round(*x*,*n*)**
3.2146	3	3.215
-3.5	0	-4
3.00	0	3
37000	-4	40000

- `rtrim(s [,c])`

  The `rtrim` function removes the specified characters from the right side of a string.

  Both *s* and *c* must be data types that can be converted to character-string values. The result is the string *s* with all occurrences of any of the characters in *c* removed from the right. If *c* is omitted, the value of the second operand is taken to be a single space character, (`' '`). If the rightmost character in *s* does not occur in *c*, the result is *s*.

  The following table shows some sample results produced by the `rtrim` function.

s	*c*	**rtrim(*s*,*c*)**
'abcd'	'cd'	'ab'
'abcd'	'dc'	'ab'
'abcd'	'bc'	'abcd'
'abcddd'	'd'	'abc'
'abcd '	'd'	'abcd '
'abcd'	Omitted	'abcd'
'abcd '	Omitted	'abcd'

  The `rtrim` built-in function is an OpenVOS extension.

- `scaneq(s,c)`

  The `scaneq` function returns the length of the longest initial substring in a string that does not contain any of the specified characters.

Both `s` and `c` must be character-string values. The result is a 2-byte binary integer indicating the length of the longest initial substring of `s` that does not contain any of the characters in the string `c`.

If `s` is the null string, the result is zero. If none of the characters in `c` occur in `s`, or if `c` is the null string, the result is the length of `s`.

The following table shows some sample results produced by the `scaneq` function.

s	***c***	**scaneq(*s*,*c*)**
'123a5'	'abc'	3
'a b,c'	' ,'	1
'abc'	''	3
''	'abc'	0
'abc'	'123'	3

The `scaneq` built-in function is an OpenVOS extension.

▶ `scanne(s,c)`
The `scanne` function returns the length of the longest initial substring of a string that consists entirely of characters present in another string.

Both `s` and `c` must be character-string values. The result is a 2-byte binary integer indicating the length of the longest initial substring of `s` consisting entirely of characters present in `c`. If `s` or `c` is the null string, the result is zero.

The following table shows some sample results of the `scanne` function.

s	***c***	**scanne(*s*,*c*)**
'123a5'	'1234567890'	3
'3153'	'1234567890'	4
'abcd'	'wxyz'	0
'abc'	''	0
''	'abc'	0

The `scanne` built-in function is an OpenVOS extension.

▶ `search(s,c)`
The `search` function returns the position of the leftmost character in a string that is found in another string.

Both `s` and `c` must be character-string values. The function returns a 2-byte binary integer indicating the position of the leftmost character in `s` that is found in `c`. If none of the characters in `s` occur in `c`, the result is zero. If either `s` or `c` is the null string, the result is zero.

The following table shows some sample results produced by the `search` function.

s	*c*	`search(`*s*`,`*c*`)`
'wait'	'abc'	2
'138b90'	'abc'	4
'abc'	''	0
''	'abc'	0

The `search` built-in function is an OpenVOS extension.

▶ `shift (`*c* [`,` *n*]`)`
The `shift` function returns a canonical NLS string.

The value of *c* can be any valid NLS character string, not necessarily nonambiguous. The result, an equivalent canonical string, is a varying-length character string, the maximum length of which is two times the current length of *c* (which, therefore, must not exceed 16,383). In a *canonical string*, each non-ASCII character is preceded by a single-shift character.

The source default character set, *n*, is an optional argument. If *n* is omitted, Latin alphabet No. 1 is assumed. Note that if the character string is a constant expression, the default character set of that constant is the character set specified in the `%options` statement (or Latin alphabet No. 1, by default). In this case, it is erroneous to specify, implicitly or explicitly, another default character set.

If *c* contains an invalid NLS string, `shift` returns an ASCII `SUB` character in place of each invalid character, and the `warning` condition is signaled. If no on-unit is established for this condition, a message describing the situation is written to the terminal and the program continues execution as though no condition were signaled. In this case, the oncode value will be one of the following (note, however, that you can examine an oncode **only** within an on-unit).

`e$invalid_right_graphic_char` (4151)
`e$missing_lockshift_selector` (4155)
`e$truncated_locking_shift` (4154)
`e$truncated_multibyte_char` (4153)
`e$truncated_single_shift` (4152)
`e$unknown_character_set` (4156)

▶ `sign(`*x*`)`
The `sign` function returns the sign of a number.

The reference *x* must represent an arithmetic value. The result is a 2-byte integer indicating the sign of *x*, as follows:

- `-1`, if the value of *x* is negative
- `0`, if the value of *x* is zero
- `1`, if the value of *x* is positive

The following table shows some sample results produced by the `sign` function.

x	sign(x)
-17	-1
1.290000E-05	1
0.00000	0

▶ `sin(`*x*`)`
The `sin` function returns the sine, in radians, of an angle.

The reference *x* must be a floating-point value representing the radian measure of an angle. The result is the sine of angle *x*. The result has the same data type as *x*.

For example, the result of `sin(1.5707E+00)` is `1.0000E+00`; the result of `sin(0.000000E+00)` is `0.000000E+00`.

▶ `sind(`*x*`)`
The `sind` function returns the sine, in degrees, of an angle.

The reference *x* must be a floating-point value representing the measure of an angle in degrees. The result is the sine of angle *x*. The result has the same data type as *x*.

For example, the result of `sind(2.700000E+02)` is `-1.000000E+00`; the result of `sind(4.500000E+01)` is `7.071068E-01`.

▶ { `singleshiftchar` | `ss` } `(`*n*`)`

The `singleshiftchar` function returns the right control character corresponding to the single-shift character for the character set.

The value of *n* is a character-set number. The result is a single byte containing the right control character that corresponds to the single-shift character for the character set represented by *n*.

The following table lists the return value of `singleshiftchar` for each character set.

Character Set Number	**Character-Set Defined Constant**	**Return Value (in Hex.)**
0	`ASCII_CHAR_SET`	0F
1	`LATIN_1_CHAR_SET`	80
2	`KANJI_CHAR_SET`	8E
3	`KATAKANA_CHAR_SET`	8F
4	`HANGUL_CHAR_SET`	81
5	`SIMPLIFIED_CHINESE_CHAR_SET`	82
6	`CHINESE1_CHAR_SET`	83
7	`CHINESE2_CHAR_SET`	84
8	`USER_DOUBLE_BYTE_CHAR_SET`	85

See the *National Language Support User's Guide* (R212) for more information about NLS.

The `singleshiftchar` built-in function is an OpenVOS extension.

▶ `sinh(`*`x`*`)`
The `sinh` function returns the hyperbolic sine of an angle.

The reference *x* must be a floating-point value representing the radian measure of an angle. The result is the hyperbolic sine of angle *x*. The result has the same data type as *x*.

For example, the result of `sinh(-1.570700E+00)` is `-2.301057E+00`; the result of `sinh(0.000000E+00)` is `0.000000E+00`.

▶ `size(`*`x`*`)`
The `size` function returns the amount of storage, in bits, needed to allocate the specified variable.

The reference *x* must be a nonsubscripted reference to a nonmember variable. The result is a 4-byte binary integer indicating the number of bits of storage necessary to allocate the variable *x*.

The following example illustrates the `size` function.

```
declare fx                fixed bin(15) based;
declare fx_size           fixed bin(31);
declare struct_size       fixed bin(31);

declare  1 struct        ,
           2  first     char(4),
           2  second    fixed bin(31),
           2  third     float dec(7);

     fx_size = size(fx);           /* fx_size = 16     */
     struct_size = size(struct);   /* struct_size = 96 */
```

The `size` built-in function is an OpenVOS extension.

▶ `sqrt(`*`x`*`)`
The `sqrt` function returns the square root of a floating-point value.

The reference *`x`* must represent a positive, nonzero floating-point value. The result is the positive square root of *`x`*. The result has the same data type as *`x`*.

For example, the result of `sqrt(4.000E+00)` is `2.000E+00`; the result of `sqrt(2.250000E+02)` is `1.500000E+01`.

▶ `string(`*`s`*`)`
The `string` function converts the specified value into a string.

The value of *`s`* must be an arithmetic value, a string value, or an array or structure that contains all bit-string values or all character-string values, and is suitable for defining string overlays, as described in Chapter 6.

The result is the value of *`s`*, converted to a string, according to the rules for data-type conversion provided in Chapter 5. If *`s`* is a bit string or an array or structure consisting entirely of bit strings, the result is a bit string; if *`s`* is a character string or an array or structure consisting entirely of character strings, the result is a character string. The length of the resultant string is determined by the standard rules for data-type conversion.

The following example illustrates the `string` function.

```
declare   array(3)  char(2);
declare   sstr      char(6);

     array(1) = 'ab';
     array(2) = 'cd';
     array(3) = 'ef';
     sstr = string(array);       /* sstr = 'abcdef' */
```

An assignment to the `string` pseudovariable assigns a string value to a string, array, or structure variable. The assignment statement takes the following form.

```
string(s) = expression;
```

The expression on the right of the assignment operator is evaluated and converted to a character-string value or a bit-string value, depending on the data type of the variable referenced within the `string` pseudovariable. The converted value is then assigned to the variable referenced within the `string` pseudovariable as if that variable were a nonvarying string variable. The length of the assigned string value is the total number of characters or bits in the variable referenced within the `string` pseudovariable.

The `string` pseudovariable is most often used to assign a value to an array or structure of string values.

In the following example, the `string` pseudovariable assigns a value to an array of characters.

```
declare   a(5)    char;
          .
          .
          .
     string(a) = 'abcde';
```

As a result of the assignment, `a(1)` has a value of `'a'`, `a(2)` has a value of `'b'`, and so forth.

▶ `substr(`*s*`,`*i* [`,`*j*]`)`
The `substr` function copies the specified part of a string.

The reference *s* must represent a bit string or a character string. The references *i* and *j* must represent fixed-point integer values. The result is a copy of the part of string *s* starting at the *i*th character or bit and having a length of *j*. If you omit *j*, the value of the third operand is `length(`*s*`) - `*i*` + 1`. The data type of the result is a character or bit string of length *j*.

The following restrictions apply to the values of *i* and *j*.

- *i* `>= 1`
- `(`*i* `+` *j* `- 1) <= length(`*s*`)`
- *j* `>= 0`

For example, the result of `substr('abcdefg',3,2)` is `'cd'`; the result of `substr('10110'b,1,5)` is `'10110'b`; the result of `substr('abc',4)` is `''`.

An assignment to the `substr` pseudovariable alters a part of the value of a string variable. The assignment statement takes the following form.

`substr(`*s*`,`*i* [`,`*j*]`) = ` *expression*

The expression on the right side of the assignment statement is evaluated and converted to a character-string value or a bit-string value, depending on the data type of *s*. The substring of the *s* string that begins with the *i* character or bit and continues for *j* characters or bits receives this converted value. The substring is treated as a nonvarying string. The following restrictions apply to the values.

```
1 <= i <= (length(s) + 1)
     (i + j - 1) <= length(s)
```

These restrictions ensure that the pseudovariable evaluates to a legitimate storage location. If *i* indexes a position within *s*, a substring of *j* characters can be accessed without exceeding the length of the string. If *i* is one greater than the length of *s*, the value of *j* is 0, and the assignment has no effect.

If *j* is omitted, its value is calculated using the following formula.

```
(length(s) - i + 1)
```

In such a case, the substring begins with the *i* character or bit and continues to the end of the string. The `length` built-in function is described earlier in this chapter.

The following example illustrates the `substr` pseudovariable.

```
declare   long_str  bit(10) aligned;
          .
          .
          .
      substr(long_str,3,4) = '101'b;
```

In the preceding example, the string `'101'b` is converted to a string of length 4, `'1010'b`, and assigned to the specified substring of `long_str`. After the assignment, the third bit of `long_str` is `'1'b`, the fourth bit is `''b`, the fifth bit is `'1'b`, and the sixth bit is `'0'b`. The other bits of `long_str` are unchanged by the assignment.

If you choose the `-check` argument when you compile your program, the compiler checks for any *i* or *j* values that index positions outside of the string value and inserts code to perform further checking when the program runs.

Note that the `substr` pseudovariable allows you to assign only within the current value of a varying-length character string. Any attempt to index a value beyond the current length is an error.

In the following example, the `substr` pseudovariable is **improperly** used.

```
declare   v_str     char(12) varying;

      v_str = ' ';
      substr(v_str,1,3) = 'xyz'; /* Incorrect use */
```

In the preceding example, because the `substr` pseudovariable references characters outside the current length, its effect is unpredictable. If you compile the program with the `-check` argument, this error is diagnosed at run time, signaling the `error` condition.

► `tan(x)`
The `tan` function returns the tangent, in radians, of an angle.

The target of the reference *x* must be a floating-point value representing the radian measure of an angle. The result is the tangent of angle *x*. The result has the same data type as *x*.

For example, the result of `tan(7.854E-01)` is `1.000E+00`; the result of `tan(0.0E+00)` is `0.0E+00`.

- ▶ `tand(`*x*`)`

  The `tand` function returns the tangent, in degrees, of an angle.

  The target of the reference *x* must be a floating-point value representing the measure of an angle in degrees. The result is the tangent of angle *x*. The result has the same data type as *x*.

  For example, the result of `tand(4.50E+00)` is `1.00E+00`; the result of `tand(0.000000E+00)` is `0.000000E+00`.

- ▶ `tanh(`*x*`)`

  The `tanh` function returns the hyperbolic tangent of an angle.

  The reference *x* must represent a floating-point value. The result is the hyperbolic tangent of the angle whose radian measure is *x*. The result has the same data type as *x*.

  For example, the result of `tanh(7.854E-01)` is `6.558E-01`; the result of `tanh(0.0E+00)` is `0.0E+00`.

- ▶ `time [()]`

  The `time` function returns the current time. This function takes no arguments.

  The result is a character string of length 6 representing the present time. The string has the form *hhmmss*. The term *hh* represents the hour and is in the range 00 to 23; the term *mm* represents minutes and is in the range 00 to 59; the term *ss* represents seconds and is in the range 00 to 59.

  You must specify the empty argument list (`()`), unless you declare the name `time` with the `builtin` attribute.

  For example, at 12:00 noon, the result of `time()` is `'120000'`; at 1:00 a.m., the result of `time()` is `'010000'`; at 3 minutes and 23 seconds after 3:00 p.m., the result of `time()` is `'150323'`.

- ▶ `translate(`*s*`,`*t* `[,`*x*`])`

  The `translate` function replaces certain characters in a string with other characters.

  The references *s*, *t*, and *x* must represent character-string values. The result is a copy of string *s* with each character that appears in *x* replaced by the corresponding character in *t*. The data type of the result is a character string with the same length as *s*.

  For each character, *s*`(`*k*`)`, in *s*, let *t*`(`*i*`)` be `index(`*x*`,s(`*k*`))`. If the value of *i* is zero, the corresponding character in the result is *s*`(`*k*`)`; otherwise, the corresponding character in the result is *t*`(`*i*`)`.

  If *x* is not specified, the value of the third operand is understood to be `collate()`. If *t* is shorter than *x*, *t* is padded on the right with space characters until the lengths are equal. If *s* is the null string, the result is the null string.

The following table shows some sample results produced by the `translate` function.

s	***t***	***x***	**`translate(`*s*`,`*t*`,`*x*`)`**
`'1#2#'`	`'0'`	`'#'`	`'1020'`
`''`	`'abc'`	`'123'`	`''`
`'AbcD!'`	`'ABCD'`	`'abcd'`	`'ABCD!'`

▶ `trim (`*s*`,`[*c1*`,`*c2*]`)`
The `trim` function removes characters from the left and right sides of a string.

The values *s*, *c1*, and *c2* must be values that can be converted to character strings. The result is the string *s* with all occurrences of any of the characters in *c1* removed from the left, and all occurrences of any of the characters of *c2* removed from the right. The characters that occur in *c1* and *c2* are removed from *s* until a character that is not a member of one of these strings occurs. If the null string is specified for either *c1* or *c2*, the corresponding `trim` operation will not be performed on that side.

The following table shows some sample results produced by the `trim` function.

s	***c1***	***c2***	**`trim(`*s*`,`*c1*`,`*c2*`)`**
`'abcde'`	`'a'`	`'e'`	`'bcd'`
`'abcabc'`	`'a'`	`'c'`	`'bcab'`
`'abcde'`	`'a'`	`'d'`	`'bcde'`
`'aabbccdd'`	`'ab'`	`'cd'`	`''`
`'abababab'`	`'a'`	`'b'`	`'bababa'`
`'abcde'`	`'a'`	`''`	`'bcde'`
`'abcde'`	`''`	`'e'`	`'abcd'`
`'abababab'`	`'ab'`	`''`	`''`

The `trim` built-in function is an OpenVOS extension.

▶ `trunc(`*x*`)`
The `trunc` function returns the integer part of a number.

The reference *x* must represent an arithmetic value. The result is the integer part of *x*.

The data type of the result is determined as shown in the following table.

Data Type of *x*	**Data Type of `trunc(`*x*`)`**
`fixed bin(`*p*`)`	`fixed bin(`*p*`+1)`
`fixed dec(`*p*`)`	`fixed dec(`*p*`+1)`
`fixed dec(`*p*`,`*q*`)`	`fixed dec(`*p*`-`*q*`+1)`
`float bin(`*p*`)`	`float bin(`*p*`)`
`float dec(`*p*`)`	`float dec(`*p*`)`

Note: If the calculated number of digits exceeds the maximum allowed for the given base and scale, the maximum is used; if the calculated number of digits is less than one, one is used. The complete precision formulae for fixed-point results are shown in the following table.

Base and Scale	Result Precision
`fixed dec`	(min(18, max(*p*-*q*+1, 1)))
`fixed bin`	(min(*N*, *p*+1))

In the preceding table, the maximum precision (*N*) is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.

The following table shows some sample results produced by the `trunc` function.

x	**trunc(*x*)**
-3.4	-3
6.9076	6
0	0

▶ `unshift(`*v*`(`*n*`)` [, n]`)`
The `unshift` function returns an NLS string from which all single-shift characters are removed.

The value of *v*(*n*) can be any valid unambiguous canonical string or a common string. A *common string* can contain characters from any right graphic character set and has a default character set of Latin alphabet No. 1. In a common string, a single-shift character precedes each right graphic character except for Latin alphabet No. 1 characters. A common string contains no locking-shift characters. Common strings can be written to a terminal or to a text file.

The value of *n* is an integer that identifies the default character set of the output. The result is an equivalent ambiguous NLS string, with the default character set indicated by the optional argument *n*, from which all single-shift characters are removed. If you do not specify *n*, Latin alphabet No. 1 is assumed, and the result is a common NLS string.

The result is a varying-length character string, the maximum length of which is two times the current length of the source string (which, therefore, must not exceed 16,383). The return value can actually be longer than the current length of the input character string **only** if the character set is not Latin alphabet No. 1.

If the input character string contains an invalid NLS string, all invalid sequences are replaced by the ASCII `SUB` character in the output.

In this case, the `warning` condition is also signaled. If no on-unit is established for this condition, a message describing the situation is written to the terminal and the program continues execution as though no condition were signaled. In this case, the oncode value will be one of the following (note, however, that you can examine an oncode **only** within an on-unit).

```
e$invalid_right_graphic_char (4151)
e$missing_lockshift_selector (4155)
e$truncated_locking_shift (4154)
e$truncated_multibyte_char (4153)
e$truncated_single_shift (4152)
e$unknown_character_set (4156)
```

The `unshift` built-in function is an OpenVOS extension.

► `unspec(`*x*`)`
The `unspec` function returns a string containing the internal storage of a variable.

The reference *x* must represent a scalar variable. The result of the function is a bit string containing the internal storage of *x*. The length of the resultant bit string depends on the data type of *x*. See Appendix B for information on data storage.

The value returned by the `unspec` function is implementation-defined.

You can use the `unspec` function with the `unspec` pseudovariable to facilitate untyped storage sharing, as shown in the following example.

```
declare   fib       fixed bin(15);
declare   cstr      char(2);

     cstr = 'ab';   /* cstr storage: '0110000101100010'b */

     unspec(fib) = unspec(cstr);   /* fib = 24930        */
```

An assignment to the `unspec` pseudovariable assigns a bit-string value to the storage of a variable without regard for data type. The assignment statement has the following form.

```
unspec(v) = expression
```

The expression on the right of the assignment operator is evaluated and converted to a bit-string value. The bit-string value is then copied into the storage of the variable that is the target of *v*. The variable referred to by *v* cannot be subsequently used unless the assigned bit-string value represents a valid storage value for that variable. See Appendix B for information on how OpenVOS PL/I values are stored.

Note: Because storage methods are not standardized, using the `unspec` pseudovariable makes a program implementation-dependent.

The following example illustrates the `unspec` pseudovariable.

```
declare   b    fixed bin(15);
declare   c    char(3) varying;

     unspec(b) = '0'b;
     unspec(c) = '000000000000000101000001'b;

     put skip list(b,c);
```

The output from the preceding example is `0 A`.

In the preceding example, because the storage of `b` contains 16 bits, the result of the expression on the right of the assignment operator is converted to a bit string of length 16: `'0000000000000000'b`. When this bit string is subsequently interpreted as a 2-byte binary integer, the string produces the value 0.

A varying-length character string of length 3 is stored in 6 bytes. Therefore, the bit-string value in the second assignment statement of the preceding example is padded on the right with 24 zero bits (to a length of 48) before being assigned to `c`. The storage is subsequently interpreted as a varying-length character-string value. The first 16 bits are interpreted as a 2-byte binary integer representing the length of the string. The length in this case is 1. The next 8 bits are interpreted as an ASCII character, producing the value `'A'`. The remaining bits are ignored.

- ▶ `valid(`*`x`*`)`
The `valid` function indicates whether a value can be edited into the specified picture.

The reference *`x`* must represent a scalar pictured value. The result is a bit string of length 1 that indicates whether the character-string value of *`x`* can be edited into the picture declared for *`x`*. The value is `'1'b` if the character-string value can be edited into the picture; otherwise, the value is `'0'b`.

The following example illustrates the `valid` function.

```
declare   p         picture 'zzz99.999cr';
declare   e         char(11) defined(p);
declare   bstr      bit(1) aligned;

     e = '0.000000000';  /* p = '0.000000000'                */
     bstr = valid(p);    /* bstr = '0'b; value of p is invalid */
```

- ▶ `verify(`*`s,c`*`)`
The `verify` function indicates the leftmost character in one string that is not found in another string.

Both *`s`* and *`c`* must be references to character-string values. The result is a 2-byte binary integer indicating the leftmost character in *`s`* that is not found in *`c`*. The result is zero if each of the characters in *`s`* occurs in *`c`*.

For example, the result of `verify('a,ba/&d','abcde')` is `2`; the result of `verify('01011001','01')` is `0`.

OpenVOS-Supplied Functions

In addition to the built-in functions described earlier in this chapter, Stratus provides a number of other functions and subroutines in the system object library `(master_disk)>system>object_library`. These functions are described in the next two sections.

Unlike built-in functions, you **must** declare OpenVOS-supplied functions in your program. If you are working on your program in an Emacs buffer, you can simplify this task by using the Emacs `declare_subroutine` request. If you type the name of an OpenVOS-supplied function or subroutine as part of a `declare` statement and specify the `declare_subroutine` request (`^Z-D`), Emacs does the following:

- reads the first word (the name of the subroutine) to the left of the cursor
- searches the list of subroutine declarations in the file `pl1_declarations` in the `>system` directory until it finds a match for the word to the left of the cursor
- copies the attributes for the name of the subroutine and inserts them into the current buffer to the right of the cursor

For example, in an Emacs buffer, position the cursor to the right of the OpenVOS-supplied function name `after` in the following `declare` statement.

```
declare after█
```

If you specify the `declare_subroutine` request (`^Z-D`), Emacs inserts the appropriate attributes to the right of the point so that the declaration now appears as shown in the following example.

```
declare after    entry (char (*) var, char (*) var)
                 returns (char (256) var);█
```

Note that attributes are not available for every OpenVOS-supplied function or subroutine. See the *VOS Emacs User's Guide* (R093) for more information on the `declare_subroutine` request.

The next two sections discuss the following topics.

- "Summary of OpenVOS-Supplied Functions"
- "OpenVOS-Supplied Function Descriptions"

Summary of OpenVOS-Supplied Functions

Table 13-11 briefly summarizes the OpenVOS-supplied functions, using the symbols provided in Table 13-1 earlier in this chapter.

Table 13-11. OpenVOS-Supplied Functions

Function	Arguments	Result	Description
`after`	*V1*, *V2*	*V*(256)	Returns substring of *V1* after first *V2*
`ascii_to_ebcdic`†	*I1*, *C1*	*C*	Converts ASCII string to EBCDIC
`before`	*V1*, *V2*	*V*(256)	Returns substring of *V1* before first *V2*
`codeptr`	*E1*	*P*	Returns pointer to first instruction
`displayptr`	*E1*	*P*	Returns display pointer of *E1*
`ebcdic_to_ascii`†	*C1*	*C*	Converts EBCDIC string to ASCII
`hash`	*V1*, *I1*	*I*	Returns hash-code less than *I1* for *V1*
`hex`	*L1*, *I1*	*V*(8)	Returns low-order digits of *L1* in hexadecimal
`hex64`	*V*, *N*	*V*(*n*)	Returns the hexadecimal value of *V*
`hexp`	*P1*, *I1*	*V*(8)	Returns low-order digits of *P1* in hexadecimal
`occurs`	*V1*, *V2*	*I*	Returns occurrences of *V2* in *V1*
`quote`	*V1*	*V*(256)	Returns quoted form of *V1*
`returnptr`	None	*P*	Returns *P* to the instruction to which the calling procedure would return once it finishes executing
`reverse`	*V1*	*V*(256)	Reverses characters in *V1*
`reverseb`†	*B1*	*B1*	Reverses bits in *B1*
`rindex`	*V1*, *V2*	*I*	Returns position of last instance of *V2* in *V1*
`rscaneq`	*V1*, *V2*	*I*	Returns length of *V1* after last *V2* character
`rscaneqf`	*U1*, *V1*	*I*	Returns length of *U1* after last *V1* character
`rscanne`	*V1*, *V2*	*I*	Returns length of *V1* after last non-*V2* character
`rscannef`	*U1*, *V1*	*I*	Returns length of *U1* after last non-*V1* character
`stackframeptr`	None	*P*	Returns pointer to current stack frame

Table 13-11. OpenVOS-Supplied Functions *(Continued)*

Function	Arguments	Result	Description
`staticptr`	*`E1`*	*`P`*	Returns biased pointer to static storage for *`E1`*
`unhex`†	*`V1,L1,I1`*	*`L`*	Converts *`V1`* from hexadecimal to binary
`unquote`	*`V1`*	*`V`*(256)	Returns unquoted form of *`V1`*

† These routines are subroutines, not functions.

OpenVOS-Supplied Function Descriptions

This section describes the OpenVOS-supplied functions in alphabetical order.

▶ `after(`*`s,c`*`)`
The `after` function returns a substring that follows the first occurrence of a string in another string.

Both *`s`* and *`c`* must be varying-length character strings of any length. The result is a varying-length character string with a maximum length of 256.

The function returns the substring of *`s`* that follows the first occurrence of *`c`* in *`s`*.

If *`s`* is the null string (`''`), the result is the null string. If *`c`* is the null string, the result is *`s`*. If *`c`* does not occur in *`s`*, the result is the null string.

The following example illustrates the `after` function.

```
declare   s         char(12) varying;
declare   c         char(4) varying;
declare   rtime     char(256) varying;

declare   after     entry (char(*) varying, char(*) varying)
                    returns (char(256) varying);

     s = 'Time: 8:30pm';
     c = ':';
     rtime = after(s,c);      /* rtime = ' 8:30pm' */
```

Note: The OpenVOS PL/I `after` function differs from the `after` function in full PL/I in two ways.

- The arguments are restricted to varying-length character strings.
- The result is restricted to 256 characters.

▶ `ascii_to_ebcdic(`*`n`*`,`*`s`*`,`*`c`*`)`
The `ascii_to_ebcdic` subroutine converts an ASCII character string to EBCDIC. Note that `ascii_to_ebcdic` is a subroutine, not a function.

The value of *`n`* must be a 2-byte binary integer. The values of *`s`* and *`c`* must be character strings of any length.

The subroutine converts *`n`* characters of *`s`* from ASCII to EBCDIC and returns the result in *`c`*.

The following example illustrates a call to the `ascii_to_ebcdic` subroutine.

```
declare n                fixed bin(15);
declare s                char(10);
declare c                char(10);

declare ascii_to_ebcdic  entry (fixed bin(15), char(*), char(*));

     n = 10;

     s = 'abcdefghij';

     call ascii_to_ebcdic(n, s, c);

     put skip list(c); /* c = `81`82`83`84`85`86`87`88`89`91 */
```

▶ `before(`*`s`*`,`*`c`*`)`
The `before` function returns a substring that precedes the first occurrence of a string in another string.

Both *`s`* and *`c`* must be varying-length character strings of any length. The result is a varying-length character string with a maximum length of 256.

The function returns the substring of *`s`* that precedes the first occurrence of *`c`* in *`s`*.

If either *`s`* or *`c`* is the null string, (`''`), the result is the null string. If *`c`* does not occur in *`s`*, the result is *`s`*.

The following example illustrates the `before` function.

```
declare   s          char(12) varying;
declare   c          char(4) varying;
declare   pstr       char(256) varying;

declare   before     entry (char(*) varying, char(*) varying)
                     returns (char(256) varying);

     s = 'Time: 8:30pm';
     c = ' ';
     pstr = before(s,c);      /* pstr = 'Time:' */
```

Note: The OpenVOS PL/I `before` function differs from the `before` function in full PL/I in two ways.

- The arguments are restricted to varying-length character strings.
- The result is restricted to 256 characters.

► `codeptr(`*e*`)`
The `codeptr` function returns a pointer to the first instruction of the specified entry point.

The reference *e* must resolve to an entry value. The result is a pointer to the first instruction of the entry point specified by *e*.

Note that the `codeptr` function does **not** accept label or format values.

The following example illustrates the `codeptr` function.

```
declare   p         pointer;

declare   codeptr   entry (entry) returns (pointer);

     p = codeptr(get_next_value);
          .
          .
          .
get_next_value: procedure;
          .
          .
          .
```

► `displayptr(`*e*`)`
The `displayptr` function returns a pointer to a stack frame.

The reference *e* must resolve to an entry value. The result is a pointer to the stack frame from which the procedure associated with *e* inherits automatic storage when *e* is activated.

Note that the `displayptr` function does **not** accept label or format values.

The following example illustrates the `displayptr` function.

```
declare   p            pointer;

declare   displayptr   entry (entry) returns (pointer);

     p = displayptr(b);
          .
          .
          .
b: procedure;
          .
          .
          .
```

► `ebcdic_to_ascii(`*n*`,`*s*`,`*c*`)`
The `ebcdic_to_ascii` subroutine converts an EBCDIC character string to ASCII. Note that `ebcdic_to_ascii` is a subroutine, not a function.

The value of *n* must be a 2-byte binary integer. The values of *s* and c must be a character string of any length.

The subroutine converts *n* characters of *s* from EBCDIC to ASCII and returns the value in *c*.

The following example illustrates a call to the `ebcdic_to_ascii` subroutine.

```
declare n                fixed bin(15);
declare s                char(10);
declare c                char(10);

declare unspec           builtin;

declare ebcdic_to_ascii  entry (fixed bin(15), char(*), char(*));

     n = 10;

     unspec(s) = '81828384858687888991'b4;

     call ebcdic_to_ascii(n,s,c);

     put skip list(c); /* c = abcdefghij */
```

- `hash(`*s*`,`*n*`)`

  The `hash` function returns a hash code.

  The value of *s* must be a varying-length character string of any length, and *n* must be a 2-byte binary integer. The result is a 2-byte binary-integer hash code for *s* in the range 0 to *n*`-1`, inclusive.

  All of the characters of *s* are involved in the computation. The function has been designed so that, for the same value of *n*, values of *s* that differ by only a few characters most often hash to different values.

  The `hash` function is repeatable; that is, for the given values of *s* and *n*, the function always returns the same result.

  The following example illustrates the `hash` function.

```
declare   s         char(12) varying;
declare   n         fixed bin(15);
declare   hash_code fixed bin(15);

declare   hash      entry (char(*) varying, fixed bin(15))
                    returns (fixed bin(15));
               .
               .
               .
     hash_code = hash(s,n);   /* 0 <= hash_code <= n-1 */
```

- `hex(`*v*`,`*n*`)`

  The `hex` function returns the hexadecimal value of an integer.

The value of *v* must be a 4-byte binary integer, and *n* must represent a 2-byte binary integer between 1 and 8, inclusive. The result is a varying-length character string with a maximum length of 8, containing the low-order *n* digits of the hexadecimal representation of *v*.

The result is unsigned and is padded with leading zeroes, if necessary, to make the length *n* characters.

The following example illustrates the `hex` function.

```
declare   v         fixed bin(31);
declare   n         fixed bin(15);
declare   result    char(8) varying;

declare   hex       entry (fixed bin(31), fixed bin(15))
                    returns (char(8) varying);

     v = 256;
     n = 4;
     result = hex(v,n);  /* result = '0100' */
```

▶ `hex64(v,n)`
The `hex64` function returns the hexadecimal value of an integer.

The value of *v* must be an 8-byte integer, and *n* must represent a 2-byte integer between 1 and 16, inclusive. The result is a varying-length character string with a maximum length of 16, containing the low-order *n* digits of the hexadecimal representation of *v*.

The result is unsigned and is padded with leading zeroes, if necessary, to make the length *n* characters.

The following example illustrates the `hex64` function.

```
declare   v         fixed bin(63);
declare   n         fixed bin(15);
declare   result    char(16) varying;

declare   hex64     entry (fixed bin(63), fixed bin(15))
                    returns (char(16) varying);

     v = 8589934592;
     n = 12;
     result = hexp64(v,n);  /* result = '000200000000' */
```

▶ `hexp(p,n)`
The `hexp` function returns the hexadecimal value of a pointer value.

The value of *p* must be a pointer, and *n* must represent a 2-byte binary integer between 1 and 8, inclusive. The result is a varying-length character string with a maximum length of 8, containing the low-order *n* digits of the hexadecimal representation of the pointer-value *p*.

Pointers are stored as 4-byte integers.

The result is unsigned and is padded with leading zeroes, if necessary, to make the length `n` characters.

The following example illustrates the `hexp` function.

```
declare   p         pointer;
declare   n         fixed bin(15);
declare   result    char(8) varying;

declare   hexp      entry (pointer, fixed bin(15))
                    returns (char(8) varying);

     p = null();
     n = 4;
     result = hexp(p,n);  /* result = '0001' */
```

▶ `occurs(s,c)`
The `occurs` function returns the number of times a specified string appears in another string.

Both `s` and `c` must be varying-length character strings of any length. The result is a 2-byte binary integer indicating the number of distinct occurrences of `c` that appear in `s`.

Distinct occurrences do not overlap. If either `s` or `c` is the null string, the result is 0.

The following example illustrates the `occurs` function.

```
declare   s         char(24) varying;
declare   c         char(7) varying;
declare   result    fixed bin(15);

declare   occurs    entry (char(*) varying, char(*) varying)
                    returns (fixed bin(15));

     s = 'abababab';
     c = 'aba';
     result = occurs(s,c);    /* result = 2 */
```

▶ `quote(s)`
The `quote` function encloses a string in single apostrophes and doubles any existing apostrophes.

The value of `s` must be a varying-length character string of any length. The result is a varying-length character string with a maximum length of 256 containing a quoted form of `s`. All apostrophes in `s` are doubled, and the entire string is enclosed in single apostrophes.

If the string that would result is longer than 256 characters, the result is the null string (`''`).

The following example illustrates the `quote` function.

```
declare   s         char(15) varying;
declare   quoted_s  char(256) varying;

declare   quote     entry (char(*) varying)
                    returns (char(256) varying);

     s = 'don''t';
     quoted_s = quote(s);

     put skip list(s, quoted_s);
```

The output is as follows: `don't 'don''t'`.

▶ `returnptr(`*`p`*`)`
The `returnptr` function returns a pointer to the instruction to which the calling procedure would return once it finishes executing.

▶ `reverse(`*`s`*`)`
The `reverse` function reverses the characters in a character string.

The value of *`s`* must be a varying-length character string with a length of no more than 256 characters. The result is a varying-length character string with a maximum length of 256 characters containing the characters of *`s`* in reverse order.

If *`s`* is the null string (`''`), the result is the null string.

The following example illustrates the `reverse` function.

```
declare   s         char(36) varying;
declare   rev_s     char(256) varying;

declare   reverse   entry (char(*) varying)
                    returns (char(256) varying);

     s = 'test.incl.pl1';
     rev_s = reverse(s);       /* rev_s = '1lp.lcni.tset' */
```

Note: The OpenVOS PL/I `reverse` function differs from the `reverse` function in full PL/I in two ways.

- The argument must be a varying-length character string.
- The result is restricted to 256 characters.

▶ `reverseb(`*`b`*`)`
The `reverseb` subroutine reverses the order of the bits in a bit string. Note that `reverseb` is a subroutine, not a function.

The value of *`b`* must be a bit string of any length, and it must be passed by reference to `reverseb`.

If *`b`* is the null bit string (`''b`), the subroutine has no effect.

The following example illustrates the `reverseb` subroutine.

```
declare   b         bit(6);

declare   reverseb  entry (bit(*));

          b = '011101'b;
          call reverseb(b);  /* b = '101110'b */
```

- `rindex(`*`s`*`,`*`c`*`)`

The `rindex` function returns the position of the last occurrence of a string within another string.

The values *s* and *c* must be varying-length character strings of any length. The result is a `fixed bin(15)` value representing the position of the last instance of *c* in *s*. If *c* does not occur in *s*, or if either *s* or *c* is null, the result is zero.

The following example illustrates the `rindex` function.

```
declare s          char(15) varying;
declare c          char(15) varying;
declare n          fixed bin(15);

declare rindex     entry (char(*) varying, char(*) varying)
                   returns (fixed bin(15));

          s = '/Let/us/begin';
          c = '/';

          n = rindex(s,c);

          put skip list(n); /* n = 8 */
```

- `rscaneq(`*`s`*`,`*`c`*`)`

The `rscaneq` function returns the length of the longest terminal substring in a string that does not contain any of the specified characters.

Both *s* and *c* must be varying-length character strings of any length. The result is a 2-byte binary integer indicating the length of the longest terminal substring of *s* that does not contain any of the characters in *c*.

If *s* is the null string, the result is 0. If none of the characters in *c* occur in *s*, or if *c* is the null string, the result is the length of *s*.

The following example illustrates the `rscaneq` function.

```
declare   s         char(25) varying;
declare   c         char(5) varying;
declare   count     fixed bin(15);

declare   rscaneq   entry (char(*) varying in, char(*) varying in)
                    returns (fixed bin(15));

     s = '1bbb23d5';
     c = 'abcde';
     count = rscaneq(s,c);  /* count = 1 */
```

Note that `rscaneq` and `rscaneqf` are identical except for the type of the first argument.

▶ `rscaneqf(`*`s`*`,`*`c`*`)`
The `rscaneqf` function returns the length of the longest terminal substring in a string that does not contain any of the specified characters.

The value of *`s`* must be a nonvarying character string of any length, and *`c`* must be a varying-length character string of any length. The result is a 2-byte binary integer indicating the length of the longest terminal substring of *`s`* that does not contain any of the characters found in *`c`*.

If *`s`* is the null string, the result is 0. If none of the characters in *`c`* occur in *`s`*, or if *`c`* is the null string, the result is the length of *`s`*.

The following example illustrates the `rscaneqf` function.

```
declare   s         char(18);
declare   c         char(9) varying;
declare   count     fixed bin(15);

declare   rscaneqf  entry (char(*) in, char(*) varying in)
                    returns (fixed bin(15));

     s = 'SMITH, RUSSEL T   ';
     c = ',;.:';
     count = rscaneqf(s,c);  /* count = 12 */
```

Note that `rscaneq` and `rscaneqf` are identical except for the type of the first argument.

▶ `rscanne(`*`s`*`,`*`c`*`)`
The `rscanne` function returns the length of the longest terminal substring consisting of characters found in the specified string.

Both *`s`* and *`c`* must be varying-length character strings of any length. The result is a 2-byte binary integer indicating the length of the longest terminal substring of *`s`* that consists entirely of characters found in *`c`*.

If *`s`* or *`c`* is the null string, the result is 0. If none of the characters in *`c`* occur in *`s`*, the result is 0.

The following example illustrates the `rscanne` function.

```
declare   s          char(64) varying;
declare   c          char(12) varying;
declare   count      fixed bin(15);

declare   rscanne    entry (char(*) varying in, char(*) varying in)
                     returns (fixed bin(15));

          s = 'Hello, world.';
          c = 'abcd,;:.';
          count = rscanne(s,c);    /* count = 2 */
```

Note that `rscanne` and `rscannef` are identical except for the type of the first argument.

▶ `rscannef(`*`s`*`,`*`c`*`)`
The `rscannef` function returns the length of the longest terminal substring consisting of characters found in the specified string.

The value of *`s`* must be a nonvarying character string of any length, and *`c`* must be a varying-length character string of any length. The result is a 2-byte binary integer indicating the length of the longest terminal substring of *`s`* that consists entirely of characters found in *`c`*.

If *`s`* or *`c`* is the null string, the result is 0. If none of the characters in *`c`* occur in *`s`*, the result is 0.

The following example illustrates the `rscannef` function.

```
declare   s          char(10);
declare   c          char(4) varying;
declare   count      fixed bin(15);

declare   rscannef   entry (char(*) in, char(*) varying in)
                     returns (fixed bin(15));

          s = 'abcd322132';
          c = '1234';
          count = rscannef(s,c);   /* count = 6 */
```

Note that `rscanne` and `rscannef` are identical except for the type of the first argument.

▶ `stackframeptr()`
The `stackframeptr` function returns a pointer to the stack frame of the procedure from which it is invoked. This function takes no arguments.

The following example illustrates the `stackframeptr` function.

```
declare   p               pointer;
declare   stackframeptr   entry returns (pointer);

          p = stackframeptr();
```

▶ `staticptr(e)`
The `staticptr` function returns a biased pointer to the storage of an entry point.

The reference *e* must resolve to an entry value. The function returns a biased pointer to the internal static storage of the entry point associated with *e*.

Note that this function does **not** accept label or format values.

The following example illustrates the `staticptr` function.

```
main: procedure;
     .
     .
     .
inner: procedure;

declare   p         pointer;
declare   staticptr entry (entry) returns (pointer);

     p = staticptr(main);
     .
     .
     .
end inner;
end main;
```

▶ `unhex(s,v,c)`
The `unhex` subroutine converts a character string from hexadecimal to binary. Note that `unhex` is a subroutine, not a function.

The value of *s* must be a varying-length character string of any length. The reference *v* must represent a 4-byte binary integer variable. The reference *c* must represent a 2-byte binary integer variable.

The subroutine converts *s* from hexadecimal to binary and returns the result in *v*. The string *s* must contain only characters from the following sets.

- the digits `0` through `9`
- the uppercase letters `A` through `F`
- the lowercase letters `a` through `f`
- an optional initial minus sign (`-`)
- an optional terminating `x`

If no error occurs, *c* is set to 0; otherwise, *c* is set to an operating-system status code.

The following example illustrates a call to the `unhex` subroutine.

```
declare   s       char(128) varying;
declare   v       fixed bin(31);
declare   c       fixed bin(15);

declare   unhex   entry (char(*) varying, fixed bin(31),
                  fixed bin(15));

     s = '-000Cx';
     call unhex(s,v,c);  /* v = -12 */
```

▶ `unquote(`*`s`*`)`
The `unquote` function removes leading and trailing apostrophes from a character string, and also changes double apostrophes to single apostrophes.

The reference *`s`* must be to a varying-length character string of any length. The result is a varying-length character string with a maximum length of 256 containing an unquoted form of *`s`*. A leading and trailing apostrophe character is removed from *`s`*, and all double apostrophe characters are changed to single apostrophes.

If *`s`* contains no apostrophes or unbalanced apostrophes, or if *`s`* does not end in an apostrophe, the function returns *`s`* unchanged.

The following example illustrates the `unquote` function.

```
declare   s              char(12) varying;
declare   c              char(6) varying;
declare   result(2)      char(256) varying;

declare   unquote   entry (char(*) varying)
                    returns (char(256) varying);

     s = '''don''''t''';
     result(1) = unquote(s);
     c = '''hello''  ';
     result(2) = unquote(c);

     put skip list(s, result(1), c, result(2));
```

The preceding example produces the following output (the ␣ character represents a space).

```
'don''t'␣␣don't␣␣␣␣␣'hello␣␣␣␣'hello␣␣
```

In the example, because the last character in `c` is not an apostrophe, the values of `result(2)` and `c` are equivalent.

Chapter 14: Input and Output

This chapter discusses the following topics related to PL/I I/O.

- "Overview"
- "File Control Blocks"
- "Stream I/O"
- "Record I/O"
- "Opening and Closing Files"

Note: Many of the examples in this chapter show space characters in output. For these examples, each ␣ character represents one space.

Overview

The operating system allows you to perform PL/I I/O in two ways: using the operating system subroutines or using PL/I language features. For information on the operating system subroutines approach, see the *OpenVOS PL/I Subroutines Manual* (R005). The discussion of I/O in this chapter is limited to the PL/I language features.

A PL/I program can take input from a terminal or other device, or from a file. Likewise, output can be written to a terminal, device, or file. In PL/I, all I/O is file I/O: the terminal and other system devices are treated as files.

PL/I recognizes two file types: stream files and record files.

A *stream file* is a sequence of characters organized into lines and, possibly, pages. The terminal is treated as a stream file.

A *record file* is a set of discrete records that can be accessed either sequentially or by keys. Records contain values stored internally by OpenVOS PL/I. A key is either a character-string valued index key or an integer representing the ordinal position of the record in the file.

File Control Blocks

PL/I I/O occurs through file control blocks. A *file control block* is a 440-byte storage area associated with a file constant. Each file constant declared in a program is associated with one file control block. For information on file data, see Chapter 4.

Before I/O can be performed on a file control block, that file control block must be opened. When you open a file control block, the OpenVOS file or device associated with the file control block is identified and an OpenVOS port is attached to that file or device. The name of the port is the same as the file ID of the file control block. Exceptions to this rule are the `sysin` and `sysprint` files; see "Terminal I/O through Predefined I/O Ports" later in this chapter. If, when you attach the port to the file, you specify the name of a file that does not exist, the operating system attempts to create the file if the `open` statement contains the `output` or `update` attribute. See "Operations on Records," later in this chapter, for more information on the record operations attributes: `input`, `output`, and `update`.

If a port has already been attached by the OpenVOS `attach_port` command, the current attachment of that port is used. The previous attachment overrides any path name specified in the `title` option of the `open` statement. The `open` statement is discussed in Chapter 12.

When you have completed all I/O operations on a device, or if you wish to change the attributes of the file control block, you can close the control block. Closing a file detaches the associated port. After a file control block is closed, it can be reopened with new file attributes or attached to a different device.

Opening and closing file control blocks is discussed later in this chapter.

Stream I/O

In stream I/O, sequences of characters are written to or read from a file. The length of these sequences can vary; the maximum length is 256 characters.

On input, you provide a list of target variables to receive values from the stream. On output, you provide a list of values to be written to the stream. These lists are called *I/O lists*.

Note that a PL/I stream file **need not** have the OpenVOS stream organization. OpenVOS sequential, fixed, or relative files can also be used as stream files for PL/I I/O. The OpenVOS file organizations are described in the *VOS Reference Manual* (R002). The term *stream file*, in this discussion, refers to a PL/I stream file unless specifically noted otherwise.

Every stream file has a line size, which you can specify when you open the file. The default is the line size of the device to which the file is attached; if the file is not attached to a device with a line size, the default line size is 80.

A stream file always has a current position. This position consists of a line and a column position within that line. When the file is opened, the current line is the first line in the file, and the column position is initialized to 1. Each character written to or read from the file increases the column position by 1. When the current column position exceeds the line size, a new line begins and the current column position is reset to 1.

The current position can also be altered by edit control formats, which are described in "Edit-Directed I/O," later in this chapter.

The next seven sections discuss the following topics.

- "The `put` and `get` Statements"
- "Print Files"
- "Terminal I/O through Predefined I/O Ports"
- "List-Directed I/O"

- "Edit-Directed I/O"
- "Data and Control Formats"
- "Stream I/O with the `read` and `write` Statements"

The `put` and `get` Statements

You use the `get` statement for stream input, and you use the `put` statement for stream output. These statements can transmit string values of up to 256 bytes or characters.

The `get` statement has the following syntax.

```
    ┌                                                  ┐
    │ [file(file_name)] ┌skip[ (number)]┐              │
get │                   └               ┘              │
    │            string(string_name)                   │
    └                                                  ┘
    ┌                                                 ┐
    │     list(input_item [,input_item]...)           │
    │ edit(input_item [,input_item]...)(format_list)  │;
    └                                                 ┘
```

The `put` statement has the following syntax.

```
    ┌                                                            ┐
    │ file(file_reference) ┌ skip [(skips)]   ┐ [page]           │
put │                      │ line(line_number)│                  │
    │                      └                  ┘                  │
    │              string(string_reference)                      │
    └                                                            ┘
    ┌                                                    ┐
    │      list(output_item [,output_item]...)           │
    │ edit(output_item [,output_item]...)(format_list)   │;
    └                                                    ┘
```

In the following example, the `put` and `get` statements are used to copy a value from one file to another.

```
declare   next_value     char(79) varying;
declare   (f,g)          file;
          .
          .
          .
     get file(f) skip list(next_value);
     put file(g) list(next_value);
```

The `get` statement in the preceding example skips to the next line of the file associated with `f` and reads a value from that line into the program variable `next_value`. The `put` statement writes the value of `next_value` at the current position of the file associated with `g`.

Although a stream file consists of sequences of characters, a value from a stream file can be read into an arithmetic, pictured, or bit-string variable, provided the file value composes a legitimate value for the variable. Similarly, the value of an arithmetic, pictured, or bit-string variable can be written to a stream file. The following example transfers an integer value from one file to another.

```
declare   number_field   fixed bin(15);
declare   (f,g)          file;
          .
          .
          .
     get file(f) list(number_field);
     put file(g) list(number_field);
```

Control characters, such as carriage return, new line, form feed, vertical tab, horizontal tab, bell, and null must **not** be written as data characters in a `put` statement and **cannot** be read by a `get` statement. Attempting to transmit such characters produces unpredictable results.

The next two sections discuss the following topics.

- "The `string` Option"
- "The Iterative-Do in I/O Lists"

The `string` Option

In addition to using the `put` and `get` statements for stream file operations, you can use these statements to transmit data to and from character-string variables. The `string` option of the I/O statements is used for this purpose, as shown in the following example.

```
declare   data_string        char(256) varying;
declare   record_value       char(25);
declare   1  struct          ,
            2  numb          fixed bin(15),
            2  data          char(25);
               .
               .
               .
     put string(data_string) list(10, record_value);
     get string(data_string) list(struct);
```

The `put` statement in the preceding example writes data to the variable `data_string` as if that variable were a line in a stream output file. The `get` statement reads data from the same variable as if it were a line in a stream input file.

You cannot use the `column`, `line`, `page`, `skip`, or `tab` control formats in the format list when you use the `string` option. See "Edit-Directed Input," later in this chapter, for more information.

The Iterative-Do in I/O Lists

You can use an iterative-do within the I/O list of a `put` or `get` statement. This is most often done to perform I/O to or from an array.

The iterative-do is appended to an item in the I/O list. Both the iterative-do and the item to which it refers must be enclosed in parentheses, as shown in the following format.

```
(item iterative-do)
```

The following example uses an iterative-do in both a `get` statement and a `put` statement to transmit values to and from a cross-section of a two-dimensional array.

```
declare   line(5,5) char(1);
declare   code      fixed bin(15);

     get list((line(k,2) do k = 1 to 5));
                .
                .
                .
     put skip list(code,(line(k,2) do k = 1 to 5));
```

The `get` statement reads five one-character values and transmits them, in order, to `line(1,2)`, `line(2,2)`, `line(3,2)`, `line(4,2)`, and `line(5,2)`. The `put` statement first outputs an integer value and then transmits the five elements of the cross-section of `line` in the same order in which they were input.

Print Files

A file opened with the `print` attribute is a special kind of stream output file known as a *print file*. Output to a print file is formatted so that the file can be spooled to a printer.

A print file always has a current line number, a current page number, and a page size. When you open the file, you can specify the page size; the line number and page number are initialized to 1. The default page size is 60 lines.

A print file has tab stops set every five columns beginning with column 1. Each value written to the file in list-directed I/O begins on a tab stop. In edit-directed I/O, you can use the `tab` format item to position to a specific tab stop. See the discussions of list-directed and edit-directed I/O later in this chapter.

Each time a line is written to a print file, the current line number is incremented by one. This occurs regardless of what caused the new line to be written. Each time a new line begins, the current column is reset to 1.

If the line number is set to a value one greater than the page size, an `endpage` condition is signaled. By default, when this condition occurs, the following actions take place.

- The remainder of the current page is filled with blank lines.
- A form-feed character is inserted.
- A new page is started.
- The current page number is incremented by 1.
- The line number is set to 1.

You can change the default actions by coding an on-unit for the `endpage` condition of the file. See Chapter 15 for information regarding I/O conditions and on-units.

You can determine the current line number and current page number using the `lineno` and `pageno` functions described in Chapter 13. You can alter the value of the page number by using the `pageno` pseudovariable, which is also described in Chapter 13.

Terminal I/O through Predefined I/O Ports

If you do not specify a file reference in a `put` statement, the `sysprint` file control block is used; `sysprint` is always associated with the `default_output` port. If you do not specify a file reference in a `get` statement, the `sysin` file control block is used; `sysin` is always associated with the `default_input` port. You need **not** declare the `sysprint` and `sysin` files.

The following predefined I/O ports are, by default, attached to the user's terminal.

- `default_input`
- `terminal_output`
- `command_input`
- `default_output`

You can access the `default_input` or `command_input` port by specifying `file(default_input)` or `file(command_input)`, respectively, in a `get` statement. Likewise, in a `put` statement, you can specify `file(default_output)` or `file(terminal_output)` to access the `default_output` or `terminal_output` port, respectively. You **must** declare these files, if you specify them in a `get` or `put` statement.

Unless the port has been directed to a file, stream I/O performed through one of these predefined terminal I/O ports is unlike ordinary stream I/O in the following respects.

- A single current column position is used for all predefined I/O ports attached to a specific terminal.
- Each `put` statement transmits data to the file without waiting for the line to be filled.
- The page size is ignored and the `endpage` condition does not occur.

See Chapter 15 for information about the `endpage` condition. See the previous section, "Print Files," for a discussion of directing output to a file.

Because a single current column position is used for the terminal, the current column position always corresponds to the cursor position.

The following example shows how the `put` and `get` statements perform terminal I/O.

```
put skip list('Enter your name:'); /* Default: file(sysprint) */
get list(user_name);               /* Default: file(sysin)    */
```

The preceding example skips to the next line on the terminal and then outputs the string `Enter your name:`. The next value typed on that line or a succeeding line is read into the variable `user_name`.

Stream input or output without a controlling format list, as shown in the preceding example, is called *list-directed* I/O. Stream input or output under control of a format list is called *edit-directed* I/O. The following sections describe these two different forms of stream I/O.

List-Directed I/O

In list-directed I/O, each line in a stream file is divided into *fields*. On input, the first field begins with the first nonspace, noncomma character on the line (except that a series of spaces followed by a comma compose an empty field). Fields are separated by one or more spaces or commas. On output, fields within a line are separated by single space characters.

A space character is considered to be appended to the end of each input line unless you specify `-noblank` in the `title` option of the `open` statement, as described in Chapter 12.

Input values are converted from character strings to the data type of the target variable. Output values are converted to character strings. See Chapter 5 for a discussion of data-type conversions.

The next two sections discuss the following topics.

- "List-Directed Input"
- "List-Directed Output"

List-Directed Input

On input, one or more space characters or a comma terminates a field. If a field is to be assigned to an arithmetic variable, any excess space characters before or after the value are ignored. Such a field must contain a valid arithmetic constant, as could appear on the right side of an assignment statement. Chapter 4 explains how the data type of a constant is determined.

If a field is to be assigned to a character-string variable, it can contain either a character-string constant enclosed in apostrophes, or any sequence of ASCII characters. A character-string constant begins with the character following the opening apostrophe and ends with the character preceding the closing apostrophe. A sequence of characters begins with the first nonspace character in the field and ends with the character immediately preceding the next comma or space character.

To include a space character within a character-string input field, you must enclose the field in apostrophes. To include an apostrophe within an input field that is enclosed in apostrophes, type two apostrophes instead of one.

If the field is enclosed in apostrophes, the value of that character-string constant is assigned to the list variable; if the field is not enclosed in apostrophes, all characters in the field are assigned to the list variable without removing apostrophes.

The following example illustrates list-directed input.

```
declare    y    fixed bin(15);
declare    z    char(5);
           .
           .
           .
      get list(y,z);
```

The following table shows the values assigned to `y` and `z` for various input lines.

Input Line	y	z
`5,abc`	`5`	`'abc  '`
`-45 'abc'`	`-45`	`'abc  '`
`-45 abc-4`	`-45`	`'abc-4'`
`-45 , abc`	`-45`	`'abc  '`

Any excess space characters on either end of the input line are ignored.

An empty field is terminated by a comma and contains only space characters or no characters. The following input line contains three fields, all of which are empty (this assumes the previous input line ended with a space character).

```
␣␣,␣,,
```

In the preceding example, an empty field causes no assignment to the corresponding list variable.

Input fields can break across two or more lines; a line boundary does not terminate a field. However, OpenVOS PL/I inserts a space character at the end of each line read from a terminal or file. This prevents a field from continuing onto another line. You can override this action on input by specifying `-noblank` in the `title` option of the `open` statement. See Chapter 12 for further information.

If more than one space character follows a field, the spaces are scanned when the field is read and the current position is set to the next nonspace character or to the end of the line, whichever comes first. The following example illustrates.

```
get file(f) list(y,z);
get file(f) list(c);
```

Assume file `f` contains the following data.

```
␣␣52␣1.07E+05␣abc
x␣7
```

The values in file `f` are read as shown in the following table.

Variable	Value
`y`	`52`
`z`	`1.07E+05`
`c`	`'abcx'`

After the second `get` statement, the position is set to column 3. If the second line in file `f` had ended after the second character (the space character), the position would be at the end of the line, and any subsequent `get` statement operating on the file would read a new line or, if no lines remained, detect the end of the file.

List-Directed Output

Values transmitted in list-directed output are first converted to character strings using the normal conversion rules described in Chapter 5.

If the original value is a bit string, the resultant character-string value is enclosed in apostrophes, and a `b` character is appended after the closing apostrophe.

If the original value was a character-string or pictured value, and the output file was opened without the `print` attribute, each apostrophe within the string is replaced by two apostrophes and the entire string value is enclosed in apostrophes.

> **Note:** The `sysprint` file is implicitly opened with the `print` attribute.

If the current position is the beginning of a line, no space characters are written before the field; otherwise, each output string is preceded by at least one space character. If the file is opened with the `print` attribute, the output value is preceded by sufficient space characters to ensure that the output begins on the next tab stop after the current position; in a nonprint file, a single space character precedes the field.

If an output field does not fit entirely on the current line, a new line is begun; the value is written on that new line and, if necessary, on subsequent lines.

The following example illustrates list-directed output.

```
declare    y     fixed bin(15);
declare    z     char(5);

     y = -75;
     z = 'dog  ';

     put list(y,z);
```

The preceding example produces the following output string.

```
␣␣␣␣␣␣-75␣dog␣␣
```

As mentioned earlier in this chapter, each ␣ character represents a space character. The output is produced by first converting `y` to the character-string value `'      -75'`. Note that on output, a single space character separates the value of `y` from the value of `z`.

The following is a more complex example of list-directed stream output.

```
open file(f) stream output linesize(20);

put file(f) list(52,1.07E+05);
put file(f) list('abcx');
```

The following example shows the contents of file `f`.

```
␣␣␣52␣␣1.07E+05␣␣␣␣␣
abcx
```

The value `52` converts to `'   52'`, and the value `1.07E+05` converts to `' 1.07E+05'`; these values are separated by a single space on output. The string `'abcx'` preceded by a separating space character would not fit on the current line; instead, the string is written on the next line. If file `f` had been opened with the `print` attribute, each value would be at a tab stop, and the last value would be written without the enclosing apostrophes.

Edit-Directed I/O

In edit-directed I/O, two lists appear in the `get` or `put` statement. The first list, called the *I/O list*, is the same as the list used in list-directed I/O. The second list, called the *format list*, contains data and control format items.

Each element in the I/O list of the `put` or `get` statement has a corresponding data format item in the format list. The *data format item* determines the size and form of the stream field. *Control format items* determine the position of fields in the file. Fields in edit-directed I/O need not be separated by spaces or commas; you can specify the position and size of fields in the stream.

All data and control format items are described in "Data and Control Formats" later in this chapter. In "Data and Control Formats," Tables 14-1 and 14-2 list the OpenVOS PL/I data and control format items, respectively.

The next three sections discuss the following topics.

- "Format Lists"
- "Edit-Directed Input"
- "Edit-Directed Output"

Format Lists

A *format list* is a series of format items separated by commas and enclosed in parentheses. You can precede each format item by an optional repetition factor, *k*, where `0 <= k <= 255`. Note that you must separate a repetition factor from a format item with a space.

The following example shows a format list. In this example, the last item will be repeated twice, as a repetition factor of 2 is specified.

```
(a(2), x(3), 2 f(10,4))
```

You can include a format list within a `put` or `get` statement, or you can set it up separately in a `format` statement. Control is transferred to the list in a `format` statement by an `r` format item in the format list of a `put` or `get` statement.

The following example shows two format lists: one in a `format` statement and one in a `put` statement.

```
rec_form:      format(f(5),a(32),f(3),a(16));

          put skip edit(count, enum, ename, dept, job)(f(3),r(rec_form));
```

The various format items are described in "Data and Control Formats" later in this chapter. See Chapter 12 for descriptions of the `format`, `put`, and `get` statements.

Each time control passes to a format list, all control format items between the last used format item and the next data format item are evaluated. That next data format item is then used to control the conversion of the data being transmitted to or from the stream file.

If control reaches the end of a format list in a `get` or `put` statement and more values remain to be transmitted in the I/O list, control transfers to the beginning of the format list. If control reaches the end of a format list in a `format` statement, control returns to the `r` format item that originally transferred control to the `format` statement.

You can use a parenthesized format list as a format item within another format list. The nested list can be preceded by a repetition count.

The format lists in the following `format` statements are equivalent.

```
forma:    format(f(10,4),2(a(5),e(14,3)),skip);

formb:    format(f(10,4),a(5),e(14,3),a(5),e(14,3),skip);
```

Edit-Directed Input

In edit-directed input, each value is converted from a character string to the type described by a data format. The result might have to be converted again before being assigned to the target variable.

If you use edit-directed input, each input line must be exactly described by the controlling format. If the current input line contains fewer characters than are required to satisfy the format, additional lines are read until the format is satisfied.

If the length of the input line is not known, you can use the `a` format with no width specification to read the line. See further information in "Data and Control Formats" later in this chapter.

The following example reads a value into a character-string variable.

```
declare   str  char(100);
          .
          .
          .
      get edit(str)(a(80));
```

If the current line contains only 60 characters, those 60 characters are read along with 20 characters from the next line. These 80 characters are converted to a character string with a length of 100 when assigned to `str`. See Chapter 5 for the conversion rules involved.

If you use the `string` option of the `get` statement, you cannot use the `column` or `skip` control format.

Edit-Directed Output

Each value transmitted in edit-directed output is converted to a character string under control of a data format. The result is written to the output stream.

The following example illustrates edit-directed stream output.

```
declare   y    fixed bin(15);
declare   z    char(4);

      y = -45;
      z = 'dogs';

      put edit(y,z)(f(7,2),a(6));
```

The preceding example produces the following output string.

```
␣-45.00dogs␣␣.
```

The value of *y* is transmitted under control of the `f(7,2)` data format. The effect of this is that the value of *y* is first converted to a fixed-point decimal value with precision `(7,2)`. This value is then converted to a character string on output.

The value of *z* is transmitted under control of the `a(6)` data format. The effect is to convert the value of *z* to a character string of length 6 before output.

See Chapter 5 for an explanation of the conversions involved. The data formats are discussed in “Data and Control Formats” later in this chapter.

In addition to data formats, a format list can include control formats. You can use control formats to change the current position in the file before each field is written.

In the following example, control formats are used to leave spaces between two fields and to force the start of a new line.

```
put edit(y,z)(f(7,2),x(3),a(6),skip);
```

The `x(3)` control format causes three spaces to be skipped between the output fields. The `skip` format starts a new line. The following example illustrates the `skip` format in edit-directed stream output.

```
declare    y     fixed bin(15);
declare    z     char(4);
declare    y1    fixed bin(15);
declare    z1    char(4);

      y = -45;
      z = 'dogs';
      y1 = 38;
      z1 = 'cats';

      put edit(y,z,y1,z1)(f(7,2),x(3),a(6),skip);
```

The preceding example produces the following output.

```
␣-45.00␣␣␣dogs␣␣
␣␣38.00␣␣␣cats␣␣
```

Since there is no more data to write after `z1`, the `skip` format is not implemented in the second line of data. Any subsequent output will begin at the end of this line.

If you use the `string` option of the `put` statement, you cannot use the `column`, `line`, `page`, `skip`, or `tab` control format.

The following section describes each data and control format.

Data and Control Formats

Table 14-1 lists the PL/I data formats.

Table 14-1. PL/I Data Formats

Format	Description
`a`	Converts a value to a character string
`b`	Converts a value to a bit string
`e`	Converts a value to a floating-point decimal number
`f`	Converts a value to a fixed-point decimal number
`p`	Converts a value to a pictured value

Table 14-2 lists the PL/I control formats.

Table 14-2. PL/I Control Formats

Format	Description
`column`	Positions to a specific column
`line`	Positions to a specific line (print files only)
`page`	Positions to the top of a new page (print files only)
`r`	Transfers control to a format list
`skip`	Skips lines
`tab`	Positions to a specific tab stop (print files only)
`x`	Writes spaces on output; skips characters on input

The following sections describe the data and control formats in alphabetical order.

- "The `a` Data Format"
- "The `b` Data Format"
- "The `column` Control Format"
- "The `e` Data Format"
- "The `f` Data Format"
- "The `line` Control Format"
- "The `p` Data Format"
- "The `page` Control Format"
- "The `r` Control Format"
- "The `skip` Control Format"
- "The `tab` Control Format"
- "The `x` Control Format"

The a Data Format

The `a` data format has the following syntax.

```
a [ (width) ]
```

The *`width`* value must be an integer constant.

Input

If you use the `a` format on input, a character-string value containing the next *`width`* characters is read from the input stream file.

You can use the `a` format without *`width`* to read variable-length input lines. If you omit *`width`*, the remainder of the current line is read from the stream file; the input value begins with the current column position and ends with the end of the line. The current column position is then reset so that the next input operation reads a new line.

Output

If you use the `a` format on output, an arithmetic or string value specified in the I/O list of a `put` statement is converted to a character string. See Chapter 5 for a discussion of data-type conversions.

If you omit *`width`*, the length of the character-string value is used by default.

The converted string is left-justified within a field of length *`width`*. The rest of the field is filled with space characters.

The `b` Data Format

The `b` data format has the following syntax.

```
⎧ b               ⎫
⎪ b1              ⎪
⎨ b2 [ (width) ]  ⎬
⎪ b3              ⎪
⎩ b4              ⎭
```

The width, if specified, must be an integer constant.

Input

If you use the `b` format on input, you must specify a value for *`width`*. The next *`width`* characters from the input stream are converted to a bit string. The conversion is performed as if the input characters appeared as a bit-string constant followed by `b`, `b1`, `b2`, `b3`, or `b4`, depending on which form of the `b` format you specify.

If the field contains a character that is invalid for the specified bit-string format, the `error` condition is signaled. Table 4-1 lists the characters allowed for each bit-string format and describes how each is interpreted.

Output

If you use the `b` format on output, an arithmetic or string value specified in the I/O list of a `put` statement is converted to a bit string using the normal conversion rules discussed in Chapter 5. The resultant bit string is then padded on the **left** with sufficient zero bits to make it a multiple of *`k`* bits in length, where *`k`* is the integer `1`, `2`, `3`, or `4` that immediately followed the `b` in the format item; if you do not specify an integer, `1` is assumed.

Each group of *`k`* characters in the padded bit string is then converted to a single output character. The result is a character string of length *`n`*, where *`n`* is the length of the padded bit string divided by *`k`*.

The resultant character string is right-justified in a field of *`width`* characters; the remainder of the field is filled with space characters. If specified, *`width`* must be sufficient to hold all characters in the string. If you omit *`width`*, the length of the resultant character string is the default.

The following table shows the results of the `b` format used on various values.

Value	**Format**	**Bit-String Value**	**Result**
`'00'`	`b`	`'00'b`	`00`
`''`	`b(4)`	`''b`	`␣␣␣␣`
`13`	`b2(4)`	`'00001101'b`	`0031`
`'110101'b`	`b3(4)`	`'110101'b`	`65`
`'10011101'`	`b4(2)`	`'10011101'b`	`9d`
`27`	`b2(4)`	`'00011011'b`	`0123`
`1`	`b(4)`	`'0001'b`	`0001`
`1`	`b(3)`	`'0001'b`	Invalid: field too small

The `column` Control Format

The `column` control format has the following syntax.

$$\left\{\begin{array}{c}\texttt{column}\\ \texttt{col}\end{array}\right\}\texttt{(}n\texttt{)}$$

A `column` format item puts spaces into an output stream or skips characters in an input stream so that the next character is read from or written to column *n* of a line.

The value of *n* must be a positive integer constant.

You cannot use the `column` format when using the `string` option of the `put` or `get` statement.

Input

If the current position is such that the next character to be read is located in column *n* of the current input line, no characters are skipped.

If the current column position is less than column *n* and *n* does not exceed the size of the current input line, all characters between the current position and the *n* column position of the current line are skipped. The next input begins from column *n*.

If *n* exceeds the size of the current line, or if the current column is greater than *n*, a new line is read. If *n* exceeds the size of the current line, input begins from column 1 of the new line; otherwise, `n-1` characters are skipped and output begins from column *n* of the new line.

The following table shows the effect of the `column` format in a file with a line size of 79.

Format	**Current Column**	**Current Line Number**	**New Column**	**New Line Number**
`col(79)`	`78`	`1`	`79`	`1`
`col(78)`	`78`	`1`	`78`	`1`
`col(77)`	`78`	`1`	`77`	`2`
`col(80)`	`78`	`1`	`1`	`2`

Output

If exactly `n-1` characters have been written to the current line of the output file, the `column` format has no effect. The next output begins in column `n` of the current line.

If less than `n-1` characters have been written to the current line of the output file and `n` is less than or equal to the line size, sufficient space characters are output onto the current line to cause the next output to begin in column `n`.

If more than `n` characters have been written to the current output line, a new line is begun. If the line size of the file is `n` or greater, `n-1` space characters are written onto the new line, causing the next output to begin in column `n`. If the line size is less than `n`, no space characters are written; the next output begins in column 1.

The `e` Data Format

The `e` data format has the following syntax.

```
e(width [,scaling_factor])
```

Both *`width`* and, if specified, *`scaling_factor`* must be integer constants.

Input

If you use the `e` format on input, a field of *`width`* characters is read from the input line and converted to a floating-point decimal value. The field of characters read must contain either all space characters or an optionally signed fixed-point or floating-point constant; the constant can be preceded or followed by a number of space characters.

If the field contains all space characters, the precision of the converted floating-point value is `15` or *`width`*, whichever is less. Otherwise, the precision of the converted floating-point value is the precision of the constant contained within the field.

If the field contains a decimal point, an exponent, or both, the value of *`scaling_factor`* is ignored. Otherwise, the last *`scaling_factor`* digits in the field are treated as fractional digits.

The following table illustrates the `e` format, used on input.

Value	Format	Result
`␣␣␣␣␣␣`	`e(6,1)`	`0.00000E+00`
`␣-1␣␣␣`	`e(6,1)`	`-1.00000E-01`
`-25e10`	`e(6,1)`	`-2.50000E+11`
`␣7.413`	`e(6,2)`	`7.41300E+00`
`␣␣150␣`	`e(6,2)`	`1.50000E+00`

Output

If you use the `e` format on output, an arithmetic or string value from the I/O list of a `put` statement is converted to a floating-point decimal value using the normal rules for data-type conversion, as described in Chapter 5. If you specify a scaling factor, the precision, *`p`*, of the result is *`scaling_factor`* `+ 1`. The value is right-justified within the output field; the rest of the field is filled with space characters.

The resultant field contains two exponent digits. The exponent is preceded by its sign, which is preceded by the letter `E`. The *`p-1`* least significant digits precede the letter `E`. These in turn are preceded by a decimal point, which is preceded by the most significant digit of the result. If the value is negative, it is preceded by a minus sign.

If the entire value (including its sign, its decimal point, and the character `E`) cannot fit into a field of *`width`* characters, the `error` condition is signaled.

The following example and table illustrate the `e` format, used on output.

```
put edit(z)(e(10,3));
```

Value	**Format**	**Result**
`0`	`e(10,3)`	`␣0.000E+00`
`-15`	`e(10,3)`	`-1.500E+01`
`12345678`	`e(10,3)`	`␣1.234E+07`
`7.3e-10`	`e(10,3)`	`␣7.300E-10`
`0` (precision 7)	`e(14)`	`␣␣0.000000E+00`
`-25` (precision 7)	`e(14)`	`␣-2.500000E+01`

The `f` Data Format

The `f` data format has the following syntax.

```
f(width [,scaling_factor])
```

The value *`width`* must be an integer constant specifying the width of the field, and *`scaling_factor`*, if specified, must be an integer constant that specifies the number of digits to the right of the decimal point in the field.

Input

If you use the `f` format on input, *`width`* characters are read from the input line and converted to a fixed-point decimal value. The field must contain one of the following:

- only space characters
- an optionally signed fixed-point constant, possibly with leading and trailing space characters

If the field contains a fixed-point constant, the precision of the converted value is the precision of the constant. If the field contains a decimal point, the number of digits following that decimal point is the scaling factor of the converted value. If the field does not contain a decimal point, the last *`scaling_factor`* digits are considered to be fractional digits.

If the field contains only space characters, the precision of the converted value is 18 or *`width`*, whichever is less. If you specify a scaling factor, the converted value has *`scaling_factor`* digits to the right of the decimal point; otherwise, all digits appear to the left of the decimal point.

The following example and table illustrate the `f` format, used on input.

```
get edit(a)(f(5,1));
```

Value	Format	Result
`0`	`f(5,1)`	`␣␣␣`
`-70.0`	`f(3,1)`	`-700␣`
`0.7`	`f(1,1)`	`␣␣␣␣7`
`0.0`	`f(1,1)`	`␣␣␣␣0`
`25`	`f(2,0)`	`25.␣␣`
Invalid	N/A	`5E+01`

Output

If you use the `f` format on output, an arithmetic or string value is converted to a fixed-point decimal value. The decimal value is then rounded and formatted as a character string of length *`width`* containing a value with *`scaling_factor`* digits to the right of the decimal point.

If *`scaling_factor`* is 0, or if *`scaling_factor`* is omitted, the source value is converted to a fixed-point decimal integer value. The resulting value is right-justified in the field; leading zeroes are suppressed and the field is filled with space characters to a length of *`width`*. If the value is negative, the most significant digit is preceded by a minus sign. If the entire value (including the sign, if the value is negative) cannot fit in *`width`* characters, the `error` condition is signaled.

The following example and table illustrate the `f` format, used on output.

```
put edit(b)(f(4));
```

Value	Result
`0`	`0`
`25`	`25`
`-8`	`-8`
`13.5`	`14`
`17.08`	`17`
`1000`	`1000`
`-1000`	Invalid

If you specify a nonzero value for *`scaling_factor`*, the source value is initially converted to a fixed-point decimal value with *`scaling_factor`* `+ 1` fractional digits. The last fractional digit is then increased by 5 and deleted, which has the effect of rounding the value off to *`scaling_factor`* decimal places. The resulting value is right-justified in a field of *`width`* characters. Leading integral zero digits are suppressed (fractional values and zero have one integral zero digit). If the value is negative, the leading digit is preceded by a minus sign. The remainder of the field is filled with space characters. The `error` condition is signaled if the value (including its decimal point) and its sign, if negative, cannot fit in *`width`* characters.

The following example and table illustrate the `f` format with a scaling factor, used on output.

```
put edit(c)(f(5,2));
```

Value	Result
0	0.00
-1	-1.00
.005	0.01
.0005	0.00
10	10.00
-10	Invalid

The `line` Control Format

The `line` control format has the following syntax.

```
line(n)
```

You can only use the `line` format to control output to a stream file that has been opened with the `print` attribute. Such files must have a page size. The `line` format positions the file to line number *n*. Lines are numbered relative to the top of the current page.

The value of *n* must be a positive integer constant.

You cannot use the `line` format when using the `string` option of the `put` statement.

If the current line number is less than *n* and *n* does not exceed the page size, sufficient lines are written to the output file so that the current line is line number *n*.

If the current line number is greater than *n*, or if *n* exceeds the page size, the remainder of the current page is filled with blank lines and a new page is begun. The `endpage` condition is signaled unless the current line number has already exceeded the page size.

If the current line is line number *n*, and the current column is not 1, a new page is begun. The `endpage` condition is signaled unless the current line number has already exceeded the page size.

If the current line is line number *n* and the current column is 1, the `line` format has no effect.

The `p` Data Format

The `p` data format has the following syntax.

```
p 'picture'
```

The value *picture* must be a valid picture description, as described in Chapter 4. The field width, *width*, is the number of characters in *picture*, excluding any `v` characters.

Input

If you use the `p` format on input, the next *width* characters are read from the input stream and assigned as a pictured value to the corresponding variable in the `get` statement.

Output

If you use the `p` format on output, an arithmetic or string value from the I/O list of a `put` statement is converted to a fixed-point decimal value described by *`picture`*. The decimal value is then edited into an output field of *`width`* characters as if the value were being assigned to a pictured variable. See the discussion of pictured conversions in Chapter 5.

The `page` Control Format

The `page` control format has the following syntax.

```
page
```

You can only use the `page` format to control output to a stream file that has been opened with the `print` attribute. You cannot use the `page` format when you use the `string` option of the `put` statement.

The `page` format changes the current position to the top of a new page. This change increments the page number by one and resets the line number to 1.

The `r` Control Format

The `r` control format has the following syntax.

```
r(format_name)
```

The `r` format transfers control to the format list of the `format` statement whose name appears in *`format_name`*. When the format list in that `format` statement is exhausted, control returns to the format item following the `r` format.

The following example illustrates the `r` format.

```
form_a:  format(a,x(3));
           .
           .
           .
      put edit(p,q)(r(form_a),e(14,3));
```

In the preceding example, the value of `p` is transmitted under control of the `a` format item contained in the `format` statement. To transmit the value of `q`, the `x(3)` item from the `format` statement is first evaluated. Then control returns to the `put` statement. The value of `q` is transmitted under control of the `e` format in the `put` statement.

For a description of the `format` statement, see Chapter 12.

The `skip` Control Format

The `skip` control format has the following syntax.

```
skip [(n)]
```

The value of *`n`* must be a positive integer constant. If you omit *`n`*, the value defaults to 1.

You cannot use the `skip` format when using the `string` option of the `put` or `get` statement.

Input

If you apply the `skip` format to an input stream, the rest of the current input line is skipped, along with *n*`-1` subsequent lines; that is, *n* line boundaries are skipped.

Output

If you apply the `skip` format to an output stream, the current line and *n*`-1` empty lines are output; that is, *n* line boundaries are written.

If the output stream file has been opened with the `print` attribute and the total number of lines written as the result of the `skip` format would exceed the page size, the current line is written, followed by sufficient empty lines to fill the page. The `endpage` condition is then signaled.

The `tab` Control Format

The `tab` control format has the following syntax.

```
tab [ (n) ]
```

The value of *n* must be a positive integer constant. If you omit *n*, the value defaults to 1.

The `tab` format can only be used to control output to a stream file that has been opened with the `print` attribute. If you use `tab` formats, the line size of the file must be equal to or greater than the first tab stop. Tab stops are set every five columns beginning with column 1.

You cannot use the `tab` format when using the `string` option of the `put` statement.

The `tab` format writes sufficient space characters to the file to change the current position to the *n*th tab stop relative to the current column of the line.

If the current column is a tab stop, the `tab(1)` format causes sufficient spaces to be written so that the next output item begins at the next tab stop.

If *n* tab stops do not remain on the current line, the current line is written and a new line is begun. The next field is written beginning in column 1 of the new line.

The `x` Control Format

The `x` control format has the following syntax.

```
x(n)
```

The value of *n* must be a positive integer constant. You must specify *n*.

Input

If you use the `x` format on input, *n* characters are skipped in the current input stream. If less than *n* characters remain in the current line, the rest of the line is skipped and additional characters are skipped on subsequent lines until a total of *n* characters are skipped.

Output

If you use the `x` format on output, *n* space characters are written to the current output stream. If less than *n* characters remain on the current line, the line is filled with space characters and additional spaces are written on subsequent lines until a total of *n* space characters are written.

Stream I/O with the `read` and `write` Statements

Unlike standard PL/I, OpenVOS PL/I allows you to use the `read` and `write` statements on a stream file. This provides an easy method for reading and writing variable-length input lines.

The `read` statement for a stream file has the following syntax.

```
read file(file_reference) into(variable);
```

The file control block associated with *`file_reference`* must have been opened for stream input, and *`variable`* must represent a scalar varying-length character-string variable. The `read` statement reads the next complete line from the stream and assigns it to that varying-length string.

The `write` statement for a stream file has the following syntax.

```
write file(file_control_block) from(variable);
```

Again, the file must have been opened for stream output, and *`variable`* must represent a scalar varying-length character-string variable. When the `write` statement is executed, the current value of *`variable`* is written as a new line in the output stream.

The `read` and `write` statements are documented in Chapter 12.

Record I/O

In record I/O, data in a file is accessed one record at a time. The record I/O statements are `read`, `write`, `rewrite`, and `delete`.

Record I/O statements transmit the **storage** of a variable to or from a record in a file. The compiler does not perform conversions and does not ensure that data is of the proper type for storage in a particular variable.

Record I/O is faster than stream I/O and requires less space.

The next three sections discuss the following topics.

- "Accessing Records"
- "Operations on Records"
- "Record File Positioning"

Accessing Records

You can access records in three ways.

- sequentially, using the `sequential` attribute
- by an index key, using the `sequential keyed` attributes
- directly by record number, using the `direct` [`keyed`] attributes

Note: *Sequential order* refers to either the order in which records appear in the file or the index order.

A record file must be opened with one of the attribute sets shown in the preceding list. The attribute set determines the manner in which records are accessed. The default is `sequential`. (The `direct` attribute implies the `keyed` attribute. See "Implied File Attributes," later in this chapter, for more information about implied attributes.)

> **Note:** Do not confuse the PL/I file attributes with OpenVOS file organizations. A file opened with the PL/I `sequential` attribute need not be an OpenVOS sequential file; see Table 14-3 for a list of OpenVOS file organizations in PL/I record files. For more information on OpenVOS file organizations, see the *Introduction to VOS* (R001).

Table 14-3. OpenVOS File Organizations for PL/I Record Files

PL/I Record File Attributes	OpenVOS File Organization
`sequential`	Sequential, stream, relative, or fixed
`keyed sequential`	Sequential, relative, or fixed (index is required or created)
`direct`	Relative or fixed

You can specify the OpenVOS file organization when you open the associated file control block. See "Opening File Control Blocks," later in this chapter, for more information.

The next three sections discuss the following topics.

- "Sequential Access"
- "Keyed Sequential Access"
- "Direct Access"

Sequential Access

If you open a file with the `sequential` attribute and without the `keyed` attribute, records are accessed in the order in which they appear in the file. The file always has a current position. Records in such a file can be read or written, but cannot be rewritten or deleted.

You can alter the order in which records are read from a sequential input file by specifying an index in the `title` option of the `open` statement. Subsequent `read` statements on that file access records in index order. You **cannot** specify an index for sequential output files.

When you open a file with the `sequential` attribute for input or update, the current position is initially set to the point immediately before the first record. The first `read` statement advances the current position to the first record, then reads it. The next `read` statement advances the current position to the second record and reads it, and so forth. The only exception to this process is if you use an OpenVOS operating system subroutine that sets the just-positioned switch for the I/O port. When this switch is set and the current record is not deleted, the `read` statement does not change the position of the file. The following subroutines set the just-positioned switch.

- `s$keyed_position`
- `s$keyed_position_delete`
- `s$rel_position`
- `s$seq_lock_record`

- s$seq_position
- s$seq_position_read

See the *OpenVOS PL/I Subroutines Manual* (R005) for more information on these subroutines.

The write statement appends records to the end of the file. It does not alter the current position of the file.

If a file opened for sequential output does not exist, the operating system creates an OpenVOS sequential file.

Keyed Sequential Access

If you open a file with both the sequential and keyed attributes, you can access records either sequentially in index order, or by specifying an index key. A file opened with the sequential and keyed attributes must have an index with character-string keys of up to 64 characters each.

By default, a separate-key index is used. To use an embedded-key index, you must describe the position of the keys in the -keyis phrase of the title option of the open statement (see the description of the open statement in Chapter 12). You **cannot** use an item index.

If a file opened for keyed sequential output does not exist, an OpenVOS sequential file with a separate key index is created. By default, the index is named primary and its keys are sorted in ascending ASCII order. To use a different index, specify the index name in the title option of the open statement.

Each record in the file must have a unique index-key value unless you specify -duplicatekeys in the title option of the open statement (see Chapter 12 for a description of the open statement). Using null keys, while technically invalid in standard PL/I, is not diagnosed as an error.

Records in a keyed sequential file can be read, written, rewritten, or deleted.

A sequential record file, with or without keys, always has a current position. When a file is opened for input or update, its current position is initially set to the point immediately before the first record. The first I/O operation on the file must be a read, or it must involve a key or keyfrom option to alter the current position.

If you open a file for keyed sequential access, you can optionally specify a key value in subsequent I/O statements on that file. If you do not specify a key value, the file is treated the same as an nonkeyed sequential file, except that records are read in index order.

If you specify a key value in a read statement, the record with that index key value is read and the current position is changed to that record.

The write statement, with or without a key value, appends records to the end of the file. The write statement alters the current position within the file if you specify a non-null value in the keyfrom clause. Note that if you write a record without a key value, you cannot read that value when the file is opened with the keyed attribute.

If you specify a `rewrite` or `delete` statement with a non-null key value, the current position is reset to the rewritten or deleted record.

Direct Access

If a file is opened with the `direct` attribute, you access a record by specifying an integer key. Direct access does not use an index; instead, the integer key refers to the ordinal position of the record in the file. This method is often more efficient than keyed sequential access.

You can read, write, rewrite, or delete records in a direct file.

The current position of a direct file is meaningless; records must always be accessed by key values. Because keys are essential to direct access, the `direct` attribute implies the `keyed` attribute; whether or not you specify the `keyed` attribute is insignificant if you specify `direct`. Implied attributes are discussed later in this chapter.

If a file opened for direct output does not exist, the operating system creates it. By default, a relative file with a record size of 1024 is created. You can specify another file organization or record size in the `title` option of the `open` statement, as described in Chapter 12.

Operations on Records

The access specified for a file limits the operations you can perform on the file. Table 14-4 explains the operations allowed on files relative to file access.

Table 14-4. Operations Allowed on Record Files

Access	Operation Allowed
Sequential	`read` or `write` statements; key options not allowed
Keyed sequential	`read`, `write`, `rewrite`, or `delete` statements; key optional
Direct	`read`, `write`, `rewrite`, or `delete` statements; key required

Each time you open a file control block, you must specify one of the following attributes.

- `input`
- `output`
- `update`

The default is `input`. These attributes further limit the allowable operations on the file. Table 14-5 explains the operations allowed on files when `input`, `output`, or `update` is specified.

Table 14-5. Record Operations Attributes

Attribute	Operation Allowed
`input`	Read only
`output`	Write only

Table 14-5. Record Operations Attributes

Attribute	Operation Allowed
`update`	Read, write, rewrite, or delete

If a file opened with the `input` attribute does not exist, an `undefinedfile` condition is signaled. The `undefinedfile` condition is discussed in Chapter 15.

If a file opened for `output` or `update` does not exist, the operating system creates it. If you specify the `keyed` and `sequential` attributes, the created file is indexed; otherwise, the file is not indexed.

If a file opened for `output` already exists, the operating system deletes it and creates a new file unless you specify `-append` or `-truncate` within the `title` option of the `open` statement. See Chapter 12 for a discussion of the `open` statement.

The next four sections discuss the following topics.

- "Reading Records"
- "Writing Records"
- "Rewriting Records"
- "Deleting Records"

Reading Records

The `read` statement reads records from a file. You can transmit a record value to a program variable, or you can transmit the value to a buffer associated with the file control block and set a pointer variable to it. The `read` statement has the following syntax.

```
                        { into(variable) }  [   key(key_value)     ]
read file(file_name)    {                }  [                      ] ;
                        { set(pointer)   }  [ keyto(key_variable)  ]
```

A variable referenced in the `into` option cannot be an unaligned bit string or a structure composed entirely of unaligned bit strings.

The `key` option is **required** if the file control block is open for direct access and is **optional** for keyed sequential access. The `key` and `keyto` options are not allowed if the file control block was opened for nonkeyed sequential access.

The following examples illustrate some forms of the `read` statement.

```
read file(f) into(x);

read file(f) set(p) key(n + n);

read file(f) keyto(next_key) into(next_rec);
```

The effect of a `read` statement on the current file position is described in "Record File Positioning" later in this chapter.

Writing Records

The `write` statement writes records to a file. The `write` statement has the following syntax.

```
write file(file_name) from(variable)[keyfrom(key_value)];
```

The variable used in the `from` option cannot be an unaligned bit string or a structure composed entirely of unaligned bit strings. The `from` option cannot contain an expression.

The `keyfrom` option is **required** if the file control block is open for direct access, but is **optional** for a keyed sequential file. The `keyfrom` option cannot be used if the file control block is open for nonkeyed sequential access.

In a sequential or keyed sequential file, the `write` statement always appends records to the end of the file. In a direct file, the value in the `keyfrom` option determines the location of the record.

The following examples illustrate the `write` statement.

```
write file(g) from(x);
write file(g) from(y) keyfrom(n + m);
```

The effect of a `write` statement on the current file position is described in "Record File Positioning" later in this chapter.

Rewriting Records

The `rewrite` statement writes over an existing record. The `rewrite` statement can only be applied to file control blocks that have been opened for direct or keyed sequential access with the `update` attribute. The `rewrite` statement has the following syntax.

```
rewrite file(file_reference)
from(record_string)[key(key_value)];
```

The `from` option of the `rewrite` statement cannot contain an expression.

The `key` option is **required** if the file control block is open for direct access, but is **optional** if the file control block is open for keyed sequential access.

The following example illustrates a `rewrite` statement.

```
rewrite file(h) from(x) key(n + m);
```

The effect of a `rewrite` statement on the current file position of a keyed sequential file is described in "Record File Positioning" later in this chapter.

Deleting Records

The `delete` statement removes an existing record from a file. The file must have been opened for direct or sequential access with the `update` attribute. The `delete` statement has the following syntax.

```
delete file(file_reference)[key(key_value)];
```

The `key` option is **required** if the file control block is open for direct access, but is **optional** if the file control block is open for keyed sequential access.

The following example illustrates the `delete` statement.

```
delete file(h) key(n + m);
```

The effect of a `delete` statement on the current file position of a keyed sequential file is described in the next section, "Record File Positioning."

Record File Positioning

Each open sequential and keyed sequential file has a current position. Table 14-6 describes how the current position of a file is initialized. Note that the initial position of an output file depends on whether or not you specify `-append` in the `title` option of the `open` statement.

Table 14-6. Initial Positions for Record Files

Operation	Initial Position
`input`	Before first record
`update`	Before first record
`output -append`	At end of file
`output [-truncate]`	File is empty

Table 14-7 describes how the current position of a file is altered by language I/O operations.

Table 14-7. Current Position in Record I/O

	Access Description	
Statement	Sequential or keyed sequential; `key` or `keyfrom` omitted	Sequential or keyed sequential; `key` or `keyfrom` specified
`read`	Positions to read record†	Positions to read record
`write`	No change	Positions to written record‡
`rewrite`	No change	Positions to rewritten record‡
`delete`	No change	Positions to deleted record‡

† Usually, this is the next record. The only exception occurs if the `just_positioned` flag for the port is set and the current record is not deleted, in which case the current record is read.
‡ If the key value specified in the `key` or `keyfrom` clause is the null string, the current position does not change.

Files do **not** retain current positions between openings.

Opening and Closing Files

A file control block must be opened before you can perform I/O through it. When you open a file control block, an OpenVOS port is attached and opened. When you have finished your I/O, you can close the file control block. Closing a file closes and detaches the port.

The next two sections discuss the following topics.

- "Opening File Control Blocks"
- "Closing File Control Blocks"

Opening File Control Blocks

If an I/O statement is encountered and the file control block is not open, the file control block is implicitly opened before the I/O operation is performed. The attributes of the implicit opening appear in Table 14-8.

Table 14-8. Attributes for Implicit File-Control-Block Opening

I/O Statement	Attributes of Implicit Opening
`get`	`stream input`
`put`	`stream output`†
`read`	`record sequential input`
`write`	`record sequential output`
`rewrite`	`record sequential update`‡
`delete`	`record sequential update`‡

† If you use the `sysprint` file control block, the `print` attribute is also included in the implicit opening.
‡ If the file is not declared to have the `keyed` attribute, the `error` condition is signaled.

You can explicitly open a file control block by coding an `open` statement. The `open` statement has the following general syntax.

open file(*file_name*) [title(*title_string*)]

[*file_attributes_and_options*];

The following is a more detailed syntax.

```
open file(file_name)[title(title_string)]

[ [stream][linesize(line_size)][             input            ]  ]
[                              [ [output][print][pagesize(page_size)] ]  ]  ;
[                                                                  ]
[            [record] [ input  ] [ sequential ] [keyed]            ]
[                     [ output ] [ direct     ]                    ]
[                     [ update ]                                   ]
```

To open a file for stream I/O, use the following syntax.

```
open file(file_name)[title(title_string)][stream]

[linesize(line_size)] [                 input              ];
                      [ [output][print][pagesize(page_size)] ]
```

To open a file for record I/O, use the following syntax.

```
open file(file_name)[title(title_string)]

        [ input  ]  [ sequential ]
record  [ output ]  [ direct     ]  [keyed];
        [ update ]
```

The `open` statement is explained in Chapter 12.

You can specify the path name of the file or device associated with the open file in the `title` option. However, if a port named the same as the file ID of *`file_name`* is already attached to an OpenVOS file or device, any path name specified in the `title` option is ignored.

The file attributes in the `open` statement are combined with any attributes declared for the file constant associated with the file control block being opened.

Certain attributes imply others. You need not explicitly state an attribute that is implied by a stated attribute. Implied attributes are explained in the next section, "Implied File Attributes."

If, after combining the attribute lists and including any implied attributes, the resulting attribute list is incomplete, defaults are added. Default attributes are discussed in "Default File Attributes," later in this chapter.

If the resulting attribute list contains inconsistent attributes, the `error` condition is signaled.

All OpenVOS PL/I attributes are described in Chapter 7.

Implied File Attributes

Certain file attributes imply others. You need not specify an attribute if it is implied by an attribute that is specified. Table 14-9 lists the attributes that have implied attributes.

Table 14-9. Implied File Attributes

Specified Attribute	Implied Attributes
`print`	`stream output`
`direct`	`record keyed`
`keyed`	`record`
`sequential`	`record`
`update`	`record`

For example, the following two statements are equivalent.

```
open file(f) print;
open file(f) stream output print;
```

The following two sample statements are also equivalent.

```
open file(f) direct;
open file(f) record direct keyed;
```

Default File Attributes

Some default file attributes and options are listed in Table 14-10. The default attribute is determined by which attributes are specified and by which attributes or options are not specified.

Table 14-10. Default File Attributes and Options

Attribute Specified	Attribute/Option Omitted	Default Attribute
N/A	`stream` or `record`	`stream`
N/A	`input`, `output`, or `update`	`input`
`record`	`sequential` or `direct`	`sequential`
`stream`	*`line_size`*	`linesize(80)` †
`print`	*`page_size`*	`pagesize(60)`

† If the device to which the file control block is attached has a line size, that value is the default.

Because of the default rules, the following two statements are equivalent.

```
open file(f) stream input;
open file(f);
```

The following statements are also equivalent.

```
open file(f) record;
open file(f) record input sequential;
```

Closing File Control Blocks

When a program terminates, all file control blocks that have been opened in that program are closed. You can explicitly close a file control block by including a `close` statement, which has the following syntax.

```
close file(file_control_block);
```

The `close` statement closes and detaches the port associated with the file control block.

Once you have closed a file control block, you can reopen it with different file attributes and associate it with a different OpenVOS file or device.

The `close` statement is explained in Chapter 12.

Chapter 15:
Exception Handling

This chapter discusses the following topics related to exception handling in PL/I.

- "Overview"
- "On-Units"
- "Computational Conditions"
- "I/O Conditions"
- "System Conditions"
- "Programmer-Defined Conditions"
- "Condition Resolution"

Overview

Certain events that sometimes arise during the execution of a PL/I program require special handling. Such events are called *exceptions*. For example, an exception occurs if you attempt to divide a number by zero, since the result is undefined. When such an event occurs, a *condition* is signaled and a special routine is activated to handle that condition. Such a routine is called an *on-unit* because it is executed only **on** a certain condition.

OpenVOS PL/I recognizes the following predefined conditions. The conditions marked with an asterisk (*) are OpenVOS extensions; the others are standard PL/I conditions.

```
*alarmtimer
*anyother
*break
*cleanup
*cputimer
 endfile(file)
 endpage(file)
 error
 fixedoverflow
 key(file)
 overflow
*reenter
*stopprocess
 undefinedfile(file)
 underflow
*warning
 zerodivide
```

You cannot use a condition name as a parameter for a procedure.

On-Units

Each condition has a predefined default on-unit. You can establish your own on-unit for each condition within each block activation using the `on` statement.

The `on` statement has the following general syntax.

```
on condition_name on_unit;
```

The condition name can be any of those condition names listed in the preceding section, or `condition(`*`programmer_condition`*`)`, where *`programmer_condition`* is a condition defined by the programmer. Programmer-defined conditions are discussed later in this chapter.

The *`on_unit`* can be any of the following:

- a `begin` block
- `system`, which invokes the default handler for that condition
- any single statement **other than** `declare`, `do`, `end`, `entry`, `if`, `on`, `procedure`, or `return`

When an on-unit is activated, a stack frame associated with it is pushed on to the stack. When the end of the on-unit is reached, the stack is popped and, if possible, execution resumes at the point at which the condition was signaled.

The activation of an on-unit can also be terminated by a `goto` statement that transfers control out of the on-unit. When such a `goto` statement is executed, the stack frame for the on-unit is popped along with all stack frames back to, but not including, the stack frame containing the statement referenced by the `goto` statement. Execution continues with that statement.

An on-unit **cannot** contain the `return` statement.

Attempting to exit the on-unit of a fatal condition produces unpredictable results. The `error` and `fixedoverflow` conditions are always fatal. In some cases, the `undefinedfile` and `zerodivide` conditions are also fatal. For more information about the `zerodivide` condition and the `undefinedfile` condition, respectively, see "The `zerodivide` Condition" and "The `undefinedfile` Condition" later in this chapter.

The following program fragment contains three examples of on-units.

```
on underflow
     put skip list('Underflow has occurred.');

on endfile(g)
     begin;
          call clean_up;
          stop;
     end;

on error system;
```

If you specify `system` as the on-unit, the default handler for that condition is used. If you use the `system` on-unit within an on-unit you define, your on-unit cannot activate itself recursively.

The following example illustrates the `system` on-unit.

```
on error
     begin;
          on error system;
          error_code = oncode();
          call s$error(error_code, MY_NAME, message);
          stop;
     end;
```

Execution of an `on` statement establishes the on-unit; it does not cause immediate execution of the on-unit.

The `on` statement is further described in Chapter 12.

Table 15-1 lists the four OpenVOS PL/I built-in functions that are used specifically in on-units.

Table 15-1. Built-In Functions Used with On-Units

Function	Description
`oncode`	Returns a 2-byte integer status code describing why the current condition was signaled
`onfile`	Returns a character string containing the file ID of the file for which the current file condition was signaled
`onkey`	Returns a character string containing the key value for which the current `key` condition was signaled
`onloc`	Returns the entry name for the block in which the current condition was signaled

None of the condition built-in functions takes arguments. See Chapter 13 for further information on the OpenVOS PL/I built-in functions.

If a condition is signaled and no on-unit is established within the current block, the most recent on-unit established for that condition in the stack is executed. If no applicable on-unit is found, a default on-unit is executed. The process of resolving a signal to an on-unit is discussed in "Condition Resolution" later in this chapter.

You can use the `signal` statement to signal a specific condition. The `signal` statement is most often used in debugging programs or to signal a programmer-defined condition.

You can use the `revert` statement to revert the on-unit established within the current block activation for a specific condition. The next time that condition is signaled, the next most recent on-unit for that condition on the stack is executed; if no other on-unit for the condition exists on the stack, the default on-unit is executed.

The `signal` and `revert` statements are described in Chapter 12.

The predefined conditions that PL/I recognizes can be classified into three groups.

- computational conditions
- I/O conditions
- system conditions

You can also establish your own conditions. See "Programmer-Defined Conditions," later in this chapter, for additional information about defining conditions.

Computational Conditions

Four computational conditions are predefined. Table 15-2 lists these conditions and describes the default handler action for each.

Table 15-2. Default Handler Actions for Computational Conditions

Condition	Default Action
`fixedoverflow`	Writes a message and signals the `error` condition
`overflow`	Writes a message and signals the `error` condition
`underflow`	Writes a message and returns to the computation with a zero or near-zero value
`zerodivide`	Writes a message and signals the `error` condition

This section discusses the following topics.

- "The `fixedoverflow` Condition"
- "The `overflow` Condition"
- "The `underflow` Condition"
- "The `zerodivide` Condition"

The `fixedoverflow` Condition

A fixed-point overflow occurs when a fixed-point operation produces a result that does not fit in the internal register used for the calculation: two, four, or eight bytes.

If you do not select the `-fixedoverflow` compiler argument, most fixed-point overflows are **not** detected. If you select `-fixedoverflow`, all fixed-point overflows are detected. If a fixed-point overflow occurs but is not detected, the results are undefined.

Exceeding the declared precision of a fixed-point target variable does **not** necessarily constitute a fixed-point overflow. A result can exceed the stated precision without exceeding the register size. For example, in the following program fragment, `fixedoverflow` is **not** signaled.

```
declare   target    fixed dec(5);

     target = 2 ** 30;               /* target = 1073741824 */
     target = divide(target,1F6,5);  /* target = 1073       */

     if target > 99999
     then goto TOO_BIG;
          .
          .
          .
TOO_BIG:
          .
          .
          .
```

The result of the exponentiation in the first assignment statement of the preceding fragment exceeds the declared precision of `target`, but it does not exceed the register size. Therefore, `fixedoverflow` is not signaled. Although the program is technically in error, arithmetic operations subsequently performed on the oversized value of `target` may yield reasonable results, depending on the implementation. For example, the `divide` built-in function reference in the second assignment statement produces the expected value. Nonarithmetic operations involving an oversized value might produce a run-time error. You can use an `if` statement, as in the example, to detect values of `target` that exceed the declared precision.

The default on-unit for the `fixedoverflow` condition writes an error message to the `terminal_output` port and signals the `error` condition.

The following example shows an on-unit you could establish for the `fixedoverflow` condition.

```
on fixedoverflow
     begin;
          put skip list('Fixed-point overflow.');
          put skip list('Occurred in ',onloc());
          stop;
     end;
```

If control reaches the end of a `fixedoverflow` on-unit, the effect is undefined.

The `overflow` Condition

The `overflow` condition is signaled whenever the exponent of a floating-point result is too large to fit in the target variable or register. The default on-unit writes an error message to the `terminal_output` port and signals the `error` condition.

The following example shows an on-unit you could establish for the `overflow` condition.

```
on overflow
     begin;
          put skip(2) list('A floating-point overflow occurred.');
          call fatal_error;
          stop;
     end;
```

If control reaches the end of the on-unit, control transfers back to the computation that caused the overflow. Execution resumes using positive or negative infinity as the calculated value.

The `underflow` Condition

The `underflow` condition is signaled whenever the exponent of a nonzero normalized floating-point result is too small to be stored in the target variable or register. The default on-unit writes an error message to the `terminal_output` port and returns to the computation.

All models provide *gentle underflow*. This means that when control returns from an `underflow` on-unit to the computation, it uses a denormalized, near-zero value. Stratus machines support denormalized values; when they return to the computation, they use the denormalized value.

Denormalized values are described in Appendix B.

The following example shows an on-unit you could establish to override the default handler for the `underflow` condition.

```
on underflow
     begin;
          put skip list('An underflow has occurred.');
          stop;
     end;
```

The `zerodivide` Condition

The `zerodivide` condition is signaled whenever division by zero is attempted. The default on-unit writes an error message to the `terminal_output` port and signals the `error` condition.

The following example shows an on-unit you could establish for the `zerodivide` condition.

```
on zerodivide
     begin;
          put skip list('A divisor is zero.');
          rate = 1;
          call rate_processor;
          stop;
     end;
```

If control reaches the end of a `zerodivide` on-unit, the effect depends on the operands of the division. If one of the operands is a floating-point value and the dividend is not zero, control transfers back to the calculation and execution continues with a value of positive or negative infinity. If both operands are fixed-point values or the dividend is zero, the `zerodivide` condition is fatal; if control reaches the end of the `zerodivide` on-unit, the effect is undefined.

I/O Conditions

Four conditions related to the status of a file control block are predefined. Table 15-3 lists these conditions and describes the default handler action for each.

Table 15-3. Default Handler Actions for I/O Conditions

Condition	Default Action
`endfile(`*`file`*`)`	Writes a message and signals the `error` condition
`endpage(`*`file`*`)`	Puts a new page in the file and returns to the point of the signal
`key(`*`file`*`)`	Writes a message and returns to the statement
`undefinedfile(file)`	Writes a message and signals the `error` condition

Note that each of the conditions listed in Table 15-3 is qualified by a file reference. Distinct I/O conditions are established for each file control block. This means that if files `f` and `g` are associated with different file control blocks, `endfile(f)` is a different condition than `endfile(g)`. However, if `f` and `g` are associated with the same file control block, `endfile(f)` and `endfile(g)` refer to the same condition.

> **Note:** If a block contains two on-units for the same condition, the most recently established on-unit is used.

In the following example, on-units are established for two different `endfile` conditions, provided that `v ^= h`.

```
declare   (f,g,h)   file;
declare   v         file variable;

     on endfile(h)
          goto NO_MORE_NAMES;

     on endfile(v)
          begin;
               if onfile() = 'f'
               then v = g;
               else goto END_INPUT;
          end;
               .
               .
               .
END_INPUT:
               .
               .
               .
```

This section discusses the following topics.

- "The `endfile` Condition"
- "The `endpage` Condition"
- "The `key` Condition"
- "The `undefinedfile` Condition"

The `endfile` Condition

An `endfile` condition is signaled when you attempt to read past the end of a file. The default on-unit for the `endfile` condition writes an error message to the `terminal_output` port and then signals the `error` condition.

If, as a result of executing the on-unit, an attempt is made to read again from the same file control block, the condition is resignaled.

If control reaches the end of an `endfile` on-unit, control transfers to the statement following the `get` or `read` statement that triggered the condition, and execution continues.

The following example illustrates an on-unit you could establish for the `endfile` condition.

```
on endfile(file2)
     more_recs = no;
```

The `endpage` Condition

An `endpage` condition is signaled when the line to be written by a `write` or `put` statement has a line number that is one greater than the page size of a stream output file with the `print` attribute. The default on-unit starts a new page in the output file, thereby resetting the line number to 1, and returns to the point of the signal.

For further information on print files, see Chapter 14.

If control reaches the end of an `endpage` on-unit, control transfers to the `write` or `put` statement that triggered the condition. Any additional output is then written to the file.

If a new page is written by the on-unit or at some point after the on-unit is executed, the line number is reset to 1 and `endpage` is signaled the next time the line number is one greater than the page size.

If execution of the on-unit completes without writing a new page, the line number of the file increases indefinitely; the `endpage` condition is **not** resignaled by subsequent output.

The following example illustrates an on-unit you could establish for the `endpage` condition.

```
if ^page_breaks
then on endpage(f);      /* Do not put page breaks */
```

The `key` Condition

A `key` condition is signaled in the following cases.

- The value referenced in the `key` option of a `read`, `rewrite`, or `delete` statement does not match the key value of a file record.
- The value specified in the `keyfrom` option of a `write` statement matches the key value of an existing file record, and you did not specify `-duplicatekeys` in the `title` option of the `open` statement.

The default on-unit writes an error message to the `terminal_output` port and signals the `error` condition.

Just before a `key` condition is signaled, the value of the `onkey` built-in function is set to the key value from the I/O statement. Therefore, you can use the `onkey` function within the on-unit to determine the key value that caused the condition to be signaled.

The following example shows an on-unit you could establish for the `key` condition.

```
on key(g)
     begin;
          if onkey() = ltrim(char(n + 1))
          then signal endfile(g);
          else put skip list('Key not found: ', key_value);
     end;
```

If control reaches the end of a `key` on-unit, control transfers to the statement following the I/O statement that triggered the condition and execution resumes.

The `undefinedfile` Condition

An `undefinedfile` condition is signaled whenever an attempt to open a file fails. This situation most often occurs when a file being opened for input does not exist, or if you specify inconsistent file attributes. The default on-unit writes an error message to the `terminal_output` port and signals the `error` condition.

The following example shows an on-unit you could establish for the `undefinedfile` condition.

```
on undefinedfile(f)
     begin;
          put skip list('Data file undefined: ');
          put skip list(full_path_name);
          put skip list('Code: ', oncode());
          call cleanup;
          stop;
     end;
```

An `undefinedfile` condition signaled from an `open` statement is not fatal. If control reaches the end of the `undefinedfile` on-unit, control transfers to the statement following the `open` statement and execution continues; the file is **not** opened.

If the `undefinedfile` condition is signaled as the result of an attempted implicit opening, the condition is fatal. If control reaches the end of the `undefinedfile` on-unit, the effect is undefined.

System Conditions

Certain conditions are related to system events. Table 15-4 lists the OpenVOS system conditions and explains the default handler action for each.

Table 15-4. Default Handler Actions for OpenVOS System Conditions

Condition	Default Action
`alarmtimer`	Signals the `error` condition.
`anyother`	Should not be signaled.
`break`	Writes "`BREAK`" and puts the process at break level.
`cleanup`	Should not be signaled.
`cputimer`	Signals the `error` condition.
`error`	Writes a message and puts the process at break level.
`reenter`	Writes a message and puts the process at break level.
`stopprocess`	Sends a `STOPPROCESS` interrupt to the process.
`warning`	Writes a message and resumes execution at the point of the signal. If the oncode is `e$abort_output`, however, no message is written.

This section describes the following topics.

- "Suspension and Re-entry Conditions"
- "Timer Conditions"
- "Nonspecific Conditions"

Suspension and Re-entry Conditions

The `break` and `reenter` conditions affect program suspension and re-entry.

The next two sections discuss the following topics.

- "The `break` Condition"
- "The `reenter` Condition"

The `break` Condition

The `break` condition is signaled whenever the user issues the CTRL BREAK request during program execution.

The default on-unit writes the following message to the terminal's screen.

```
BREAK
Request?  (stop, continue, debug, keep, login, re-enter)
```

The process is then at break level. The user must then choose one of the requests shown in the preceding message. Table 15-5 explains the effect of each request.

Table 15-5. Break-Level Requests

Break-Level Request	Action
`stop`	Terminates program execution
`continue`	Continues program execution, if possible
`debug`	Invokes the debugger
`keep`	Creates a keep module
`login`	Clones a subprocess
`re-enter`	Signals the `reenter` condition

The next section, "The `reenter` Condition," explains the `reenter` condition.

If you issue the `continue` request and execution cannot resume, you are returned to break level.

The following example shows an on-unit you could establish for the `break` condition.

```
on break
     put skip list('You cannot break out of this program.');
```

If control reaches the end of a `break` on-unit, control returns to the point of the signal and execution continues. Therefore, the on-unit in the preceding example prevents the user from breaking out of the program.

The `reenter` Condition

The `reenter` condition is signaled when the user issues the `re-enter` request at break level. The `re-enter` request allows continued execution of the program from a clean point, usually the beginning of a request-processing loop.

To use the `re-enter` request, you must explicitly establish an on-unit for the `reenter` condition.

The following example shows an on-unit you could establish for the `reenter` condition.

```
          on reenter
               goto REQUEST_LOOP;
               .
               .
               .
     REQUEST_LOOP:
          do while(^end_flag);
               call read_next_request;
               call execute_request(end_flag);
          end REQUEST_LOOP;
```

If no on-unit is established for the `reenter` condition, typing `re-enter` at break level returns the process to break level.

Timer Conditions

The `alarmtimer` and `cputimer` conditions are signaled as a result of elapsed timers. They are not dependent upon any other activity occurring in the program.

The next two sections discuss the following topics.

- "The `alarmtimer` Condition"
- "The `cputimer` Condition"

The `alarmtimer` Condition

The `alarmtimer` condition is signaled when a specified amount of time has elapsed. You set the amount of time using the `s$set_alarm_timer` subroutine. You can use this condition to send an interrupt when a specified time has elapsed.

The following example shows an on-unit you could establish for the `alarmtimer` condition.

```
declare set_time                fixed bin(31);
declare error_code              fixed bin(15);

declare s$set_alarm_timer entry (fixed bin(31),
                                fixed bin(31),
                                fixed bin(15));

     set_time = 2048; /* Sets the timer (in 1/1024’s) */
     call s$set_alarm_timer(set_time, (0), error_code);

     on alarmtimer
          begin;
               put skip list ('Allowed time has elapsed.');
                    .
                    .
                    .
               call s$set_alarm_timer(set_time, (0), error_code);
          end;
```

See the *OpenVOS PL/I Subroutines Manual* (R005) for additional information about the `s$set_alarm_timer` subroutine.

The `cputimer` Condition

The `cputimer` condition is signaled when a specified amount of CPU time has elapsed. You set the amount of CPU time using the `s$set_cpu_timer` subroutine. You can use this condition to send an interrupt when a process has used a specified amount of CPU time.

The following example shows an on-unit you could establish for the `cputimer` condition.

```
declare set_cpu_time          fixed bin(31);
declare error_code            fixed bin(15);

declare s$set_cpu_timer entry (fixed bin(31),
                              fixed bin(31),
                              fixed bin(15));

     set_cpu_time = 5000; /* Sets the timer (in CPU seconds) */
     call s$set_cpu_timer(set_cpu_time, (0), error_code);

     on cputimer
          begin;
               put skip list ('Allowed CPU time has elapsed.');
                    .
                    .
                    .
               call s$set_cpu_timer(set_cpu_time, (0), error_code);
          end;
```

See the *OpenVOS PL/I Subroutines Manual* (R005) for additional information about the `s$set_cpu_timer` subroutine.

Nonspecific Conditions

The `anyother`, `cleanup`, `error`, `stopprocess`, and `warning` conditions are nonspecific conditions. The `error` and `warning` conditions are signaled when an uncategorized exception occurs. An on-unit for the `anyother` condition is executed only when no on-unit for the current condition has been established in the current block. The `cleanup` condition is used to regain control during a nonlocal `goto`. The `stopprocess` condition is signaled when the process is stopped by a `stop_process` command or a `s$stop_process` subroutine issued by a different process.

The next five sections discuss the following topics.

- "The `anyother` Condition"
- "The `cleanup` Condition"
- "The `error` Condition"
- "The `stopprocess` Condition"
- "The `warning` Condition"

The `anyother` Condition

If you establish an on-unit for the `anyother` condition, that on-unit is executed whenever a condition is signaled and no other on-unit has been established in the current block for that condition.

The following example shows an on-unit you could establish for the `anyother` condition.

```
declare   s$continue_to_signal     entry;

     on anyother
          begin;
               put skip list('Other condition signaled.');
               call s$continue_to_signal;
          end;
```

In the preceding example, the call to `s$continue_to_signal` tells the operating system to resignal, in the preceding stack frame, the condition that caused the `anyother` on-unit to be executed. See the *OpenVOS PL/I Subroutines Manual* (R005) for more information.

You can specify `system` as an on-unit for the `anyother` condition.

```
on anyother system;
```

In this example, the on-unit inhibits the search of preceding blocks for condition handlers. If the current block does contain an on-unit for a signaled condition, the system default handler for that condition is invoked. However, if you call `s$continue_to_signal` within an on-unit, previous blocks are searched. Note that the `anyother` condition is signaled in certain situations, such as when a `stop_process` command has been executed. Therefore, you should always include the call to `s$continue_to_signal`, so that preceding blocks can be searched.

Notes:

1. Because the `anyother` condition may produce **unpredictable** results, you should be familiar with all appropriate operating system conditions before you choose this condition. Rather than use this condition as a catchall condition, you should explicitly enable the specific conditions you want to trap.

2. Before coding an on-unit for the `anyother` condition, make sure you are familiar with the rules in "Condition Resolution" later in this chapter.

The `cleanup` Condition

The `cleanup` condition is used to gain control when a block is aborted due to a nonlocal `goto`. The `cleanup` condition cannot be signaled by a program or by a call to `s$signal_condition`. Rather, the nonlocal `goto` examines each block between the current block and the target block of the nonlocal `goto` (including the block starting the nonlocal `goto` but not that of the target) to test whether a handler for the `cleanup` condition has been established.

If a handler for the `cleanup` condition has been established, the handler is reverted and then called. When the handler returns, or if no cleanup handler exists, all on-units are reverted and the block is popped from the stack.

A handler for the cleanup must return normally. It must not perform a nonlocal `goto` itself, as this will abort the processing of the first nonlocal `goto`.

Other condition handlers for the block remain in effect while the cleanup handler executes. Thus, any condition that is raised while executing the cleanup handler can be caught and handled. This may not produce the desired behavior. A cleanup handler can attempt to set up a system handler for the `error` condition that guarantees that problems will be handled, as shown in the following example.

```
on cleanup
begin;
     on error system;         /* Perform cleanup actions */
end;
```

A handler for the `anyother` condition is never invoked for the `cleanup` condition. It is unnecessary to call the `s$continue_to_signal` subroutine from a cleanup handler; this action is automatic.

The `error` condition is signaled with an oncode that indicates the nature of the problem if the stack is destroyed.

See the *OpenVOS PL/I Subroutines Manual* (R005) for additional information about the condition-handling subroutines `s$enable_condition`, `s$signal_condition`, and `s$continue_to_signal`.

The `error` Condition

The `error` condition is signaled when certain fatal errors or exceptional conditions occur that are not covered by other predefined conditions. Several of the default condition handlers signal the `error` condition after writing a message to the `terminal_output` port.

Just before the `error` condition is signaled, the value of the `oncode` built-in function is set to an OpenVOS status code indicating the nature of the condition or error.

The default on-unit for the `error` condition writes an error message to the `terminal_output` port and puts the process at break level.

The following example shows an on-unit you could establish for the `error` condition.

```
on error
     begin;
          on error system;
          if oncode() = ENDCODE
          then stop;
          else signal error;
     end;
```

If control reaches the end of an `error` on-unit, the effect is undefined.

The `stopprocess` Condition

The `stopprocess` condition is signaled when a process is stopped either by the `stop_process` command or by the `s$stop_process` subroutine issued by a different process. After the condition is signaled, the operating system allows one minute (in real time) for the code defined in the condition handler to execute and then stops the process. The operating system performs the following actions when it stops the process.

- closes all files opened by the process that can be closed
- detaches all ports that the process attached
- detaches all events that the process attached
- unlocks all locks that the process locked

The following example shows an on-unit you could establish for the `stopprocess` condition.

```
on stopprocess
     begin;
          put skip list('Process has been stopped');
          call cleanup_routine;
     end;
```

For more information on stopping a process, see the description of the `stop_process` command in the *OpenVOS Commands Reference Manual* (R098) or the description of the `s$stop_process` subroutine in the *OpenVOS PL/I Subroutines Manual* (R005).

The `warning` Condition

The `warning` condition is signaled only when certain nonfatal conditions occur, such as when the user aborts output by pressing the key that invokes the `CANCEL` function.

Just before the `warning` condition is signaled, the value of the `oncode` built-in function is set to an OpenVOS status code indicating the nature of the warning.

The default on-unit for the `warning` condition writes an error message and continues execution at the point of the signal. If the error code is `e$abort_output` (`1279`), however, no error message is written; in order for output to occur again, you must reset the terminal using the `s$control` subroutine with the `RESET_OUTPUT (224)` opcode. See the *OpenVOS PL/I Subroutines Manual* (R005) for more information on the `s$control` subroutine; see the manual *VOS Communications Software: Asynchronous Communications* (R025) for more information on the `RESET_OUTPUT` opcode.

The following example shows an on-unit you could establish for the `warning` condition.

```
declare   e$abort_output      fixed bin(15) static external;

   on warning
      begin;
        if oncode() = e$abort_output
        then goto REQUEST_LOOP;      /* Re-enables terminal */
        else put skip list('Warning: ',ltrim(char(oncode())));
      end;
```

If control reaches the end of a `warning` condition on-unit, control transfers to the point of the signal and execution continues.

See the *OpenVOS PL/I Subroutines Manual* (R005) for additional information about the `s$control` subroutine.

Programmer-Defined Conditions

You can define your own conditions within a program and signal them with the `signal` statement.

First, you must declare the name of the condition with the `condition` attribute in a `declare` statement, as shown in the following example.

```
declare   too_big   condition;
```

Note: Condition names cannot be array elements or structure members.

To set up an on-unit for the condition, use an `on` statement with the following syntax.

```
on condition(condition_name) on_unit;
```

To signal the condition within the program, use a `signal` statement with the following syntax.

```
signal condition(condition_name);
```

The following example shows how a programmer-defined condition works.

```
declare   i         fixed bin(15);
declare   too_big   condition;

     on condition(too_big)
          begin;
               put skip list('The value is too large.');
               stop;
          end;
          .
          .
          .
     if i > 15
     then signal condition(too_big);
```

If control reaches the end of the on-unit for a programmer-defined condition, control transfers back to the point of the signal and execution continues.

A programmer-defined condition can be signaled only as the result of a `signal` statement.

The default handler for programmer-defined conditions writes an error message to the `terminal_output` port and signals the `error` condition.

Condition Resolution

When a condition is signaled, the compiler performs the following steps to locate a handler for that condition.

1. The current block is searched for an on-unit for the specific condition.
2. If no specific on-unit is found, and you have established an on-unit for the `anyother` condition within the current block, that on-unit is executed.
3. If no specific on-unit or `anyother` on-unit is found, these steps are repeated for the previous stack frame.

If no user on-unit is found, eventually the stack frame on the bottom of the stack is searched. The bottom stack frame is an operating system routine that establishes the default on-units for all conditions.

Appendix A: Abbreviations

This appendix discusses the abbreviations that OpenVOS PL/I allows.

PL/I provides abbreviations for certain keywords and built-in function names. These abbreviations are recognized as synonyms for the full words in every respect except one: the abbreviations of built-in function names have separate declarations (explicit or contextual) and separate name scopes. For example, `char` is the abbreviation for the `character` built-in function. Declaring `char` to be something other than a built-in function has no effect on the use of `character` as a built-in function name or data type.

In the following example, `char` is declared to be a bit string.

```
declare   char      bit(8) aligned;
declare   str       character(9);
declare   count     fixed bin(15);

     char = '00010011'b;

     count = 14;

     str = character(count);
```

In the preceding example, all references to `char` refer to a bit-string variable. All references to `character` refer to the `character` built-in function. (If `char` had not been declared as a variable, `char` and `character` would be synonymous.) The third assignment statement sets the value of `str` to `'       14'`. If the abbreviation `char` had been used for `character` in that assignment statement, you would receive a compiler error message. In some contexts, such an error would not be detected.

To maintain readability, you should use each abbreviation consistently or not at all.

Table A-1 lists the OpenVOS PL/I keyword abbreviations.

Table A-1. PL/I Keyword Abbreviations

Keyword	Abbreviation
allocate	alloc
automatic	auto
binary	bin
character	char
column	col
condition	cond
decimal	dec
declare	dcl
defined	def
dimension	dim
external	ext
fixedoverflow	fofl
initial	init
internal	int
lockingshiftintroducer	lsi
lockingshiftselector	lss
overflow	ofl
picture	pic
pointer	ptr
procedure	proc
sequential	seql
singleshiftchar	ss
undefinedfile	undf
underflow	ufl
varying	var
zerodivide	zdiv

Appendix B:
Internal Storage

This appendix discusses the following storage-related topics.

Overview

Internally, all data is stored as a series of bits. A value's data type determines how the value is converted to and from its internal representation.

Storage is divided into groups of eight bits, called *bytes*. Some illustrations in this appendix show the hexadecimal representation of storage. For example, the following represents two bytes of storage.

```
3142x
```

Each hexadecimal digit in the preceding example is equivalent to half of a byte (four bits). The first byte contains `3x` and `1x`, and the second contains `4x` and `2x`. The bit pattern of the storage for `3142x` follows.

```
00110001 01000010
```

Data Alignment

As described in Chapter 4, data can be aligned according to longmap or shortmap alignment rules. Data alignment is described using the modulus operation; the 0 (as in 0 mod2) is understood. Boundaries are described, for example, as mod2, mod4, or mod8, indicating that data begins on a boundary that is evenly divisible by 2, 4, or 8, respectively.

Under the shortmap alignment rules, most static and automatic data that is not contained in an array or structure is aligned on mod2 boundaries. Storage for pictured data and unaligned nonvarying character-string data is *byte-aligned*, meaning that storage begins on the first available byte. Unaligned bit-string data is stored beginning with the first available bit. See Table 4-5 for a list of the shortmap alignment rules.

Under the longmap alignment rules, most data is aligned on a boundary that is equivalent to the data type's size. A structure is aligned according to the boundary requirement of the largest (in bytes) member. See Table 4-6 for a list of the longmap alignment rules.

On ftServer V Series modules, longmap alignment is available and is **far more efficient** in terms of access time than shortmap alignment. However, prior to VOS Release 9.0, longmap alignment was not available on the operating system. Therefore, unless you specify longmap alignment rules, shortmap alignment rules are the default. Note, however, that the default alignment rules are site-settable.

Use either the mapping-rules options of the `%options` statement or the `longmap` or `shortmap` attribute to specify alignment.

See Chapter 4 for more information about longmap and shortmap alignment rules. See Chapter 7 for more information about the `longmap` and `shortmap` attributes. See Chapter 11 for more information on the mapping-rules options of the `%options` statement.

Arithmetic Data

All arithmetic data is stored in two, four, or eight bytes, as described later in this section. The first bit is always the sign of the value: `'0'b` for positive and `'1'b` for negative.

This section discusses the following topics.

- "Fixed-Point Data"
- "Floating-Point Data"

Fixed-Point Data

Fixed-point values are stored in two, four, or eight bytes. Table B-1 lists the storage requirements for different types of fixed-point data and the alignment for each.

Table B-1. Storage Sizes and Alignment for Fixed-Point Data *(Page 1 of 2)*

Base	Precision	Size	Shortmap Alignment	Longmap Alignment
Binary	`p <= 15`	2 bytes (16 bits)	mod2	mod2
Binary	`p > 15` but `< 32`	4 bytes (32 bits)	mod2	mod4
Binary	`p > 31` but `< 64`	8 bytes (64 bits)	mod2	mod8
Decimal	`p <= 9`	4 bytes (32 bits)	mod2	mod4

Table B-1. Storage Sizes and Alignment for Fixed-Point Data *(Page 2 of 2)*

Base	**Precision**	**Size**	**Shortmap Alignment**	**Longmap Alignment**
Decimal	`p > 9`	8 bytes (64 bits)	mod2	mod8

Note: The scaling factor of a fixed-point decimal value has no effect on its storage size.

In each case, in Table B-1, the high-order bit is the sign bit. The remaining bits contain the magnitude of the value. Negative values are stored in two's complement form, which means that the bit pattern representing the value `-x` is stored as the one's complement of `x-1`.

The following example shows the storage of the `fixed bin(15)` value `12`.

```
0 0000000 00001100
```

The following example shows the storage of the `fixed bin(15)` value `-12`.

```
1 1111111 11110100
```

The following example shows the storage for the `fixed bin(31)` values `2000000000` and `-2000000000`, respectively.

```
0 1110111 00110101 10010100 00000000
1 0001000 11001010 01101100 00000000
```

Nonintegral fixed-point decimal values are stored as if they were integers. The following formula determines the integer representation.

```
stored_value = value * 10 ** scaling_factor
```

The resulting value is stored as if it were a fixed-point integer. For example, the `fixed dec(9,2)` value `12.45` is stored as the 4-byte integer `1245`.

Floating-Point Data

Floating-point values are stored in either four or eight bytes. Table B-2 lists the storage requirements for different types of floating-point data and the alignment for each.

Table B-2. Storage Sizes and Alignment for Floating-Point Data

Base	**Precision**	**Size**	**Shortmap Alignment**	**Longmap Alignment**
Binary	`p <= 24`	4 bytes (32 bits)	mod2	mod4
Decimal	`p <= 7`	4 bytes (32 bits)	mod2	mod4
Binary	`p > 24`	8 bytes (64 bits)	mod2	mod8
Decimal	`p > 7`	8 bytes (64 bits)	mod2	mod8

The storage for decimal values is exactly the same as storage for binary values of the same storage size.

The format for both 4-byte and 8-byte floating-point values conforms to that documented in Draft 10.0 of the IEEE Task P754. Denormalized values are supported on all Stratus modules.

The storage for a floating-point value consists of three parts: a sign, a base-2 exponent, and a mantissa.

For a 4-byte value, the high-order bit, *s*, is the sign. The base-2 exponent, *e*, is stored in 8 bits, with a bias of 127. This makes the range of stored exponent values 0 to 255. The remaining 23 bits of storage hold *f*, the fractional part of the mantissa.

The general formula for determining the value of 4-byte floating-point storage follows.

$$(-1^{s})(1 + f) * (2^{(e-127)})$$

The preceding rule has four exceptions.

- If `e = 255` and `f = 0`, the value is positive or negative infinity, depending on the value of *s*.
- If `e = 255` and `f ^= 0`, the value is not a number.
- If `e = 0` and `f = 0`, the value is zero.
- If `e = 0` and `f ^= 0`, the value is as follows:

  $$(-1^{s})(f) * (2^{-126})$$

The last item in the preceding list is a denormalized value produced by gentle underflow. This situation occurs when a calculated value is near zero. Denormalized values are supported on all Stratus modules.

Eight-byte floating-point values also have a high-order sign bit. This is followed by an 11-bit base-2 exponent, *e*, with a bias of 1023. The range of the biased exponent value is 0 to 2047. The value *f*, the fractional part of the mantissa, is stored in the remaining 52 bits.

The general formula for determining the value of 8-byte floating-point storage follows.

$$(-1^{s})(1 + f) * (2^{(e-1023)})$$

The preceding rule has four exceptions.

- If `e = 2047` and `f = 0`, the value is positive or negative infinity, depending on the value of *s*.
- If `e = 2047` and `f ^= 0`, the value is not a number.
- If `e = 0` and `f = 0`, the value is zero.

- If e = 0 and f ^= 0, the value is as follows:

  `(-1`^{s}`)(`f`) * (2`$^{-1022}$`)`

The last item in the preceding list is a denormalized value produced by gentle underflow. This situation occurs when a calculated value is near zero. Denormalized values are supported on all Stratus modules.

Character-String Data

Each character in a character string is stored as a 1-byte unsigned integer. Table B-3 lists the storage requirements for different types of character strings and the alignment for each. The mapping between characters and integers is according to the ASCII collating sequence as documented in Appendix D.

Table B-3. Storage Sizes and Alignment of Character Data

Base	Shortmap Size in Bytes	Longmap Size in Bytes	Shortmap Alignment	Longmap Alignment
`char(`*n*`)`	*n*	*n*	Byte	Byte
`char(`*n*`) varying`	`2 * ceil(`*n*`/2) + 2`	`2 * ceil(`*n*`/2) + 2`	mod2	mod2
`char(`*n*`) aligned`	`2 * ceil(`*n*`/2)`	`4 * ceil(`*n*`/4)`	mod2	mod4

This section discusses the following topics.

- "Nonvarying Character Strings"
- "Varying-Length Character Strings"

Nonvarying Character Strings

Unaligned nonvarying character strings begin on byte boundaries in storage. An unaligned nonvarying character string of length *n* is always stored in exactly *n* bytes.

Under shortmap alignment rules, aligned character strings are aligned on a mod2 boundary. An aligned nonvarying character string of length *n* is stored in an even number of bytes: `2 * ceil(`*n*`/2).`

Under longmap alignment rules, aligned character strings begin on a mod4 boundary. The storage for an aligned nonvarying character string of length *n* is calculated as `4 * ceil(`*n*`/4).`

Varying-Length Character Strings

A varying-length character string with a maximum length of *n* is stored as a 2-byte integer followed by *n* bytes of ASCII data. The integer contains the current length of the string value.

A varying-length character string of length *n* requires at least *n* `+ 2` bytes of storage. This provides space for *n* ASCII bytes and the 2-byte integer length.

Under both longmap and shortmap alignment rules, varying-length character strings are always aligned on mod2 boundaries and require an even number of storage bytes: `n + 2` or `n + 3`, depending on whether *n* is even or odd.

Bit-String Data

Unaligned bit-string data is stored without regard to byte boundaries. An unaligned bit string of length *n* requires *n* bits of storage.

Aligned bit strings begin on a mod2 boundary under shortmap alignment rules, or a mod4 boundary under longmap alignment rules. An aligned bit string of length *n* requires the following storage.

For shortmap: `2 * ceil(n/16)`

For longmap: `4 * ceil(n/32)`

That is, the storage required for an aligned bit string is the smallest multiple of two (for shortmap alignment) or four (for longmap alignment) bytes that contains at least *n* bits.

Pictured Data

A pictured value is stored as an unaligned nonvarying character string. The length of the string is the number of characters in the associated picture, excluding any `v` characters.

Pointer Data

A pointer value is stored as a 4-byte integer storage address.

Under shortmap alignment rules, pointers are aligned on a mod2 boundary; under longmap alignment rules, they begin on a mod4 boundary.

Label Data

A label value is stored in eight bytes and consists of two 4-byte integer addresses: a statement address and a stack frame address.

Under shortmap alignment rules, label values are aligned on a mod2 boundary; under longmap alignment rules, they begin on a mod8 boundary.

Entry Data

An entry value is stored in 12 bytes and consists of the following three 4-byte integer addresses.

- The *display pointer* is the address of the stack frame from which the procedure inherits automatic storage when it is called.
- The *code pointer* is the address of the code that represents the `entry` or `procedure` statement associated with the entry point.

- The *static pointer* is the biased address of the location of the static storage for the program.

Under shortmap alignment rules, entry values are aligned on a mod2 boundary; under longmap alignment rules, they begin on a mod4 boundary.

File Data

Every file constant has an associated 440-byte file control block. A file value is a pointer to such a file control block.

Under shortmap alignment rules, file values are aligned on a mod2 boundary; under longmap alignment rules, they begin on a mod4 boundary.

Arrays

Array elements are stored consecutively in row-major order. The size of an array is the sum of the sizes of its elements.

If the elements of an array are unaligned nonvarying character strings or pictured values, each element is aligned on the next available byte boundary; the array contains no extra bytes.

The elements of an array of unaligned bit strings are stored in contiguous bits. No consideration is given to byte boundaries.

In an array of structures, if the size of each array element is not a multiple of its required alignment, the compiler adds enough bytes of padding to start subsequent structures in the array on the correct boundary. Consider the following declaration.

```
declare 1 struct(2),
          2 a         fixed bin(15),
          2 b         char(1);
```

The following table illustrates how the array of structures is stored.

Element	**Alignment**	**Byte Offset**	**Size (in Bytes)**
`struct`	mod2	0	8 bytes
`struct(1)`	mod2	0	4 bytes
`struct(1).a`	mod2	0	2 bytes
`struct(1).b`	Byte	2	1 byte
Alignment padding	N/A	3	1 byte
`struct(2)`	mod2	4	4 bytes
`struct(2).a`	mod2	4	2 bytes
`struct(2).b`	Byte	6	1 byte

Element	Alignment	Byte Offset	Size (in Bytes)
Alignment padding	N/A	7	1 byte

See Chapter 4 for more information on alignment requirements.

Structures

The immediate members of a structure are stored in the order in which they are specified in the structure declaration.

For example, the following declaration shows a shortmapped structure named `struct`.

```
declare   1  struct       shortmap,
            2  a          char(1),
            2  b          fixed bin(31),
            2  c          bit(3),
            2  d          bit(2),
            2  e          fixed bin(15);
```

If you use shortmap alignment rules, the preceding structure is aligned on a mod2 boundary. The members are stored as shown in the following table.

Element	Alignment	Byte Offset	Size (in Bytes)
`struct`	mod2	0	10 bytes
`struct.a`	Byte	0	1 byte
Alignment padding	N/A	1	1 byte
`struct.b`	mod2	2	4 bytes
`struct.c`	Bit	6	3 bits
`struct.d`	Bit	6 bytes and 3 bits	2 bits
Alignment padding	N/A	6 bytes and 5 bits	11 bits
`struct.e`	mod2	8	2 bytes

The size of the entire structure `struct` is 10 bytes.

Note that you can conserve storage space by juxtaposing all unaligned bit-string data within a structure.

Consider the same structure aligned using longmap rules.

```
declare   1  struct      longmap,
            2  a         char(1),
            2  b         fixed bin(31),
            2  c         bit(3),
            2  d         bit(2),
            2  e         fixed bin(15);
```

If you use longmap alignment rules, the preceding structure is aligned on a mod4 boundary. The members are stored as shown in the following table.

Element	**Alignment**	**Byte Offset**	**Size (in Bytes)**
`struct`	mod4	0	12 bytes
`struct.a`	Byte	0	1 byte
Alignment padding	N/A	1	3 bytes
`struct.b`	mod4	4	4 bytes
`struct.c`	Bit	8	3 bits
`struct.d`	Bit	8 bytes and 3 bits	2 bits
Alignment padding	N/A	8 bytes and 5 bits	11 bits
`struct.e`	mod2	10	2 bytes

The size of the entire structure is 12 bytes.

In the following example, the nested structure `flags` is aligned on a mod2 boundary because it contains an aligned bit string.

```
declare   1  struct2     shortmap,
            2  a         char(1),
            2  b         fixed bin(31),
            2  flags ,
              3 x        bit(3) aligned,
              3 y        bit(2),
              3 unused   bit(3),
            2 c          fixed bin(15);
```

The full structure is stored as shown in the following table.

Element	**Alignment**	**Byte Offset**	**Size (in Bytes)**
`struct2`	mod2	0	†
`struct2.a`	Byte	0	1 byte
Alignment padding	N/A	1	1 byte
`struct2.b`	mod2	2	4 bytes
`struct2.flags`	mod2	6	‡
`struct2.flags.x`	mod2	6	2 bytes
`struct2.flags.y`	Bit	8	2 bits
`struct2.flags.unused`	Bit	8 bytes and 2 bits	3 bits
Alignment padding	N/A	8 bytes and 5 bits	11 bits
`struct2.c`	mod2	10	2 bytes

† The size of the entire structure `struct2` is 12 bytes.
‡ The size of the entire structure `flags` is 21 bits (2 bytes and 5 bits).

If `flags` contained all unaligned bit-string data, it would not need to be mod2-aligned; `struct2.flags.y` would have an offset of 6 bytes and 3 bits, meaning that all of the bit strings would be packed together.

A structure is aligned according to the requirements of the largest of its members, and each member is aligned according to its alignment requirements. In the following example, the members of the nested structure `flags` are aligned on a mod4 boundary because the structure contains an aligned bit string.

```
declare   1  struct2     longmap,
            2  a         char(1),
            2  b         fixed bin(31),
            2  flags ,
              3 x        bit(3) aligned,
              3 y        bit(2),
              3 unused   bit(3),
            2 c          fixed bin(15);
```

The full structure is stored as shown in the following table.

Element	Alignment	Byte Offset	Size (in Bytes)
`struct2`	mod4	0	†
`struct2.a`	Byte	0	1 byte
Alignment padding	N/A	1	3 bytes
`struct2.b`	mod4	4	4 bytes
`struct2.flags`	mod4	8	‡
`struct2.flags.x`	mod4	8	4 bytes
`struct2.flags.y`	Bit	12	2 bits
`struct2.flags.unused`	Bit	12 bytes and 2 bits	3 bits
Alignment padding	N/A	12 bytes and 5 bits	11 bits
`struct2.c`	mod2	14	2 bytes

† The size of the entire structure `struct2` is 16 bytes.
‡ The size of the entire structure `flags` is 37 bits (4 bytes and 5 bits).

If `flags` contained all unaligned bit-string data, it would not need to be mod4-aligned; `struct2.flags.y` would have an offset of 8 bytes and 3 bits, meaning that all of the bit strings could be packed together.

Storage Examples

The following tables demonstrate how values are stored for different data types.

Table B-4 shows how sample values are stored for common arithmetic data types.

Table B-4. Examples of Arithmetic Storage

Data Type	Value	Storage	Description
`fixed bin(15)`	`12` `-12`	`000Cx` `FFF4x`	Stored as a 15-bit value with high-order sign bit
`fixed bin(31)`	`20` `-20`	`00000014x` `FFFFFFECx`	Stored as a 31-bit value with high-order sign bit
`fixed dec(9,2)`	`12.45`	`04DDx`	The integer 1245 stored in 2 bytes
`fixed dec(18,1)`	`2.0`	`00000014x`	The integer 20 stored in 4 bytes
`float bin(24)`	`1.0E+10`	`501502F9x`	Stored in 4-byte IEEE format
`float bin(53)`	`1.0E+10`	`4202A05F 20000000x`	Stored in 8-byte IEEE format
`float dec(7)`	`1.0E+10`	`501502F9x`	Stored in 4-byte IEEE format
`float dec(15)`	`1.0E+10`	`4202A05F 20000000x`	Stored in 8-byte IEEE format

Table B-5 shows how sample values are stored in string and pictured data types. Note that the table shows only **shortmapped** examples.

Table B-5. Examples of Character-String, Bit-String, and Pictured Storage

Data Type	Value	Storage	Description
`char(5)`	`'abcde'`	`6162636465x`	Stored in 5 bytes as 5 ASCII characters
`char(5) aligned`	`'abcde'`	`6162636465..x`	Stored in 6 bytes as 5 ASCII characters
`char(5) varying`	`'abc'`	`0003616263....x`	Stored as a 2-byte integer followed by up to 5 bytes of ASCII characters
`bit(3)`	`'010'b`	`0002x`	Stored in 3 bits: `010`
`bit(3) aligned`	`'010'b`	`4000x`	Stored in 2 bytes: `'01000000 00000000'b`†
`picture '$zzv.99cr'`	`9.00`	`2420392E 30302020x`	Stored as the 8-byte ASCII character string `'$ 9.00  '`

† In this case, only the contents of the first three bits are guaranteed.

Table B-6 shows examples of how pointer, label, entry, and file values are stored.

Table B-6. Examples of Pointer, Label, Entry, and File Storage

Data Type	Value	Storage	Description
`pointer`	`null`	`00000001x`	4-byte integer address
`label`	N/A	`00E00024` `00FD6F10x`	Statement address and stack frame address

`entry variable`	N/A	`00000001`
		`00E00016`
		`00E0B000x`
			Display pointer, code pointer, and static pointer

`file`	N/A	`00E04000x`	Pointer to a 440-byte file control block

Appendix C:
Nonstandard OpenVOS PL/I Features

This appendix summarizes those features of OpenVOS PL/I that are not part of the ANSI Programming Language PL/I General-Purpose Subset standard (ANSI X3.74-1981), also known as PL/I Subset G.

This appendix discusses the following topics related to nonstandard OpenVOS PL/I features.

- "Features from Full PL/I"
- "Unique OpenVOS PL/I Extensions"
- "Implementation-Defined Features"

Features from Full PL/I

This section describes the OpenVOS PL/I features that are available in full PL/I but are not part of the PL/I Subset G standard. It discusses the following topics.

- "Embedded Comments"
- "Implicit Declarations"
- "Negative Scaling Factors"
- "Additional Picture Characters"
- "Secondary Entry Points"
- "Asterisks in Array References"
- "Additional Attributes"
- "Exception Handling"
- "Extended Syntax of the `do` Statement"

Embedded Comments

The OpenVOS PL/I compiler allows you use a slant-asterisk combination (`/*`) within a comment.

Implicit Declarations

If you reference an undeclared name, the OpenVOS PL/I compiler assumes that it is a variable with the attributes `fixed bin(15) automatic` and issues a severity-1 error message. See the *VOS PL/I User's Guide* (R145) for a description of compiler-error severity levels.

Negative Scaling Factors

PL/I Subset G does not support negative scaling factors for fixed-point decimal data. OpenVOS PL/I allows a scaling factor, *q*, in the following range.

```
-18 <= q <= 18
```

Additional Picture Characters

The following picture characters do not appear in PL/I Subset G but are recognized by OpenVOS PL/I: `t`, `i`, `r`, and `y`.

Secondary Entry Points

The OpenVOS PL/I compiler recognizes the `entry` statement and allows for procedures with more than one entry point.

Asterisks in Array References

The OpenVOS PL/I compiler allows you to use asterisk subscripts when referencing an entire array.

Note: OpenVOS PL/I does not support cross-section references.

Additional Attributes

The OpenVOS PL/I compiler allows you to use the `condition` and `like` attributes.

In OpenVOS PL/I, the `dimension` attribute is a keyword.

Exception Handling

OpenVOS PL/I allows you to do the following:

- declare, write on-units for, and signal programmer-defined conditions
- specify `system` as an on-unit
- return to the point where an `underflow` condition was signaled and continue execution with the value zero

Extended Syntax of the `do` Statement

In OpenVOS PL/I, the index variable of the `do` statement is not restricted to an integer or pointer variable.

OpenVOS PL/I does not restrict the finish and increment values of the `do` statement to integers.

Unique OpenVOS PL/I Extensions

This section describes the OpenVOS PL/I extensions that do not appear in either PL/I Subset G or full PL/I. It discusses the following topics.

- "Additional Characters"
- "Preprocessor Statements"
- "Infinite Values"
- "Pointer Values"
- "Variable Attributes"
- "Default Labels"

- "Stream I/O Extensions"
- "Exception Handling"
- "Built-In Functions"

Additional Characters

OpenVOS PL/I allows the use of the dollar-sign character (`$`) in program object names.

OpenVOS PL/I allows you to use the exclamation-point character (`!`) instead of the vertical-line character (|) as a bit-string operator and double exclamation points (`!!`) instead of the double vertical lines (||) as the concatenate operator.

Preprocessor Statements

The OpenVOS PL/I compiler recognizes the nonstandard preprocessor statements shown in the following table.

PL/I Preprocessor Statements	OpenVOS Preprocessor Statements
`%` (null)	`$define`
`%do`	`$else`
`%else`	`$elseif`
`%end`	`$endif`
`%if`	`$if`
`%list`	`$undefine`
`%nolist`	
`%options`	
`%page`	
`%replace`	
`%then`	

Infinite Values

A floating-point overflow or division by zero can produce an infinite floating-point value. However, subsequent operations on infinite values produce reasonable results.

Pointer Values

You can compare pointer values with any of the relational operators.

Variable Attributes

The `in`, `inline`, `shared`, `longmap`, `shortmap`, `volatile`, and `type` attributes are nonstandard.

OpenVOS PL/I does not recognize the `environment` attribute.

OpenVOS PL/I allows you to use the `like` attribute in the description of a parameter within an entry declaration and in parameter declarations. Furthermore, the structure referenced within a `like` attribute can itself be declared with `like`.

Default Labels

OpenVOS PL/I accepts a default label for a label array.

Stream I/O Extensions

OpenVOS PL/I allows you to use the `read` and `write` statements to operate on stream files.

OpenVOS PL/I allows you to use the `a` format without a width to read variable-length input lines from a stream file.

Unless you specify otherwise in the `title` option of the `open` statement, OpenVOS assumes that a space character is appended to the end of each line read from a file opened for stream input.

Exception Handling

OpenVOS PL/I recognizes the following nonstandard predefined conditions.

```
alarmtimer
anyother
break
cleanup
cputimer
reenter
stopprocess
warning
```

Built-In Functions

OpenVOS PL/I recognizes the nonstandard built-in functions shown in the following table.

addrel	lockingcharcode	scanne
byte	lockingshiftintroducer	search
bytesize	lockingshiftselector	shift
charcode	ltrim	singleshiftchar
charwidth	maxlength	size
collateascii	paramptr	trim
convert	pointer	unshift
datetime	rank	
entryinfo	rel	
iclen	rtrim	
	scaneq	

Note: Full PL/I includes a `pointer` built-in function that differs from the OpenVOS PL/I `pointer` built-in function.

Implementation-Defined Features

This section describes the OpenVOS implementation of certain features termed "implementation-defined" in the ANSI Programming Language PL/I standard (ANSI X3.53-1976). It discusses the following topics.

- "Collating Sequence"
- "Arithmetic Precisions"
- "Data Size and Alignment"
- "Maximum Lengths of Declared Names, Strings, and Text Lines"
- "Input and Output"
- "Exception Handling"
- "Built-In Functions"
- "Miscellaneous Implementation-Defined Features"

Collating Sequence

OpenVOS PL/I uses the 7-bit ASCII character set, implemented in an 8-bit format. The high-order bit of each character is zero.

The `collate` built-in function returns a string of 256 characters. The first 128 characters contain the ASCII character set; the remaining 128 characters contain the values `byte(128)` through `byte(255)`.

All character operations and built-in functions work on any 8-bit character value.

Arithmetic Precisions

Table C-1 lists the maximum and default precisions for each arithmetic scale and base in OpenVOS PL/I.

Table C-1. Maximum and Default Precisions for Arithmetic Data

Scale and Base	Maximum Precision	Default Precision
`fixed bin`	*`N`*	`15`
`fixed dec`	`18`	`9`
`float bin`	`53`	`24`
`float dec`	`15`	`7`

In the preceding table, the maximum precision (*`N`*) is either 31 or 63, depending on the value of the `$MAX_FIXED_BIN` PL/I preprocessor symbol; see "Fixed-Point Binary Data" in Chapter 4 for more information.

> **Note:** The scaling factor for fixed-point decimal values can range from `-18` to `18`, regardless of the number of digits; the default scaling factor is always `0`.

Data Size and Alignment

The allocation of automatic and static data that is not contained in an array or structure depends on the mapping rules that are in effect for each data item. The allocation of a structure is determined by the sizes and arrangement of its members and the mapping rules that are in effect.

See Chapter 4 for information about data alignment and mapping rules. See Appendix B for information about internal storage.

Maximum Lengths of Declared Names, Strings, and Text Lines

The maximum lengths of declared names, string values, and text lines are not defined by either of the PL/I standards.

Declared Names

The maximum length of a declared name is 32 characters.

Strings

The maximum length of a string value is 32,767 characters or bits.

The maximum length of a string-constant representation is 2048 characters, including the enclosing apostrophes and, in the case of a bit string, the following `b` character.

The maximum length of a string transmitted by a `get` or `put` statement is 256 characters or bits.

Text Lines

The maximum length of a line of source-program text is 300 characters.

Input and Output

This section describes implementation features related to I/O. It discusses the following topics.

- "The `title` Option of the `Open` Statement"
- "Ports and File Control Blocks"
- "Predefined I/O Ports"
- "Stream I/O"
- "Record I/O"

The `title` Option of the `Open` Statement

You can specify the following in the `title` option of the `open` statement.

- the path name of the file or device
- the OpenVOS file organization
- the name of a file index
- the position of an embedded index key
- duplicate keys
- the type of locking
- whether to append or truncate an existing output file
- whether to delete the file when it is closed
- the file ID, volume ID, and owner ID in a tape label
- whether to append a space to the end of each line read from a stream file
- the `dirty_input` I/O type

Ports and File Control Blocks

When an OpenVOS PL/I file control block is opened, an OpenVOS port with the same name as the file control block is attached. The only exceptions to this rule are the `sysin` and `sysprint` files, which are always associated with the `default_input` and `default_output` ports, respectively.

Predefined I/O Ports

Any OpenVOS PL/I file attached to a predefined I/O port that is itself attached to the terminal is unlike other stream files. (The predefined ports are `default_input`, `terminal_output`, `command_input`, and `default_output`.) OpenVOS PL/I uses a single current column position for all such ports, and the current column position always corresponds to the current cursor position on the terminal's screen.

- Each `put` statement transmits data to the file without waiting for the line to be filled.
- A `get` statement resets the current column position used by subsequent `put` statements.
- The page size is ignored, and the `endpage` condition does not occur.

Stream I/O

In a stream file, the default line size is the line size of the device to which the file is attached. If that device does not have a specified line size, the default is 80.

In a print file, tab stops are set every 5 columns, beginning with column 1. The default page size for a print file is 60 lines.

Record I/O

Record I/O keys are restricted to a maximum of 64 characters.

New records written to a file opened for keyed sequential access are always appended to the end of the file.

Exception Handling

By default, the `error` condition writes a message to the `terminal_output` port and puts the process at break level.

The `key` condition is signaled in the following cases.

- when a `read`, `rewrite`, or `delete` statement contains a `key` option value that does not match the key value of any record in the file
- when a `write` statement specifies a `keyfrom` value that matches the key value of an existing record

Built-In Functions

The `oncode` built-in function returns a 2-byte integer OpenVOS status code.

The `round` built-in function cannot operate on a floating-point value.

The length of the string returned by the `time` built-in function is 6. The string has the following form: *`hhmmss`*.

Miscellaneous Implementation-Defined Features

The file name you specify in an `%include` preprocessor statement must be a full or relative path name enclosed in apostrophes; you can omit the suffix `.incl.pl1`. If the file name is not a full path name, the compiler searches for the file in the directories on your include library search list.

Floating-point values are written with a two-digit exponent, unless three digits are required to hold the exponent value.

The `options(c)` attribute allows you to declare entry points for C language functions that expect argument values to be pushed on to the stack.

The `options(main)` clause of the `procedure` statement allows you to designate the main procedure of a program.

The `options(max_optimization_level)` attribute of the `procedure` statement allows you to specify a maximum optimization level for the source module.

Appendix D:
OpenVOS Internal Character Code Set

Table D-1 shows the OpenVOS internal character code set.

Table D-1. OpenVOS Internal Character Code Set *(Page 1 of 10)*

Decimal Code	Hex Code	Symbol	Name
0	00	NUL	Null
1	01	SOH	Start of Heading
2	02	STX	Start of Text
3	03	ETX	End of Text
4	04	EOT	End of Transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal Tabulation
10	0A	LF	Linefeed
11	0B	VT	Vertical Tabulation
12	0C	FF	Form Feed
13	0D	CR	Carriage Return
14	0E	SO	Shift Out
15	0F	SI	Shift In
16	10	DLE	Data Link Escape
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3

Table D-1. OpenVOS Internal Character Code Set *(Page 2 of 10)*

Decimal Code	Hex Code	Symbol	Name
20	14	DC4	Device Control 4
21	15	NAK	Negative Acknowledge
22	16	SYN	Synchronous Idle
23	17	ETB	EOT Block
24	18	CAN	Cancel
25	19	EM	End of Medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File Separator
29	1D	GS	Group Separator
30	1E	RS	Record Separator
31	1F	US	Unit Separator
32	20	SP	Space
33	21	!	Exclamation Mark
34	22	“	Quotation Marks
35	23	#	Number Sign
36	24	$	Dollar Sign
37	25	%	Percent Sign
38	26	&	Ampersand
39	27	’	Apostrophe
40	28	(	Opening Parenthesis
41	29	)	Closing Parenthesis
42	2A	*	Asterisk
43	2B	+	Plus Sign
44	2C	,	Comma
45	2D	-	Hyphen, Minus Sign
46	2E	.	Period
47	2F	/	Slant

Table D-1. OpenVOS Internal Character Code Set *(Page 3 of 10)*

Decimal Code	Hex Code	Symbol	Name
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less-Than Sign
61	3D	=	Equals Sign
62	3E	>	Greater-Than Sign
63	3F	?	Question Mark
64	40	@	Commercial “at” Sign
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K

Table D-1. OpenVOS Internal Character Code Set *(Page 4 of 10)*

Decimal Code	Hex Code	Symbol	Name
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Opening Bracket
92	5C	\	Reverse Slant
93	5D	]	Closing Bracket
94	5E	^	Circumflex
95	5F	_	Underline
96	60	`	Grave Accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g

Table D-1. OpenVOS Internal Character Code Set *(Page 5 of 10)*

Decimal Code	Hex Code	Symbol	Name
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Opening Brace
124	7C	|	Vertical Line
125	7D	}	Closing Brace
126	7E	~	Tilde
127	7F	DEL	Delete
128	80	SS1	Single-Shift 1
129	81	SS4	Single-Shift 4
130	82	SS5	Single-Shift 5
131	83	SS6	Single-Shift 6

Table D-1. OpenVOS Internal Character Code Set *(Page 6 of 10)*

Decimal Code	Hex Code	Symbol	Name
132	84	SS7	Single-Shift 7
133	85	SS8	Single-Shift 8
134	86	SS9	Single-Shift 9
135	87	SS10	Single-Shift 10
136	88	SS11	Single-Shift 11
137	89	SS12	Single-Shift 12
138	8A	SS13	Single-Shift 13
139	8B	SS14	Single-Shift 14
140	8C	SS15	Single-Shift 15
141	8D		(Not Assigned)
142	8E	SS2	Single-Shift 2
143	8F	SS3	Single-Shift 3
144	90	LSI	Locking-Shift Introducer
145	91	WPI	Word Processing Introducer
146	92	XCI	Extended-Control Introducer
147	93	BDI	Binary-Data Introducer
148	94		(Not Assigned)
149	95		(Not Assigned)
150	96		(Not Assigned)
151	97		(Not Assigned)
152	98		(Not Assigned)
153	99		(Not Assigned)
154	9A		(Not Assigned)
155	9B		(Not Assigned)
156	9C		(Not Assigned)
157	9D		(Not Assigned)
158	9E		(Not Assigned)
159	9F		(Not Assigned)

Table D-1. OpenVOS Internal Character Code Set *(Page 7 of 10)*

Decimal Code	Hex Code	Symbol	Name
160	A0	NBSP	No Break Space
161	A1	¡	Inverted Exclamation Mark
162	A2	¢	Cent Sign
163	A3	£	British Pound Sign
164	A4	¤	Currency Sign
165	A5	¥	Yen Sign
166	A6	¦	Broken Bar
167	A7	§	Paragraph Sign
168	A8	¨	Dieresis
169	A9	©	Copyright Sign
170	AA	ª	Feminine Ordinal Indicator
171	AB	«	Left-Angle Quote Mark
172	AC	¬	“Not” Sign
173	AD	-	Soft Hyphen
174	AE	®	Registered Trademark Sign
175	AF	¯	Macron
176	B0	°	Degree Sign, Ring Above
177	B1	±	Plus-Minus Sign
178	B2	^{2}	Superscript 2
179	B3	^{3}	Superscript 3
180	B4	´	Acute Accent
181	B5	µ	Micro Sign
182	B6	¶	Pilcrow Sign
183	B7	·	Middle Dot
184	B8	¸	Cedilla
185	B9	^{1}	Superscript 1
186	BA	º	Masculine Ordinal Indicator
187	BB	»	Right-Angle Quote Mark

Table D-1. OpenVOS Internal Character Code Set *(Page 8 of 10)*

Decimal Code	Hex Code	Symbol	Name
188	BC	¼	One-Quarter
189	BD	½	One-Half
190	BE	¾	Three-Quarters
191	BF	¿	Inverted Question Mark
192	C0	À	A with Grave Accent
193	C1	Á	A with Acute Accent
194	C2	Â	A with Circumflex
195	C3	Ã	A with Tilde
196	C4	Ä	A with Dieresis
197	C5	Å	A with Ring Above
198	C6	Æ	Diphthong A with E
199	C7	Ç	C with Cedilla
200	C8	È	E with Grave Accent
201	C9	É	E with Acute Accent
202	CA	Ê	E with Circumflex
203	CB	Ë	E with Dieresis
204	CC	Ì	I with Grave Accent
205	CD	Í	I with Acute Accent
206	CE	Î	I with Circumflex
207	CF	Ï	I with Dieresis
208	D0	Đ	D with Stroke
209	D1	Ñ	N with Tilde
210	D2	Ò	O with Grave Accent
211	D3	Ó	O with Acute Accent
212	D4	Ô	O with Circumflex
213	D5	Õ	O with Tilde
214	D6	Ö	O with Dieresis
215	D7	×	Multiplication Sign

Table D-1. OpenVOS Internal Character Code Set *(Page 9 of 10)*

Decimal Code	Hex Code	Symbol	Name
216	D8	Ø	O with Oblique Stroke
217	D9	Ù	U with Grave Accent
218	DA	Ú	U with Acute Accent
219	DB	Û	U with Circumflex
220	DC	Ü	U with Dieresis
221	DD	Ý	Y with Acute Accent
222	DE	Þ	Uppercase Thorn
223	DF	β	Sharp s
224	E0	à	a with Grave Accent
225	E1	á	a with Acute Accent
226	E2	â	a with Circumflex
227	E3	ã	a with Tilde
228	E4	ä	a with Dieresis
229	E5	å	a with Ring Above
230	E6	æ	Diphthong a with e
231	E7	ç	c with Cedilla
232	E8	è	e with Grave Accent
233	E9	é	e with Acute Accent
234	EA	ê	e with Circumflex
235	EB	ë	e with Dieresis
236	EC	ì	i with Grave Accent
237	ED	í	i with Acute Accent
238	EE	î	i with Circumflex
239	EF	ï	i with Dieresis
240	F0	ð	Lowercase Eth
241	F1	ñ	n with Tilde
242	F2	ò	o with Grave Accent
243	F3	ó	o with Acute Accent

Table D-1. OpenVOS Internal Character Code Set *(Page 10 of 10)*

Decimal Code	Hex Code	Symbol	Name
244	F4	ô	o with Circumflex
245	F5	õ	o with Tilde
246	F6	ö	o with Dieresis
247	F7	÷	Division Sign
248	F8	ø	o with Oblique Stroke
249	F9	ù	u with Grave Accent
250	FA	ú	u with Acute Accent
251	FB	û	u with Circumflex
252	FC	ü	u with Dieresis
253	FD	ý	y with Acute Accent
254	FE	þ	Lowercase Thorn
255	FF	ÿ	y with Dieresis

Index

C

D

E

P

S