PDP-11 FORTRAN-77
Language Reference Manual

Order Number: AA-V193B-TK

August 1988

This document describes the syntax and semantics of the FORTRAN-77 implementation of PDP-11 FORTRAN. It does not, however, present information specific to any operating system.

Revision/Update Information:	This revised document supersedes PDP-11 FORTRAN-77 Language Reference Manual, AA-V193A-TK.
Operating System and Version:	RSX-11M Version 4.3 RSX-11M/M-PLUS Version 4.1 RSTS/E Version 9.6 VAX/VMS Version 4.7
Software Version:	FORTRAN-77 Version 5.3

digital equipment corporation
maynard, massachusetts

First Printing, July 1983
Revised, August 1988

The postpaid Reader's Comments forms at the end of this document request the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital™

ZK4339

Contents

EXAMPLES

FIGURES

TABLES

Preface

Manual Objectives

This manual describes the elements of PDP–11 FORTRAN–77 and serves as the PDP–11 FORTRAN–77 language reference manual for several operating systems that run on the PDP–11 family of computers. No information specific to any operating system is presented here. For information on a particular operating system, refer to the user's guide for that system or the *PDP–11 FORTRAN–77 User's Guide*.

Intended Audience

Readers who have a basic understanding of the FORTRAN programming language will derive maximum benefit from this manual.

Structure of this Document

This manual presents the information in ten chapters and three appendixes, as follows:

- Chapter 1 contains general information about how PDP–11 FORTRAN–77 adheres to standards and provides extensions to those standards. It also discusses how to write a PDP–11 FORTRAN–77 program.
- Chapter 2 describes the data types, data items, and expressions that can be used in PDP–11 FORTRAN–77 programs.

- Chapter 3 describes the assignment statement, which defines the values of data items.
- Chapter 4 describes specification statements, which are nonexecutable statements. Specification statements allocate and initialize data items and define various characteristics of symbolic names used in a program.
- Chapter 5 describes control statements, which specify when and where control transfers from one point in a program to another.
- Chapter 6 discusses subprograms (subroutines and functions), both those written by users and those supplied by PDP–11 FORTRAN–77.
- Chapter 7 describes I/O (input/output) statements, which physically transfer data, both internally within memory and to and from output storage devices.
- Chapter 8 describes formatting statements, which are used together with formatted I/O statements.
- Chapter 9 describes auxiliary I/O statements, which manage files.
- Appendix A describes some statements and language features that support programs written in older versions of FORTRAN.
- Appendix B summarizes the character sets supported by PDP–11 FORTRAN–77.
- Appendix C summarizes PDP–11 FORTRAN–77 features: operators used in expressions, statements, intrinsic functions and their arguments, and system subroutines and bit manipulation functions.

Associated Documents

The following documents are of interest to PDP–11 FORTRAN–77 programmers:

- *PDP–11 FORTRAN–77 User's Guide*
- *PDP–11 FORTRAN–77 Object Time System Reference Manual*
- *PDP–11 FORTRAN–77 Installation Guide/Release Notes*

Conventions Used in this Document

The following syntactic conventions are used in this manual:

- All references to FORTRAN-77 denote PDP-11 FORTRAN-77, unless otherwise specified.
- Uppercase type is used in text to indicate system commands and command options.
- Lowercase letters are used in syntax specifications and examples to indicate variables; anything that is not a variable (for example, statement names and keywords) appears in uppercase.
- Brackets ([]) indicate optional elements within statements.
- Braces ({}) are used to enclose lists from which one element is to be chosen.
- Horizontal ellipses (. . .) indicate that the preceding item(s) can be repeated one or more times.
- "Real" (lowercase) is used to refer to the REAL*4 (REAL), REAL*8 data types as a group; likewise, "complex" (lowercase) is used to refer to COMPLEX*8; "logical" (lowercase) is used to refer to the LOGICAL*2 and LOGICAL*4 data types as a group; and "integer" (lowercase) is used to refer to the INTEGER*2 and INTEGER*4 data types as a group.
- Extensions to the FORTRAN-77 standard are printed in blue.

In addition, the following notations denote special nonprinting characters:

Tab character	<TAB>
Space character	Δ

Chapter 1

Introduction to PDP-11 FORTRAN-77

1.1 Language Overview

The PDP-11 FORTRAN-77 language comprises the American National Standard FORTRAN-77 subset language (ANSI X3.9-1978), DIGITAL-supplied enhancements to the FORTRAN-77 subset standard, and certain features of full-language FORTRAN as defined by the ANSI Standard. For information on how to obtain a copy of the ANSI standard, write to the American National Standards Institute, Inc., 1430 Broadway, New York, New York 10018.

The DIGITAL-supplied enhancements to the FORTRAN-77 subset standard follow:

- You can use any arithmetic expression as an array subscript. If the expression is not an integer type, it is converted to integer type.
- Mixed-mode expressions can contain elements of any data type except character.
- The LOGICAL*1 and LOGICAL*2 data types have been added.
- The IMPLICIT statement redefines the implied data type of symbolic names.
- The following input/output (I/O) statements have been added:

ACCEPT TYPE PRINT	Device-oriented I/O
READ (u'r) WRITE (u'r) FIND (u'r)	Unformatted direct-access I/O
READ (u'r,fmt) WRITE (u'r,fmt)	Formatted direct-access I/O
DEFINE FILE	File control and attribute specification
ENCODE DECODE	Formatted data conversion in memory
READ (u,f,key) READ (u,key)	Indexed I/O
REWRITE DELETE UNLOCK	Record control and update

- You can include an explanatory comment on the same line as any statement. These comments begin with an exclamation point (!).
- You can include debugging statements in a program by placing the letter D in column 1. These statements are compiled only when you specify a compiler command qualifier; otherwise, they are treated as comments.
- You can use any arithmetic expression as the control parameter in the computed GO TO statement.
- Virtual arrays provide large data areas outside of normal program address space.
- You can include the specification ERR=s in any OPEN, CLOSE, FIND, DELETE, UNLOCK, BACKSPACE, REWIND, or ENDFILE statement to transfer control to the statement specified by s when an error condition occurs.
- The INCLUDE statement incorporates FORTRAN statements from a separate file into a FORTRAN program during compilation.
- The INTEGER*4 data type provides a sign bit and 31 data bits.
- You can use octal and hexadecimal constants in place of any numeric constants.
- You can use character substrings and all the character intrinsic functions defined in the full language except CHAR.

In addition, PDP–11 FORTRAN–77 includes the following features of full-language FORTRAN as defined by the ANSI Standard:

- Double-precision and complex data types
- Function subprograms, including LEN, ICHAR, and INDEX
- Exponentiation forms, including double-precision
- Format edit descriptors, including S,SP,SS,T,TL, and TR
- Generic function selection based on argument data type for FORTRAN-defined functions
- Use of a real or double-precision variable as a DO statement control variable
- Use of any arithmetic expression as the initial value, increment, or final value in a DO statement
- CLOSE and OPEN statements
- Use of the specification ERR=s in READ or WRITE statements to transfer control to the statement specified by s when an error occurs
- Use of list-directed I/O to perform formatted I/O without a format specification
- Use of constants and expressions in the I/O lists of WRITE, REWRITE, TYPE, and PRINT statements
- Specification of lower bounds for array dimensions in array declarators
- Use of ENTRY statements in SUBROUTINE and FUNCTION subprograms to define multiple entry points
- Use of PARAMETER statements to assign symbolic names to constant values

1.2 Program Elements

All FORTRAN programs consist of statements and optional comments. The statements are organized into program units. A program unit is a sequence of statements that defines a computing procedure and terminates with an END statement. A program unit can be either a main program or a subprogram. An executable program consists of one main program and one or more optional subprograms.

1.2.1 Statements

Statements are grouped into two general classes: executable and nonexecutable. Executable statements specify the actions of a program; nonexecutable statements describe data arrangement and characteristics, and provide editing and data-conversion information.

Statements are divided into physical sections called lines. A line is a string of up to 80 characters. If a statement is too long to fit on one line, it can be continued on one or more additional lines, called continuation lines. A continuation line is identified by a continuation character in the sixth column of that line. (For further information on continuation characters, see Section 1.3.4.)

You can identify a statement with a label to enable other statements to reference it: that is, either to transfer control to it or to obtain the information it contains. A statement label is an integer number placed in the first five columns of a statement's initial line. Any statement can have a label; however, only executable and FORMAT statements can be referenced with a label.

1.2.2 Comments

Comments do not affect program processing in any way; they are merely a documentation aid. You can, and are encouraged to, use comments freely to describe the actions of a program, to identify program sections and processes, and to facilitate the reading of source-program listings. The letter C or an asterisk (*) in the first column of a source line identifies that line as a comment. In addition, if you place an exclamation point (!) in any column of a line except column 6, the rest of that line is treated as a comment. (However, if you place an exclamation point in column 6 of a line, that line will be treated as a continuation line.)

Any printable character can appear in a comment.

1.2.3 Character Set

The PDP-11 FORTRAN-77 character set consists of:

1. All uppercase and lowercase letters (A through Z, a through z)
2. The numerals 0 through 9
3. The special characters listed below:

Character	Name
△ or TAB	Space or tab
=	Equal sign
+	Plus sign
–	Minus sign
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Period
'	Apostrophe
"	Quotation mark
$	Dollar sign
!	Exclamation point
:	Colon
<	Left angle bracket
>	Right angle bracket

Other printable ASCII characters can appear in a FORTRAN statement only as part of a character or Hollerith constant (see Appendix B for a list of printable characters).

Except in character and Hollerith constants, the compiler makes no distinction between uppercase and lowercase letters.

1.3 Formatting a Fortran Line

Every FORTRAN line has four fields:

- A statement label field
- A continuation indicator field
- A statement field
- A sequence number field

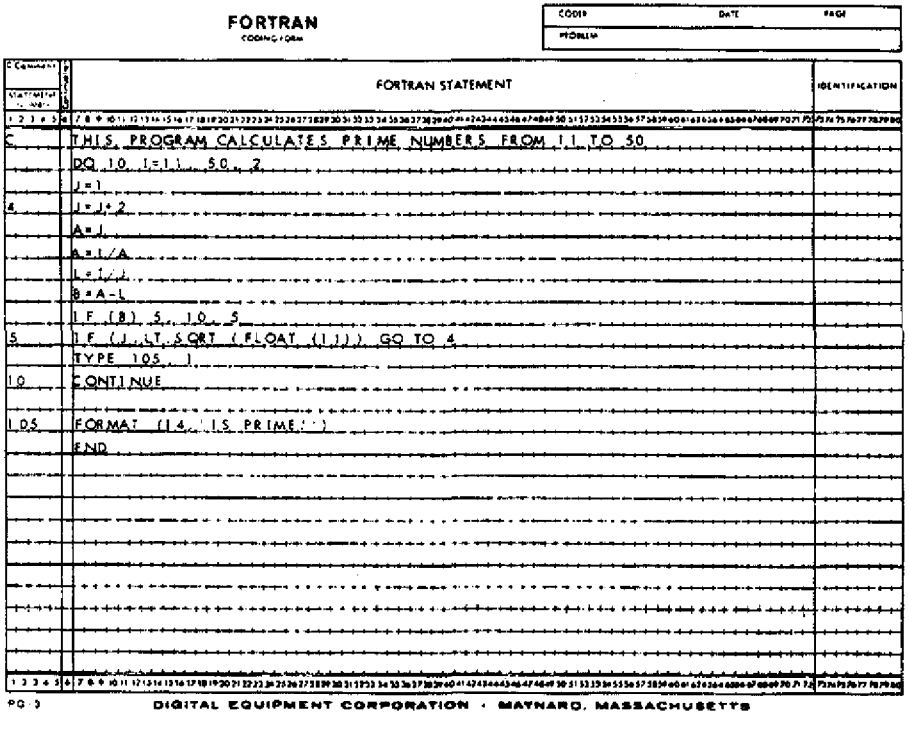
You can format a FORTRAN line in two ways: 1) by typing one character per column (character-per-column formatting); or 2) by using, in conjunction with character-per-column formatting, the tab character (tab-character formatting) to get from field to field. You can use character-per-column formatting when punching cards, writing on a coding form, or typing on a terminal keyboard; you can use tab-character formatting, however, only when you are typing at a terminal keyboard.

1.3.1 Character-per-Column Formatting

As shown in Figure 1–1, a FORTRAN line is divided into four separate fields: a statement label, a continuation indicator, statement text, and a sequence number. (Sections 1.3.3 through 1.3.6 describe the use of each field.)

Each column represents a single space, into which can be placed a single character. To get from one field to another, you type each space individually until you arrive at the correct position. For example, in Figure 1–1, to enter the comment, after typing 'C' you press the space bar five times and then begin typing the comment.

Figure 1–1: FORTRAN Coding Form


FORTRAN
CODING FORM
PROGRAMMER
DATE
PAGE
PROBLEM
C Comment
Statement Number
Continuation
FORTRAN STATEMENT
IDENTIFICATION
C THIS PROGRAM CALCULATES PRIME NUMBERS FROM 1 TO 50
DO 10 I=1, 50, 2
J=1
4 J=J+2
A=J
A=I/A
L=I/J
B=A-L
IF (B) 5, 10, 5
5 IF (J .LT. SQRT (FLOAT (I))) GO TO 4
TYPE 105, I
10 CONTINUE
105 FORMAT (I4, ' IS PRIME')
END
PG 3
DIGITAL EQUIPMENT CORPORATION · MAYNARD, MASSACHUSETTS

ZK-203-81

Field	Column(s)
Statement label	1 through 5
Continuation indicator	6
Statement	7 through 72
Sequence number	73 through 80

1.3.2 Tab-Character Formatting

You can press the tab character to move to the continuation-indicator field from the statement label field, or to the statement field from the continuation-character field; however, you cannot move to the sequence-number field by using the tab. Figure 1–2 compares keystrokes in lines typed with tab-character formatting with those in lines typed with character-per-column formatting.

Figure 1–2: Line Formatting Example

Format Using TAB Character

C TAB FIRST VALUE

10 TAB I = J + 5*K +

TAB 1 L*M

TAB IVAL = I+2

Character-per-Column Format

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C						F	I	R	S	T		V	A	L	U	E			
1	0					I		=		J		+		5	*	K		+	
					1		L	*	M										
						I	V	A	L		=		I	+	2				

ZK-204-81

In tab-character formatting, the statement-label field consists of the characters you type before you type the first tab character; however, the statement-label field cannot have more than five characters.

After you type the first tab character, you can enter either a continuation-indicator field or a statement field.

To enter a continuation-indicator field, you type any digit; the statement field then consists of all the characters you type after this digit, to the end of the line.

To enter a statement field without entering a continuation-indicator field, you simply type the statement immediately after the first tab. (No FORTRAN statement can start with a digit.)

Many text editors and terminals advance the terminal print carriage to a predefined print position when you type the TAB key. However, this action is not related to the PDP-11 FORTRAN-77 compiler's interpretation of the tab character described above.

If you use the tab character to improve the legibility of a FORTRAN statement, the spaces introduced into the statement are ignored by the compiler but are printed in a source listing. Tab characters in a statement field are ignored by the compiler as well. In a source listing, a tab causes the character following the tab to be printed at the next tab stop (which is located at columns 9, 17, 25, 33, and so forth).

1.3.3 Statement Label Field

A statement label, or number, consists of up to five decimal digits in the statement-label field of a statement's initial line. Spaces and leading 0s are ignored. (An all-zero statement label is invalid.)

Any statement referenced by another statement must have a label. No two statements within a program unit can have the same label.

You can use two special indicators in the first column of a label field: the comment indicator and the debugging-statement indicator. These indicators are described in Sections 1.3.3.1 and 1.3.3.2.

The statement label field of a continuation line must be blank.

1.3.3.1 Comment Indicators

The letter C or an asterisk (*) in column 1 of a line indicates that the entire line is a comment. An exclamation point (!) in any column of a line except column 6 indicates that the remainder of the line is a comment. All blanks indicate a blank comment.

The compiler prints a comment line in a source-program listing and then ignores it.

1.3.3.2 Debugging-Statement Indicator

The letter D in column 1 of a line designates the contents of the statement field as a debugging statement. A debugging-statement line can have a statement label in the four remaining columns of the label field. If a debugging statement is continued to one or more other lines, every continuation line must have a D in column 1 and a continuation indicator in column 6.

Debugging statements are not compiled unless you use a compiler command to specify that they be compiled. If you do not specify debugging-statement compilation, any debugging statements are treated as comments. For a description of the available compilation commands, refer to the *PDP-11 FORTRAN-77 User's Guide*.

1.3.4 Continuation Field

A continuation indicator is any character (except 0 or space) in column 6 of a line, or any digit (except 0) after the first tab.

A statement can be divided into continuation lines at any point.

The compiler considers the characters after the continuation character to follow the last character of the previous line, as if there were no physical breaks at that point. If a continuation indicator is 0, the compiler considers the line containing it to be the first line of a statement.

Comment lines cannot be continued. They can, however, occur between a statement's initial line and its continuation line or lines, and between successive continuation lines.

1.3.5 Statement Field

The text of a statement is placed in a statement field. Because the compiler ignores all tab characters and spaces in a statement field except those in Hollerith constants and alphanumeric literals, you can space the text in a statement field in any way you desire to maximize legibility. The use of tabs for spacing is discussed in Section 1.3.2.

NOTE

If a line extends beyond character position 72, the text following position 72 is ignored; no warning message is printed.

1.3.6 Sequence Number Field

A sequence number or other identifying information can appear in columns 73 through 80 of any line; the compiler ignores characters in this field.

Remember that you cannot move to the sequence-number field by tab-character formatting.

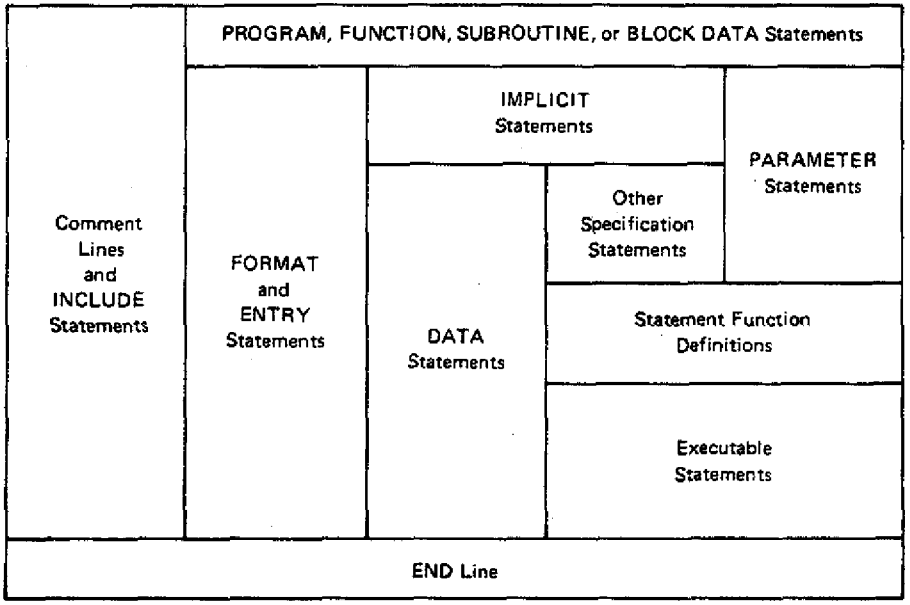
1.4 Program Unit Structure

Figure 1–3 shows the allowed order of statements in a PDP–11 FORTRAN–77 program unit. In this figure, vertical lines separate statement types that may be interspersed with one another—that is, occur in any order relative to each other. For example, comment lines and FORMAT statements may occur before, between, or after DATA statements and executable statements (see next paragraph) in the body of a program. Horizontal lines indicate statement types that cannot be interspersed but must occur in a prescribed order within a program. For example, an IMPLICIT or IMPLICIT NONE statement cannot occur before a PROGRAM statement or after an END statement.

The "executable" statements mentioned in Figure 1–3 include: assignment, ASSIGN, GOTO, arithmetic IF, logical IF, block IF, ELSE IF, ELSE, ENDIF, CONTINUE, STOP, PAUSE, DO, READ, WRITE, PRINT, TYPE, ACCEPT, FIND, DELETE, REWRITE, BACKSPACE, ENDFILE, REWIND, UNLOCK, OPEN, CLOSE, CALL, RETURN, and END.

The "specification" statements mentioned in Figure 1–3 include: DIMENSION, COMMON, EQUIVALENCE, EXTERNAL, INTRINSIC, SAVE, and type declaration.

Figure 1–3: Required Order of Statements and Lines



ZK-205-81

1.5 INCLUDE Statement

The INCLUDE statement specifies that the contents of a designated file are to be incorporated into a compilation directly following the statement. INCLUDE has no effect on program execution.

The INCLUDE statement has the form:

```
INCLUDE 'filespec[/[NO]LIST]'
```

filespec
A file specification in the form of a character constant string that represents the file to be included in a compilation. This file specification must be acceptable to the operating system. (See the *PDP-11 FORTRAN-77 User's Guide* for the form of a file specification.)

The /LIST qualifier specifies that the statements in the designated file are to be included in the compilation source listing; an asterisk (*) precedes each statement included. The /NOLIST qualifier specifies that the statements in the designated file are not to be included in the compilation source listing. The default is /LIST; that is, the compiler assumes /LIST if you do not specify either qualifier.

When the compiler encounters an INCLUDE statement, it stops reading statements from the current file and begins reading statements from the designated, or included, file. When it reaches the end of this file, the compiler reads the next statement after the INCLUDE statement.

An INCLUDE statement can be contained in an included file.

An included file cannot begin with a continuation line; each statement included must be completely contained within a single file.

The INCLUDE statement can appear anywhere that a comment line can appear.

Any PDP–11 FORTRAN–77 statement can appear in an included file; however, all the statements in an included file, when combined with the other statements in a compilation, must satisfy the requirements shown in Figure 1–3.

In the following example, the included file COMMON.FTN defines the size of the blank COMMON block and the size of the arrays X, Y, and Z.

Main Program File

```
      INCLUDE 'COMMON.FTN'
      DIMENSION Z(M)
      CALL CUBE
      DO 5, I=1,M
5     Z(I) = X(I)+SQRT(Y(I))
            .
            .
            .
      SUBROUTINE CUBE
      INCLUDE 'COMMON.FTN'
      DO 10, I=1,M
10    X(I) = Y(I)**3
      RETURN
      END
```

File COMMON.FTN

```
      PARAMETER (M = 100)
      COMMON X(M),Y(M)
```

Chapter 2

Statement Components

PDP–11 FORTRAN–77 statements are composed of five basic components:

- Constants—fixed values, such as numbers. These values cannot be changed by program statements.
- Variables—symbolic names that represent stored values. These values can be changed by program statements.
- Arrays—groups of values that are stored contiguously and can be referenced individually by a symbolic name with a subscript, or collectively by a symbolic name only. Individual values are called array elements.
- Function references—function names optionally followed by lists of arguments. A function is a program unit that performs a specified computation (for example, computing a trigonometric sine) using arguments supplied by a function reference; the resulting value is then used in place of the function reference.
- Expressions—combinations of constants, variables, array elements, function references, and operators. An operator is a unique symbol for a particular operation (such as multiplication) that obtains a single result.

Variables, arrays, and functions have symbolic names. A symbolic name is a string of characters that identifies an entity in a program.

Constants, variables, arrays, expressions, and functions can have the following data types:

- Logical
- Integer
- Real

- Double-precision
- Complex
- Character (except for functions)

Symbolic names, data types, and all the statement components except function references are discussed in this chapter; function references are discussed in Chapter 6.

2.1 Symbolic Names

Symbolic names identify the entities that can appear in a program unit. The entities that symbolic names identify are listed in Table 2–1, where the column labelled "Typed" indicates whether an entity has a data type (such as real or integer). (Data types are discussed in Section 2.2.)

Table 2–1: Entities Identified by Symbolic Names

Entity	Typed
Variables	Yes
Arrays	Yes
Statement functions	Yes
Intrinsic functions	Yes
Function subprograms	Yes
Subroutine subprograms	No
Common blocks	No
Main programs	No
Block data subprograms	No
Dummy arguments	Yes
Function entry points	Yes
Subroutine entry points	No
Parameter constants	Yes

A symbolic name is a string of characters (letters and digits) totaling a maximum of six; the first character must be a letter. If more than six characters are used, the system will automatically truncate the name to six characters during compilation.

Examples of valid and invalid symbolic names are:

Valid	Invalid	Explanation
NUMBER	5Q	Begins with a numeral
K9	B.4	Contains a special character

Symbolic names must be unique within a program unit—that is, the same symbolic name cannot be used to identify two or more entities in the same program unit.

In executable programs consisting of two or more program units, a symbolic name for any of the following entities must be unique throughout all the program units:

- Intrinsic functions
- Function subprograms
- Subroutine subprograms
- Common blocks
- Main programs
- Block data subprograms
- Function entries
- Subroutine entries

Therefore if, for example, one of your program units contains a function named UMP, you cannot use UMP as the symbolic name for any other entity anywhere else in your program, even in a completely separate program unit.

2.2 Data Types

Each basic statement component in a PDP–11 FORTRAN–77 program (constant, variable, array, function reference, or expression) has assigned to it one of six data types that specifies the kind of value it can represent. The data types and the values they represent are:

- Integer—for a whole number
- Real—for a decimal number: that is, a whole number, a decimal fraction, or a combination of a whole number and a decimal fraction

- Double precision—for a real number with more than twice as many maximum significant digits as real
- Complex—for a pair of real numbers representing a complex number: the first value representing the real part, the second representing the imaginary part
- Logical—for the value true or the value false
- Character—for a sequence of characters

The data type of a basic component can be assigned in one of three ways: it can be inherent in the component's construction (as in constants); it can be implied by a naming convention (with or without an IMPLICIT statement); or it can be explicitly declared.

Whenever a value of one data type is converted to a value of another type, the conversion is performed according to the rules for assignment statements (see Table 3–1).

For the purpose of facilitating control of processing performance and memory requirements, PDP–11 FORTRAN–77 provides several data types (or data type variations) in addition to the six basic data types listed above. These data types are included in Table 2–2, which lists all PDP–11 FORTRAN–77 data types, as well as the amount of memory each data type requires for storage.

The form *n appended to a data type name is called a data-type length specifier.

2.3 Constants

A constant represents a fixed value and can be a number, the logical values true or false, or a character string.

Octal, hexadecimal, and Hollerith constants have no data type; these constants assume the data type prescribed by the context in which they appear (see Section 2.3.8.).

Table 2–2: Data Type Storage Requirements

Data Type	Storage Requirements
BYTE	1a
LOGICAL	2 or 4b
LOGICAL*1	1a
LOGICAL*2	2
LOGICAL*4	4
INTEGER	2 or 4b
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*4	4
REAL*8	8
DOUBLE PRECISION	8
COMPLEX	8
COMPLEX*8	8
CHARACTER*len	len c

a. The 1-byte storage area can contain the logical values true or false, a single character, or integers in the range –128 to +127.

b. Either two or four bytes are allocated depending on the compiler command qualifier specified. The default allocation is two bytes. When four bytes are allocated, all four bytes are used for computation.

c. The value of len is the number of characters specified; this number can be any integer within the range 1 to 255.

BYTE and LOGICAL*1 are synonymous.

2.3.1 Integer Constants

An integer constant is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number. An integer constant has the form:

```
snn
```

s
An optional sign.

nn
A string of numeric characters.

Leading 0s, if any, are ignored.

A minus sign must appear before a negative integer constant; a plus sign is optional before a positive constant (an unsigned constant is assumed to be positive).

Except for the sign, an integer constant cannot contain a character other than the numerals 0 through 9.

The absolute value of an integer constant cannot be greater than 2147483647.

Examples of valid and invalid integer constants are:

Valid	Invalid	Explanation
0	99999999999	Too large
-127	3.14	Decimal point not allowed
+32123	32,767	Comma not allowed

If the value of a constant is within the range –32768 to +32767, this value represents a 2-byte signed quantity and is treated as an INTEGER*2 data type; if a value is outside this range, it represents a 4-byte signed quantity and is treated as an INTEGER*4 data type.

2.3.2 Real Constants

A real constant is a number with a decimal point and can occur in any one of three forms:

- As a basic real constant
- As a basic real constant followed by a decimal exponent
- As an integer constant followed by a decimal exponent

A basic real constant is a string of decimal digits in one of three formats:

```
  s.nn
snn.nn
snn.
```

s
An optional sign.

nn
A string of decimal digits.

The decimal point can appear anywhere in the string. The number of digits is not limited, but only the leftmost 7 digits are significant. Leading 0s (0s to the left of the first nonzero digit) are ignored in counting the leftmost 7 digits; therefore, in the constant 0.00001234567, all of the nonzero digits are significant, but none of the 0s is significant.

A decimal exponent has the form:

```
Esnn
```

s
An optional sign.

nn
An integer constant.

The exponent represents a power of 10 by which the preceding real or integer constant is to be multiplied; for example, 1.0E6 represents the value 1.0 * 10 ** 6.

A real constant occupies four bytes and is interpreted as a real number with a precision, typically, of seven decimal digits.

A minus sign must appear between the letter E and a negative exponent; a plus sign is optional between the letter E and a positive exponent.

Except for algebraic signs and a decimal point, and the letter E if used, a real constant cannot contain a character other than the numerals 0 through 9.

If the letter E appears in a real constant, an integer constant exponent must follow it. The exponent cannot be omitted; however, it can be 0.

The magnitude of a nonzero real constant cannot be smaller than approximately 0.29E-38 or greater than approximately 1.7E38.

Examples of valid and invalid real constants are:

Valid	Invalid	Explanation
3.14159	1,234,567	Commas not allowed
621712.	325E-45	Too small
-.00127	-47.E47	Too large
+5.0E3	100	Decimal point missing
2E-3	$25.00	Special character not allowed

2.3.3 Double-Precision Constants

A double-precision constant is a basic real constant or an integer constant followed by a decimal exponent of the form:

```
Dsnn
```

s
An optional sign.

nn
An integer constant.

A double-precision constant occupies eight bytes and is interpreted as a real number with a precision, typically, of 16 decimal digits. The number of digits that precede the exponent is not limited, but only the leftmost 16 digits are significant.

A minus sign must appear before a negative double-precision constant; a plus sign is optional before a positive constant. A minus sign must appear between the letter D and a negative exponent; a plus sign is optional between the letter D and a positive exponent.

The exponent following the letter D cannot be omitted; however, it can be 0.

The magnitude of a nonzero double-precision constant cannot be smaller than approximately 0.29D-38 or greater than approximately 1.7D38.

Examples of valid and invalid double-precision constants are:

Valid	Invalid	Explanation
1234567890D+5	1234567890D45	Too large
+2.71828182846182D00	1234567890.0D-89	Too small
-72.5D-15	+2.7182812846182	No Dsnn present
1D0		This is a valid real constant

2.3.4 Complex Constants

A complex constant is a pair of integer or real constants separated by a comma and enclosed in parentheses. The first constant represents the real part of a complex number, the second constant the imaginary part.

A complex constant has the form:

```
(rc,rc)
```

rc
A real constant.

The parentheses and comma are part of the complex constant and are required. See Section 2.3.2 for the rules for forming real constants.

A complex constant occupies eight bytes and is interpreted as a complex number.

Examples of valid and invalid complex constants are:

Valid	Invalid	Explanation
(1.70391,-1.70391)	(1.23,)	Second real constant is missing
(+12739E3,0.)	(1.0,1.0D0)	Double-precision constants are not allowed

2.3.5 Octal and Hexadecimal Constants

Octal and hexadecimal constants are alternative ways to represent numeric constants; you can use them wherever numeric constants are allowed.

An octal constant is an unsigned string of octal digits enclosed by apostrophes and followed by the alphabetic character O. An octal constant has the form:

```
'c1c2c3...cn'O
```

c
A digit in the range 0 to 7.

A hexadecimal constant is an unsigned string of hexadecimal digits enclosed by apostrophes and followed by the alphabetic character X. A hexadecimal constant has the form:

```
'c1c2c3...cn'X
```

c
A hexadecimal digit in the range 0 to 9, or a letter in the range A to F or a to f.

Leading zeros are ignored in octal and hexadecimal constants. You can specify up to 32 bits (11 octal digits, 8 hexadecimal digits).

Examples of valid and invalid octal constants are:

Valid	Invalid	Explanation
07737'O	'7782'O	Invalid character
'1'O	7772'O	No initial apostrophe
	'0737'	No O after second apostrophe
	'-4367'	Signed

Examples of valid and invalid hexadecimal constants are:

Valid	Invalid	Explanation
'AF9730'X	'999.'X	Invalid character
'FFABC'X	'F9X	No apostrophe before the X
	'-ACF4'	Signed

Octal and hexadecimal constants are typeless numeric constants; they assume data types that are based on the way they are used (and thus they are not converted before use), as follows:

- When the constant is used with a binary operator, including the assignment operator, the data type of the constant is the data type of the other operand. For example:

Statement	Data Type of Constant	Length of Constant
REAL*8 DOUBLE		
INTEGER*4 N		
RAPHA = '99AF2'X	REAL*4	4
JCOUNT = ICOUNT + '777'O	INTEGER*2	2
DOUBLE = 'FFF99A'X	REAL*8	8
IF(N.EQ.'123'O) GO TO 10	INTEGER*4	4

- When a specific data type—generally integer—is required, this type is assumed for the constant. For example:

Statement	Data Type of Constant	Length of Constant
Y(IX)=Y('15'O)+3.	INTEGER*2	2

- When the constant is used as an actual argument, no data type is assumed; however, a length of two bytes is always used. For example:

Statement	Data Type of Constant	Length of Constant
CALL APAC('34BC'x)	None	2

- When the constant is used in any other context, INTEGER*2 data type is assumed. For example:

Statement	Data Type of Constant	Length of Constant
IF('AF77'X) 1,2,3	INTEGER*2	2
I = '7777'O – 'A39'X	INTEGER*2	2
J = .NOT.'73777'O	INTEGER*2	2

An octal or hexadecimal constant actually specifies as much as 4 bytes of data. When the data type implies that the length of the constant is more

than the number of digits specified, the leftmost digits have a value of zero. When the data type implies that the length of the constant is less than the number of digits specified, the constant is truncated on the left. An error results if any nonzero digits are truncated. Table 2–2 lists the number of bytes that each data type requires.

2.3.6 Logical Constants

A logical constant specifies true or false; therefore, only the following two logical constants are possible:

```
.TRUE.
.FALSE.
```

The delimiting periods are a required part of each constant.

2.3.7 Character Constants

A character constant is a string of printable ASCII characters enclosed by apostrophes.

A character constant has the form:

```
'c1c2c3...cn'
```

c
A printable character.

Both delimiting apostrophes must be present.

The value of a character constant is the string of characters between the delimiting apostrophes. The value does not include the delimiting apostrophes, but does include all spaces or tabs within the apostrophes.

Within a character constant, the apostrophe character is represented by two consecutive apostrophes (with no space or other character between them).

The length of the character constant is the number of characters between the delimiting apostrophes (two consecutive internal apostrophes counting as one character). The length of a character constant must be in the range 1 through 255.

Examples of valid and invalid character constants are:

Valid	Invalid	Explanation
'WHAT?'	'HEADINGS	No trailing apostrophe
'TODAY"S DATE IS:'	''	Character constant must contain at least one character
'HE SAID, "HELLO"'	"NOW OR NEVER"	Quotation marks cannot be used in place of apostrophes

If a character constant appears in a numeric context (for example, as the expression on the right side of an arithmetic assignment statement), it is considered a Hollerith constant (see Section 2.3.8.).

2.3.8 Hollerith Constants

A Hollerith constant is a string of printable characters preceded by a character count and the letter H.

A Hollerith constant has the form:

```
nHc1c2c3...cn
```

n
An unsigned, nonzero integer constant stating the number of characters in the string (including spaces and tabs).

c
A printable character.

The maximum number of characters is 255.

Hollerith constants are stored as byte strings, one character per byte.

Hollerith constants have no data type; they assume a numeric data type according to the context in which they are used. Hollerith constants cannot assume a character data type and cannot be used where a character value is expected.

Examples of valid and invalid Hollerith constants are:

Valid	Invalid	Explanation
16HTODAY'S DATE IS: 1HB	3HABCD	Wrong number of characters

When Hollerith constants are used in numeric expressions, they assume a data type according to the following rules:

- When the constant is used with a binary operator, including the assignment operator, the data type of the constant is the data type of the other operand. For example:

Statement	Data Type of Constant	Length of Constant
INTEGER*2 ICOUNT REAL*8 DOUBLE		
RALPHA = 4HABCD	REAL*4	4
JCOUNT = ICOUNT +2HXY	INTEGER*2	2
DOUBLE = 8HABCDEFGH	REAL*8	8

- When a specific data type is required, this type is assumed for the constant. For example:

Statement	Data Type of Constant	Length of Constant
X=Y(1HA)	INTEGER*2	2

- When the constant is used as an actual argument, no data type is assumed. For example:

Statement	Data Type of Constant	Length of Constant
CALL APAC (9HABCDEFGHI)	None	9

- When the constant is used in any other context, INTEGER*2 data type is assumed. For example:

Statement	Data Type of Constant	Length of Constant
IF (2HAB) 1,2,3	INTEGER*2	2
I= 1HC - 1HA	INTEGER*2	2
J= .NOT. 1HB	INTEGER*2	2

When the length of the constant is less than the length implied by the data type, spaces are appended to the constant on the right; when the length of the constant is greater than the length implied by the data type, the constant is truncated on the right. An error results if any nonblank characters are truncated.

Table 2–2 lists the number of characters required for each data type. Each character occupies one byte of storage.

2.4 Variables

A variable is a symbolic name associated with a storage location (see Section 2.1 for the form of a symbolic name). The value of the variable is the value currently stored in that location; however, you can change that value by assigning a new value to the variable.

Like constants, variables are classified by data type. The data type of a variable indicates the type of data the variable represents, its precision, and its storage requirements. When data of any type is assigned to a variable, this data is converted, if necessary, to the data type of the variable. You can establish the data type of a variable by using type declaration statements or IMPLICIT statements, or by choosing names that begin with certain letters (I—N for integer; any other for real).

Two or more variables are associated with each other when they refer to the same memory location. They are partially associated when part of the location to which one variable refers is the same as part or all of the location to which the other variable refers. Association and partial association occur when you use COMMON statements, EQUIVALENCE statements, and actual and dummy arguments in subprogram references.

A variable is considered defined if the storage location associated with it contains data of the same type as the variable name. A variable can be defined before program execution by a DATA statement, or during execution by an assignment or input statement.

If variables of different data types are associated (or partially associated) with the same storage location, and if the value of one variable is defined (for example, by assignment), the value of the other variable becomes undefined; that is, its value cannot be predicted.

2.4.1 Data Typing by Specification

To specify the data types of variables, you use type declaration statements (see Section 5.2). For example, the statements

```
COMPLEX VAR1
DOUBLE PRECISION VAR2
```

assign the COMPLEX data type to the variable VAR1 and the DOUBLE PRECISION data type to the variable VAR2; that is, they cause the variable VAR1 to be associated with an 8-byte storage location that will contain complex data, and the variable VAR2 to be associated with an 8-byte double-precision storage location.

Character type declaration statements assign the character data type and a value length to specified variables. For example, the statements

```
CHARACTER*72 INLINE
CHARACTER NAME*12, NUMBER*9
```

cause the variables INLINE, NAME, and NUMBER to be associated with storage locations containing character data of lengths 72, 12, and 9, respectively.

The IMPLICIT statement (see Section 5.1) has a more general effect: it assigns, in the absence of an explicit type declaration, a specified data type to any variable beginning with a specified letter or any letter within a specified range.

You can explicitly declare the data type of a variable only once. An explicit declaration takes precedence over an IMPLICIT statement.

2.4.2 Data Typing by Implication

In the absence of either IMPLICIT statements or type declaration statements, all variables you use that have names beginning with I, J, K, L, M, or N are assumed to be integer variables, and those that have names beginning with any other letter are assumed to be real variables. For example:

Real Variables	Integer Variables
ALPHA	JCOUNT
BETA	ITEM
TOTAL	NTOTAL

2.5 Arrays

An array is a group of contiguous storage locations associated with a single symbolic name (the array name). The individual storage locations, called array elements, are referred to by a subscript appended to the array name. (Section 2.5.2 discusses subscripts.)

In PDP-11 FORTRAN-77, an array can have from one to seven dimensions. A single column of figures, for example, is an array having only one dimension—or a one-dimensional array; to refer to a value in this array, you need only specify the value's row number. Similarly a table of more than one column of figures is a two-dimensional array; to refer to a value in this array, you must specify both the value's row number and its column number. And a table of figures that covers several pages is a three-dimensional array; to refer to a value in this array, you must specify the value's row number, its column number, and its page number.

The following PDP-11 FORTRAN-77 statements establish arrays:

- Type declaration statements (see Section 5.2)
- The DIMENSION statement (see Section 5.3)
- The COMMON statement (see Section 5.4)
- The VIRTUAL statement (see Section 5.5)

These statements may contain array declarators (see Section 2.5.1) that define the name of the array, the number of dimensions in the array, and the number of elements in each dimension.

An element of an array is considered defined if the storage location associated with it contains data of the same type as the array name (see Section 2.5.4). An array element or an entire array can be defined before program execution by a DATA statement. An array element can be defined during program execution by an assignment or input statement; an entire array can be defined during program execution by an input statement.

2.5.1 Array Declarators

An array declarator specifies the symbolic name that is to identify an array within a program unit, and it specifies the properties of this array.

An array declarator has the form:

```
a (d[,d] ...)
```

a
The symbolic name of the array—that is, the array name. (Section 2.1 gives the form of a symbolic name.)

d
A dimension declarator.

The number of dimension declarators indicates the number of dimensions in the array; the number of dimensions can range from one to seven.

For example, in

```
DIMENSION IUNIT (10,10,10)
```

IUNIT is a three-dimensional array.

The value of a dimension declarator specifies the number of elements in that dimension: in the example above, each dimension of IUNIT consists of 10 elements.

The number of elements in an array is equal to the product of the values of the dimension declarators; IUNIT above contains 1000 elements (10 X 10 X 10).

An array name can appear in only one array declarator within a program unit.

Dimension declarators that vary in value are not permitted in a main program, but they are permitted in a subprogram in order to define adjustable arrays. You can use adjustable arrays within a single subprogram—to process arrays with different dimension declarators—by specifying the declarators as well as the array name as subprogram arguments. (See Section 6.1.2 for more information.)

A dimension declarator in PDP–11 FORTRAN–77 can specify both a lower bound and an upper bound, as follows:

```
[dl:] du
```

dl
The lower bound of the dimension.

du
The upper bound of the dimension. (Can be an asterisk (*); see below.)

The number of elements specified by a dimension with upper and lower bounds is du–dl+1.

Specifying the lower bound of an array allows you to use a range of subscripts that does not begin with 1. For example, to reference an array storing data for the years 1964 to 1974, you could specify an upper bound of 74 and a lower bound of 64 as follows:

```
DIMENSION KYEAR (64:74)
```

The value of the lower bound, dl, can be negative, 0, or positive. The value of the upper bound, du, must be greater than or equal to the corresponding lower bound. If a lower bound is not specified, it is assumed that the lower bound is 1 and that the value of the upper bound is the number of elements in the dimension.

For example, in the statement

```
DIMENSION NUM (0:9,-1:1)
```

NUM contains 30 elements.

The upper bound in the last dimension declarator in a list of dimension declarators may be an asterisk; an asterisk marks the declarator as an assumed-size array declarator (see Section 6.1.3).

Each dimension bound is an integer arithmetic expression in which:

- Each operand is an integer constant, an integer dummy argument, or an integer variable in a COMMON block

- Each operator is a +, –, *, /, or ** operator

Array references and function references are not allowed in dimension bounds expressions.

2.5.2 Subscripts

A subscript is a list of expressions, called subscript expressions, enclosed in parentheses, that specify, or reference, a particular element in an array; a subscript is said to "qualify" an array name. A subscript is appended to the array name it qualifies.

A subscript has the form:

```
(s[,s]...)
```

s
A subscript expression.

A subscript expression can be a constant, a variable, or an arithmetic expression. If the value of a subscript is not of type integer, it is converted to integer by truncating any fractional part.

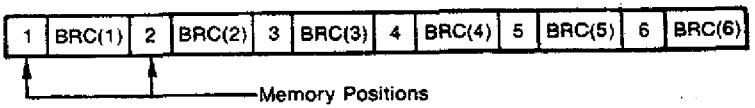
A subscripted array reference must contain one subscript expression for each dimension defined for the array being referenced (one for each dimension declarator).

2.5.3 Array Storage

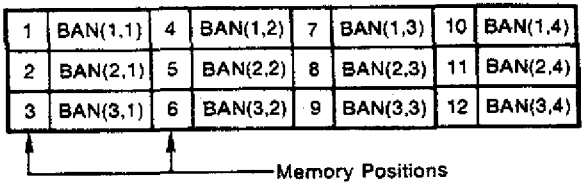
As suggested earlier in Section 2.5, you can think of an array as an arrangement of values in rows, columns, and pages (or planes)—that is, as an arrangement of values in other than a strictly linear sequence. An array of any size is always stored in memory, however, as a linear sequence of values: A one-dimensional array is stored with its first element in the first storage location, and its last element in the last storage location of the sequence; a multidimensional array is stored so that the leftmost subscripts vary most rapidly. This storage arrangement for arrays is called the "order of subscript progression." Figure 2–1 shows array storage in one, two, and three dimensions.

Figure 2–1: Array Storage

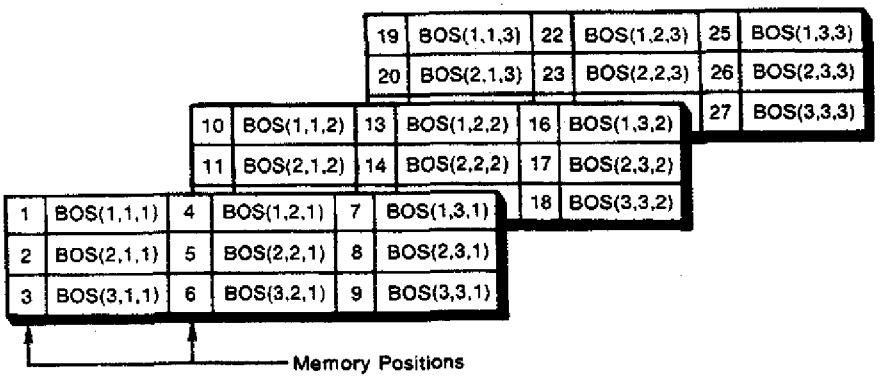
One-Dimensional Array BRC (6)



Two-Dimensional Array BAN (3,4)



Three-Dimensional Array BOS (3,3,3)



ZK-7627-HC

2.5.4 Data Type of an Array

The data type of an array is specified in the same way that the data type of any other variable is specified—that is, implicitly by the initial letter of the name, or explicitly by a type declaration statement.

All the values in an array have the same data type. Any value assigned to an array element is converted to the data type of the array. If an array is named in a DOUBLE PRECISION statement, for example, the compiler allocates an 8-byte storage location for each element of the array. When a value of any type is assigned to any element of this array, it is converted to double precision.

2.5.5 Array References Without Subscripts

In the following types of statements, you can indicate that an entire array is to be used (or defined) simply by specifying the array name without its subscript:

- Type declaration statements
- COMMON statement
- DATA statement
- EQUIVALENCE statement
- FUNCTION statement
- SUBROUTINE statement
- Input/output statements
- ENTRY statement
- SAVE statement

You can also use unsubscripted array names as actual arguments in references to external procedures. Unsubscripted array names are not permitted in any other type of statement.

2.5.6 Adjustable Arrays

Adjustable arrays allow subprograms to manipulate arrays of variable dimensions. To use an adjustable array in a subprogram, you specify the array bounds and the array name as subprogram arguments. (See Chapter 6 for more information.)

2.6 Character Substrings

A character substring is a contiguous segment of a character variable or character array element.

A character substring reference has one of the forms:

```
v([e1]:[e2])

a(s[,s]...) ([e1]:[e2])
```

v
A character variable name.

a
A character array name.

s
A subscript expression.

e1
A numeric expression that specifies the leftmost character position of a substring.

e2
A numeric expression that specifies the rightmost character position of a substring.

Character positions within a character variable or array element are numbered from left to right, beginning with 1. For example, LABEL (2:7) specifies the substring beginning with the second character position and ending with the seventh character position of the character variable LABEL. If the CHARACTER*8 variable LABEL has a value of XVERSUSY, then the substring LABEL(2:7) has a value of VERSUS.

If the value of the numeric expression e1 or e2 is not of type integer, it is converted to an integer value before use by truncating any fractional part.

The values of the numeric expressions e1 and e2 must meet the following conditions:

```
1   .LE.   e1   .LE.   e2   .LE.   len
```

len
The length of the character variable or array element.

If e1 is omitted, FORTRAN–77 assumes that e1 equals 1; if e2 is omitted, FORTRAN–77 assumes that e2 equals len.

For example, NAMES(1,3)(:7) specifies the substring starting with the first character position and ending with the seventh character position of the character array element NAMES(1,3).

2.7 Expressions

An expression consists of a single basic component (such as a constant or a variable) or a combination of basic components with one or more operators that represents a single value. Operators specify computations to be performed on the values of the basic components.

Expressions are classified as arithmetic, character, relational, or logical. Arithmetic expressions produce numeric values, character expressions produce character values, and relational and logical expressions produce logical values.

2.7.1 Arithmetic Expressions

Arithmetic expressions are expressions that are formed with arithmetic elements and arithmetic operators. The evaluation of an arithmetic expression yields a single numeric value.

An arithmetic element can be any of the following:

- A numeric, Hollerith, octal, or hexadecimal constant
- A numeric variable
- A numeric array element
- An arithmetic expression enclosed in parentheses
- An arithmetic function reference

The term "numeric," as used above, includes logical data, because logical data is treated as integer data when used in an arithmetic context.

Arithmetic operators specify a computation that is to be performed on the values of arithmetic elements to produce a numeric value as a result. The operators and their meanings are:

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition and unary plus
-	Subtraction and unary minus

These operators are called binary operators because each is used with two elements. When written immediately preceding an arithmetic element, to denote a positive or negative value, the plus (+) and minus (-) symbols are unary operators.

You can use any arithmetic operator with any valid arithmetic element, except as noted in Table 2–3.

A variable or array element must have a defined value before it can be used in an arithmetic expression.

Table 2–3 shows the allowed combinations of base and exponent data types, and also shows the data type of the result of exponentiation.

Table 2–3: Exponentiation Data Types

	Exponent Types			
Base Type	**Integer**	**Real**	**Double**	**Complex**
Integer	Integer	Real	Double	Complex
Real	Real	Real	Double	Complex
Double	Double	Double	Double	No
Complex	Complex	Complex	No	Complex

NOTE

A negative element can be exponentiated only by an integer element; and an element with a 0 value cannot be exponentiated by another 0-value element.

In any valid exponentiation, the result has the same data type as the base element, except in two cases: (1) a real base and a double-precision exponent produces a double-precision result; and (2) a base of any type and a complex exponent produces a complex result.

Arithmetic expressions are evaluated in an order that is determined by the operators involved. The five operators in FORTRAN are performed in the following order of precedence:

Operator	Precedence
**	First
* and /	Second
+ and –	Third

When two or more operators of equal precedence (such as + and –) appear, they are evaluated by the compiler in any order that is algebraically equivalent to a left-to-right order of evaluation. For example, in 3+4–1, the addition is performed before the subtraction. Exponentiation, however, is evaluated right to left. For example, in the expression A**B**C, B**C is evaluated first, and then A is raised to the result of B**C.

2.7.1.1 Use of Parentheses

You can use parentheses to force a particular order of evaluation. When part of an expression is enclosed in parentheses, this part is evaluated first, and then the result is used in the evaluation of the remainder of the expression. In the following examples, the numbers below the operators indicate the order of evaluation:

```
4 + 3 * 2 - 6 / 2 = 7
  ⇑   ⇑   ⇑   ⇑
  2   1   4   3

(4 + 3) * 2 - 6 / 2 = 11
   ⇑   ⇑   ⇑   ⇑
   1   2   4   3

(4 + 3 * 2 - 6) / 2 = 2
   ⇑   ⇑   ⇑   ⇑
   2   1   3   4
```

```
((4 + 3) * 2 - 6) / 2 = 4
  ⇑     ⇑   ⇑   ⇑
  1     2   3   4
```

As shown in the third and fourth examples above, expressions within parentheses are evaluated according to the normal order of precedence, unless you override the order by using parentheses within parentheses.

Using parentheses to specify evaluation order is often important in high-accuracy computations because rounding and normalizations may cause algebraically equivalent evaluation orders not to be computationally equivalent.

Using parentheses to specify evaluation order is also important in complex expressions, where it is difficult during the process of writing a program to analyze visually what is to happen to what. If you have any doubt about accuracy, use parentheses.

2.7.1.2 Data Type of an Arithmetic Expression

If every element in an arithmetic expression is of the same data type, the value produced by the expression is also of this data type. If elements of different data types are combined in an expression, the data type of the result of each operation is determined by a rank associated with each data type. The data types are ranked as follows:

Data Type	Rank
Logical	1 (Low)
Integer	2
Real	3
Double Precision	4
Complex	5 (High)

The data type of the value produced by an operation on two arithmetic elements of different data types is the data type of the highest-ranked element in the operation. For example, the value resulting from an operation on an integer and a real element is real.

The data type of an expression is the data type of the result of the last operation performed in that expression.

Operations are classified by data type as follows:

- Integer operations—Integer operations are performed only on integer elements. (Logical entities used in an arithmetic context are treated as integers.) In integer arithmetic, any fraction that results from division is truncated, not rounded. For example:

  ```
  1/3 + 1/3 + 1/3
  ```

  The value of this expression is 0, not 1.

  In PDP–11 FORTRAN–77, an operation involving an INTEGER*2 element and an INTEGER*4 element is carried out as an INTEGER*4 operation and produces an INTEGER*4 result.

- Real operations—Real operations are performed only on real elements or combinations of real, integer, and logical elements. Any integer elements present are converted to real data type by giving each integer element a fractional part equal to 0. The expression is then evaluated using real arithmetic. Note, however, that in the statement Y = (I/J)*X an integer division operation is performed on I and J, and then a real multiplication is performed on the result and X.

- Double-precision operations—Any real or integer element in a double-precision operation is converted to double precision, by making the real or integer element the most significant portion of a double-precision element; the least significant portion is given the value 0. The expression is then evaluated in double-precision arithmetic.

  Converting a real element to a double-precision element does not increase its accuracy. For example, the real number

  ```
  0.3333333
  ```

  is converted to (approximately):

  ```
  0.333333134651184D0
  ```

  not to either:

  ```
  0.33333300000000000D0
  ```

  or:

  ```
  0.33333333333333333D0
  ```

- Complex operations—In an operation on an expression containing a complex element, integer elements are converted to real data type, as previously described, and double-precision elements are converted to real data type, by rounding the least-significant portion. The real element obtained is designated as the real part of a complex number;

the imaginary part is given the value 0. The expression is then evaluated with complex arithmetic, and the resulting value is complex.

2.7.2 Character Expressions

Character expressions consist of character elements. The evaluation of a character expression yields a single value of character data type.

A character element can be any one of the following:

- A character constant
- A character variable
- A character array element
- A character substring

A character expression has the form:

```
character element
```

and can be enclosed with parentheses.

2.7.3 Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator. The value of the expression is true if the stated relationship exists, false if it does not.

A relational operator tests for a relationship between two arithmetic expressions. These operators are:

Operator	Relationship
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

The delimiting periods are a required part of each operator.

Complex expressions can be related only by the .EQ. and .NE. operators. Complex entities are equal if their corresponding real and imaginary parts are both equal.

In an arithmetic relational expression, the arithmetic expressions are evaluated, and then the resulting values are compared with each other to determine whether the relationship stated by the operator exists. For example, the expression

```
APPLE+PEACH .GT. PEAR+ORANGE
```

states the relationship: "The sum of the real variables APPLE and PEACH is greater than the sum of the real variables PEAR and ORANGE." If this relationship exists, the value of the expression is true; if not, the value of the expression is false.

In a character relational expression, the character expressions are evaluated and then the resulting values are compared with each other to determine whether the relationship stated by the operator exists; PDP–11 FORTRAN–77 uses the ASCII collating sequence in comparing character values. In character relational expressions, "less than" means "precedes in the ASCII collating sequence," and "greater than" means "follows in the ASCII collating sequence." For example, the expression

```
'ABZZZ'  .LT.  'CCCCC'
```

states that 'ABZZZ' is less than 'CCCCC'. Because this relationship does exist, the value of the expression is true. If the relationship stated did not exist, the value of the expression would be false.

If the two character expressions in a relational expression are not the same length, the shorter of the two is padded on the right with spaces until the lengths are equal. For example, in the relational expressions

```
'ABC'  .EQ.  'ABC  '

'AB'  .LT  'C'
```

the first has a value of true, even though the lengths of the expressions are not equal; and the second has a value of true even though 'AB' is longer than 'C'.

All relational operators have the same precedence; however, the arithmetic operators have a higher precedence than the relational operators.

You can use parentheses to alter the order of evaluation of arithmetic expressions in a relational expression; however, because arithmetic operators are evaluated before relational operators, you need not enclose the whole of an arithmetic expression in parentheses.

You can compare two numeric expressions of different data types in a relational expression. To make such a comparison, the system converts the value of the expression with the lower-ranked data type to the data type of the expression with the higher-ranked data type.

2.7.4 Logical Expressions

Logical expressions are formed with logical elements and logical operators. A logical expression yields a single logical value—either true or false.

A logical element can be any of the following:

- An integer or logical constant
- An integer or logical variable
- An integer or logical array element
- A relational expression
- A logical expression enclosed in parentheses
- An integer or logical function reference

The logical operators are:

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction: The expression is true if, and only if, both A and B are true.
.OR.	A .OR. B	Logical disjunction (inclusive OR): The expression is true if either A or B, or both, is true.
.XOR.	A .XOR. B	Logical exclusive OR: The expression is true if A is true and B is false, or vice versa; but the expression is false if both elements have the same value.
.NEQV.	A .NEQV. B	Same as .XOR.

Operator	Example	Meaning
.EQV.	A .EQV. B	Logical equivalence: The expression is true if, and only if, both A and B have the same logical value, whether true or false.
.NOT.	.NOT. A	Logical negation: The expression is true if, and only if, A is false.

The delimiting periods of logical operators are required.

A logical expression is evaluated in accordance with the precedence of the arithmetic, relational, and logical operators. The following list gives the order in which the operators in a logical expression are evaluated:

Operator	Precedence
**	First (Highest)
* and /	Second
+ and -	Third
'The relational operators	Fourth
.NOT.	Fifth
.AND.	Sixth
.OR.	Seventh
.XOR.,.EQV.,.NEQV.	Eighth

Operators of equal rank are evaluated from left to right. For example, in the expression

```
A*B+C*ABC .EQ. X*Y+DM/ZZ .AND. .NOT. K*B .GT. TT
```

the sequence in which evaluation occurs is:

```
(((A*B)+(C*ABC)).EQ.((X*Y)+(DM/ZZ))).AND.(.NOT.((K*B).GT.TT))
```

As in arithmetic expressions, you can use parentheses to alter the normal sequence of evaluation.

Two consecutive logical operators are not allowed unless the second is .NOT.

Some logical expressions are evaluated before all their subexpressions are evaluated. For example, if A is .FALSE. in the expression

```
A .AND. (F(X,Y) .GT. 2.0) .AND. B
```

the value of the expression can be determined by testing A without evaluating F(X,Y); therefore, the function subprogram F is not called and consequences resulting from a call, such as changing variables in COMMON, do not occur.

When a logical operator operates on logical elements, the resulting data type is logical. When a logical operator operates on integer elements, the logical operation is carried out bit-by-bit on the corresponding bits of the internal (binary) representation of the integer elements; the resulting data type is integer. When a logical operator combines integer and logical values, the logical value is first converted to an integer value and then the operation is carried out as it would be for any two integer elements; the resulting data type is integer.

For example, in the sequence

```
INTEGER I, J, K
I = '65'O
J = I.OR.'100'O
K = I.AND.'23'O
```

J has the value '165'O and K has the value '21'O.

Chapter 3

Assignment Statements

Assignment statements assign a value to (or "define") a variable, an array element, or a character substring; that is, assignment statements evaluate an expression and assign the resulting value to a specified variable, array element, or character substring.

The four kinds of assignment statements are:

- Arithmetic
- Logical
- Character
- ASSIGN

3.1 Arithmetic Assignment Statement

An arithmetic assignment statement assigns an arithmetic value to a variable or array element.

The arithmetic assignment statement has the form:

```
v = e
```

v
A numeric variable or array element.

e
An arithmetic expression.

The equal sign does not mean "is equal to," as in mathematics; rather, it means "is replaced by." For example, the assignment statement

```
KOUNT = KOUNT + 1
```

means "Replace the current value of the integer variable KOUNT with the sum of the current value of KOUNT and the integer constant 1."

Although the symbolic name to the left of the equal sign can be undefined, values must have been previously assigned to all symbolic references in the expression to the right of the equal sign.

The expression must yield a value of the proper size. For example, a real expression that produces a value greater than 32767 is invalid if the entity to the left of the equal sign is an INTEGER*2 variable.

If v and e have the same data types, the statement assigns the value of e directly to v. If the data types are different, the value of e is converted to the data type of v before it is assigned. Table 3–1 summarizes the data conversion rules for assignment statements.

A character element cannot be assigned to a numeric entity.

Table 3–1: Conversion Rules for Assignment Statements

	Expression (E)			
Variable or Array Element (V)	**Integer or Logical**	**Real**	**Double Precision**	**Complex**
Integer or Logical	Assign E to V	Truncate E to integer and assign to V	Truncate E to integer and assign to V	Truncate real part of E to integer and assign to V; imaginary part of E is not used
Real	Append fraction (.0) to E and assign to V	Assign E to V	Assign MS[1] portion of E to V; LS[1] portion of E is rounded	Assign real part of E to V; imaginary part of E is not used

[1] MS = most significant (high order); LS = least significant (low order).

Table 3–1 (Cont.): Conversion Rules for Assignment Statements

Variable or Array Element (V)	Expression (E)			
	Integer or Logical	**Real**	**Double Precision**	**Complex**
Double Precision	Append fraction (.0) to E and assign to MS[1] portion of V; LS[1] portion of V is 0	Assign E to MS[1] portion of V; LS[1] portion of V is 0	Assign E to V	Assign real part of E to MS[1] portion of V; LS[1] portion of V is 0, imaginary part of E is not used
Complex	Append fraction (.0) to E and assign to real part of V; imaginary part of V is 0.0	Asign E to real part of V; imaginary part of V is 0.0	Assign MS[1] portion of E to real part of V; LS[1] portion of E is rounded; imaginary part of V is 0.0	Assign E to V

[1] MS = most significant (high order); LS = least significant (low order).

Examples of valid and invalid assignment statements are:

Valid

```
BETA = -1./(2.*X)+A*A/(4.*(X*X))

PI = 3.14159

SUM = SUM+1.
```

Invalid

```
3.14 = A-B                      (entity on the left must be a variable or
                                 array element)

-J = I**4                       (entity on the left must be a variable or
                                 array element)

ALPHA = ((X+6)*B*B/(X-Y)        (entity on the right is an invalid
                                 expression because the parentheses
                                 are not balanced)
```

3.2 Logical Assignment Statement

The logical assignment statement assigns a logical value (true or false) to a variable or array element.

The logical assignment statement has the form:

```
v = e
```

v
A logical variable or array element.

e
A logical expression.

V must be of logical data type and e must yield a logical value; otherwise, conversions will be made according to Table 3–1 and the resultant values will not be meaningful.

Values, either numeric or logical, must have been previously assigned to all variables or array elements in e.

Examples of logical assignment statements are:

Valid

```
LOGICAL PAGEND, PRNTOK, ABIG
PAGEND = .FALSE.
PRNTOK = LINE .LE. 132 .AND. .NOT. PAGEND
ABIG = A .GT. B .AND. A .GT. C .AND. A .GT. D
```

Invalid

```
X=.TRUE.                    (entity on the left must be logical)
```

3.3 Character Assignment Statement

The character assignment statement assigns the value of a character expression to a character variable, array element, or substring.

The character assignment statement has the form:

```
v = e
```

v
A character variable, array element, or substring.

e
A character expression.

If the length of the character expression is greater than the length of the character variable, array element, or substring, the character expression is truncated on the right.

If the length of the character expression is less than the length of the character variable, array element, or substring, the character expression is filled on the right with spaces.

The expression must be of character data type: you cannot assign a numeric value to a character variable, array element, or substring.

Note that assigning a value to a character substring does not affect character positions in the character variable or array element not included in the substring. If a character position outside of the substring has a value previously assigned, it remains unchanged; and if the character position is undefined, it remains undefined.

Examples of valid and invalid character assignment statements follow. (All variables and arrays in these examples are assumed to be of character data type.)

Valid

```
FILE = 'PROG2'

REVOL(1) = 'MARCIA'

LOCA(3:8) = 'PLANT5'

TEXT(I,J+1) (2:N-1) = 'NAMEX'
```

Invalid

```
'ABC' = CHARS    (element on the left must be a character variable, array
                  element, or substring reference)

CHARS = 25       (expression on the right must be of character data type)
```

3.4 Assigning Statement Labels

The ASSIGN statement assigns a statement label value to an integer variable. The variable can then be used to specify a transfer destination in a subsequent assigned GO TO statement (see Section 4.1.3).

The ASSIGN statement has the form:

```
ASSIGN s TO v
```

s
A label of an executable statement or a FORMAT statement in the same program unit as the ASSIGN statement.

v
An integer variable.

The ASSIGN statement assigns a statement label to a variable. The ASSIGN statement is similar to an arithmetic assignment statement in that it assigns a value to a variable, but differs in that the variable becomes defined for use as a statement-label reference and undefined as a variable; that is, the assigned value cannot be used for output or arithmetic computations.

The statement label must refer to an executable statement or a FORMAT statement in the same program unit.

The ASSIGN statement must be executed before the assigned GO TO statement or statements in which the assigned variable is to be used are executed. The ASSIGN statement and the assigned GO TO statements must occur in the same program unit.

For example, the statement

```
ASSIGN 100 TO NUMBER
```

associates the variable NUMBER with the statement label 100; arithmetic operations on the variable are now invalid. For example, the variable NUMBER in the statement

```
NUMBER = NUMBER+1
```

is undefined and does not result in a value of 101 being stored in NUMBER.

An associated variable can become defined again with an assignment statement. For example, assigning NUMBER a value with an arithmetic assignment statement as follows:

```
NUMBER=10
```

dissociates the variable from statement 100. The variable now has the arithmetic value 10 and can no longer be used in an assigned GO TO statement, but can be used for output and arithmetic computations.

Examples:

Valid

```
ASSIGN 10 TO NSTART
ASSIGN 99999 TO KSTOP
```

Invalid

```
ASSIGN 250 TO ERROR  (variable must be integer)
```

Chapter 4

Control Statements

Statements are normally executed in the order in which they are written. However, you may use control statements to transfer control to another point within the same program unit or to another program unit. You can also use control statements to govern iterative processing, suspension of program execution, and program termination.

The control statements are as follows:

- GO TO statement—transfers control within a program unit
- IF statement—conditionally transfers control or conditionally executes a statement
- IF THEN, ELSE IF THEN, ELSE, and END IF statements – conditionally execute blocks of statements
- DO statement—specifies iterative processing of a specified group of statements a specified number of times.
- CONTINUE statement—transfers control to the next executable statement
- CALL statement—transfers control to a subprogram
- RETURN statement—returns control from a subprogram to the calling program unit
- PAUSE statement—temporarily suspends program execution
- STOP statement—terminates program execution
- END statement—marks the end of a program unit

The following sections describe these statements.

4.1 GO TO Statements

GO TO statements transfer control to a point within the program unit containing the GO TO statement. The three types of GO TO statements are:

- Unconditional
- Computed
- Assigned

4.1.1 Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the same statement every time it is executed.

The unconditional GO TO statement has the form:

```
GO TO s
```

s
A statement label.

The statement identified by s must be an executable statement in the same program unit as the GO TO statement.

Examples:

```
GO TO 7734
GO TO 99999
```

4.1.2 Computed GO TO Statement

The computed GO TO statement transfers control to a statement specified by the value of an arithmetic expression.

The computed GO TO statement has the form:

```
GO TO (slist)[,] e
```

slist
A list, called the transfer list, of one or more labels of executable statements, separated by commas.

e
An arithmetic expression whose value is in the range 1 to n, where n is the number of statement labels in the transfer list.

The computed GO TO statement evaluates e and, if necessary, converts the result to integer data type. Control is transferred to the statement label in position e in the transfer list.

If the value of e is less than 1 or greater than the number of labels in the transfer list, control is transferred to the first executable statement after the computed GO TO.

Examples:

```
GO TO (12,24,36),INDEX
GO TO (320,330,340,350,360), SITU(J,K)+1
```

In the first example, if INDEX has a value of 2, execution is transferred to statement 24. In the second example, if SITU(J,K)+1 has a value of 3, execution is transferred to statement 340.

4.1.3 Assigned GO TO Statement

The assigned GO TO statement transfers control to a statement whose label has been placed in a variable by an ASSIGN statement. Therefore, the transfer destination may depend on the most recently executed ASSIGN statement.

The assigned GO TO statement has the form:

```
GO TO v[[,](slist)]
```

v
An integer variable.

slist
A list of one or more labels of executable statements separated by commas.

The assigned GO TO statement transfers control to the statement whose label was most recently assigned to the variable v. (See Section 3.4 on the ASSIGN statement.)

The GO TO statement, the associated ASSIGN statement or statements, and the statements to which control is transferred must be executable statements in the same program unit. If slist is used, the assigned value of v must be a member of slist.

In PDP-11 FORTRAN-77, if the statement label value of v is not present in slist (if slist is specified), control is transferred to the next executable statement following the assigned GO TO statement.

Examples of assigned GO TO statements are:

```
ASSIGN 200 TO IGO
GO TO IGO
```

This example is equivalent to GO TO 200.

```
ASSIGN 450 TO IBEG
GO TO IBEG, (300,450,1000,25)
```

This example is equivalent to GO TO 450.

4.2 IF Statements

An IF statement transfers control or executes a statement (or a block of statements) if a specified condition is met. The three types of IF statements are:

- Arithmetic IF statement
- Logical IF statement
- Block IF statement

The decision to transfer control or to execute a statement is based on the evaluation of an expression contained in the IF statement.

4.2.1 Arithmetic IF Statement

The arithmetic IF statement transfers control to one of three statements, on the basis of the value of an arithmetic expression.

The arithmetic IF statement has the form:

```
IF (e) s1, s2, s3
```

e
An arithmetic expression.

s1,s2,s3
Labels of executable statements in the same program unit.

All three labels (s1,s2,s3) are required; however, they need not refer to three different statements.

The arithmetic IF statement evaluates expression e. If e is less than zero, control passes to label s1; if e is equal to zero, control passes to label s2; if e is greater than zero, control passes to label s3.

Some examples:

```
IF (THETA-CHI) 50,50,100
```

This statement transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI, to statement 100 if THETA is greater than CHI.

```
IF (NUMBER/2*2-NUMBER) 20,40,20
```

This statement transfers control to statement 40 if the value of the integer variable NUMBER is even, to statement 20 if the value is odd.

4.2.2 Logical IF Statement

The logical IF statement conditionally executes a single FORTRAN statement on the basis of the evaluation of a logical expression.

The logical IF statement has the form:

```
IF (e) st
```

e
A logical expression.

st
A complete FORTRAN statement. The statement can be any executable statement except a DO statement, an END statement, a block IF statement, or another logical IF statement.

The logical IF statement first evaluates logical expression e. If the value of the expression is true, statement st is executed. If the value of the expression is false, control transfers to the next executable statement after the logical IF, and statement st is not executed. Note that e must yield a logical value.

Examples of logical IF statements:

```
IF (J .GT. 4 .OR. J .LT. 1) GO TO 250

IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K)*(-1.5D0)

LOGICAL ENDRUN
IF (ENDRUN) CALL EXIT
```

4.2.3 Block IF Statements

Block IF statements conditionally execute blocks (or groups) of statements.

The four block IF statements are:

- IF THEN
- ELSE IF THEN
- ELSE
- END IF

These statements are used in block IF constructs. The block IF construct has the form:

```
IF (e) THEN
   block

ELSE IF (e) THEN
   block
     .
     .
     .

ELSE
   block

END IF
```

e
A logical expression.

block
A sequence of zero or more complete FORTRAN statements. This sequence is called a statement block.

Figure 4–1 describes the flow of control for four examples of block IF constructs.

Each block IF statement, except the END IF statement, has an associated statement block. The statement block consists of all the statements following the block IF statement up to (but not including) the next block IF statement in the block IF construct. The statement block is conditionally executed based on the values of logical expressions in the preceding block IF statements.

The IF THEN statement begins a block IF construct. The block following it is executed if the value of the logical expression in the IF THEN statement is true.

The ELSE IF THEN statement is an optional statement within a block IF construct. The statement instructs the system to execute a statement block if the value of the logical expression in the ELSE IF THEN statement is true, **and** if no preceding statement block in the block IF construct was executed. A block IF construct can contain any number of ELSE IF THEN statements.

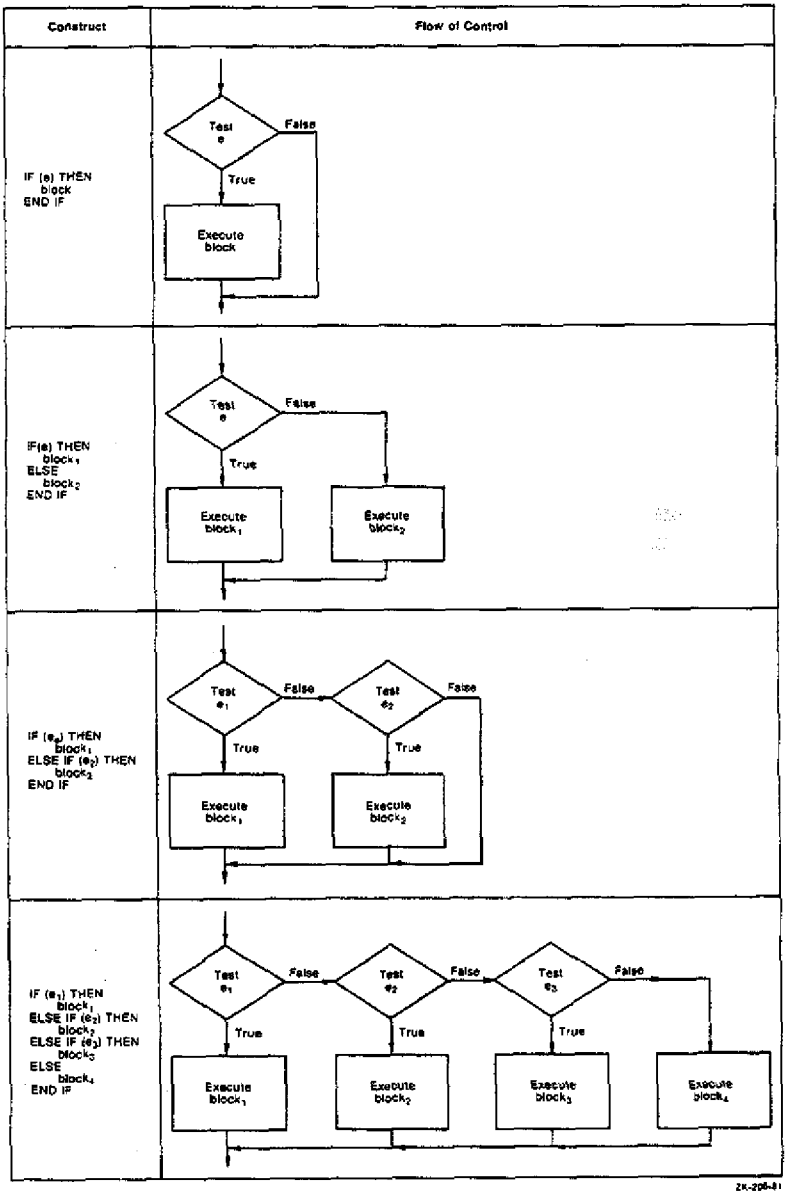
The ELSE statement is an optional statement within a block IF construct that specifies a statement block to be executed if no preceding statement block in the block IF construct was executed. Except for the END IF statement, no block IF statement can follow the ELSE statement.

The END IF statement terminates the block IF construct.

After the last statement in a statement block is executed, control passes to the next executable statement following the END IF statement. Consequently, only one statement block in a block IF construct can be executed each time an IF THEN statement is executed.

ELSE IF THEN and ELSE statements can have statement labels, but these labels cannot be referenced. The END IF statement can have a statement label to which control can be transferred, but control can be transferred only from within the block IF construct.

Figure 4–1: Examples of Block IF Constructs



Section 4.2.3.1 describes restrictions on statements in a statement block. Section 4.2.3.2 describes examples of block IF constructs. Section 4.2.3.3 describes nested block IF constructs.

4.2.3.1 Statement Blocks

A statement block can contain any executable FORTRAN statement except an END statement. You can transfer control out of a statement block, but you cannot transfer control back into the block. You cannot transfer control from one statement block to another.

DO loops cannot partially overlap statement blocks. When a statement block contains a DO statement, it must also contain the DO loop's terminal statement. Conversely, when a statement contains a DO loop's terminal, it must also contain the DO statement. If you use DO loops with statement blocks, each loop must be wholly contained within one statement block.

4.2.3.2 Block IF Examples

The simplest block IF construct consists of the IF THEN and END IF statements; this construct conditionally executes one statement block. An example follows:

Form	Example

`IF (e) THEN`
`  block`
`END IF`
	`IF (ABS(ADJU).GE.1.0E-6) THEN`
	`  TOTERR=TOTERR+ABS(ADJU)`
	`  QUEST=ADJU/FNDVAL`
	`END IF`

The statement block consists of all the statements between the IF THEN and the END IF statements.

The IF THEN statement first evaluates logical expression e, ABS(ADJU).GE.1.0E-6. If the value of e is true, the statement block is executed. If the value of e is false, control transfers to the next executable statement after the END IF statement; the block is not executed.

The following example shows a block IF construct with an ELSE IF THEN statement:

Form	Example

`IF (e1) THEN`
`  block1`
`ELSE IF (e2) THEN`
`  block2`
`END IF`
	`IF (A .GT. B) THEN`
	`  D = B`
	`  F = A - B`
	`ELSE IF (A .GT. B/2.) THEN`
	`  D = B/2.`
	`  F = A - B/2.`
	`  END IF`

Block1 consists of all the statements between the IF THEN and the ELSE IF THEN statements; block2 consists of all the statements between the ELSE IF THEN and the END IF statements.

If A is greater than B block1 is executed.

If A is not greater than B but A is greater than B/2, block2 is executed.

If A is not greater than B and A is not greater than B/2, neither block1 nor block2 is executed; control transfers directly to the next executable statement after the END IF statement.

The following example shows a block IF construct with an ELSE statement:

Form	Example

`IF (e) THEN`
`  block1`
`ELSE`
`  block2`
`END IF`
	`IF (NAME .LT. 'N') THEN`
	`  IFRONT = IFRONT + 1`
	`  FRLET(IFRONT)=NAME(1:2)`
	`ELSE`
	`  IBACK=IBACK + 1`
	`END IF`

Block1 consists of all the statements between the IF THEN and ELSE statements; block2 consists of all the statements between the ELSE and the END IF statements.

If the value of the character variable NAME is less than 'N', block1 is executed.

If the value of NAME is greater than or equal to 'N', block2 is executed.

The following example shows a block IF construct with several ELSE IF THEN statements and an ELSE statement:

Form	Example

`IF (e1) THEN`
`  block1`
	`IF (A .GT. B) THEN`
	`  D = B`
	`  F = A - B`

`ELSE IF (E2) THEN`
`  block2`
	`ELSE IF (A .GT. C) THEN`
	`  D = C`
	`  F = A - C`

`ELSE IF (e3) THEN`
`  block3`
	`ELSE IF (A .GT. Z) THEN`
	`  D = Z`
	`  F = A - Z`

`ELSE`
`  block4`
	`ELSE`
	`  D = 0.0`
	`  F = A`

`END IF`	`END IF`

The above example contains four statement blocks. Each block consists of all the statements between the block IF statements listed below.

Block	Delimiting Block IF Statements
block1	IF THEN and first ELSE IF THEN
block2	First ELSE IF THEN and second ELSE IF THEN
block3	Second ELSE IF THEN and ELSE
block4	ELSE and END IF

If A is greater than B, block1 is executed.

If A is not greater than B but is greater than C, block2 is executed.

If A is not greater than B or C but is greater than Z, block3 is executed.

If A is not greater than B, C, or Z, block4 is executed.

4.2.3.3 Nested Block IF Constructs

A block IF construct can be included in a statement block of another block IF construct. But the nested block IF construct must be completely contained within a statement block; it must not overlap statement blocks.

The following example contains a nested block IF construct:

Form	Example
`IF (e) THEN`	`IF (A .LT. 100) THEN`
`  IF (e) THEN`	`  INRAN=INRAN + 1`
`  blocka`	`  IF (ABS (A-AVG) .LE. 5.) THEN`
`block1  ELSE`	`  INAVG = INAVG + 1`
`  blockb`	`  ELSE`
`  END IF`	`  OUTAVG = OUTAVG + 1`
`ELSE`	`  END IF`
`  block2`	`ELSE`
`END IF`	`  OUTRAN = OUTRAN + 1`
	`END IF`

If A is less than 100, block1 is executed. Block1 contains a nested block IF construct. If the absolute value of A minus AVG is less than or equal to 5, blocka is executed. If the absolute value of A minus AVG is greater than 5, blockb is executed. If A is greater than or equal to 100, block2 is executed; the nested IF construct is not executed because it is not in block2.

4.3 DO Statement

The DO statement specifies iterative processing of a sequence of statements. The sequence of statements is called the range of the DO statement, and the DO statement together with its range is called a DO loop.

The DO statement has the form:

```
DO  s[,]   v=e1,e2[,e3]
```

s
The label of an executable statement. This executable statement must physically follow the DO statement, in the same program unit.

v
Usually an integer variable but may be a real or double-precision variable.

e1,e2,e3
Usually integer expressions but may be real or double-precision expressions.

The variable v is called the control variable; e1, e2, and e3 are the initial, terminal, and increment parameters, respectively. If you omit the increment parameter, a default increment value of 1 is used. In FORTRAN-77, v can be a real or double-precision variable, and el, e2, and e3 can be any arithmetic expressions. If necessary, evaluated expressions are converted to the data type of the control variable before they are used. If the data type of the control variable is real or double-precision, the number of iterations of the DO range might not be what is expected because of the effects of floating-point rounding.

The label s identifies the terminal statement of the DO loop. The terminal statement must not be:

- A GO TO statement
- An arithmetic IF statement
- Any block IF statement
- An END statement
- A RETURN statement
- A DO statement

The range of the DO statement consists of all the statements that follow the DO statement, up to and including the terminal statement.

The DO statement first evaluates the expressions e1, e2, and e3 to determine values for the initial, terminal, and increment parameters, respectively. The value of the initial parameter is assigned to the control variable. The executable statements in the range of the DO loop are then executed repeatedly. The exact mechanism is explained in Section 4.3.1.

The number of executions of the DO range, called the iteration count, is given by:

```
[(e2 - e1 + e3)/e3]
```

where, letting X represent the above expression, [X] is the largest integer whose magnitude does not exceed the magnitude of X and whose sign is the same as the sign of X (for example, [-3.5] = -3). The increment parameter, e3, cannot be zero.

If the iteration count is zero or negative, the body of the loop is not executed. If the /NOF77 compiler qualifier is specified and the iteration count is zero or negative, the body of the loop is executed once.

4.3.1 DO Iteration Control

After each execution of the DO range, the following actions are taken:

1. The value of the increment parameter is algebraically added to the control variable.
2. The iteration count is decremented by 1.
3. If the iteration count is greater than 0, control is transferred to the first executable statement after the DO statement, for another iteration of the range.
4. If the iteration count is 0, execution of the DO statement is terminated.

You can also cause execution of a DO statement to be terminated by using a statement within the range that transfers control outside the loop. If control is transferred outside the loop, the control variable of the DO statement remains defined with its current value.

When execution of a DO loop terminates, but other DO loops share this loop's terminal statement, control transfers outward to the next DO loop in the nesting structure (see Section 4.3.2). If no other DO loop shares a DO loop's terminal statement, or if a DO loop is outermost, control transfers to the first executable statement after the terminal statement.

You cannot alter the value of the control variable within the range of the DO loop; however, you can reference it for purposes other than altering it.

The range of a DO statement can contain other DO statements (nested DO loops), as long as these DO statements meet certain requirements. See Section 4.3.2.

You cannot transfer control into the range of a DO loop. Exceptions to this rule are described in Sections 4.3.3 and 4.3.4.

You can modify variables holding the initial, terminal, or increment parameters within the loop without affecting the iteration count.

Examples of DO statements follow.

Valid

```
DO 100 K=1,50,2
```

This statement specifies 25 iterations; K=49 during the final iteration.

```
DO 350 J=50,-2,-2
```

This statement specifies 27 iterations; J=–2 during the final iteration.

```
DO 25 IVAR=1,5
```

This statement specifies 5 iterations; IVAR=5 during the final iteration.

Invalid

```
DO NUMBER=5,40,4      (the statement label is missing)
DO 40 M=2.10          (a decimal point has been typed for a comma)
```

Note that in the last invalid example, the statement

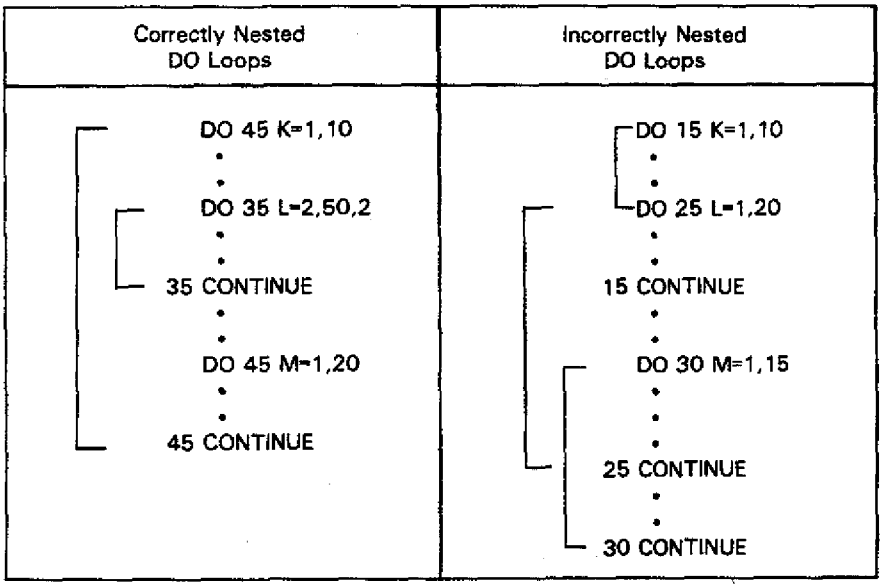
```
DO40M = 2.10
```

is an unintentionally valid arithmetic assignment statement.

4.3.2 Nested DO Loops

A DO loop can include one or more complete DO loops called nested DO loops. The range of a nested DO loop must lie completely within the range of the next outer loop. Nested loops can share a terminal statement. Figure 4–2 illustrates nested loops.

Figure 4–2: Nested DO Loops



ZK-7629-HC

4.3.3 Control Transfers in DO Loops

Within a nested DO loop, you can transfer control from an inner loop to an outer loop; however, you cannot transfer control from an outer loop to an inner loop.

If two or more nested DO loops share the same terminal statement, you can transfer control to this shared terminal statement only from within the range of the innermost loop. Because this shared terminal statement is part of the innermost loop, any transfer to it from an outer loop is a transfer from an outer loop to an inner loop, and is therefore invalid.

4.3.4 Extended Range

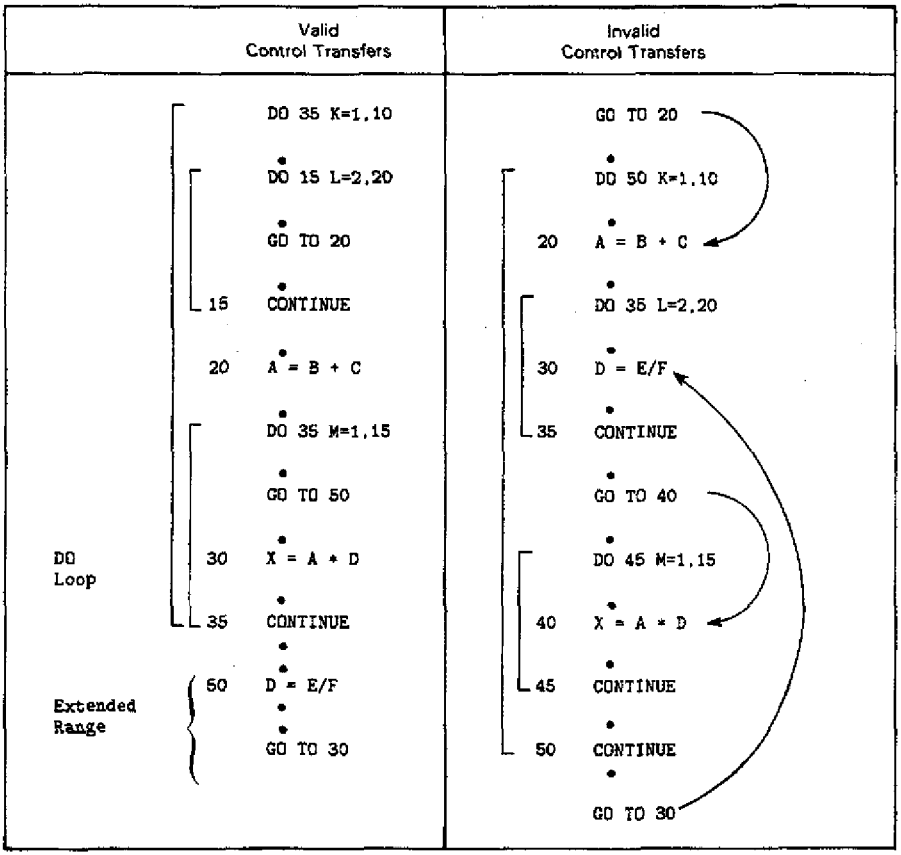
A DO loop has an extended range if a control statement transfers control out of the loop and then, after execution of one or more statements, another control statement returns control into the loop. The range of this DO loop includes all executable statements between the destination statement of the first transfer and the statement that returns control to the loop.

The following rules govern the use of a DO statement with an extended range:

- A transfer into the range of a DO statement is permitted only from its extended range.
- Statements in the extended range must not change the control variable.

Figure 4–3 illustrates valid and invalid extended range control transfers.

Figure 4–3: Control Transfers and Extended Range



ZK-7628-HC

4.4 CONTINUE Statement

The CONTINUE statement transfers control to the next executable statement. It is primarily used as the terminal statement of a DO loop that would otherwise end with a prohibited control statement such as a GO TO or an arithmetic IF.

The CONTINUE statement has the form:

```
CONTINUE
```

4.5 CALL Statement

The CALL statement executes a SUBROUTINE subprogram or other external procedure. It can also specify an argument list for the subroutine. (See Chapter 6 for details on the definition and use of a subroutine.)

The CALL statement has the following form:

```
CALL s[([a][,[a]]...)]
```

s
The name of a SUBROUTINE subprogram or other external procedure, or a dummy argument associated with a SUBROUTINE subprogram or other external procedure.

a
An actual argument. (Section 6.1 describes actual arguments.)

If you specify an argument list, the CALL statement associates the values in the list with the dummy arguments in the subroutine. It then transfers control to the first executable statement of the subroutine.

The arguments in the CALL statement must agree in number, order, and data type with the dummy arguments in the subroutine. These arguments can be variables, arrays, array elements, substring references, constants, expressions, Hollerith constants, character constants, or subprogram names. An unsubscripted array name in the argument list refers to the entire array.

Examples of CALL statements are:

```
CALL CURVE (BASE,3.14159+X,Y,LIMIT,R(LT+2))
CALL PNTOUT (A,N,'ABCD')
CALL EXIT
```

4.6 RETURN Statement

The RETURN statement is used to return control from a subprogram to the calling program. It has the form:

```
RETURN
```

When a RETURN statement is executed in a function subprogram, control is returned to the statement that contains the function reference (see Chapter 6). When a RETURN statement is executed in a subroutine subprogram, control is returned to the first executable statement following the CALL statement.

RETURN statement example:

```
      SUBROUTINE SIZCHK (N,K)
      IF (N) 10,20,30
10    K=-1
      RETURN
20    K=0
      RETURN
30    K=+1
      RETURN
      END
```

4.7 PAUSE Statement

The PAUSE statement temporarily suspends program execution and displays a message on the terminal to permit you to take some action.

The PAUSE statement has the form:

```
PAUSE  [disp]
```

disp
An alphanumeric literal, a decimal digit string of one to five digits, or an octal constant.

The disp argument is optional. The effect of a PAUSE statement depends on how your program is being executed. If it is running as a batch job, the contents of disp are written to the system output file, and the program is not suspended.

If the program is running in interactive mode, the contents of disp are displayed at your terminal, followed by a prompt sequence indicating that the program is suspended. After you then enter the proper control command, execution resumes with the first executable statement following the PAUSE. The proper control command is specific to the operating system (refer to the *PDP-11 FORTRAN-77 User's Guide*).

Some examples of PAUSE statements are:

```
PAUSE    999
PAUSE    'MOUNT NEXT TAPE'
```

4.8 STOP Statement

The STOP statement terminates program execution and returns control to the operating system.

The STOP statement has the form:

```
STOP  [disp]
```

disp
A character constant, a decimal digit string of one to five digits, or an octal constant.

The disp argument, if present, specifies a message to be displayed when execution stops.

Examples of STOP statements are:

```
STOP 98
STOP 'END OF RUN'
STOP
```

4.9 END Statement

The END statement marks the end of a program unit. It must be the last source line of every program unit.

The END statement has the form:

```
END
```

The END statement must not occur on a continuation line and must not itself be continued.

In a main program, if no STOP statement prevents execution from reaching the END statement, program execution terminates; in a subprogram, a RETURN statement is implicitly executed.

Chapter 5

Specification Statements

Specification statements are nonexecutable statements that let you allocate and initialize variables and arrays, and define other characteristics of the symbolic names used in the program.

The specification statements are:

- IMPLICIT statement overrides the implied (default) data-typing of symbolic names
- Type declaration statement—explicitly declares the data type of specified symbolic names
- DIMENSION statement—declares the number of dimensions in an array, and the number of elements in each dimension
- COMMON statement—reserves one or more contiguous areas of storage
- VIRTUAL statement—reserves space for one or more arrays to be located outside normal program storage
- EQUIVALENCE statement—associates the same storage location with two or more entities
- SAVE statement—retains the definition status of an entity after execution of a RETURN statement in a subprogram
- EXTERNAL statement—declares the specified symbolic names to be external procedure names
- INTRINSIC statement—declares one or more symbolic names to be FORTRAN intrinsic functions
- DATA statement—assigns initial values to variables, arrays, and array elements before program execution
- PARAMETER statement—assigns a symbolic name to a constant value

- PROGRAM statement—assigns a symbolic name to a main program unit
- BLOCK DATA statement—establishes a BLOCK DATA program unit in which initial values may be assigned to entities contained in common blocks

The following sections describe these statements.

5.1 IMPLICIT Statement

The IMPLICIT statement permits you to change the default data-typing rules. By default, all names beginning with the letters I through N are interpreted to be of integer data type, and all names beginning with any other letter are interpreted to be of real data type; the IMPLICIT statement allows you to alter these interpretations.

The IMPLICIT statement has one of the following forms:

```
IMPLICIT typ(a[,a]...)[,typ(a[,a]...)]...
IMPLICIT NONE
```

typ
One of the data-type specifiers. (See Table 2–2.)

a
An alphabetic specification in one of two forms: c or c1–c2, where c is an alphabetic character. The c1–c2 form specifies a range of letters (from c1 through c2), that must occur in alphabetical order.

The IMPLICIT statement assigns the specified data type to all symbolic names that begin with any of the specified letters and that have no explicit data-type declaration. Explicit declarations take precedence over implicit declarations.

The IMPLICIT statement also affects symbolic names defined in a PARAMETER statement (see Section 5.11).

For example, the statements

```
IMPLICIT INTEGER (I,J,K,L,M,N)
IMPLICIT REAL (A-H, O-Z)
```

specify the default in the absence of any explicit statement.

The IMPLICIT NONE statement is used to override all implicit defaults. You must then explicitly declare the data types of all symbolic names in the program unit. If you specify IMPLICIT NONE, no other IMPLICIT statement can be included in the program unit.

IMPLICIT statements must precede all other specification statements except PARAMETER statements, and they must precede all executable statements.

You can use the IMPLICIT statement to set a default length for the character data type; simply specify typ as CHARACTER*len, where len is the default length. Typ must be an unsigned integer constant or a positive integer constant in parentheses, in the range 1 through 255.

Any data type can be specified in an IMPLICIT statement, as the following examples demonstrate:

```
IMPLICIT DOUBLE PRECISION (D)
IMPLICIT COMPLEX (S,Y), LOGICAL*1 (L,A-C)
```

NOTE

The IMPLICIT statement has no effect on default types of intrinsic functions.

5.2 Type Declaration Statements

Type declaration statements explicitly define the data type of specified symbolic names. There are two forms of type declaration statements: numeric type declarations (see Section 5.2.1) and character type declarations (see Section 5.2.2).

The following rules apply to type declaration statements:

- Type declaration statements must precede all executable statements.
- The data type of a symbolic name can be declared only once.
- You can use a type declaration statement to declare an array by appending an array declarator (see Section 2.5.1) to an array name.

5.2.1 Numeric Type Declaration Statements

Numeric type declaration statements has the form:

```
typ v[,v]
```

typ
Any data type specifier (see Table 2–2) except CHARACTER

v
The symbolic name of a variable, array, statement function, function subprogram, or an array declarator.

A symbolic name can be followed by a data-type length specifier of the form *s, where s is one of the acceptable lengths for the data type being declared (see Table 2–2). Such a specification overrides the length attribute that the statement implies, and assigns a new length to the specified item. If you specify both a data-type length specifier and an array declarator, the data type length specifier goes first. Examples of type declaration statements are as follows.

```
INTEGER COUNT, MATRIX(4,4), SUM
REAL MAN,IABS
LOGICAL SWITCH

INTEGER*2 Q, M12*4, IVEC*4(10)
REAL*8 WX1, WX3*4, WX5, WX6*8
```

5.2.2 Character Type Declaration Statements

Character type declaration statements have the form:

```
CHARACTER[*len[,]] v[*len][,v[*len]]...
```

v
The symbolic name of a constant, variable, array, or array declarator. (You cannot declare a function subprogram, a statement function, or a virtual-array name to be of character data type.)

len
An unsigned integer constant or an integer-constant expression enclosed in parentheses. The value of len specifies the length of the character data elements.

If you specify CHARACTER*len, len becomes the default length specification for the specified list. If an item in this list does not have its own length specification, the item's length is len. However, if an item does have its own length specification, this specification overrides the default length specified in CHARACTER*len.

If you do not specify a length, a length of 1 is assumed. The length specification must be in the range 1 to 255; a length specification of zero is invalid. You can use a character type declaration statement to define arrays by including array declarators (see Section 2.5.1) in the list. If you specify both an array declarator and a length, the array declarator goes first (the reverse of the rule for numeric type declarations).

Examples of character type declaration statements follow:

```
CHARACTER*32 NAMES(100), SOCSEC (100)*9, NAMETY*10
```

This statement specifies an array NAMES comprising one hundred 32-character elements, an array SOCSEC comprising one hundred 9-character elements, and a variable NAMETY, which is 10 characters long.

```
PARAMETER (LENGTH=4)
CHARACTER*(4+LENGTH) LAST, FIRST
```

The latter statement specifies two 8-character variables, LAST and FIRST. (The PARAMETER statement is described in Section 5.11.)

```
CHARACTER LETTER(26)
```

This statement specifies an array LETTER comprising twenty-six 1-character elements.

```
CHARACTER*16 BIGCHR*(30000*2),QUEST*(5*INT(A))
```

This statement is invalid; the value specified for BIGCHR is too large, and the length specifier for QUEST is not an integer constant expression.

5.3 DIMENSION Statement

The DIMENSION statement specifies the number of dimensions in an array and the number of elements in each dimension.

The DIMENSION statement has the form:

```
DIMENSION a(d)[,a(d)]...
```

a(d)
An array declarator (see Section 2.5.1).

a
The symbolic name of an array.

d
A dimension declarator.

The DIMENSION statement allocates one storage element to each element in each dimension of an array. The data type of the array determines the length of the storage element.

The total number of storage elements assigned to an array is equal to the product of the array's individual dimension declarators. For example, the statement

```
DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

defines ARRAY as having 16 (4x4) real elements of 4 bytes each and defines MATRIX as having 125 (5x5x5) integer elements of 2 bytes each.

In addition to DIMENSION statements, you can use array declarators in type declaration, COMMON, and VIRTUAL statements. However, within a program unit, you can use an array name in only one array declarator.

Examples of DIMENSION statements are:

```
DIMENSION BUD(12,24,10)
DIMENSION X(5,5,5),Y(4,85),Z(100)
DIMENSION MARK(4,4,4,4)
```

For further information on arrays and on storing array elements, see Section 2.5.

5.4 COMMON Statement

A COMMON statement reserves one or more contiguous blocks of storage. A symbolic name is used to identify each contiguous block; however, you can omit a symbolic name for a blank common block in a program unit. COMMON statements also specify the order of variables and arrays in each common block.

The COMMON statement has the form:

```
COMMON [/[cb]/] nlist[[,]/[cb]/ nlist]...
```

cb
A symbolic name, called a common block name; cb can be blank. (If the first cb is blank, you can omit the first pair of slashes.)

nlist
A list of variable names, array names, and array declarators separated by commas. (You cannot use a virtual-array name in a COMMON statement.)

A common block can have the same name as a variable or an array in the same executable program. However, it cannot have the same name as a function, subroutine, or entry in the same executable program (see Section 2.1).

When common blocks having the same name but located in separate programs are made part of the same executable program, the individual names become associated with the same storage area. Consider the following example:

```
PROGRAM MAIN
COMMON/BLOCK1/ICOUN,IHOL/BLOCK2/ICHK(10)
  .
  .
CALL GSUB
  .
  .
END

SUBROUTINE GSUB
COMMON/BLOCK2/JCHK(10)/BLOCK1/JCOUN,JHOL
  .
  .
  .
END
```

In this example, BLOCK1 in MAIN and BLOCK1 in GSUB are associated with the same storage area; likewise, the two BLOCK2s are associated with a single storage area.

You can have only one blank common block in an executable program, but you can have up to 250 named common blocks.

Entities are assigned storage in common blocks on a one-for-one basis. In the above example, ICOUN and JCOUN are associated with the same storage space in BLOCK1, because each entity occurs first in its respective list.

Entities placed in a one-to-one correspondence in the same common block should agree in data type. For example, if one program unit contains the statement

```
COMMON CENTS
```

and another program unit contains the statement

```
INTEGER*2 MONEY
COMMON MONEY
```

incorrect results may occur when these program units are combined, because the 2-byte integer variable MONEY is made to correspond to the high-order 2 bytes of the real variable CENTS.

You must not assign LOGICAL*1 (BYTE) or character variables or arrays to a common block in such a way that subsequent data of any other type is allocated on an odd byte boundary. The compiler supplies no filler space for common blocks; however, all common blocks are begun on a word (even byte) boundary. In addition, you must not mix character and numeric data in COMMON blocks. The data in a COMMON block must be entirely of numeric data type or entirely of character data type.

Examples of COMMON statements follow.

Main Program

```
COMMON HEAT,X/BLK1/KILO,Q
  .
  .
CALL FIGURE
  .
  .
  .
```

Subprogram

```
SUBROUTINE FIGURE
COMMON /BLK1/LIMA,R/ /ALFA,BET
  .
  .
  .
RETURN
END
```

The COMMON statement in the main program puts HEAT and X in a blank common block, and puts KILO and Q in a named common block, BLK1. The COMMON statement in the subroutine makes ALFA and BET correspond to HEAT and X in the blank common block, and makes LIMA and R correspond to KILO and Q in BLK1.

Valid	Invalid
`INTEGER CHARS(9)`	`CHARACTER CHARS(9)`
`COMMON/STRING/ILEN,CHARS`	`COMMON/STRING/CHARS,ILEN`

In this example, the integer variable ILEN is allocated on the same block as a character variable.

```
BYTE B0,B1
COMMON/STRING/B0,ILEN,B1
```

In this example, the integer variable ILEN is allocated on an odd byte address.

5.5 VIRTUAL Statement

A virtual array is an array whose storage is allocated in physical main memory outside of the program's directly addressable main memory. The use of virtual arrays in a program frees directly addressable memory for executable code and other data storage.

The VIRTUAL statement names a virtual array and specifies the number of dimensions and the number of elements in each dimension. The VIRTUAL statement has the form:

```
VIRTUAL a(d) [,a(d)]...
```

a(d)
An array declarator (see Section 2.5.1).

a
The symbolic name of an array.

d
A dimension declarator.

The maximum total directly addressable memory available to user programs executing on a computer in the PDP-11 family is 64K, or 65,536 bytes. In light of the allowable sizes of PDP-11 FORTRAN-77 arrays, it is easy to see how quickly directly addressable main memory can be used up. A numeric array, for instance, can have a maximum of 32,767 elements of from 1 to 8 bytes in length. Therefore, a maximum LOGICAL*1 array of 1 byte per element would require 32,767 bytes of storage space, and a maximum COMPLEX array of 8 bytes per element would require 262,136 bytes of storage space, a requirement far beyond the 64K limit on directly addressable memory.

NOTE

Virtual arrays are not supported on RSTS/E operating systems.

The data type of a virtual array is specified in the same way that the data type of any other array is specified, that is, either implicitly by the first letter of the name, or explicitly, by a type declaration statement.

An example of a VIRTUAL statement follows:

```
VIRTUAL A(1000), LARG(180,180), Mult (4,4,4,4,4,4,4)
```

This statement defines a one-dimensional array named A of 1000 elements, a two-dimensional array named LARG of 32400 elements, and a seven-dimensional array named MULT of 16384 elements. These arrays are placed in external main memory and therefore do not significantly diminish the 64K of directly addressable memory.

For further information concerning arrays and their storage, see Section 2.5.

5.5.1 Restrictions on Using Virtual Arrays

Virtual arrays and virtual array elements are subject to the following limitations:

- A virtual array name must not be used in a COMMON statement (see Section 5.4).
- The name of a virtual array or virtual array element must not be used in an EQUIVALENCE statement (see Section 5.6).
- A virtual array or virtual array element cannot be assigned an initial value by a DATA statement (see Section 5.10).
- Virtual arrays cannot be used to contain run-time format specifications (see Section 8.6). The name of a virtual array or virtual array element must not appear as a format specifier in an I/O statement.
- The name of a virtual array or virtual array element must not be specified as the buffer argument (third argument inside parentheses) of an ENCODE or DECODE statement (see Section A.1).
- The name of a virtual array element must not be used as an actual argument to a subprogram if the subprogram assigns a value to the corresponding dummy argument (see Section 6.1).
- The name of a virtual array or virtual array element cannot be used to specify the FILE keyword in an OPEN statement (see Section 9.1.10).

- The name of a virtual array cannot be used to specify a key expression in a keyed I/O statement.
- A virtual array name must not be of data type character.

Below are examples of valid and invalid use of virtual arrays:

Valid

```
      VIRTUAL A(1000),B(2000)
      READ(1,*) A
      DO 10,I=1,1000
10    B(I)=-A(I)*2
      WRITE(2,*) (A(I),I=1,1000)
      CALL SUB (A,B)
```

Invalid

```
      VIRTUAL A(10)
      CHARACTER A                    (declared as type character)
      DATA A(1)/2.5/                 (used in DATA statement)
      COMMON /X/  A                  (used in COMMON statement)
      EQUIVALENCE  (A(1),Y)          (used in EQUIVALENCE statement)
      WRITE(1,A) X,Y                 (used as format specifier)
      ENCODE (4,100,A(3)) X,Y        (used as ENCODE output buffer)
```

5.5.2 Virtual Array References in Subprograms

A dummy argument that is the name of a virtual array can become associated with an actual argument that is also the name of a virtual array.

An actual argument that is a reference to a virtual array element can become associated only with a dummy argument that is a simple variable (see Section 2.4). In effect, an actual argument that is a virtual array element is treated as if it were an expression.

Furthermore, a value must be assigned to a virtual array element before this element is used as an actual argument and the subprogram must not alter the value of the corresponding dummy argument.

Below are examples of valid and invalid virtual array references in subprograms:

Valid Usage

```
      VIRTUAL A(1000),B(1000)
      B(3)=0.5
      CALL SCALE(A,1000,B(3))
      END
```

```
      SUBROUTINE SCALE (X,N,W)
      VIRTUAL X(N)
      S=0
      DO 10, I=1,N
10    S=S+X(I)*W
      TYPE *,S
      END
```

Invalid Usage

```
      VIRTUAL A(1000)
      REAL B(4000)
      CALL ABC(A,B,A(3))
      END

      SUBROUTINE ABC(X,Y,Z)
      REAL X(1000)                  (actual argument is virtual)
      VIRTUAL Y(4000)               (actual argument is nonvirtual)
      Z=2.3                         (actual argument is virtual array
      END                            element)
```

5.6 EQUIVALENCE Statement

The EQUIVALENCE statement partially or totally associates two or more entities in the same program unit with the same storage location.

The EQUIVALENCE statement has the form:

```
EQUIVALENCE (nlist) [,(nlist)]...
```

nlist

A list of variables, array elements, arrays, or character substring references, separated by commas. You must specify at least two of these entities in each list.

The EQUIVALENCE statement allocates storage that begins at the same location to all of the entities in its list.

In an EQUIVALENCE statement, each expression in a subscript or substring reference must be an integer constant or integer constant expression.

Dummy arguments, virtual arrays, and virtual array elements may not be used in an EQUIVALENCE statement.

The entities in nlist must be either entirely of numeric data type or entirely of character data type: you cannot make numeric entities and character entities equivalent.

You must not equivalence LOGICAL*1 arrays with other elements in such a way that subsequent data of any other type is allocated on an odd byte boundary.

An array name used in an EQUIVALENCE statement refers to the first element of the array.

You can equivalence variables of different numeric data types; that is, you can store them such that each entity begins at the same address. Furthermore, you can store multiple components of one data type with a single component of a higher-ranked data type. For example, if you make an integer variable equivalent to a complex variable, the integer variable shares storage with the real part of the complex variable.

Examples of valid and invalid EQUIVALENCE statements are:

Valid

```
DOUBLE PRECISION DVAR
INTEGER*2  IARR(4)
EQUIVALENCE (DVAR,IARR(1))
```

This EQUIVALENCE statement makes the four elements of the integer array IARR occupy the same storage as the double-precision variable DVAR.

```
CHARACTER KEY*16, STAR*10
EQUIVALENCE (KEY,STAR)
```

This EQUIVALENCE statement makes the first character of the character variables KEY and STAR share the same storage location. The character variable STAR is equivalent to the substring KEY (1:10).

Invalid

```
LOGICAL*1 BYTES(10)
EQUIVALENCE (ILEN, BYTES(8))
```

In this example, the integer variable ILEN is allocated on an odd byte address.

5.6.1 Making Arrays Equivalent

When you make an element of one array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between corresponding elements of the two arrays. Therefore, if the first elements of two equal-sized arrays are made equivalent, both arrays share the same storage space. And, for example, if the third element of a 7-element array is made equivalent to the first element of another array, the last five elements of the first array overlap the first five elements of the second array.

You must not use the EQUIVALENCE statement to assign the same storage location to two or more elements of the same array. You also must not attempt to assign memory locations in a way that is inconsistent with the normal linear storage of array elements. For example, you cannot make the first element of one array equivalent to the first element of another array and then attempt to set an equivalence between the second element of the first array and the sixth element of the other array.

Some examples of the use of the EQUIVALENCE statement follow:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE (TABLE(2,2), TRIPLE(1,2,2))
```

As a result of these statements, the entire array TABLE shares part of the storage space allocated to array TRIPLE. Table 5–1 shows how these statements align the arrays.

Table 5–1: Equivalence of Array Storage

Array TRIPLE		Array TABLE	
Array Element	Element Number	Array Element	Element Number
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7	TABLE(2,2)	4
TRIPLE(2,2,2)	8		

Each of the following statements also aligns the two arrays as shown in Table 5–1:

```
EQUIVALENCE (TABLE,TRIPLE(2,2,1))
EQUIVALENCE (TRIPLE(1,1,2), TABLE(2,1))
```

You can identify an array element in an EQUIVALENCE statement with a single subscript (that is, with the linear element number), even though the array is multidimensional. For example, the following statement aligns arrays TRIPLE and Table 5–1:

```
EQUIVALENCE (TABLE(4), TRIPLE(7))
```

Similarly, you can make arrays equivalent with nonunity lower bounds. For example, an array defined as A(2:3,4) is a sequence of eight values. A reference to A(2,2) refers to the third element in the sequence. To make array A(2:3,4) share storage with array B(2:4,4), you can use the statement

```
EQUIVALENCE (A(3,4), B(2,4))
```

The whole of array A now shares part of the storage space allocated to array B. Table 5–2 shows how the above statement aligns the arrays.

Table 5–2: Equivalence of Arrays with Nonunity Lower Bounds

Array B		Array A	
Array Element	Element Number	Array Element	Element Number
B(2,1)	1		
B(3,1)	2		
B(4,1)	3	A(2,1)	1
B(2,2)	4	A(3,1)	2
B(3,2)	5	A(2,2)	3
B(4,2)	6	A(3,2)	4
B(2,3)	7	A(2,3)	5
B(3,3)	8	A(3,3)	6
B(4,3)	9	A(2,4)	7
B(2,4)	10	A(3,4)	8
B(3,4)	11		
B(4,4)	12		

5.6.2 Making Substrings Equivalent

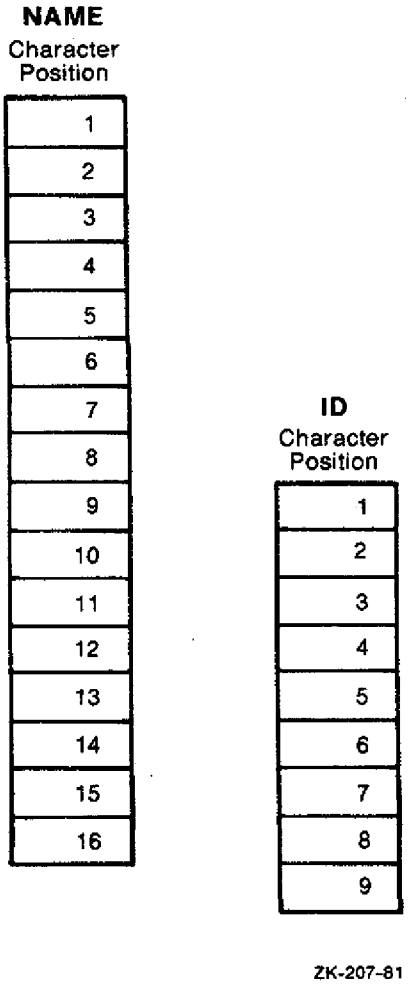
When you make one character substring equivalent to another character substring, the EQUIVALENCE statement also sets equivalences between the other corresponding characters in the character entities.

For example, as a result of statements

```
CHARACTER NAME*16, ID*9
EQUIVALENCE (NAME(10:13), ID(2:5))
```

the character variables NAME and ID share space as illustrated in Figure 5–1.

Figure 5–1: Equivalence of Substrings



The following statement also aligns arrays NAME and ID as they are aligned in Figure 5–1:

```
EQUIVALENCE (NAME(9:9),ID(1:1))
```

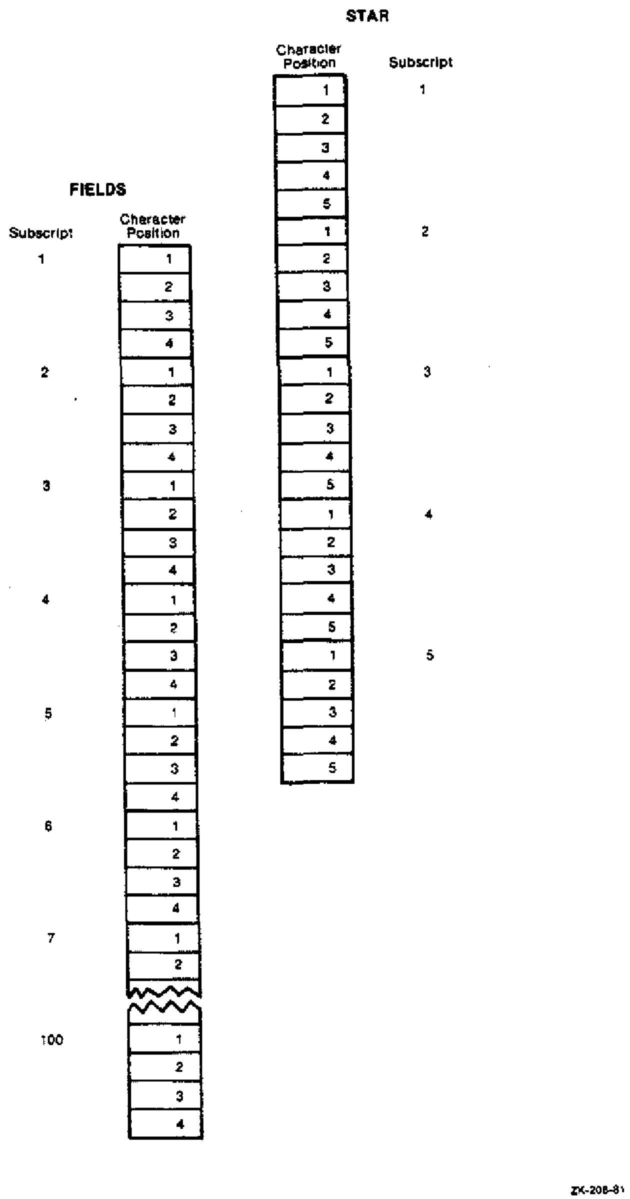
If the character substring references are array elements, the EQUIVALENCE statement sets equivalences between the other corresponding characters in the complete arrays.

Character elements of arrays can overlap at any character position. For example, as a result of statements

```
CHARACTER FIELDS(100)*4, STAR(5)*5
EQUIVALENCE (FIELDS(1)(2:4), STAR(2)(3:5))
```

the character arrays FIELDS and STAR share storage space as shown in Figure 5–2.

Figure 5-2: Equivalence of Character Arrays



You cannot use the EQUIVALENCE statement to assign the same storage location to two or more substrings that start at different character positions in the same character variable or character array.

You also cannot use the EQUIVALENCE statement to assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

The following statements also align the arrays as shown in Table 5–2

```
EQUIVALENCE (A,B(4,1))
EQUIVALENCE (B(3,2), A(2,2))
```

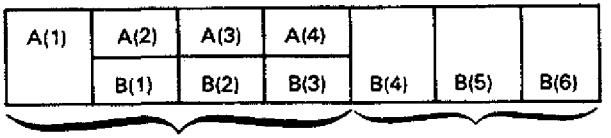
5.6.3 Extending Common Blocks

When you make entities equivalent to entities stored in a common block, the common block can be extended beyond its original boundaries to include the entities specified in the EQUIVALENCE statement. However, you can extend the common block in only one direction. That is, you can only extend it beyond the last element of the previously established common block. You cannot place the extended portion before the first element of the existing common block. The following examples show valid and invalid extensions of the common block:

Figure 5–3: Common Block

Valid

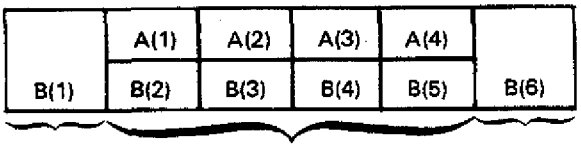
```
DIMENSION A(4),B(6)
COMMON A
EQUIVALENCE (A(2),B(1))
```



ZK-7625-HC

Invalid

```
DIMENSION A(4),B(6)
COMMON A
EQUIVALENCE (A(2),B(3))
```



ZK-7626-HC

If you assign two entities to common blocks, you cannot make them equivalent to each other.

5.7 SAVE Statement

The SAVE statement retains the definition status of an entity after execution of a RETURN or END statement in a subprogram.

The SAVE statement has the form:

```
SAVE [a[,a]...]
```

a

A unique common-block name (preceded and followed by a slash), a variable name, or an array name.

Dummy argument names, procedure names, and names of entities contained in common blocks must not appear in a SAVE statement. If you violate these restrictions, a multiple definition of the name used illegally occurs.

An entity contained in a common block specified in a SAVE statement does not become undefined upon execution of a RETURN or END statement contained in the same program unit. However, it may become undefined (or redefined) in another program unit.

Because a variable, an array element, or a common block contained in one overlay segment can become undefined when this segment is replaced by another overlay segment, the SAVE statement can be especially useful in overlaid programs. To retain the definition of an entity when you are using overlays, you can simply specify that entity in a SAVE statement, within the proper program unit.

A SAVE statement that does not explicitly contain a list is treated as though it contained a list consisting of all allowable items in the program unit in which the SAVE statement resides.

If a common block name is specified in a SAVE statement within a subprogram of an executable program, this common block name must be specified in a SAVE statement in every subprogram in which the common block appears.

The following example demonstrates use of the SAVE statement:

```
DIMENSION A(100)
COMMON /CMN2/B(100),C,D(50)
SAVE A,/CMN2/,E
```

The SAVE statement in this example preserves the current definitions of the array A, the named common block CMN2, and the local variable E.

5.8 EXTERNAL Statement

The EXTERNAL statement allows you to use external subprogram names as arguments to other subprograms.

The subprograms to be used as arguments can never be FORTRAN intrinsic functions; they can only be user-supplied functions and subroutines. The INTRINSIC statement discussed in Section 5.9 allows intrinsic function names to be used as arguments.

The EXTERNAL statement has the form:

```
EXTERNAL v[,v]...
```

v
The symbolic name of user-supplied subprogram, or the name of a dummy argument associated with the name of a subprogram.

The EXTERNAL statement declares each symbolic name included in it to be the name of an external procedure. This name can then be used as an actual argument to a subprogram that can use the corresponding dummy argument in a function reference or a CALL statement.

Note that a complete function reference used as an argument—FUNC(B) in CALL SUBR (A, FUNC(B), C), for example—represents a value, not a subprogram. A complete function reference is not, therefore, defined in an EXTERNAL statement.

The interpretation of the EXTERNAL statement described above is different from that of earlier versions of DIGITAL FORTRAN. See Appendix A for the earlier interpretation.

5.9 INTRINSIC Statement

The INTRINSIC statement allows you to use intrinsic function names as arguments to subprograms. Section C.3 contains the names and descriptions of the individual PDP–11 FORTRAN–77 intrinsic functions; for further information on intrinsic functions, see Chapter 6.

The INTRINSIC statement has the form:

```
INTRINSIC v[,v]...
```

v
The symbolic name of an intrinsic function.

The INTRINSIC statement declares a symbolic name the name of an intrinsic procedure. This symbolic name can then be passed as an actual argument to a subprogram, which can use it in a function reference or a CALL statement.

An example of the use of the INTRINSIC statement follows:

Main Program

```
      .
      .
      .
EXTERNAL CTN
INTRINSIC SIN, COS
      .
      .
      .
CALL TRIG (ANGLE, SIN, SINE)
      .
      .
      .
CALL TRIG (ANGLE, COS, COSINE)
      .
      .
      .
CALL TRIG (ANGLE, CTN, COTANGENT)
      .
      .
      .
```

Subprograms

```
SUBROUTINE TRIG (X,F,Y)
Y=F(X)
RETURN
END

FUNCTION CTN(X)
CTN=COS(X)/SIN(X)
RETURN
END
```

In this example, when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the FORTRAN library functions SIN and COS; but when TRIG is called with a second argument of CTN, F(X) references the user function CTN.

5.10 DATA Statement

The DATA statement assigns initial values to variables, arrays, and array elements before program execution.

The DATA statement has the form:

```
DATA nlist/clist/[[,]nlist/clist/]...
```

nlist
A list of one or more variable names, array names, array element names, or character substring names, separated by commas. Subscript expressions and expressions in substring references must be integer expressions containing integer constants.

clist
A list of constants, separated by commas, to be assigned to nlist. Clist constants have one of the following forms:

```
val
n * val
```

n
The number of times the same value is to be assigned to successive entities in the associated nlist. The value of n is a nonzero, unsigned integer constant or the symbolic name of an integer constant.

Subscript expressions and constant values may be integer constant expressions.

The DATA statement assigns the constant values in each clist to the entities in the preceding nlist. Values are assigned in the order they appear, from left to right.

The number of constants must correspond exactly to the number of entities in the preceding nlist.

When an unsubscripted array name appears in a DATA statement, values are assigned to every element of that array. The associated constant list must therefore contain enough values to fill the array. Array elements are filled in the order of subscript progression.

If both the constant value in the clist and the entity in the nlist have numeric data types, the conversion is based on the following rules:

- The constant value is converted, if necessary, to the data type of the variable being initialized.
- When an octal or hexadecimal constant is assigned to a variable or array element, the number of digits that can be assigned depends on the data type of the component. If the constant contains fewer digits than the capacity of the variable or array element, the constant is extended on the left with zeros. If the constant contains more digits than can be stored, the constant is truncated on the left.

- When a Hollerith or character constant is assigned to a numeric variable or numeric array element, the number of characters that can be assigned depends on the data type of the component (see Table 2–2). If the Hollerith or character constant contains fewer characters than the capacity of the variable or array element, the constant is extended on the right with spaces. If the constant contains more characters than can be stored, the constant is truncated on the right.

If the constant value in the clist and the entity in the nlist are both character data type, the conversion is based on the following rules:

- If the constant contains fewer bytes than the length of the entity, the rightmost character positions of the entity are initialized with spaces.
- If the constant contains more bytes than the length of the entity, the character constant is truncated on the right.

If the constant value is numeric data type and the entity in the nlist is character data type, the constant and the entity must conform to these restrictions:

- The character entity must have a length of one character.
- The constant must be an integer, octal, or hexadecimal constant and must have a value in the range 0 through 255.

When the constant and the entity conform to these restrictions, the entity is initialized with the character that has the ASCII code specified by the constant; a character entity, then, can be initialized to any 8-bit ASCII code.

Dummy arguments, virtual arrays, and virtual array elements may not be initialized in DATA statements.

In the example

```
INTEGER A(10)
BYTE BELL,TAB,LF,FF,ACHR,ZCHR
DATA A, BELL,TAB,LF,FF,ACHR,ZCHR /10*0,7,9,10,12,'A',1HZ/
```

the DATA statement assigns 0 to all 10 elements of array A, and ASCII control character codes to byte variables BELL, TAB, LF, and FF. It assigns values 'A' and 1HZ to ACHR and ZCHR, respectively.

Some other examples of the DATA statement are included in the following segment:

```
CHARACTER*4 STRING
REAL X(6)
COMPLEX Z
DATA X/2*-3.,4.,2*0.37/,Z/(1.0,-3.0)/
DATA STRING/'ABCD'/
```

5.11 PARAMETER Statement

The PARAMETER statement assigns a symbolic name to a constant.

The PARAMETER statement has the form:

```
PARAMETER (p=c[,p=c]...)
```

p
A symbolic name.

c
Any valid FORTRAN constant, the symbolic name of any valid FORTRAN constant, or an integer expression.

Each symbolic name in a PARAMETER statement becomes a constant and is defined to be the value to which it is equated.

The data type of a symbolic name defined to be a constant is determined by the same implicit-typing rules that determine the data type of any other symbolic name, or by a preceding type declaration. Therefore, MU=1.23 in a PARAMETER statement is interpreted as MU=1, unless the PARAMETER statement is preceded by an appropriate type declaration or IMPLICIT statement (for example, REAL*8 MU).

Once a symbolic name is defined to be a constant, it can appear any place in a program that an ordinary constant can appear. The effect of using a symbolic name defined to be a constant is that of using the constant itself.

The symbolic name of a constant cannot appear as part of another constant; however, it can appear as either the real or imaginary part of a complex constant.

You can use a symbolic name defined to be a constant only within the program unit containing PARAMETER statement that defined it. Also, a symbolic name can be defined only once within the same program unit.

The form and the interpretation of the PARAMETER statement described above are different from the form and interpretation of the PARAMETER statement provided in earlier versions of DIGITAL FORTRAN. However, PDP-11 FORTRAN-77 provides both the FORTRAN-77 and the earlier form of the PARAMETER statement; see Appendix A for information on the earlier form and interpretation.

The following sequence demonstrates the use of the FORTRAN-77 PARAMETER statement:

```
INTEGER BYTS1Z, WRDS1Z
REAL*4 PI
REAL*8 DPI
LOGICAL FLAG
CHARACTER*25 LONGNAME
PARAMETER (PI=3.1415927, DPI=3.14159265358979323SD0)
PARAMETER (BYTS1Z=2, WRDS1Z=BYTS1Z/2)
PARAMETER (FLAG=.TRUE.,LNGNAM='A STRING OF 25 CHARACTERS')
```

5.12 PROGRAM Statement

The PROGRAM statement assigns a symbolic name to a main program unit.

The PROGRAM statement has the form:

```
PROGRAM nam
```

nam
A symbolic name.

The PROGRAM statement is optional. If you use it, it must be the first statement in the main program. The symbolic name must not be the name of any entity within the main program. It also must not be the name of any subprogram, entry, or common block in the same executable program (see Section 2.1).

5.13 BLOCK DATA Statement

The BLOCK DATA statement begins a special type of program unit that declares common blocks and defines data in common blocks.

The BLOCK DATA statement has the form:

```
BLOCK DATA [nam]
```

nam
A symbolic name.

You can use only type declaration, IMPLICIT, DIMENSION, COMMON, EQUIVALENCE, and DATA statements between a BLOCK DATA statement and its terminal statement. The last statement in a BLOCK DATA program unit must be an END statement.

A BLOCK DATA program unit must not contain any executable statements and must not have a statement label.

If you initialize any entity in a common block declared in a BLOCK DATA program unit, you must provide a complete set of data-type specification statements for all the entities in the block, even though some of the entities are not assigned an initial value. You can use the same BLOCK DATA program unit to define initial values for more than one common block.

An example of a BLOCK DATA program unit follows:

```
BLOCK DATA BLKDAT
INTEGER S,X
LOGICAL T,W
DOUBLE PRECISION U
DIMENSION R(3)
COMMON /AREA1/R,S,T,U/AREA2/W,X,Y
DATA R/1.0,2*2.0/,T/.FALSE./,U/0.214537D-7/,W/.TRUE./,Y/3.5/
END
```

In this example, enough information is provided to declare explicitly or implicitly the data type of every variable in the common blocks AREA1 and AREA2. Not all the variables appear in the DATA statement.

Chapter 6

Subprograms

A subprogram is a statement or group of statements that defines a computing procedure. A subprogram is invoked with a referencing statement. This referencing statement can be located either in the same program unit as the subprogram or in a different program unit.

There are two kinds of subprograms: user written and system supplied. User-written subprograms consist of statement functions, functions, and subroutines; system-supplied subprograms consist of intrinsic functions and generic functions.

In many cases, a program referencing a subprogram passes values, called actual arguments, to that subprogram for it to use in making computations. The subprogram specifies entities, called dummy arguments, to receive these actual arguments.

Section 6.1 describes actual and dummy arguments; Section 6.2 describes user-written subprograms; and Section 6.3 describes system-supplied subprograms.

6.1 Subprogram Arguments

A subprogram argument is an entity that passes a value to or from a subprogram. There are two kinds of arguments: actual and dummy. Actual arguments are specified in the statement referencing the subprogram. Dummy arguments are specified in the definition of the subprogram and, when control is transferred to the subprogram, are associated with actual arguments on a one-to-one basis. Each dummy argument takes on the value of the corresponding actual argument; in turn, any value assigned to the dummy argument in the subprogram is assigned to the corresponding actual argument. When control is returned to the main program from

the subprogram, the association of actual and dummy arguments ends: there is no retention of argument association from one reference of a subprogram to the next.

If (I,J(3),4) is a list of actual arguments and (K,L,M) is an associated list of dummy arguments, K is associated with I, L is associated with J(3), and M is assigned a value of 4.

6.1.1 Rules Governing Subprogram Arguments

Actual arguments can be constants, variables, expressions, arrays, array elements, substrings, or subprogram names. Actual arguments must agree in order, number, and data type with the dummy arguments with which they are associated.

Dummy arguments are symbolic names that become associated with variables or arrays, or with subprograms defined or declared in other program units; they are not in themselves variables or arrays or subprograms. A dummy argument is undefined if it is not currently associated with an actual argument.

Although dummy arguments are not variables, arrays, or subprograms, each dummy argument may be declared as though it were a variable, array, or subprogram. Each dummy argument name is declared to have the attributes of its associated actual argument.

If the actual argument is a constant, an expression, a subprogram name, or a virtual array element reference, the corresponding dummy argument may not be modified.

A dummy argument declared to be an array can be associated only with an actual argument that is an array or array element of the same data type. If the actual argument is an array, the dummy argument array must not be larger than the actual argument array; that is, it can be equal to or smaller than the number of elements in the actual argument.

If an actual argument is an element of an array, this element and succeeding elements of the array are associated with elements of the corresponding dummy argument array. The number of actual argument array elements associated depends on the size of the dummy argument array. The dummy argument array must not be larger than the number of elements in the actual argument array involved in the reference; that is, it can be equal to or smaller than the number of elements in the actual argument.

Valid

```
PROGRAM MAIN
DIMENSION A(10), B(5,5)
    .
    .
    .
CALL X(A, B(1,21))
END
SUBROUTINE X(Y,Z)
DIMENSION Y(10), Z(5,2)
END
```

Invalid

```
PROGRAM MAIN
DIMENSION A(10), B(5,5)
    .
    .
    .
CALL X(A,B(1,21))
END
SUBROUTINE X(C,D)
DIMENSION C(12)   (dummy array must
                  not be larger than
                  actual array)

DIMENSION D(5,5)  (dummy array must
                  not be larger than
                  number of elements
                  of actual array
                  included)
```

6.1.2 Adjustable Arrays

An adjustable array is a dummy argument array, declared in a subprogram, whose dimensions can be changed, or "adjusted," to match the dimensions of an associated actual argument array in a referencing program. The dimension declaration of a dummy argument array contains one or more integer variables and, optionally, an asterisk. (See Section 6.1.3 for information on the use of the asterisk.)

The following rules govern the use of adjustable arrays:

- An adjustable array must be a dummy argument.
- An adjustable array must become associated with an actual argument that is an array.
- The size of an adjustable array must be less than or equal to the size of a corresponding actual array.
- Variables in an adjustable array declarator must be dummy arguments, and the corresponding actual arguments must have a defined value.
- Variables in an adjustable array declarator must become defined; you can assign values to these variables through dummy arguments or through common blocks.
- Variables in an adjustable array declarator may be of any data type; assigned values of other than integer data type are converted to integer data type before use.

The following examples demonstrate the use of adjustable arrays:

```
      PROGRAM MAIN
      DIMENSION A1(10,35), A2(3,56)
      SUM1 = SUM(A1,10,35)
      SUM2 = SUM(A2,3,56)
      SUM3 = SUM(A1,10,10)
        .
        .
        .
      END
```

```
      FUNCTION SUM(A,M,N)
      DIMENSION A(M,N)
      SUM = 0.0
      DO 10 J = 1,N
      DO 10 I = 1,M
10    SUM = SUM + A(I,J)
      RETURN
      END
```

or

```
      FUNCTION SUM(A,M,N)
      DIMENSION A(M,*)
      SUM = 0.0
      DO 10 J = 1,N
      DO 10 I = 1,M
10    SUM = SUM + A(I,J)
      RETURN
      END
```

In this example, A1 and A2 are actual arrays and A is the adjustable array. The function subprogram computes the sum of specified sections of A1 or A2. Note that the dummy arguments M and N are used to control the DO statement iteration as well as to specify the size of A.

For more information on array declarators, see Section 2.5.1.

Upper-and lower-bound values can be specified for an adjustable array. These values do not change during subprogram execution, even if the values of variables contained in the array declaration are changed. For example:

```
      DIMENSION ARRAY (11,5)
      L = 9
      M = 5
      CALL SUB(ARRAY,L,M)
      END

      SUBROUTINE SUB(X,I,J)
      DIMENSION X(-I/2:I/2,J)
      J = 1
      I = 2
      END
```

In this example, the adjustable array X is declared to be X(-4:4,5); the subsequent assignments to I and J do not affect this declaration.

Note that argument association is not retained in the interim between one reference to a subprogram and the next.

```
REAL B
DIMENSION B(10)
CALL S(B,2,3.0)
CALL S1(5,B,3,2)
   .
   .
   .
SUBROUTINE S(A,I,J)
DIMENSION A(I)
A(I) = J
RETURN
ENTRY S1 (I,A,K,L)
A(I) = A(I) + 1
RETURN
END
```

In this example, B is declared to be a real array with 10 elements by the statement

```
DIMENSION B(10)
```

The statement

```
CALL S(B,2,3)
```

sets B(2) = 3; the next statement

```
CALL S1(5,B,3,2)
```

increments B(5) by 1, but only because it provides actual argument A, which was not retained in the subroutine after the first reference.

6.1.3 Assumed-Size Dummy Arrays

An assumed-size dummy array is a dummy array (argument) for which the upper bound of the last dimension is specified as *. For example:

```
SUBROUTINE SUB(A,N)
DIMENSION A(N,*)
   .
   .
   .
```

The size of an assumed-size array and the number of elements that can be referenced are determined as follows:

- If the actual argument corresponding to the dummy array is a non-character array name, the size of the dummy array is the size of the actual argument array.

- If the actual argument corresponding to the dummy argument is a noncharacter array element name, with a subscript value of s in an array of size a, the size of the dummy array is a+1–s.
- If the actual argument is a character array name, character array element name, or character array element substring name, and begins at character storage unit b of an array with n character storage units, the size of the dummy array is INT(n+1–b)/y, where y is the length of an element of the dummy array.

Because the actual size of an assumed-size array is not known, an assumed-size array name cannot be used as any of the following:

- An array name in the list of an I/O statement
- A unit identifier for an internal file in an I/O statement
- A run-time format specifier in an I/O statement
- A key specifier in an I/O statement
- A buffer specifier for ENCODE/DECODE statements

6.2 User-Written Subprograms

A user-written subprogram is a statement or group of statements that performs a computing procedure. A computing procedure can be a series of either arithmetic operations or FORTRAN statements.

User-written subprograms are useful in avoiding having to duplicate the same series of operations or statements in two or more different locations in a single program.

There are three types of user-written subprograms. Table 6–1 lists each type, the statements needed to define each type, and the method used to transfer control to each type.

Table 6–1: Types of User_Written Subprograms

Subprogram	Defining Statements	Control Transfer Method
Statement function	Statement-function definition	Function reference
Function subprogram	FUNCTION ENTRY RETURN	Function reference
Subroutine subprogram	SUBROUTINE ENTRY RETURN	CALL statement

A function reference (Table 6–1) consists of a function name and function arguments, and is used in an expression. The CALL statement is discussed in Section 4.5.

Function and subroutine subprograms can change the values of their arguments, and the calling program can use these changed values.

A subprcgram can refer to other subprograms but it cannot, either directly or indirectly, refer to itself.

6.2.1 Statement Functions

A statement function is a single-statement computation specified by a symbolic name. When you reference a statement function name in an expression, the computation defined by the statement function name is performed and the value produced is used to replace the statement function name in the expression. Statement functions are defined and referenced within a single program unit.

A statement function has the form:

```
f ([p[,p]...])=e
```

f
The name of a statement function.

p
A dummy argument.

e
An expression.

The expression (*e*) is an arithmetic or logical expression that defines the computation to be performed.

A reference to a statement function has the form:

```
f ([a[,a]...])
```

f
The name of the function.

a
An actual argument.

When a statement function reference appears in an expression, the values of the actual arguments are associated with the dummy arguments in the statement function. The expression in the statement function is then evaluated, and the result is used to complete the evaluation of the expression containing the reference.

The following rules govern the use of statement functions:

- A statement function may not return a value of type CHARACTER.
- Statement function names must be unique within the same program unit.
- A statement function reference must appear in the same program unit as the statement function.
- Statement functions can include a reference to another statement function (defined earlier in the same program unit).
- Statement functions must be placed before all executable statements (see Figure 1–3).
- The data type of a value computed by a statement function is determined either by the first letter of the function name or by a type declaration statement.
- Statement function dummy arguments serve only to indicate order, number, and data type of arguments for the statement function.
- Names of statement function dummy arguments must be unique only within each statement function. Variables or arrays having the same names as dummy arguments can be declared and used within the same program unit.

- The data type of statement function dummy arguments is determined either by the first letter of the argument name or by a type declaration statement.
- A statement function cannot be used as an EXTERNAL argument in a subroutine.

Examples of valid and invalid statement functions are:

Valid

```
VOLUME(RADIUS) = 4.189*RADIUS**3
AVG (A,B,C) = (A+B+C)/3
SINH (X) = (EXP(X) - EXP (-X))*0.5
```

Invalid

```
AXG(A,B,C,3.) = (A+B+C)/3      (a constant cannot be a dummy argument)
```

The examples of statement function references below refer to the second valid statement function above.

Valid

```
GRADE = AVG (TEST1,TEST2,XLAB)
IF (AVG (P,D,Q).LT.AVG(X,Y,Z))30 TO 300
```

Invalid

```
FINAL = AVG(TEST3,TEST4,LAB2)      (LAB2 is integer, but C is real)
```

6.2.2 Function Subprograms

A function subprogram consists of a FUNCTION statement followed by a series of statements that make up a computing procedure. It is invoked with a function reference.

The FUNCTION statement has the following form:

```
[typ] FUNCTION nam[*m][([p[,p]...])]
```

typ
Any data type specifier except CHARACTER (see Table 2-2).

nam
The name of a function.

m
A data type length specifier (see Table 2-2).

p
A dummy argument.

The function reference that invokes, or transfers control to, a function subprogram has the form:

```
nam ([a[,a]...])
```

nam
The symbolic name of the function.

a
An actual argument.

When a function reference in an expression is executed, control is transferred to the referenced subprogram and the values of the actual arguments (if any) in the function reference are associated with the dummy arguments in the FUNCTION statement of the subprogram. The statements in the subprogram are then executed and a computed value is assigned to the function name (as if this name were a variable). Finally, a RETURN statement is executed in the function and control is returned to the calling program unit. (An END statement used in place of a RETURN acts as an implied RETURN.) The value assigned to the function name is now used to complete the evaluation of the expression containing the name.

The following rules govern the use of function subprograms:

- A function may not return a value of type CHARACTER.
- A FUNCTION statement must be the first statement of a function subprogram.
- A FUNCTION statement must not have a statement label.
- A function subprogram must not contain the following statements: SUBROUTINE, BLOCK DATA, or FUNCTION.
- A function subprogram can reference another subprogram, but it cannot reference itself, either directly or indirectly.
- The data type of a function name can be specified either in the FUNCTION statement or in a type declaration statement.
- A function name must have the same data type in a subprogram as in a referencing program, and vice versa.

- ENTRY statements can be included in a function subprogram to provide one or more other entry points to the subprogram (see Section 6.2.4).

An example of a function subprogram is the function ROOT:

```
      FUNCTION ROOT(A)
      X = 1.0
    2 EX = EXP(X)
      EMINX = 1./EX
      ROOT = ((EX+EMINX)*.5+COS(X)-A)/((EX - EMINX)*.5-SIN(X))
      IF (ABS(X-ROOT).LT.1E-6) RETURN
      X = ROOT
      GO TO 2
      END
```

The function in this example uses the Newton-Raphson iteration method to obtain the root of the following function:

```
F(X) = cosh(X) + cos(X) - A = 0
```

The value of A is passed as an argument. The iteration formula for this root is:

$$Xi+1 = Xi - \frac{\cosh(Xi)+\cos(Xi)-A}{\sinh(Xi)-\sin(Xi)}$$

The calculation is repeated until the difference between Xi and Xi+1 is less than 1.0E–6.

The function uses the FORTRAN library functions EXP, SIN, COS, and ABS (see Section 6.3).

6.2.3 Subroutine Subprograms

A subroutine subprogram is a computing procedure referenced by a symbolic name in a CALL statement. A subroutine subprogram consists of a SUBROUTINE statement followed by a series of statements.

The SUBROUTINE statement has the form:

```
SUBROUTINE nam [([p[,p]...])]
```

nam
The name of the subroutine.

p
A dummy argument.

You must use a CALL statement to transfer control to a subroutine subprogram, and a RETURN statement to return control to the calling program unit. Section 4.5 describes the CALL statement.

When control is transferred to a subroutine, the values of the actual arguments (if any) in the CALL statement are associated with corresponding dummy arguments in the SUBROUTINE statement. The statements in the subprogram are then executed until a RETURN statement returns control to the calling program. (An END statement acts as an implied RETURN.) Unlike a function, a subroutine does not return a value to the referencing program.

The following rules govern the use of subroutine subprograms:

- The SUBROUTINE statement must be the first statement of a subroutine.
- A subroutine subprogram must not contain a FUNCTION, a BLOCK DATA, or another SUBROUTINE statement.
- A subroutine subprogram can reference another subprogram, but it cannot reference itself, either directly or indirectly.
- ENTRY statements can be included in a subroutine subprogram to provide one or more other entry points to the subprogram (see Section 6.2.4).

The subroutine in the following example computes the volume of a regular polyhedron, given the number of faces and the length of one edge. It uses the computed GO TO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron. The GO TO statement also transfers control to the proper procedure for calculating the volume. If the number of faces is not 4, 6, 8, 12, or 20, the subroutine displays an error message on the user's terminal.

Example:

```
      Main Program

      COMMON NFACES,EDGE,VOLUME
      ACCEPT *, NFACES,EDGE
      CALL PLYVOL
      TYPE *, 'VOLUME=',VOLUME
      STOP
      END

      Subroutine

      SUBROUTINE PLYVOL
      COMMON NFACES,EDGE,VOLUME
      CUBED = EDGE**3
      GOTO (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,6,5),NFACES
      GOTO 6
    1 VOLUME = CUBED * 0.11785
      RETURN
    2 VOLUME = CUBED
      RETURN
    3 VOLUME = CUBED * 0.47140
      RETURN
    4 VOLUME = CUBED * 7.66312
      RETURN
    5 VOLUME = CUBED * 2.18170
      RETURN
    6 TYPE 100, NFACES
  100 FORMAT(' NO REGULAR POLYHEDRON HAS ',I3, ' FACES.'/)
      VOLUME=0.0
      RETURN
      END
```

6.2.4 ENTRY Statement

The ENTRY statement is a nonexecutable statement that provides multiple entry points to a subprogram. It can appear within a function or subroutine subprogram after the FUNCTION or SUBROUTINE statement. Execution in a subprogram containing an ENTRY statement begins with the first executable statement following the ENTRY statement.

The ENTRY statement has the form:

```
ENTRY nam [([p[,p]...])]
```

nam
The entry name.

p
A dummy argument.

CALL statements are used to refer to entry names within subroutine subprograms; function references are used to refer to entry names within function subprograms.

The following rules govern the use of ENTRY statements:

- Within a function subprogram, an entry name can appear in a type declaration statement.
- You can specify an entry name in an EXTERNAL statement and then use entry name as an actual argument (but not as a dummy argument).
- You must not use an entry name in executable statements (in a subprogram) that precede or follow an ENTRY statement.
- You can use dummy arguments in ENTRY statements that differ in order, number, type, and name from the dummy arguments you use in the FUNCTION, SUBROUTINE, and ENTRY statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.
- A dummy argument can be referred to only in the executable statements that follow the first SUBROUTINE, FUNCTION, or ENTRY statement in which the dummy argument is specified.
- You must not use an ENTRY statement within a DO loop.

6.2.4.1 ENTRY in Function Subprograms

The name of a function subprogram and all the entry names contained in the subprogram are mutually associated; therefore, a value assigned to any one name is assigned to all the names. However, only names of the same data type can be mutually defined at any one time, because conversions between data types are not made.

A referenced entry name must be assigned a value before control is transferred back to the calling program. Example 6–1 illustrates the use of an ENTRY statement in a function subprogram that computes the hyperbolic functions sinh, cosh, and tanh of a variable x.

Example 6–1: Multiple Functions in a Function Subprogram

```
PROGRAM MAIN
EXTERNAL TANH, SINH, COSH
    .
    .
    .
X = 24.0
TANHX = TANH (X)
SINHX = SINH (X)
COSHX = COSH (X)
    .
    .
    .
END
   REAL FUNCTION TANH(X)
C
C  STATEMENT FUNCTION TO COMPUTE TWICE SINH
C
   TSINH(X) = EXP(X) - EXP (-X)
C
C  STATEMENT FUNCTION TO COMPUTE TWICE COSH
C
   TCOSH(X) = EXP(X) + EXP(-X)
C
C  COMPUTE TANH
C
   TANH = TSINH(X) / TCOSH(X)
   RETURN
C
C  COMPUTE SINH
C
   ENTRY SINH(X)
   SINH = TSINH(X) / 2.0
   RETURN
C
C  COMPUTE COSH
C
   ENTRY COSH(X)
   COSH = TCOSH(X) / 2.0
   RETURN
   END
```

6.2.4.2 ENTRY in Subroutine Subprograms

To reference an entry point in a subroutine, you execute a CALL statement that includes the entry point name. The following example demonstrates the use of the CALL statement to reference an entry point:

```
Main Program
CALL SUBA(A,B,C)
      .
      .
      .
Subroutine
SUBROUTINE SUB (X,Y,Z)
      .
      .
      .
ENTRY SUBA(Q,R,S)
```

In this example, the CALL is to an entry point (SUBA) within the subroutine (SUB). Execution begins with the first statement following ENTRY SUBA (Q,R,S), using the actual arguments (A,B,C) passed in the CALL statement.

6.3 Intrinsic and Other Library Functions

FORTRAN library functions consist of intrinsic functions, provided to perform commonly used mathematical computations, and character and lexical comparison functions. Character and lexical comparison functions are discussed in Section 6.3.4.

The FORTRAN intrinsic functions are listed in Appendix C. Function references to these functions are written in the same way function references to user-defined functions are written. For example, as a result of the reference to ABS in

```
R = 3.14159 * ABS(X-1)
```

the absolute value of X–1 is calculated and multiplied by the constant 3.14159, and the result is assigned to the variable R.

Appendix C gives the data type of each intrinsic function and that of its actual arguments.

6.3.1 Intrinsic Function References

Normally, a name in the table of intrinsic function names (Table C-2) refers to the FORTRAN library function with that name. However, the name can refer to a user-defined function under any of the following conditions:

- The name is used in a function reference with arguments of a different data type from that shown in the table.
- The name appears in an EXTERNAL statement in accordance with the rules provided in Section 5.8.

Except when they are used in an EXTERNAL statement, intrinsic function names are local to the program unit that refers to them. Thus, they can be used for other purposes in other program units. In addition, the data type of an intrinsic function does not change if you use an IMPLICIT statement to change the implied data type rules.

You cannot have an intrinsic function and a user-defined function with the same name in the same program unit.

6.3.2 Generic Function References

Some intrinsic functions perform the same computation but handle different data types. These functions are referenced with the same categorical, or generic, name. A generic function reference refers to the category of the computation to be performed, not to a specific function within the category. The selection of a specific function—that is, the actual computing procedure for a specific data type—is left to the compiler, which chooses a specific function within a category on the basis of the data type of the relevant actual argument. For example, if D is a double-precision variable, the generic function SIN(D) refers to the double-precision sine function, not to the real sine function. (Therefore, you need not write DSIN(D).)

Generic function references are independent from one another. Therefore, you could use both SIN(X) and SIN(D) in the same program unit in the example in the preceding paragraph.

Table 6–2 lists the generic function names. These names can be used only with the argument data types shown in the table.

You cannot use the names in Table 6–2 for generic function selection if you use them in a program unit in either of the following ways:

- As the name of a statement function
- As a dummy argument name, common block name, variable name, or array name

Generic function selection does not apply to a generic function name declared in an EXTERNAL statement and used as an actual argument, because there is no argument list on which to base the function selection. The name is treated according to the rules for nongeneric FORTRAN functions described in Section 6.3.1. For example, in

```
EXTERNAL EXP
CALL SUB (EXP(D))
```

EXP (D) is a generic function reference, not a generic function name; therefore, generic function selection applies. However, in

```
EXTERNAL SQRT
CALL SUB (SQRT)
```

SQRT is a generic function name being used as a nongeneric function; therefore, generic function selection does not apply.

Generic function names are local to the program unit that references them. Therefore, they can be used for other purposes in other program units.

Table 6–2: Generic Function Name Summary

Generic Name	Data Type of Argument	Data Type of Result
ABS	Integer	Integer
	Real	Real
	Double	Double
	Complex	Real
AINT, ANINT	Real	Real
	Double	Double
INT,NINT	Real	Integer
	Double	Integer

Table 6–2 (Cont.): Generic Function Name Summary

Generic Name	Data Type of Argument	Data Type of Result
REAL	Integer	Real
	Real	Real
	Double	Real
	Complex	Real
DBLE	Integer	Double
	Real	Double
	Double	Double
MOD, MAX, MIN, SIGN, DIM	Integer	Integer
	Real	Real
	Double	Double
EXP, LOG, SIN COS, SQRT	Real	Real
	Double	Double
	Complex	Complex
LOG10, TAN, ATAN, ATAN2, ASIN, ACOS, SINH, COSH, TANH	Real	Real
	Double	Double

6.3.3 Intrinsic and Generic Function Usage

Example 6–2 demonstrates the use of intrinsic and generic function names. In this example, a single executable program uses the name SIN in four distinct ways:

- As the name of a statement function
- As a generic function name
- As an intrinsic function name
- As a user-defined function

Using the name in these four ways emphasizes the local and global properties of the name.

In Example 6–2, the parenthetical references are keyed to the notes that follow the example.

Example 6–2: Multiple Function Name Usage

```
C
C     COMPARE WAYS OF COMPUTING SINE.
C
      PROGRAM SINES
      REAL*8 X, PI
      PARAMETER (PI = 3.14159265358979323SD0)
      COMMON V(3)
C     DEFINE SIN AS A STATEMENT FUNCTION  (Note 1)
      SIN(X) = COS(PI/2-X)
      DO 10 X = -PI, PI, 2*PI/100
      CALL COMPUT(X)
C     REFERENCE THE STATEMENT FUNCTION SIN  (Note 2)
10    WRITE(6,100) X,V, SIN(X)
100   FORMAT (5('     ',F10.7))
      END
C
C
      SUBROUTINE COMPUT(Y)
      REAL*8 Y
C     USE INTRINSIC FUNCTION SIN AS ACTUAL ARGUMENT (Note 3)
      INTRINSIC SIN
      COMMON V(3)
C     GENERIC REFERENCE TO DOUBLE PRECISION SINE (Note 4)
      V(1) = SIN(Y)
C     INTRINSIC FUNCTION SINE AS ACTUAL ARGUMENT (Note 5)
      CALL SUB(REAL(Y), SIN)
      END
C
C
     SUBROUTINE SUB(A,S)
C    DECLARE SIN AS NAME OF USER FUNCTION (Note 6)
     EXTERNAL SIN
C    DECLARE SIN AS TYPE REAL*8 (Note 7)
     REAL*8 SIN
      COMMON V(3)
C    EVALUATE INTRINSIC FUNCTION SIN (Note 8)
     V(2) = S(A)
C    EVALUATE USER DEFINED SIN FUNCTION (Note 9)
     V(3) = SIN(A)
     END
```

Example 6–2 (Cont.): Multiple Function Name Usage

```
C
C
C   DEFINE THE USER SIN FUNCTION (Note 10)
    REAL*8 FUNCTION SIN(X)
    INTEGER FACTOR
    SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)
   1        - X**7/FACTOR(7)
    END

    INTEGER FUNCTION FACTOR(N)
    FACTOR = 1
    DO 10 I=N, 1, -1
  10 FACTOR = FACTOR * I
    END
```

1. A statement function named SIN is defined in terms of the generic function name COS. Because the argument of COS is double precision, the double-precision cosine function is evaluated. The statement function SIN is itself single precision.
2. The statement function SIN is called.
3. The name SIN is declared intrinsic so that the single-precision intrinsic sine function can be passed as an actual argument at 5.
4. The generic function name SIN is used to refer to the double-precision sine function.
5. The single-precision intrinsic sine function is used as an actual argument.
6. The name SIN is declared a user-defined function name.
7. The type of SIN is declared double precision.
8. The single-precision sine function passed at 5 is evaluated.
9. The user-defined SIN function is evaluated.
10. The user-defined SIN function is defined as a simple Taylor series using user-defined function FACTOR to compute the factorial function.

6.3.4 Character and Lexical Comparison Library Functions

Character library functions are functions that take character arguments; lexical comparison library functions are functions that take character arguments and return logical values.

Three character functions are provided with PDP–11 FORTRAN–77, as follows:

- LEN

  The LEN function returns the length of a character expression. The LEN function has the form:

  ```
  LEN(c)
  ```

  c
  A character expression. The value returned indicates how many bytes there are in the expression.

- INDEX

  The INDEX function searches for a substring (c2) in a specified character string (c1) and, if it finds the substring, returns the substring's starting position. If c2 occurs more than once in c1, the starting position of the first (leftmost) occurrence is returned. If c2 does not occur in c1, the value zero is returned. The INDEX function has the form:

  ```
  INDEX (c1, c2)
  ```

  c1
  A character expression specifying the string to be searched for the substring specified by c2.

  c2
  A character expression specifying the substring for which the starting location is to be determined.

- ICHAR

  The ICHAR function converts a character expression to its equivalent ASCII code and returns the ASCII value. ICHAR has the form:

  ```
  ICHAR (c)
  ```

c
The character to be converted to an ASCII code. If c is longer than one byte, only the value of the first byte is returned; the remainder is ignored.

An example illustrating the LEN and INDEX functions follows:

```
          CHARACTER BUFR*80
          INTEGER COMPOS,INIPOS
 1        COMPOS = INDEX(BUFR(INIPOS:),',')
          IF (LEN(BUFR(INIPOS:COMPOS)) .GT. 8) THEN
              TYPE *,'NAME IS TOO LONG, IT HAS BEEN TRUNCATED.'
          ENDIF
```

Four lexical comparison functions are provided with PDP–11 FORTRAN–77, as follows:

- LLT, where LLT(X,Y) is equivalent to (X .LT. Y)
- LLE, where LLE(X,Y) is equivalent to (X .LE. Y)
- LGT, where LGT(X,Y) is equivalent to (X .GT. Y)
- LGE, where LGE(X,Y) is equivalent to (X .GE. Y)

The lexical functions have the form

```
func(c,c)
```

func
One of the symbolic names: LLT, LLE, LGT, or LGE.

c
A character expression.

The lexical library functions are guaranteed to make comparisons according to the ASCII collating sequence, even on non-ASCII processors. On PDP–11 systems, the lexical library functions are identical to the corresponding character relationals.

An example of the use of the lexical library functions follows:

```
CHARACTER*10 CH2
IF (LGT(CH2, 'SMITH')) STOP
```

The IF statement in this example is equivalent to:

```
IF (CH2 .GT. 'SMITH') STOP
```

Chapter 7

Input/Output Statements

FORTRAN programs use READ and ACCEPT statements for input, and WRITE, REWRITE, TYPE, and PRINT statements for output.

Some forms of these statements are used with format specifiers that control the translation and editing of data between internal (binary) form and external (readable character) form.

The READ and WRITE statements reference a logical unit to or from which data is to be transferred. The ACCEPT, TYPE and PRINT statements do not reference a logical unit; rather, they transfer data between a program and an implicit logical unit (the user's terminal for example). Normally the ACCEPT and TYPE statements are connected to the user's terminal; the PRINT statement, to the system line printer.

Input/output (I/O) statements are grouped into four categories:

- Sequential I/O—transfers records sequentially to and from files, or to and from an I/O device such as a terminal.
- Direct Access I/O—transfers records selected by record number to and from direct-access files.
- Indexed I/O—transfers records selected by data values contained in the records to and from indexed files.
- Internal I/O—translates and transfers data between variables and arrays within a program.

The I/O statement forms can be classified as formatted, list-directed, or unformatted.

Formatted I/O statements contain explicit format specifiers that are used to control the translation of data from internal (binary) form within a program to external (readable character) form in records, or vice versa.

List-directed I/O statements are similar to formatted statements in function, but differ in that they use data types instead of explicit format specifiers to control the translation of data from one form to the other.

Unformatted I/O statements do not contain format specifiers of any kind and therefore are not used to translate data being transferred. Unformatted I/O saves execution time, by eliminating data translation; preserves the precision of external data; and usually conserves file storage space. Unformatted I/O is especially useful when data to be output is subsequently to be used as input.

Table 7–1 shows the various I/O statements, by category, that can be used in PDP–11 FORTRAN–77 programs.

Table 7–1: Available I/O Statements

	Statement Category			
Statement Name	**Sequential**	**Direct**	**Indexed**	**Internal**
	F L U	F U	F U	F
READ	X X X	X X	X X	X
WRITE	X X X	X X	X X	X
REWRITE	- - -	- -	X X	-
ACCEPT	X X -	- -	- -	-
TYPE	X X -	- -	- -	-
PRINT	X X -	- -	- -	-

F—Formatted
L—List-Directed
U—Unformatted

I/O statements transfer data in units of records (see Section 7.1.1). The amount of data that one of these records can contain depends on whether unformatted or formatted I/O is used to transfer the data. With unformatted I/O, the I/O statement alone specifies the amount of data to be transferred; with formatted I/O, the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.

Normally, the data transferred by an I/O statement is read from or written to only one record. It is possible, however, for formatted I/O statements to transfer data from or to more than one record.

Section 7.1 describes general FORTRAN input/output concepts.
Section 7.2 describes the components of FORTRAN I/O statements.
Section 7.3 describes the syntactical rules that govern the I/O statements.
Sections 7.4 through 7.8 describe the individual I/O statements in detail.

7.1 I/O Overview

The following sections describe in general terms the characteristics of FORTRAN I/O processing: records, files, internal files, and access modes. See the *PDP-11 FORTRAN-77 User's Guide* for specific detail on FORTRAN I/O processing.

7.1.1 Records

A record is a collection of data items, called fields, that are logically related and that are processed as a unit; that is, the I/O statements transfer data to and from files and internal files in units of records. Normally, each I/O statement processes one record, though formatted I/O statements may transfer more than one record.

If an input statement does not use all the data fields it reads from a record, the remaining fields are ignored. If an input statement requires more data fields than the record contains, either an error condition occurs or, in the case of formatted input, all fields are read as spaces.

If an output statement attempts to write more data fields than the record can contain, an error condition occurs. If an output statement transfers fewer data than required to fill a fixed-length record, the record is filled with spaces (if a formatted record) or zeros (if an unformatted record).

7.1.2 Files

A file is a collection of logically related records arranged in a specific order and treated as a unit. The arrangement or organization of a file is determined when the file is created.

A file can have one of three possible arrangements or organizations: sequential, relative, or indexed.

Files are normally stored on disk; however, sequential files may be stored on magnetic tape. Peripheral devices such as terminals, card readers, and line printers are treated as sequential files.

7.1.2.1 Sequential Organization

In a sequential file, records appear in physical sequence. The physical order in which records appear is always identical to the order in which the records are written to the file.

7.1.2.2 Relative Organization

A relative file consists of a sequence of fixed-length cells numbered from 1 (the first) to n (the last). A cell's number represents its location relative to the beginning of the file. A cell can contain a single record or it can be empty. The cell number, or record number, is used to refer to a specific record in a relative file.

7.1.2.3 Indexed Organization

Records in an indexed file are ordered—not necessarily in physical sequence—by fields in the records that have been designated to be keys.

A key is a data field that is contained, in the same relative position, in all the records in an indexed file. When creating an indexed file, you decide which data field in the file's records is to be a key; the contents of this field in any one record are then used to identify that record for subsequent processing. The length of a key field, as well as its relative position, is the same in each of the records in a file.

You must define at least one key for an indexed file. This mandatory key is the primary key of the file. Optionally, you can define additional keys called alternate keys. Each alternate key represents an additional field that is contained in all the records in a file. The key value in any one of these additional fields can be used to identify the record containing them for retrieval. More than one record can have the same key value.

7.1.3 Internal Files

An internal file is internal storage space that is manipulated to facilitate internal I/O.

An internal file is not a real file; it consists of a character variable, a character array element, a character array, or a character substring. A record in an internal file consists of any of the above except a character array.

If an internal file is a single character variable, array element, or substring, this file comprises a single record whose length is the same as the length of the variable, array element, or substring. If an internal file is a character array, this file comprises a sequence of records, with each record consisting of a single array element. The sequence of records in an internal file is determined by the order of subscript progression.

A record in an internal file can be read only if the character variable, array element, or substring comprising the record has been defined—that is, assigned a value.

Prior to data transfer, an internal file is always positioned at the beginning of the first record.

7.1.4 Access Modes

Access mode is the method a program uses to retrieve and store records in a file. The access mode is specified as part of each I/O statement. PDP-11 FORTRAN-77 supports three access modes: sequential, direct, and keyed.

Table 7-2 shows the valid access modes for each file organization.

Table 7-2: Access Modes for Each File Organization

	Access Mode		
File Organization	**Sequential**	**Direct**	**Keyed**
Sequential	Yes	Yes[1]	No
Relative	Yes	Yes	No
Indexed	Yes	No	Yes

[1]Records must be fixed length.

7.1.4.1 Sequential Access

Sequential access means that records are processed in physical, numerical, or chronological order. In a sequential file, this processing order is the physical sequence of the records; in a relative file, it is the sequence of ascending cell numbers; and in an indexed file, it is the sequence of ascending key values.

If two records in an indexed file have the same key value, the processing sequence is the order in which the records were inserted in the file.

7.1.4.2 Direct Access

Direct access means that the record to be processed is specified by a direct access record number in an I/O statement. For records in a sequential file to be directly accessed, the file must consist wholly of fixed-length records.

7.1.4.3 Keyed Access

Keyed access means that the record to be processed is specified by a key specification (Section 7.2.1.5) in an I/O statement.

You can mix in the same program keyed access and sequential access I/O statements that reference the same file. Therefore, you can use keyed I/O statements to position a file to a particular place and then use sequential I/O statements to process successive records.

The key specificaton in an I/O statement may specify an exact match by providing a complete key value, or it may specify a generic match by providing a partial key value. In the case of a generic match, the first record whose leftmost characters match the partial key value is the record selected.

A match criterion calls for either an exact match or an approximate match. An approximate match can be either a greater-than match or a greater-than-or-equal-to match.

7.2 I/O Statement Components

I/O statements consist of three basic components: a statement keyword, a control list, and an I/O list.

There are six basic statement keywords: READ, ACCEPT, WRITE, REWRITE, TYPE, and PRINT. The first two of these represent input operations, the remaining four output operations.

The control list and the I/O list are discussed below.

7.2.1 The Control List

The control list of an I/O statement is a list of one or more specifiers that perform the following functions:

- Specify the logical unit to be acted upon
- Specify the internal file to be acted upon
- Specify whether formatting is to be used for data editing and, if it is, the format specification
- Specify the number of a direct access record to be accessed
- Specify the key and key-of-reference of a keyed access record to be accessed
- Specify where control is to be transferred in the event of an error or end-of-file condition

The category of a statement can always be determined by the contents of its control list. For example, the control list of a formatted I/O statement always contains a format specifier (FMT=f or f), and the control list of a list-directed I/O statement always contains an asterisk in place of a format specifier.

The control list has the form:

```
(p[,p]...)
```

p
A specifier of the form: keyword = value.

The control list specifiers are discussed in the following sections.

7.2.1.1 Logical Unit Specifier

The logical unit specifier specifies the logical unit that is to be accessed. It has one of the forms:

```
[UNIT=]u
[UNIT=]*
```

u
An integer expression, with a value in the range 0 through 99, that refers to a specific file or I/O device. If necessary, the value is converted to integer data type before being used.

****
Specifies that the default input or output unit is to be accessed.

The keyword UNIT= is optional only if the logical unit specifier is the first parameter in the control list.

A logical unit number is connected to a file or device in one of two ways:

- Explicitly, by an OPEN statement (see Section 9.1).
- Implicitly, by the system. The *PDP-11 FORTRAN-77 User's Guide* describes the use of implicitly connected logical unit numbers in greater detail.

7.2.1.2 Internal File Specifier

An internal file specifier specifies the internal file to be used.

The internal file specifier has the form:

```
[UNIT=]cv
```

cv
The name of a character variable, character array, character array element, or character substring.

The logical unit specifier and the internal file specifier are mutually exclusive. The keyword UNIT= is optional if the internal file specifier is the first parameter in the control list.

See Section 7.1.3 for more information on internal files.

7.2.1.3 Format Specifier

The format specifier specifies whether explicit or list-directed formatting is to be used and, in the case of explicit formatting, identifies the parameter that will control the formatting. The format specifier has the form:

```
[FMT=]f
[FMT=]*
```

f

The statement label of a FORMAT statement, an integer variable that has been assigned (with an ASSIGN statement) a FORMAT statement-label value, the name of an array or array element, or a character expression containing a run-time format.

*

Specifies list-directed formatting.

The keyword FMT= is optional only if the format specifier is the second parameter in the control list and the first parameter is a logical unit or internal file specifier without the optional keyword UNIT=.

Chapter 8 describes FORMAT statements. Section 8.7 describes the interaction between formats and I/O statements.

You can use an asterisk in sequential I/O statements, instead of a format specifier, to denote list-directed formatting. See Sections 7.4.1.2 and 7.5.1.2 on list-directed I/O.

7.2.1.4 Record Specifier

The record specifier specifies the number of the direct access record to be accessed. The record specifier has the forms:

```
REC= r
'r
```

r

A numeric expression with a value that represents the position, in a direct access file, of the record to be accessed. The value must be greater than or equal to 1, and less than or equal to the maximum number of record cells allowed in the file. If necessary, a record number is converted to integer data type before being used.

7.2.1.5 Key Specifier

The key specifier specifies the key of an indexed file record to be accessed, the index in which this key is located, and the match criterion to be used in a key search.

An indexed file contains an index for each designated key field. In this index are listed the keys and the locations of the records containing them; records are ordered sequentially in order of increasing key value. Once supplied with the key of the desired record, the system looks up the key in the appropriate index and finds the location of the proper record. It then accesses this record. Using a key to obtain a specific record is called keyed access (see Section 7.4.1.3).

The indexes of a file are denoted by numbers from 0 to n, where n is the maximum number of indexes defined for the file. The value of n must be less than 255. Index number 0 is called the primary index or primary key. The other indexes are called alternate indexes or alternate keys; for example, index number 3 specifies the third alternate key.

Keyed access to indexed files is specified by key specifications in READ statements.

The key specification of a key specifier has three components:

1. A key expression, which specifies the key
2. A key-of-reference specifier, which specifies the index
3. A match criterion, which specifies the selection constraints

A key specification has the form:

```
KEY
KEYEQ
KEYGE  =ke[,KEYID=kn]
KEYGT
```

ke
A key expression.

kn
An integer expression, the value of which, called the key-of-reference number, specifies the index to be searched.

The KEY and KEYID parameters may appear in any order, but must follow the logical unit and format specifiers.

1. **Key Expressions**—Two types of key expressions are supported:
   - Character key expressions
   - Integer key expressions

Character key expressions must be used with character keys, and integer key expressions must be used with integer keys. A character key expression may be specified in one of the following forms:

- CHARACTER variable or substring
- CHARACTER array element
- CHARACTER constant
- A BYTE (LOGICAL*1) array name containing Hollerith data

For example, you can now specify keys as follows:

```
CHARACTER*5 CKEY
OPEN (UNIT=3, STATUS='OLD', ACCESS='KEYED',
1     ORGANIZATION='INDEXED', FORM='UNFORMATTED',
2     KEY=(1:5,18:23))
CKEY='SMITH'
READ(3,KEYGE=CKEY) ALPHA,BETA
END
```

The length of the character key expression is the length of the character value or the length of the BYTE array. If the length of the key expression is greater than the length of the key field, an error occurs. If the length of the key expression is less than the length of the key field, a generic key search is made rather than an exact key search. (See "Match Criterion" below.)

An integer key expression is an integer expression. Real, double-precision, and complex values are not permitted.

The name of a virtual array cannot be used to specify a key expression.

2. **Key-of-Reference Specifier**—The key-of-reference specifier specifies the index to be searched for the locations of a record. Its value, or key-of-reference number, must be an integer in the range 0 to the maximum number of keys defined for the file. A value of 0 specifies the primary key; a value of 1 specifies the first alternate key; and so forth.

If no key-of-reference specifier is included in a key specification, the key-of-reference is assumed to be what it was in the specification given in the last keyed I/O statement for the given logical unit.

3. **Match Criterion**—The match criterion specifies whether the match key must be equal to, greater than, or greater than or equal to the key specified by the key expression.

The match criterion has the forms:

EQ —specifies equal to
GT —specifies greater than
GE —specifies greater than or equal to

The match criterion is appended to KEY as follows:

KEY
KEYEQ
KEYGT
KEYGE

For character keys, matching comparisons are made on the basis of the ASCII collating sequence.

For integer keys, matching comparisons are made on the basis of the signed integer sequence.

If no match criterion is specified, equal matching is assumed.

For character keys, either generic matching or exact matching is used. Generic matching applies if the key expression in the I/O statement is shorter than the key field in the record. In generic matching, only the leftmost characters of the key field are used for the match.

For example, if the key expression is 'ABCD', and the key field is ten characters long, an equal match is obtained for the first record that contains 'ABCD' as the first four bytes of the key. The remaining six characters are arbitrary.

Approximate generic matching occurs when approximate matching (KEYGT or KEYGE) is selected in addition to generic matching. In approximate generic matching, only the leftmost characters are used for comparison.

For example, if the key expression is 'ABCD', and the key field is five characters long, and a greater-than match is selected, the value 'ABCDA' does not match. The value 'ABCEA', however, does match.

7.2.1.6 Transfer-of-Control Specifiers

The transfer-of-control specifiers specify a statement to which program control is to be transferred in the event of an end-of-file condition or an error condition. The transfer-of-control specifiers have the form:

```
END=s

ERR=s
```

s
The label of an executable statement.

A READ, WRITE, REWRITE, ENCODE, or DECODE statement can include either or both of the above specifiers in any order. The transfer-of-control specifiers must follow the logical unit, record, and format specifiers.

The statement label in a transfer-of-control specifier must refer to an executable statement that is located within the same program unit as the I/O statement.

An end-of-file condition occurs when no more records exist in a sequential file or when an end-file record produced by the ENDFILE statement (see Section 9.7) is encountered. If a READ statement encounters an end-of-file condition during an I/O operation, it transfers control to the statement named in the END=s specification. If no END=s specification is present, an error condition occurs.

If a READ, WRITE, REWRITE, ENCODE or DECODE statement encounters an error condition during an I/O operation, it transfers control to the statement whose label appears in the ERR=s specification. If no ERR=s is present, the I/O error terminates program execution.

An END= specification in a WRITE or REWRITE statement, direct access READ statement, or keyed access READ statement is ignored. If you attempt to read or write a record using a record number greater than the maximum specified for the logical unit, an error condition occurs.

The *PDP-11 FORTRAN-77 User's Guide* describes system subroutines that you can use to control error processing. These subroutines can also be used to obtain information from the I/O system on errors that occur.

Examples of the use of transfer-of-control specifiers in I/O statements follow.

```
READ (8,END=550) (MATRIX(K),K=1,100)
```

This statement transfers control to statement 550 if an end-of-file condition occurs on logical unit 8.

```
WRITE (6,50,ERR=390)
```

This statement transfers control to statement 390 if an error occurs during execution.

```
READ (1,FORM,ERR=150,END=200) ARRAY
```

This statement transfers control to statement 150 if an error occurs during execution, and to statement 200 if an end-of-file condition occurs.

7.2.2 I/O List

The I/O list in an input or output statement contains the names of variables, arrays, array elements, and character substrings from which or to which data is to be transferred. The I/O list in an output statement can also contain constants and expressions to be output.

An I/O list has the form:

```
s[,s]...
```

s
A simple list or an implied DO list.

The I/O statement assigns values to, or transfers values from, the list elements in the order in which they appear, from left to right.

7.2.2.1 Simple List

A simple I/O list consists of either a simple I/O list element or a group of two or more simple I/O list elements separated by commas. A simple I/O list element can be a single variable, an array, an array element, a constant, or an expression. For example, in the following statement J, K(3), 4, (L+4)/2, and N are simple I/O list elements.

```
WRITE (5,10) J, K(3), 4, (L+4)/2, N
```

When you use an unsubscripted array name in an I/O list, a READ or ACCEPT statement reads enough data to fill every element of the array; a WRITE, TYPE, or PRINT statement writes all the values in the array. Data transfer begins with the initial element of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most

rapidly. For example, the following statement defines a two-dimensional array:

```
DIMENSION ARRAY(3,3)
```

If the name ARRAY, with no subscripts, appears in a READ statement, this statement assigns values from the input record or records to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on through ARRAY(3,3).

In a READ or ACCEPT statement, variables in the I/O list can be used in array subscripts later in the list. For example, if you are given the following statements.

```
    READ (1,1250) J,K,ARRAY(J,K)
1250 FORMAT (I1,X,I1,X,F6.2)
```

and an input record that contains the following values.

```
1,3,721.73
```

When the READ statement is executed, the first input value is assigned to J and the second to K; the actual subscript values are now established for ARRAY(J,K). The value 721.73 is then assigned to ARRAY(1,3). Variables that are to be used as subscripts in this way must appear before (to the left of) their use as the array subscripts in the I/O list.

An output-statement I/O list may contain any valid expression. However, this expression must not attempt any I/O operations. For example, an output statement I/O list must not contain an expression that refers to a function subprogram that performs an I/O operation.

An input statement I/O list must not contain an expression used other than as a subscript expression in an array reference.

7.2.2.2 Implied DO List

An implied DO list is an I/O list element that functions as if it were a part of an I/O statement within a DO loop. Implied DO lists can be used to:

- Specify iteration of part of an I/O list
- Transfer part of an array
- Transfer array elements in an order that differs from the order of subscript progression

An implied DO list has the form:

```
(list,i=e1,e2[,e3])
```

list
An I/O list.

i
An integer variable.

e1,e2,e3
Arithmetic expressions.

The variable i and the parameters e1, e2, and e3 have the same forms and functions that they have in the DO statement (see Section 4.3). The list immediately preceding the DO loop parameter is the range of the implied DO loop. Elements in that list can reference i but they must not alter the value of i.

For example, the statement

```
WRITE (3,200) (A,B,C, I=1,3)
```

behaves as if you had written the statement

```
WRITE (3,200) A,B,C,A,B,C,A,B,C
```

In the statement

```
WRITE (6) (I,(J,P(I),Q(I,J),J=1,L),I=1,M)
```

the I/O list consists of an implied DO list that contains another implied DO list nested within it. The implied DO lists vary the Js for each value of I and write a total of (1+3*L)*M fields.

In a series of nested implied DO lists, the parentheses indicate the nesting (see Section 4.3.2). Execution of the innermost list is repeated most often. In the following example

```
      WRITE (6,150) ((FORM(K,L), L=1,10), K=1,10,2)
150   FORMAT (F10.2)
```

because the inner DO loop is executed 10 times for each iteration of the outer loop, the second subscript advances from 1 through 10 for each increment of the first subscript—that is, in the reverse of the standard order of subscript progression. In addition, because K is incremented by 2, only the odd-numbered rows of the array are output.

The entire list of an implied DO list is transmitted before the control variable is incremented. For example, in the statement

```
READ (5,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

P(1), Q(1,1), Q(1,2) . . . ,Q(1,10) is read before I is incremented to 2.

When processing multidimensional arrays, you can use a combination of fixed subscripts and subscripts that vary according to an implied DO list. For example, the statement

```
READ (3,5555) (BOX(1,J), J=1,10)
```

assigns input values to BOX(1,1) through BOX(1,10), and then terminates without affecting any other element of the array.

The value of the control variable can also be output directly. For example, the statement

```
WRITE (6,1111) (I, I=1,20)
```

simply prints the integers 1 through 20.

7.3 Syntactical Rules

The FORTRAN I/O statements described in Sections 7.4 through 7.8 are subject to the following syntactical rules.

- When in keyword form, the control parameters can appear in any order in a control list.
- The nonkeyword form of either the logical unit specifier or the internal file specifier must occupy the first (leftmost) position in a control list.
- When used with a logical unit specifier or internal file specifier, the nonkeyword form of the format specifier must occupy the second position in the control list; the unit or internal file specifier must also be in nonkeyword form (and therefore occupy the first position in the control list).
- If you use the nonkeyword form of a direct access record specifier, it must immediately follow a nonkeyword form of the logical unit specifier.

7.4 The READ Statements

The READ statements transfer input data to internal storage from records contained in external logical units, or to internal storage from internal files. There are four categories of READ statements: sequential, direct access, indexed, and internal.

7.4.1 The Sequential READ Statements

Sequential READ statements transfer input data to internal storage from external records accessed under the sequential mode of access. There are three classes of sequential READ statements: formatted, list-directed, and unformatted.

The three classes of sequential READ statements have the following forms.

Formatted Sequential READ Statement

```
READ(extu, fmt [,err][,end])[list]
READ f[,list]
```

List-Directed READ Statements

```
READ (extu, *  [,err] [,end]) [list]
READ *[, list]
```

Unformatted Sequential READ Statements

```
READ (extu  [,err] [,end]) [list]
```

extu
A logical unit specifier. See Section 7.2.1.1.

fmt
A format specifier. See Section 7.2.1.3.

f
The nonkeyword form of a format specifier. See fmt, above.

*
Specifies list-directed formatting.

err
end
Transfer-of-control specifiers. See Section 7.2.1.6.

list
An I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.4.1.1 The Formatted Sequential READ Statement

The formatted sequential READ statement does the following:

- Reads character data from one or more external records accessed under the sequential or keyed mode of access
- Translates the data from character to binary form using format specifications to provide editing
- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list

If the number of I/O list elements in a statement is less than the number of fields in an input record, the statement ignores the excess fields.

7.4.1.2 The List-Directed READ Statement

The list-directed READ statement does the following:

- Reads character data from records accessed under the sequential mode of access
- Translates the data from external to binary form using the data types of the elements in the I/O list, and the forms of the data, to provide editing
- Assigns the translated data to the elements in the I/O list in the order, from left to right, in which those elements appear in the list

The external records from which list-directed READ statements read data contain a sequence of values and value separators.

A value in one of these records may be any one of the following:

- A constant
- A null value
- A repetition of constants in the form r*c
- A repetition of null values in the form r*

Each constant has the form of the corresponding FORTRAN constant. A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Spaces can occur between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis. A logical constant represents true or false values—that is, .TRUE. or any value beginning with T, .T, t, or .t; or .FALSE. or any

value beginning with F, .F, f, or .f. A character constant is delimited by apostrophes, with an apostrophe that occurs within a character constant being represented by two consecutive apostrophes. Hollerith, octal, and hexadecimal constants are not permitted.

A null value is specified by two consecutive commas with no intervening constant, or by an initial comma or a trailing comma. Spaces can occur before or after the commas. A null value either indicates that the corresponding list element remains unchanged, or it represents an entire complex constant (but not just one part of a complex constant).

The form r*c specifies r occurrences of c, where r is a nonzero, unsigned integer constant and c is a constant. Spaces are not permitted except within the constant c as specified above.

The form r* specifies r occurrences of a null value, where r is an unsigned integer constant.

A value separator in a record may be any one of the following:

- One or more spaces or tabs
- A comma, with or without surrounding spaces or tabs
- A slash, with or without surrounding spaces or tabs

The slash terminates execution of the input statement and processing of the record; all remaining I/O list elements are left unchanged.

When any of the above appear in a character constant, they are considered part of the constant, not value separators.

The end of a record is equivalent to a space character except when it occurs in a character constant. When the end of a record occurs in a character constant, the end of the record is ignored and the character constant is continued with the next record. That is, the last character in the previous record is followed immediately by the first character in the next record.

Spaces at the beginning of a record are ignored unless they are part of a character constant continued from a previous record. When spaces are part of a continued character constant, they are considered part of that constant.

Input constants can be any of the following data types: integer, real, logical, complex, and character. The data type of a constant determines the data type of its value and the translation from external to internal form.

A numeric list element can correspond only to a numeric constant, and a character list element can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see Table 3–1).

Each input statement reads whatever number of records is required to satisfy its I/O list. If a slash separator occurs, or if the I/O list is exhausted before all the values in a record are used, the remainder of the record is ignored.

An example of the use of list-directed READ statements follows.

A program unit consists of the following:

```
CHARACTER*14 C
DOUBLE PRECISION T
COMPLEX D,E
LOGICAL L,M
READ (1,*) I,R,D,E,L,M,J,K,S,T,C,A,B
      .
      .
      .
```

And the external record to be read contains:

```
4 6.3 (3.4,4.2), (3, 2 ) , T,F,,3*14.6 ,'ABC,DEF/GHI''JK'/
```

Upon execution of the program unit, the following values are assigned to the I/O list elements:

I/O List Element	Value
I	4
R	6.3
D	(3.4,4.2)
E	(3.0,2.0)
L	.TRUE.
M	.FALSE.
K	14
S	14.6
T	14.6D0
C	ABC,DEF/GHI'JK

A, B, and J are unchanged.

7.4.1.3 The Unformatted Sequential READ Statement

The unformatted sequential READ statement reads an external record accessed under the sequential or keyed mode of access. It assigns the fields of binary data contained in that record to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list. The data is not translated. The amount of data assigned to each element is determined by the element's data type.

The unformatted sequential READ statement reads exactly one record. If the I/O list does not use all the values in a record—that is, if there are more values in the record than elements in the list—the remainder of the record is discarded. If the number of list elements is greater than the number of values in the record, an error occurs.

If a statement contains no I/O list, it skips over one full record, positioning the file to read the succeeding record on the next execution of a READ statement.

The unformatted sequential READ statement can only read records created by unformatted sequential WRITE statements.

Some examples of the use of the unformatted sequential READ statement follow.

```
READ(UNIT=1) FIELD1, FIELD2
```

In this example, the READ statement reads one record from logical unit 1 and assigns values of binary data to variables FIELD1 and FIELD2, in the order indicated.

```
READ (8)
```

In this example, the READ statement advances logical unit 8 one record.

7.4.2 The Direct Access READ Statements

Direct access READ statements transfer input data to internal storage from external records accessed under the direct mode of access. There are two classes: formatted and unformatted.

The two classes of direct access READ statement have the following forms.

Formatted Direct Access READ Statements

```
READ( extu, rec, fmt [,err])[list]
```

Unformatted Direct Access READ Statements

```
READ( extu, rec [,err])[list]
```

extu
A logical unit specifier. See Section 7.2.1.1.

rec
A record specifier. See Section 7.2.1.4.

fmt
A format specifier. See Section 7.2.1.3.

err
A transfer-of-control specifier. See Section 7.2.1.6.

list
An I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.4.2.1 The Formatted Direct Access READ Statement

The formatted direct access READ statement does the following:

- Reads character data from one or more external records accessed under the direct mode of access
- Translates the data from character to binary form using format specifications to provide editing
- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list

If the I/O list and formatting do not use all the characters in a record, the remainder of the record is discarded; if the I/O list and the formatting require more characters than are contained in the record, the remaining fields are read as spaces.

An example of the use of the formatted direct access READ statement follows:

```
      READ (2, REC=35, FMT=10) (NUM(K), K=1,10)
10    FORMAT (10I2)
```

In this example, the READ and FORMAT statements read the first ten fields from record 35 in logical unit 2, translate the values to binary form, and then assign the translated values to the internal storage locations of the ten elements of the array NUM.

7.4.2.2 The Unformatted Direct Access READ Statement

The unformatted direct access READ statement reads an external record accessed under the direct mode of access and assigns the fields of binary data contained in this record to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list. The data is not translated. The amount of data assigned to each element is determined by that element's data type.

The unformatted direct access READ statement reads exactly one record. If this record contains more fields than there are elements in the I/O list of the statement, the unused fields are discarded; if there are more elements than fields, an error occurs.

Examples of the use of unformatted direct access READ statements follow.

```
READ (1'10) LIST(1), LIST(8)
```

In this example, the READ statement reads record 10 in logical unit 1 and assigns binary integer values to elements 1 and 8 of the array LIST.

```
READ (4, REC=58, ERR=500) (RHO(N), N=1,5)
```

In this example, the READ statement reads record 58 in logical unit 4 and assigns binary values to 5 elements of the array RHO.

7.4.3 The Indexed READ Statements

The indexed READ statement transfers input data to internal storage from external records accessed under the keyed mode of access. There are two classes: formatted and unformatted.

A series of records in an indexed file may be read in key-of-reference sequence by using a sequential READ statement in conjunction with an indexed READ statement. The first record in the sequence is found using the indexed statement, the rest using sequential statements.

The two classes of indexed READ statement have the following forms.

Formatted Indexed READ Statement

```
READ( extu, fmt, key [,keyid] [,err])[list]
```

Unformatted Indexed READ Statement

```
READ( extu, key [,keyid] [,err])[list]
```

extu
A logical unit specifier. See Section 7.2.1.1.

fmt
A format specifier. See Section 7.2.1.3.

key
A key specifier. See Section 7.2.1.5.

keyid
A key-of-reference specifier. See Section 7.2.1.5 (2).

err
end
Transfer-of-control specifiers. See Section 7.2.1.6.

list
An I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.4.3.1 The Formatted Indexed READ Statement

The formatted indexed READ statement does the following:

- Reads character data from one or more external records accessed under the keyed mode of access
- Translates the data from character to binary form using format specifications to provide editing
- Assigns the translated values to the elements in the I/O list, in the order, from left to right, in which they appear in the list

The formatted indexed READ statement may only be used on indexed files. If the I/O list and format specifications specify that additional records are to be read, the statement reads those additional records sequentially, using the current key-of-reference value.

If the KEYID parameter is omitted, the key-of-reference remains unchanged from the most recent specification.

If the specified key value is shorter than the key field referred to, the key value is matched against the leftmost characters of the appropriate key field until a match is found; the record supplying the match is then read. If the key value is longer than the key field referred to, an error occurs.

An example of the use of the formatted indexed READ statement follows:

```
READ(3,KAT,KEY='ABCD') A,B,C,D
```

In this example the READ statement retrieves a record with the value of 'ABCD' in the primary key, and then uses the format contained in the array KAT to read the first four fields from the record into variables A,B,C, and D.

7.4.3.2 The Unformatted Indexed READ Statement

The unformatted indexed READ statement reads an external record accessed under the keyed mode of access and assigns the fields of binary data contained in that record to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list. The data is not translated. The amount of data assigned to each element is determined by the element's data type.

The unformatted indexed READ statement reads exactly one record and can be used only on indexed files. If the number of I/O list elements is less than the number of fields in the record being read, the unused fields in the record are discarded. If the number of I/O list elements is greater than the number of fields, an error occurs.

If a specified key value is shorter than the key field referred to, the key value is matched against the leftmost characters of the appropriate key field until a match is found; the record supplying the match is then read. If the specified key value is longer than the key field referred to, an error occurs.

Some examples of the use of the unformatted indexed READ statement follow.

```
OPEN (UNIT=3, STATUS='OLD',
1         ACCESS='KEYED', ORGANIZATION='INDEXED',
2         FORM='UNFORMATTED',
3         KEY=(1:5, 30:37, 18:23))

READ (3,KEY='SMITH') ALPHA,BETA
```

In this example, the READ statement reads from the file connected to logical unit 3 and retrieves the record with the value 'SMITH' in the primary key field (bytes 1 to 5). The first two fields of the record retrieved are placed in variables ALPHA and BETA, respectively.

```
READ (3,KEYGE='XYZDEF',KEYID=2,ERR=99) IKEY
```

In this example, the READ statement retrieves the first record having a value equal to or greater than 'XYZDEF' in the second alternate key field (bytes 18 to 23). The first field of that record is placed in the variable IKEY.

7.4.4 The Internal READ Statement

The internal READ statement transfers input data to internal storage from an internal file.

The DECODE statement discussed in Appendix A may be used as an alternative to the internal READ statement.

The internal READ statement is always formatted and has the form:

```
READ (intu, fmt[,err][,end])[list]
```

intu
An internal file specifier. See Section 7.2.1.2.

fmt
A format specifier. See Section 7.2.1.3.

err
end
Transfer-of-control specifiers. See Section 7.2.1.6.

list
An I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

The internal READ statement does the following:

- Reads character data from an internal file
- Translates the data from character to binary form using format specifications to provide editing
- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list

Refer to Section 7.1.3 for information on the characteristics and use of internal files.

The following program segments demonstrate the use of internal file reads:

```
CHARACTER*80 BUFFER
ACCEPT *, BUFFER
READ(BUFFER, '(I4.4)')I
```

This segment reads the first four characters in the variable BUFFER as an integer and assigns this integer value to the variable I.

```
INTEGER IVAL
CHARACTER TYPE, RECORD*80
CHARACTER*5 AFMT, IFMT, OFMT, ZFMT
PARAMETER (AFMT='(Q,A)', IFMT= '(I10)', OFMT= '(O11)',
1   ZFMT= '(Z8)')
ACCEPT AFMT, ILEN, RECORD
TYPE= RECORD (1:1)
IF (TYPE .EQ. 'D') THEN
    READ (RECORD (2:MIN(ILEN, 11)), IFMT)IVAL
ELSEIF (TYPE .EQ. 'O') THEN
    READ (RECORD (2:MIN(ILEN, 12)), OFMT)IVAL
ELSEIF (TYPE .EQ. 'X') THEN
    READ (RECORD (2:MIN(ILEN, 9)),ZFMT)IVAL
ELSE
    PRINT *, 'ERROR'
ENDIF
END
```

This program segment reads a record and examines the first character to determine whether the remaining data should be interpreted as decimal, octal, or hexadecimal. It then uses internal file reads to make appropriate conversions from character string representations to binary.

7.5 The WRITE Statements

The WRITE statements transfer output data from internal storage to records contained in user-specified external logical units, or from internal storage to internal files. There are four categories of WRITE statements: sequential, direct access, indexed, and internal.

WRITE statements cannot write to existing records in an indexed file. For statements that can perform this function, refer to the REWRITE statement discussed in Section 7.6.

7.5.1 The Sequential WRITE Statements

Sequential WRITE statements transfer output data from internal storage to external records accessed under the sequential mode of access. There are three classes of sequential WRITE statements: formatted, list directed, and unformatted.

The three classes of sequential WRITE statement have the forms:

Formatted Sequential WRITE Statements

```
WRITE( extu,fmt [,err]) [list]
```

List-Directed WRITE Statements

```
WRITE( extu, * [,err]) [list]
```

Unformatted Sequential WRITE Statement

```
WRITE( extu [,err]) [list]
```

extu
A logical unit specifier. See Section 7.2.1.1.

fmt
A format specifier. See Section 7.2.1.3.

err
A transfer-of-control specifier. See Section 7.2.1.6.

list
An I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.5.1.1 The Formatted Sequential WRITE Statement

The formatted sequential WRITE statement does the following:

- Reads specified data from internal storage
- Translates the data from binary to character form using format specifications to provide editing
- Writes the translated values to an external record accessed under the sequential mode of access

The length of the records written to a user-specified output device (for example, a line printer) must not exceed the maximum record length that this device can process. In the case of a line printer, the maximum record length is usually 132 characters.

Using an appropriate format specification, a formatted sequential WRITE statement can write more than one record.

Because numeric data transferred by formatted output statements is always rounded during its conversion from binary to character form, a loss of precision may result if this data is subsequently used as input. It is recommended, therefore, that whenever numeric output is to be used subsequently as input, unformatted output and input statements be used for data transfer.

Some examples of the use of formatted sequential WRITE statements follow

```
      WRITE (6,650)
650   FORMAT (' HELLO THERE')
```

In this example, the WRITE statement writes one record, consisting of the contents of the character constant in the format statement, to logical unit 6.

```
      WRITE (1,95) AYE,BEE,CEE
95    FORMAT (3F8.5)
```

In this example, the WRITE statement writes one record consisting of fields AYE, BEE, and CEE to logical unit 1.

```
      WRITE (1,900) DEE,EEE,EFF
900   FORMAT (F8.5)
```

In this example, the WRITE statement writes three separate records to logical unit 1; each record consists of only one field.

7.5.1.2 The List-Directed WRITE Statement

The list-directed WRITE statement does the following:

- Retrieves specified data from internal storage
- Translates that data from binary to character form using the data type of the elements in the I/O list to provide editing
- Writes the translated values to an external record accessed under the sequential mode of access

The values transferred as output by the list-directed WRITE statement have the same forms as the constant values transferred as input by the list-directed READ and ACCEPT statements, with the following exception: Character constants are transferred without delimiting apostrophes, and each internal apostrophe is represented by only one apostrophe instead of two. As a consequence of this exception, records containing list-directed character output data can be printed but cannot be used for list-directed input. (Refer to Section 7.4.1.2 for a full discussion on list-directed value forms.)

Table 7–3 below shows the default output formats for each data type.

Table 7–3: List-Directed Output Formats

Data Type	Output Formats
LOGICAL*1	I5
LOGICAL*2	L2
LOGICAL*4	L2
INTEGER*2	I7
INTEGER*4	I12
REAL*4	1PG15.7
REAL*8	1PG25.16
COMPLEX*8	1X,'(',1PG14.7, ',', 1PG14.7,')'
CHARACTER	1X, An (where n is the length of the character expression)

Note the following:

- List-directed output statements do not produce octal values, hexadecimal values, null values, slash separators, or repeated forms of values.

- List-directed output removes from a complex value any embedded spaces.
- Each output record begins with a space for carriage control.
- Each output statement writes one or more complete records.
- Each individual output value is contained within a single record, with the exception of character constants longer than one record length, and complex constants that can be split after the comma.

An example of the use of the list-directed WRITE statement follows:

```
DIMENSION A(4)
DATA A/4*3.4/
WRITE(1,*) 'ARRAY VALUES FOLLOW'
WRITE(1,*) A,4
```

In this example, the WRITE statements write the following records to logical unit 1:

```
ARRAY VALUES FOLLOW
   3.400000        3.400000        3.400000        3.400000        4
```

7.5.1.3 The Unformatted Sequential WRITE Statement

The sequential unformatted WRITE statement transfers specified binary data from internal storage to an external record accessed under the sequential mode of access. The data are not translated.

The sequential unformatted WRITE statement writes exactly one record; if there is no I/O list, the statement writes one null record.

Some examples of the use of the unformatted sequential WRITE statement follow.

```
WRITE(1) (LIST(K),K=1,5)
```

In this example, the WRITE statement writes to logical unit 1 a record containing the values, in binary form, of elements 1 through 5 of the array LIST.

```
WRITE(4)
```

In this example, the WRITE statement writes one null record to logical unit 4.

7.5.2 The Direct Access WRITE Statements

Direct access WRITE statements transfer output data from internal storage to external records accessed under the direct mode of access. There are two classes of direct access WRITE statements: formatted and unformatted.

Using an OPEN statement is one method of establishing attributes of a direct access file.

The two classes of direct access WRITE statement have the forms:

Formatted Direct Access WRITE Statements

```
WRITE( extu, rec, fmt [,err])[list]
```

Unformatted Direct Access WRITE Statements

```
WRITE( extu, rec [,err])[list]
```

extu
A logical unit specifier. See Section 7.2.1.1.

rec
A record specifier. See Section 7.2.1.4.

fmt
A format specifier. See Section 7.2.1.3.

err
A transfer-of-control specifier. See Section 7.2.1.6.

list
An I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.5.2.1 The Formatted Direct Access WRITE Statement

The formatted direct access WRITE statement does the following:

- Retrieves binary values from internal storage
- Translates those values to character form using format specifications to provide editing
- Writes the translated data to a user-specified external record accessed under the direct mode of access

If the values specified by the I/O list and formatting do not fill the output record being written, the unused portion of the record is filled with space characters. If the values overfill the record, an error occurs.

7.5.2.2 The Unformatted Direct Access WRITE Statement

The unformatted direct access WRITE statement retrieves binary values from internal storage and writes those values to a user-specified external record accessed under the direct mode of access. The values are not translated.

If the values specified by the I/O list do not fill the output record being written, the unused portion of the record is filled with zeros. If the values do not fit in the record, an error occurs.

7.5.3 The Indexed WRITE Statements

The indexed WRITE statements transfer output data from internal storage to external records accessed under the keyed mode of access. There are two classes of indexed WRITE statements: formatted and unformatted.

The indexed WRITE statement always writes a new record. The REWRITE statement discussed in Section 7.6 is used to update an existing record.

Using an OPEN statement is one method of establishing the attributes of an indexed file.

The syntactical form of the indexed WRITE statement is identical to that of the sequential WRITE statement; the two statements differ only in that the indexed WRITE statement refers to a logical unit connected to an indexed file, whereas the sequential WRITE statement refers to a logical unit connected to a sequential file.

The two classes of indexed WRITE statement have the forms:

Formatted Indexed WRITE Statements

```
WRITE( extu, fmt [,err])[list]
```

Unformatted Indexed WRITE Statements

```
WRITE( extu [,err])[list]
```

extu
A logical unit specifier. See Section 7.2.1.1.

fmt
A format specifier. See Section 7.2.1.3.

err
A transfer-of-control specifier. See Section 7.2.1.6.

list
An I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.5.3.1 The Formatted Indexed WRITE Statement

The formatted indexed WRITE statement does the following:

- Retrieves binary values from internal storage
- Translates those values to character form using format specifications to provide editing
- Writes the translated data to one or more external records accessed under the keyed mode of access

No key parameters are required in the list of control parameters, because all necessary key information is contained in the output record.

If the values specified by the I/O list and formatting do not fill a fixed-length record being written, the unused portion of the record is filled with space characters. If additional records are specified, these are inserted in the file logically according to the key values contained in each record.

An example of the use of formatted indexed WRITE statement follows:

```
      WRITE (4,100) KEYVAL, (RDATA (I), I=1,20)
100   FORMAT (A10,20F15.7)
```

In this example, the WRITE statement writes the translated values of KEYVAL and each of the 20 elements of the array RDATA to a new formatted record in the indexed file connected to logical unit 4.

7.5.3.2 The Unformatted Indexed WRITE Statement

The unformatted indexed WRITE statement retrieves binary values from internal storage and writes those values to an external record accessed under the keyed mode of access. The values are not translated.

No key parameters are required in the list of control parameters because all necessary key information is contained in the output record.

If the values specified by the I/O list do not fill a fixed-length record being written, the unused portion of the record is filled with zeros; if the values specified overfill the record, an error occurs.

7.5.4 The Internal WRITE Statement

The internal WRITE statement transfers output data from internal storage to an internal file.

You can also use the ENCODE statement discussed in Appendix A to control internal output.

The internal WRITE statement is always formatted and has the form:

```
WRITE (intu, fmt[,err])[list]
```

intu
An internal file specifier. See Section 7.2.1.2.

fmt
A format specifier. See Section 7.2.1.3.

err
A transfer-of-control specifier. See Section 7.2.1.6.

list
An I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

The internal WRITE statement does the following:

- Retrieves data from internal storage
- Translates this data from binary to character form using format specifications to provide editing
- Writes the translated values to an internal file

Refer to Section 7.1.3 for information on the characteristics and use of internal files.

The following example demonstrates the use of the internal WRITE statement:

```
CHARACTER*80 BUFFER
ACCEPT *,I
WRITE(BUFFER,'(I4.4)')I ! Start buffer with 4 digits from input
END
```

7.6 The REWRITE Statement

The REWRITE statement transfers output data from internal storage to the current record in an indexed file. There is only one category of REWRITE statement: indexed.

7.6.1 The Indexed REWRITE Statement

The indexed REWRITE statement transfers output data from internal storage to the last record in an indexed file to be accessed by a READ statement. There are two classes of indexed REWRITE statements: formatted and unformatted.

The OPEN statement is used to establish the attributes of an indexed file.

The two classes of indexed REWRITE statement have the forms:

Formatted Indexed REWRITE Statement

```
REWRITE( extu,fmt [,err]) [list]
```

Unformatted Indexed REWRITE Statement

```
REWRITE( extu [,err])[list]
```

where extu, fmt, err, and list are defined as they are for the indexed WRITE statements discussed in Section 7.5.3. Refer to Section 7.3 for applicable syntactical rules.

7.6.1.1 The Formatted Indexed REWRITE Statement

The formatted indexed REWRITE statement does the following:

- Retrieves binary values from internal storage
- Translates those values to character form using format specifiers to provide editing
- Writes the translated data to an existing record in an indexed file

The record written to is the current record in the file—that is, the last record to be accessed by a preceding indexed or sequential READ statement.

Changing the primary key value results in an error, and attempting to rewrite more than one record causes an error. Any unused space in a rewritten fixed-length record is filled with spaces; if the record is too long, an error occurs.

An example of the use of a formatted indexed REWRITE statement follows:

```
      REWRITE(3, 10, ERR=99) NAME, AGE, BIRTH
10    FORMAT (A16, I2, A8)
```

In this example, the REWRITE statement updates the current record contained in the indexed file connected to logical unit 3 with the values represented by NAME, AGE, and BIRTH.

7.6.1.2 The Unformatted Indexed REWRITE Statement

The unformatted indexed REWRITE statement retrieves binary values from internal storage and writes those values to an existing record in an indexed file. The values are not translated.

The record written to is the current record in the file—that is, the last record to be accessed by a preceding indexed or sequential READ statement.

Changing the primary key value results in an error. Any unused space in a rewritten, fixed-length record is filled with zeros; if the record is too long, an error occurs.

7.7 The ACCEPT Statement

The ACCEPT statement transfers input data to internal storage from external records accessed under the sequential mode of access.

ACCEPT statements can only be used on implicitly connected logical units.

The ACCEPT statement has the forms:

```
ACCEPT f[,list]
ACCEPT *[,list]
```

f
The nonkeyword form of a format specifier. See Section 7.2.1.3.

****
Specifies list-directed formatting.

list
An I/O list. See Section 7.2.2.

The ACCEPT statement functions exactly as the formatted sequential READ statement discussed in Section 7.4.1.1, with one important exception: The ACCEPT statement can never be connected to user-specified logical units.

An example of the use of the formatted ACCEPT statement follows:

```
      CHARACTER *10 CHARAR(5)
      ACCEPT 200, CHARAR
200   FORMAT (5A10)
```

In this example, the ACCEPT statement reads character data from the implicit unit and assigns binary values to each of the five elements of the array CHARAR.

7.8 The TYPE and PRINT Statements

The TYPE and PRINT statements transfer output data from internal storage to external records accessed under the sequential mode of access.

TYPE and PRINT statements have the forms:

```
TYPE f[,list]
PRINT f [,list]

TYPE * [, list]
PRINT * [, list]
```

f
The nonkeyword form of a format specifier. See Section 7.2.1.3.

*
Specifies list-directed formatting.

list
An I/O list. See Section 7.2.2.

TYPE and PRINT statements function exactly as the formatted sequential WRITE statement discussed in Section 7.5.1.1, with one important exception: The formatted sequential TYPE and PRINT statements can never be used to transfer data to user-specified logical units.

An example of the use of a formatted sequential PRINT statement follows:

```
      CHARACTER*16 NAME, JOB
      PRINT 400, NAME, JOB
400   FORMAT ('NAME=', A, 'JOB=', A)
```

In this example, the PRINT statement writes one record to the implicit output device; the record consists of four fields of character data.

Chapter 8

Format Statements

FORMAT statements are nonexecutable statements used with formatted I/O statements (and with ENCODE and DECODE statements) to describe the format in which data is to be transferred, and to specify the kind of conversion and the editing required to achieve this format.

Throughout this chapter a distinction is made between "external form" and "internal form." "External form" refers to the ASCII characters in a data field of a formatted record; "internal form" refers to the binary representation of a data value.

FORMAT statements have the form:

```
FORMAT (q1f1s1f2s2 ... fnqn)
```

q
Zero or more slash (/) record terminators.

f
A field or edit descriptor, or a group of field or edit descriptors enclosed in parentheses.

s
A field separator.

The entire list of field and edit descriptors, field separators, and record terminators, including the enclosing parentheses (which must be present), is called the format specification.

The field separators are the comma and the slash. The slash is also a record terminator. Section 8.5 describes in detail the functions of the field separators.

The field and edit descriptors have the forms:

```
[r]c[w[.d[Ee]]]    . [r]cw.m
```

r
The number of times the field or edit descriptor is to be repeated (repeat count). If you omit r, it is assumed to be 1.

c
A field or edit descriptor code (S, SP, SS, I, O, Z, F, E, D, G, L, A, H, X, T, P, Q, $, BN, BZ, TL, or TR).

w
The external field width, in characters.

d
The number of characters to the right of the decimal point.

E
In this context, identifies an exponent field.

e
The number of characters in the exponent.

m
The minimum number of characters that must appear within the field (including leading zeros).

The terms r, w, m, and d must all be unsigned integer constants; r, w, m, d, and e must be less than or equal to 255, and r and w must be nonzero. The r term is always an optional element in those descriptors in which it can be used. The d and e terms are required in some field descriptors and are invalid in others.

You are not allowed to use parameter constants for the terms r, w, m, d, or e.

The field and edit descriptors are:

- Integer field—Iw, Ow, Zw, Iw.m, Ow.m, Zw.m
- Logical field—Lw
- Real, double-precision, and complex field—Fw.d, Ew.d, Dw.d, Gw.d, Ew.dEe, Gw.dEe
- Character field—Aw

- Edit (and control)—BN, BZ, SP, SS, S, nX, Tn, TLn, TRn, nP, Q, $, : (where n is a number of characters or character positions)
- Character and Hollerith constant field—nH, ' . . . '

Section 8.1 describes each field and edit descriptor in detail.

The first character in an output record generally contains carriage-control information (see Section 8.3).

During data transfers, the format specification is scanned from left to right and the elements in the I/O list are correlated one-for-one with corresponding field descriptors in the specification, except in the case of edit descriptors and character- and Hollerith-constant field descriptors, which do not require corresponding I/O list elements.

Section 8.7 describes in detail the interaction between format specifiers and the I/O list.

You use an I, O, Z, or L field descriptor to process integer and logical data. You use an F, E, D, G, O, or Z field descriptor to process real, double-precision, and complex data.

You use an A, O, or Z field descriptor to process character data.

You can create a format specification before program execution with the FORMAT statement. Section 8.8 summarizes the rules for writing FORMAT statements. You can create a format during program execution by using a run-time format instead of a FORMAT statement. Section 8.6 describes run-time formats.

8.1 Field and Edit Descriptors

A field descriptor describes the size and format of a data item or items. (Data items in an external medium are called external fields.) An edit descriptor specifies an editing function to be performed on a data item or items. (Some edit descriptors, such as the Scale Factor P, actually perform control functions but are included among the edit descriptors for the sake of simplicity.)

The numeric field descriptors ignore leading spaces in the external field; however, they treat embedded and trailing spaces as zeros unless the BN edit descriptor is in effect, or unless BLANK = 'NULL' is in effect for the logical unit, in which case all spaces are ignored.

At the beginning of the execution of each formatted input statement, the BLANK attribute for the unit determines the interpretation of spaces; the PDP-11 FORTRAN-77 defaults are BLANK = 'NULL' when an OPEN statement has been executed, and BLANK= 'ZERO' when no OPEN statement has been executed. During the execution of a formatted input statement, the interpretation of spaces may be controlled by BN and BZ edit descriptors—that is, the default interpretation may be superseded by either of these. The BN and BZ edit descriptors affect only the formatted I/O statement of which they are a part.

The field and edit descriptors are described in detail in Sections 8.1.1 through 8.1.23. Sections 8.1.24, 8.1.25, and 8.1.26 discuss complex-data editing, repeat counts, and default descriptors, respectively.

8.1.1 BN Edit Descriptor

The BN edit descriptor causes the processor to ignore all the embedded and trailing blanks it encounters within a numeric input field. It has the form:

```
BN
```

The effect is that of actually removing the blanks and right-justifying the remainder of the field. A field of all blanks is treated as zero. The BN descriptor affects only I, O, Z, F, E, D, and G editing, and only upon the execution of an input statement.

8.1.2 BZ Edit Descriptor

The BZ edit descriptor causes the processor to treat all the embedded and trailing blanks it encounters within a numeric input field as zeros. It has the form:

```
BZ
```

The BZ descriptor affects only I, O, Z, F, E, D, and G editing, and only upon the execution of an input statement.

8.1.3 SP Edit Descriptor

The SP edit descriptor causes the processor to produce a plus character in any position where this character would otherwise be optional. It has the form:

```
SP
```

The SP descriptor affects only I, F, E, D, and G editing, and only upon the execution of an output statement.

8.1.4 SS Edit Descriptor

The SS edit descriptor causes the processor to suppress a leading plus character from any position where this character would normally be produced as an optional character; it has the opposite effect of the SP field descriptor described above. The SS descriptor has the form:

```
SS
```

The SS descriptor affects only I, F, E, D, and G editing, and only upon execution of an output statement.

8.1.5 S Edit Descriptor

The S edit descriptor reinvokes optional plus characters (+) in numeric output fields. It has the form:

```
S
```

The S descriptor counters the action of either the SP or SS descriptor by restoring to the processor the decision-making ability to produce plus characters on an optional basis.

The same restrictions apply as for the SP and SS descriptors.

8.1.6 I Field Descriptor

The I field descriptor specifies decimal integer values. It has the form:

```
Iw[.m]
```

The corresponding I/O list element must be of either integer or logical data type.

Rules in Effect for Data Input

- The I field descriptor specifies that w characters are to be read from an external field, interpreted as a decimal integer value, and assigned to the corresponding I/O list element.
- The external data value must be an integer constant; it cannot contain a decimal point or an exponent field.
- If the external value exceeds the maximum allowed magnitude of the corresponding list element, an error occurs.
- If the first nonblank character of the external field is a minus sign, the field is treated as a negative value.
- If the first nonblank character is a plus sign, or if no sign appears in the field, the field is treated as a positive value.
- An all-blank field is treated as a value of 0.

Input Examples

Format	External Field	Internal Value
I4	2788	2788
I3	-26	-26
I9	312	312

Rules in Effect for Data Output

- The I field descriptor specifies that the value of the corresponding I/O list element is to be transferred as a decimal value, right justified, to an external field w characters long.
- If m is present, the external field consists of at least m digits; if necessary, zeros are added on the left to bring the total digits to m.
- If the value exceeds the field width, the entire field is filled with asterisks.

- If the value of the list element is negative, the field will have a minus sign as its leftmost, nonblank character, provided the term w is large enough.
- Plus signs are suppressed, unless SP is specified.

Output Examples

Format	Internal Value	External Representation
I3	284	284
I4	-284	-284
I5	174	ΔΔ174
I2	3244	**
I3	-473	***
I7	29.812	Not permitted: error
I4.2	1	ΔΔ01
I4.4	1	0001

8.1.7 O Field Descriptor

The O field descriptor specifies octal integer values. It has the form:

```
Ow[.m]
```

The corresponding I/O list element can be any data type.

Rules in Effect for Data Input

- The O field descriptor specifies that w characters are to be read from an external field, interpreted as an octal value, and assigned to the corresponding I/O list element.
- The external field can contain only the numerals 0 through 7; it cannot contain a sign, a decimal point, or an exponent field.
- An all-blank field is treated as a value of 0.
- If the value of the external data exceeds the allowed size of the corresponding list element, an error occurs.

Input Examples

Format	External Field	Internal Decimal Value
O5	77777	32767
O4	31274	1623
O6	15	53248
O3	97	Not permitted: error

Rules in Effect for Data Output

- The O field descriptor specifies that the octal value of the corresponding I/O list element is to be transferred as an octal integer, right justified, to an external field w characters long.
- No signs are output; a negative value is transmitted in its octal (two's complement) form.
- If the value does not fill the field, leading spaces are inserted.
- If the value exceeds the field width, the entire field is filled with asterisks.
- If m is present, the external field consists of at least m digits; if necessary, zeros are added on the left to bring the total digits to m.

Output Examples

Format	Internal (Decimal) Value	External (Octal) Representation
O6	32767	Δ77777
O6	-32767	100001
O2	14261	**
O4	27	ΔΔ33
O11	13.52	12173041130
O4.2	7	ΔΔ07
O4.4	7	0007

8.1.8 Z Field Descriptor

The Z field descriptor specifies hexadecimal (base 16) values. It has the form:

```
Zw[.m]
```

The Z field descriptor can be used with a corresponding I/O list element of any data type.

Rules in Effect for Data Input

- The Z field descriptor specifies that w characters are to be read from an external field, interpreted as a hexadecimal value, and assigned to the corresponding I/O list element.
- The external field can contain only the numerals 0 through 9 and the letters A (or a) through F (or f); it cannot contain a sign, a decimal point, or an exponent field.
- An all-blank field is treated as a value of zero.
- If the value of the external field exceeds the range of the corresponding list element, an error occurs.

Input Examples

Format	External Field	Internal Hexadecimal Value
Z3	A94	A94
Z5	A23DEF	A23DE
Z5	95.AF2	Not permitted: error

Rules in Effect for Data Output

- The Z field descriptor specifies that the value of the corresponding I/O list element is to be transferred as a hexadecimal value, right justified, to an external field w characters long.
- No signs are output.
- A negative value is transferred in its hexadecimal (two's complement) form.
- If the value does not fill the external field, leading spaces are inserted; if the value exceeds the field, the entire field is filled with asterisks.
- If m is present, the external field consists of at least m digits; if necessary, the field is zero filled on the left.

Output Examples

Format	Internal (Decimal) Value	External Representation
Z4	32767	7FFF
Z5	-32767	Δ8001
Z2	16	10
Z4	-10.5	C228
Z3.3	2708	A94
Z6.4	2708	ΔΔ0A94

Note that if m is zero, and the internal representation is zero, the external field is blank filled.

8.1.9 F Field Descriptor

The F field descriptor specifies real or double-precision values. It has the form:

```
Fw.d
```

The corresponding I/O list element must be of real or double-precision data type, or it must be either the real or the imaginary part of a complex data type.

Rules in Effect for Data Input

- The F field descriptor specifies that w characters are to be read from an external field, interpreted as a real or double-precision value, and assigned to the corresponding I/O list element. Any decimal point, signs, or exponent field present in the external field are included in the w count, and d is part of w.
- If the w characters include a decimal point, the position of the decimal point is used. If the w characters do not include a decimal point, the decimal point is placed before the rightmost d digits of w.
- If the w characters include an exponent field (see Section 2.3.2 for real constant exponents and Section 2.3.3 for double-precision constant exponents), the exponent is used to evaluate the number's magnitude before the decimal point position is determined.
- If the first nonblank character of the external field is a minus sign, the field is treated as a negative value.

- If the first nonblank character is a plus sign, or if no sign appears in the field, the field is treated as a positive value.
- An all-blank field is treated as a value of 0.
- The term w must be greater than or equal to d+1.

Input Examples

Format	External Field	Internal Value
F8.5	123456789	123.45678
F8.5	-1234.567	-1234.56
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

Rules in Effect for Data Output

- The F field descriptor specifies that the value of the corresponding I/O list element is to be transferred as a real or double-precision value, rounded to d decimal positions and right justified, to an external field w characters long.
- If the value does not fill the field, leading spaces are inserted.
- If the value exceeds the field width, the entire field is filled with asterisks.
- Plus signs are suppressed, unless SP is specified.
- The term w must be greater than or equal to d+1; however, the field width should be large enough to contain the number of digits after the decimal point, plus 1 for the decimal point, plus the number of digits to the left of the decimal point, plus 1 for a possible negative sign.

Output Examples

Format	Internal Value	External Representation
F8.5	2.3547188	Δ2.35472
F9.3	8789.7361	Δ8789.736
F2.1	51.44	**
F10.4	-23.24352	ΔΔΔ-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

8.1.10 E Field Descriptor

The E field descriptor specifies real or double-precision values in exponential form. It has the form:

```
Ew.d[Ee]
```

The corresponding I/O list element must be of real or double-precision data type, or it must be either the real or the imaginary part of a complex data type.

Rules in Effect for Input

On input, the E field descriptor does not differ from the F field descriptor.

Input Examples

Format	External Field	Internal Value
E9.3	734.432E3	0.734432E+6
E12.4	1022.43E-6	0.102243E-2
E15.3	52.3759663	0.523759E+2
E12.5	210.5271D+10	0.2105271E+13

In the last example, note that the E field descriptor treats the D exponent field indicator as an E indicator if the I/O list element is single precision.

Rules in Effect for Output

- The E field descriptor specifies that the value of the corresponding I/O list element is to be transferred as a real or double-precision value in exponential form, rounded to d decimal digits and right justified, to an external field w characters long.
- If the value does not fill the w characters, leading spaces are inserted.
- If the value exceeds the w characters, the entire field is filled with asterisks.
- Output is in a standard form; that is, it has a minus sign if the value is negative, an optional 0, a decimal point, d digits to the right of the decimal point, and an e+2 character exponent in one of the forms:

```
E+nn
                          Ew.d (for exponent < 99)
E-nn
E+n(1)n(2)...n(e)
                          Ew.dEe
E-n(1)n(2)...n(e)
```

n
A digit of an integer.

- The exponent field width specification is optional; if it is omitted, the value of e defaults to 2. If the exponent value is too large to be output in one of the above forms, an error occurs.
- The d digits to the right of the decimal point represent the entire value, scaled to a decimal fraction.
- Plus signs are suppressed, unless SP is specified.
- The term w must be large enough to include: a minus sign when necessary (plus signs are optional); a zero; a decimal point; d digits; and an exponent. Therefore, w must be greater than or equal to d+7, or to d+e+5 if e is present.

Output Examples

Format	Internal Value	External Representation
E9.2	475867.222	∆0.48E+06
E12.5	475867.222	∆0.47587E+06
E12.3	0.00069	∆∆∆0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****
E14.5E4	-1.001	-0.10010E+0001
E14.3E6	0.000123	∆0123E-000003

8.1.11 D Field Descriptor

The D field descriptor specifies with a D instead of an E real or double-precision values in exponential form. It has the form:

```
Dw.d
```

The corresponding I/O list element must be of real or double-precision data type, or it must be either the real or the imaginary part of a complex data type.

Rules in Effect for Input

On input, the D field descriptor does not differ from the F or E field descriptors.

Input Examples

Format	External Field	Internal Value
D10.2	12345	0.1234500000D+8
D10.2	123.45	0.1234500000D+3
D15.3	367.4981763D-04	0.3674981763D-1

Rules in Effect for Output

There is only one difference between the D and E descriptors on output: If you use the D descriptor, the letter D is output instead of the letter E.

Output Examples

Format	Internal Value	External Value
D14.3	0.0363	ΔΔΔΔΔ0.363D-01
D23.12	5413.87625793	ΔΔΔΔΔ0.541387625793D+04
D9.6	1.2	*********

8.1.12 G Field Descriptor

The G field descriptor specifies real or double-precision values, combining E- or F-type formats according to the size of the number being output. It has the form:

```
Gw.d[Ee]
```

The corresponding I/O list element must be of real or double-precision data type, or it must be either the real or the imaginary part of a complex data type.

Rules in Effect for Input

On input, the G field descriptor does not differ from the F, E, or D descriptors.

Rules in Effect for Output

- The G field descriptor specifies that the value of the corresponding I/O list element is to be transferred as a real or double-precision value in either exponential or fixed-point form, rounded to d decimal positions and right justified, to an external field w characters long.
- The form in which the value is written is a function of the magnitude of the value, as described in Table 8–1.

Table 8–1: Effect of Data Magnitude on G Formats

Data Magnitude	Effective Format
m < 0.1	Ew.d[Ee]
0.1 < m < 1.0	F(w–4).d, ' '
1.0 < m < 10.0	F(w–4).(d–1), ' '
.	.
.	.
.	.
10d–2 < m < 10d–1	F(w–4).1, ' '
10d–1 < m < 10d	F(w–4).0, ' '
m > 10d	Ew.d[Ee]

Note: The ' ' in the second column of Table 8–1 specifies that four spaces are to follow the numeric data representation.

- Plus signs are suppressed.
- The term w must be large enough to include: a minus sign when necessary (plus signs are optional); a decimal point; d digits to the right of the decimal point; and either a 4-character or an (e+2)-character exponent. Therefore, w must be greater than or equal to d+7 or d+5+e.

Output Examples

Format	Internal Value	External Representation
G13.6	0.01234567	Δ0.123457E–01
G13.6	–0.12345678	–0.123457ΔΔΔΔ
G13.6	1.23456789	ΔΔ1.23457ΔΔΔΔ

Format	Internal Value	External Representation
G13.6	12.34567890	ΔΔ12.3457ΔΔΔΔ
G13.6	123.45678901	ΔΔ123.457ΔΔΔΔ
G13.6	-1234.56789012	Δ-1234.57ΔΔΔΔ
G13.6	12345.67890123	ΔΔ12345.7ΔΔΔΔ
G13.6	123456.78901234	ΔΔ123457.ΔΔΔΔ
G13.6	-1234567.89012345	-0.123457E+07

Compare the above examples with the following examples, which show the same values output with an equivalent F field descriptor.

Format	Internal Value	External Representation
F13.6	0.01234567	ΔΔΔΔΔ0.012346
F13.6	-0.12345678	ΔΔΔΔ-0.123457
F13.6	1.23456789	ΔΔΔΔΔ1.234568
F13.6	12.34567890	ΔΔΔΔ12.345679
F13.6	123.45678901	ΔΔΔ123.456789
F13.6	-1234.56789012	Δ-1234.567890
F13.6	12345.67890123	Δ12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	**********

8.1.13 L Field Descriptor

The L field descriptor specifies logical data. It has the form:

```
Lw
```

The corresponding I/O list element must be of either integer or logical data type.

Rules in Effect for Input

- The L field descriptor specifies that w characters are to be read from the external field.
- If the first nonblank character of the field is the letter T, t, .T, or .t, the value .TRUE. is assigned to the corresponding I/O list element.
- If the first nonblank character of the field is the letter F, f, .F, or .f, or if the entire field is blank, the value .FALSE. is assigned.
- Any other value in the external field produces an error.

Rules in Effect for Output

- The L field descriptor specifies that either the letter T (if the value of the corresponding I/O list element is .TRUE.) or the letter F (if the value of the corresponding I/O list element is .FALSE.) is to be transferred to an external field w characters long.
- letter T or F is in the rightmost position of the field, preceded by w–1 spaces.

Output Examples

Format	Internal Value	External Representation
L5	.TRUE.	ΔΔΔΔT
L1	.FALSE.	F

8.1.14 A Field Descriptor

The A field descriptor specifies character or Hollerith values. It has the form:

```
A[w]
```

The corresponding I/O list element can be of any data type, because variables of any data type can be used to store Hollerith data.

The value of w must be less than or equal to 255.

Rules in Effect for Input

- The A field descriptor transfers w characters from the external record and assigns them to the corresponding I/O list element.
- The maximum number of characters that can be stored depends on the size of the I/O list element.
- The size of a character I/O list element is the length of the character variable, character substring reference, or character array element that makes up the element. The size of a numeric I/O list element depends on the data type of the element, as follows:

I/O List Element	Maximum Number of Characters
BYTE	1
LOGICAL*1	1
LOGICAL*2	2
LOGICAL*4	4
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*8	8
DOUBLE PRECISION	8
COMPLEX	8
CHARACTER*n	n

- If w is greater than the maximum number of characters that can be stored in the corresponding I/O list element, only the rightmost characters are assigned to the element. Leftmost excess characters are ignored.
- If w is less than the number of characters that can be stored, w characters are assigned to the list element, left justified, and trailing spaces are added to fill the element.

Input Examples

Format	External Field	Internal Representation	
A6	PAGEΔ#	#	(CHARACTER*1)
A6	PAGEΔ#	EΔ#	(CHARACTER*3)
A6	PAGEΔ#	PAGEΔ#	(CHARACTER*6)
A6	PAGEΔ#	PAGEΔ#ΔΔ	(CHARACTER*8)
A6	PAGEΔ#	#	(LOGICAL*1)
A6	PAGEΔ#	Δ#	(INTEGER*2)
A6	PAGEΔ#	GEΔ#	(REAL)
A6	PAGEΔ#	PAGEΔ#ΔΔ	(DOUBLE PRECISION)

Rules in Effect for Output

- The A field descriptor specifies that the contents of the corresponding I/O list element are to be transferred to an external field w characters long.
- If w is greater than the size of the list element, the data appears in the field, right justified, with leading spaces.
- If w is less than the size of the list element, only the leftmost w characters are transferred.

Output Examples

Format	Internal Value	External Representation
A5	OHMS	ΔOHMS
A5	VOLTS	VOLTS
A5	AMPERES	AMPER

If you omit w in an A field descriptor, a default value is supplied. If the I/O list element is of character data type, the default value is the length of the I/O list element. If the I/O list element is of numeric data type, the default value is the maximum number of characters that can be stored in a variable of that data type.

8.1.15 H Field Descriptor

The H field descriptor specifies that data is to be transferred between an external record and the storage location of the H field descriptor itself. It has the form (of a Hollerith constant):

```
nHc1c2c3 ... cn
```

n
The number of characters to be transferred.

c
An ASCII character.

Rule in Effect for Input

- The H field descriptor specifies that n characters be accepted from an external field and assigned to the same storage location as the characters of the H descriptor. The characters of the H descriptor are overlaid by the input data, character for character.

Rule in Effect for Output

- The H field descriptor specifies that n characters following the letter H be transferred to the external field.

An example of H field-descriptor usage follows.

```
      TYPE  100
100   FORMAT (41H ENTER PROGRAM TITLE, UP TO 20 CHARACTERS)
      ACCEPT  200
200   FORMAT (20H  TITLE GOES HERE   )
```

The TYPE statement transfers the characters from the H field descriptor in statement 100 to the user's terminal. The ACCEPT statement accepts the response from the keyboard and places the input data in the H field descriptor in statement 200. The new characters replace the words TITLE GOES HERE. If the user enters fewer than 20 characters, the remainder of the H field descriptor is filled with spaces to the right.

You can use a character constant instead of an H field descriptor. Both types of format specifier function identically.

In a character constant, the apostrophe is written as two apostrophes. For example:

```
50    FORMAT ('TODAY''S DATE IS: ',I2,'/',I2,'/',I2)
```

A pair of apostrophes used in this way is considered a single character.

8.1.16 X Edit Descriptor

The X edit descriptor specifies that a number of character positions be skipped. It has the form:

```
nX
```

The term n specifies the number of character positions to be skipped. The value of n must be greater than or equal to 1, and less than or equal to 255.

Rule in Effect for Input

- The X edit descriptor specifies that the next n characters in the input record are to be skipped.

Rule in Effect for Output

- The X edit descriptor tabs right n spaces; it does not write over anything already written. For example, the WRITE statement in:

```
      WRITE (6,90) NPAGE
90    FORMAT (13H1PAGE NUMBER ,I2,16X,23HGRAPHIC ANALYSIS, CONT.)
```

prints a record similar to the following:

```
 PAGE NUMBER nn            GRAPHIC ANALYSIS, CONT.
```

where nn is the current value of the variable NPAGE. Note that the numeral 1 in the first H field descriptor is not printed but is used to advance the printer paper to the top of a new page. (Section 8.3 describes printer carriage control.)

8.1.17 T Edit Descriptor

The T edit descriptor specifies the position, relative to the start of an external record, of the next character to be processed. It has the form:

```
Tn
```

where the term n indicates the position in the external record of the next character to be processed. The value of n must be greater than or equal to 1, but not greater than the number of characters allowed in the record.

Rule in Effect for Input

- In an input statement, the T field descriptor specifies that data starting with the nth character position is to be transferred as input. For example, the statements

```
      READ (5,10) J,K
10    FORMAT (T7,A3,T1,A3)
```

  specify that a 3-character string starting at character position 7 in the external record is to be read first, followed by a 3-character string starting at character position 1.

Rule in Effect for Output

- In an output statement, the T field descriptor specifies that data output is to begin at the nth character position of the external record. For example, the statements

```
      PRINT 25
25    FORMAT (T50,'COLUMN 2',T20,'COLUMN 1')
```

  print "COLUMN 1" at position 20 and "COLUMN 2" at position 50. The remainder of the line contains blank characters.

8.1.18 TL Edit Descriptor

The TL edit descriptor is a relative tabulation specifier for tabbing to the left. It has the form:

```
TLn
```

The term n specifies that the next character to be transferred from or to a record is the nth character to the left of the current character. The value of n must be greater than or equal to 1. If the value of n is greater than or equal to the current character position, the first character in the record is specified.

8.1.19 TR Edit Descriptor

The TR edit descriptor is a relative tabulation specifier for tabbing to the right. It has the form:

```
TRn
```

The term n indicates that the next character to be transferred from or to a record is the nth character to the right of the current character. The value of n must be greater than or equal to 1.

8.1.20 Q Edit Descriptor

The Q edit descriptor specifies that the count of the characters (**not** the characters themselves) remaining in a record being read are to be assigned to a corresponding variable in an I/O list. It has the form:

```
Q
```

A corresponding I/O list element must be of integer or logical data type.

For example, the input statements

```
    READ (4,1000) XRAY,KK,NCHRS,(ICHR(I),I=1,NCHRS)
1000 FORMAT (E15.7,I4,Q,80A1)
```

read two fields into the variables XRAY and KK. The count of the characters remaining in the record is then stored in NCHRS, and exactly this number of characters is read into the array ICHR. By placing the Q descriptor in the first position in a format specification, you can determine the actual length of an input record.

In an output specification, the Q edit descriptor has no effect except to cause a corresponding I/O list element to be skipped.

8.1.21 Dollar Sign Edit Descriptor

In an output specification, the dollar sign ($) edit descriptor suppresses a carriage return at the end of a line whose first character is a space or a plus sign (see Section 8.3 on carriage control characters). In an input specification, the dollar sign descriptor is ignored. The dollar sign descriptor is intended primarily for interactive I/O; it leaves the terminal print position at the end of the output text (rather than returning it to the left margin) so that a response can be typed immediately after the text.

For example, the statements

```
      TYPE 100
100   FORMAT (' ENTER RADIUS VALUE: ',$)
      ACCEPT 200
200   FORMAT (F6.2)
```

will produce the message

```
ENTER RADIUS VALUE:
```

on your terminal.

Your response (say, in this case, it is 12.0) can then go on the same line, as follows:

```
ENTER RADIUS VALUE:12.0
```

Note that the dollar sign descriptor used as a carriage control character instead of as a field descriptor accomplishes the same result. The following two formats are equivalent:

```
200 FORMAT (11H SIGN HERE:,$)

200 FORMAT (11H$SIGN HERE:)
```

8.1.22 Colon Edit Descriptor

The colon (:) edit descriptor terminates format control if no more items are in an I/O list. The colon descriptor has no effect if I/O list items remain. For example, the statements

```
      PRINT   100,3
      PRINT   200,4
100   FORMAT(' I=',I2,  ' J=' ,I2)
200   FORMAT(' K=',I2,:,' L=' ,I2)
```

print the two lines:

```
I= 3 J=
K= 4
```

Section 8.7 describes format control in detail.

8.1.23 Scale Factor

A scale factor is a value used in a format specifier that determines the location of the decimal point in real, double precision, or complex values.

The scale factor has the form:

```
nP
```

n
A signed or unsigned integer constant in the range –127 through +127. This integer constant specifies the number of positions to the left or right that the decimal point is to move.

Rules in Effect for Both Input and Output

- If you do not use a scale factor, a default scale factor of 0P applies.
- The scale factor is set to 0P at the start of every I/O statement.
- A scale factor applies to all subsequent F, E, D, or G field descriptors until a new scale factor is specified.
- The scale factor can appear as a field descriptor. For example, in the statement

  ```
  10 FORMAT (X, I4, E6.3, 3P, 2A3, 2I2, 2F5.3, E8.5)
  ```

  the 3P applies to the 2F5.3 descriptor and to the E8.5 descriptor, but not to the E6.3 descriptor or to the X, I, or A descriptors.

- A scale factor can appear as a prefix to an F, E, D or G field descriptor. For example, in the statement

```
10 FORMAT (3P2F5.3, E8.5)
```

  3P applies both to 2F5.3 and to E8.5.
- Format reversion (see Section 8.7) has no effect on the scale factor. For example, given the statement

```
10 FORMAT (X, F3.2, E3.2, 2PE4.2, F4.2, 3PE4.2)
```

  suppose two records are read, with reversion occurring to the start of the format. In the second record, the active scale factor 3P now applies to F3.2.
- A scale factor of 0P can be reinstated only by an explicit 0P specification in the format.

Additional Rules in Effect for Input

- If the external field contains an exponent, the scale factor has no effect.
- If the external field does not contain an exponent, the scale factor specifies multiplication of the value by 10**-n and assignment of the result to the corresponding I/O list element.

  For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left. A -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right.

Input Examples

Format	External Field	Internal Value
3PE10.5	37.614	.037614
3PE10.5	37.614E2	3761.4
-3PE10.5	37.614	37614.0

Additional Rules in Effect for Output

- Scale factors apply only to data output. The values of the I/O list variables do not change.
- For the F field descriptor, the value of the I/O list element is multiplied by 10**n before this value is transferred to the external record. Therefore, a positive scale factor moves the decimal point to the right, and a negative scale factor moves the decimal point to the left.

- For the E or D field descriptor, the basic real constant part of the value (see Section 2.3.2) is multiplied by 10**n, and n is subtracted from the exponent. Therefore, a positive scale factor moves the decimal point to the right and decreases the exponent, and a negative scale factor moves the decimal point to the left and increases the exponent.
- Because the G field descriptor supplies its own scaling function, a scale factor has no effect on a G field descriptor when the magnitude of the data to be output is within the effective range of the descriptor. When the magnitude of the data value is outside the range of the G field descriptor, the G field descriptor functions as an E field descriptor; therefore, the scale factor has the same effect as it does for the E field descriptor.

Output Examples

Format	Internal Value	External Representation
1PE12.3	-270.139	ΔΔ-2.701E+02
1PE12.2	-270.139	ΔΔΔ-2.70E+02
-1PE12.2	-270.139	ΔΔΔ-0.03E+04

8.1.24 Complex Data Editing

Input and output of complex values is governed by pairs of successive real field descriptors that use any combination of the forms Fw.d, Ew.dEe, Dw.d, or Gw.dEe.

Rule in Effect for Input

- During input, the two successive fields comprising a complex value are read under the control of repeated or successive real field descriptors and assigned to a complex I/O list element as the value's real and imaginary parts, respectively.

Input Examples

Format	External Field	Internal Value
F8.5,F8.5	1234567812345.67	123.45678, 12345.67
E9.1,F9.3	734.432E8123456789	734.432E8, 123456.789

Rules in Effect for Output

- During output, the two parts of a complex value are transferred to an external record under the control of repeated or successive field descriptors.
- The two parts are transferred consecutively, without punctuation or spacing, unless the format specifier states otherwise.

Output Examples

Format	Internal Value	External Representation
2F8.5	2.3547188, 3.456732	Δ2.35472 Δ3.45673
E9.2,'Δ,Δ',E5.3	47587.222, 56.123	Δ0.48E+06Δ,Δ*****

8.1.25 Repeat Counts and Group Repeat Counts

You can apply any field descriptor except H, T, P, or X to a number of successive data fields by preceding the field descriptor with an unsigned nonzero integer constant that specifies the number of applications, or repetitions, desired. This integer constant is called a repeat count. For example, the following two statements are equivalent:

```
20   FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)

20   FORMAT (3E12.4,4I5)
```

Similarly, you can apply a group of field descriptors repeatedly by enclosing the group in parentheses and preceding it with an unsigned nonzero integer constant. This integer constant is called a group repeat count. For example, the following two statements are equivalent:

```
50    FORMAT (2I8,3(F8.3,E15.7))

50    FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7)
                      \ /       \ /       \ /
                       1         2         3
```

To repeat an H or X field specification (for example, 20H), you can enclose it in parentheses and treat it as a group repeat specification (for example, 5(20H)).

If you do not specify a group repeat count, a default count of 1 is assumed.

8.1.26 Default Field Descriptors

If you write the field descriptors I, O, Z, L, F, E, D, G, or A without specifying a field width value, default values for w, d, and e are supplied on the basis of the data type of the I/O list element. Note that for F, E, D, and G, you must specify w.d[Ee] or nothing.

Table 8–2 lists the default values for w, d, and e. Notice that for the A field descriptor, the default for w is the length of the corresponding I/O list element.

Table 8–2: Default Field Widths

Field Descriptor	List Element Data Type	w	d	e
I, O, Z	INTEGER*2	7		
I, O, Z	INTEGER*4	12		
O, Z	CHARACTER*n	(see Note)		
O, Z	LOGICAL*1, BYTE	7		
O, Z	REAL	12		
O, Z	DOUBLE PRECISION	23		
L	LOGICAL	2		
F, E, G, D	REAL, COMPLEX	15	7	2
F, E, G, D	DOUBLE PRECISION	25	16	2
A	LOGICAL*1 or BYTE	Δ1		
A	LOGICAL*2,INTEGER*2	Δ2		
A	LOGICAL*4,INTEGER*4	Δ4		
A	REAL, COMPLEX	Δ4		
A	DOUBLE PRECISION	Δ8		
A	CHARACTER*n	Δn		

Note: The default value of w is:

(n*8)/3+1 if (n*8 MOD 3) = 0
(n*8)/3+2 otherwise

8.2 Variable FORMAT Expressions

You can use an expression in a FORMAT statement any place you can use an integer (except in the specification of the number of characters in the H field), by enclosing it in angle brackets. For example, the statement

```
FORMAT (I<J+1>)
```

performs an I (integer) data transfer with a field width one greater than the value of J at the time the format is scanned. The expression is reevaluated each time it is encountered in a normal format scan. If the expression is not of integer data type, its evaluated result is converted to integer data type before it is used. You can use any valid FORTRAN expression, including function calls and references to dummy arguments.

The value of a variable format expression must obey the same restrictions on magnitude that apply to an integer constant in a format, or an error occurs.

The following example shows a variable format expression.

```
      DIMENSION A(5)
      DATA A/1.,2.,3.,4.,5./
C
      DO 10 I = 1,10
      WRITE (6,100) I
100   FORMAT(I<MAX(I,5)>)
10    CONTINUE
C
      DO 20 I = 1,5
      WRITE (6,101) (A(I),J=1,I)
101   FORMAT (<I>F10.<I--1>)
20    CONTINUE
      END
```

On execution, these statements produce the following output:

```
1
2
3
4
5
 6
  7
   8
    9
    10
    1.
   2.0          2.0
  3.00         3.00         3.00
 4.000        4.000        4.000        4.000
5.0000       5.0000       5.0000       5.0000       5.0000
```

8.3 Carriage Control Characters

The first character of every record transferred to a printer is assumed by the system to be a carriage control character (except when overridden by the OPEN statement specification CARRIAGECONTROL = 'LIST' or 'NONE'); this character is not printed. The FORTRAN I/O system recognizes certain characters as carriage control characters. Table 8–3 lists these characters and their effects.

Table 8–3: Carriage Control Characters

Character	Effect
△ (space)	Advances one line
0 (zero)	Advances two lines
1 (one)	Advances to top of the next page
+ (plus)	Does not advance (allows overprinting)
$ (dollar sign)	Advances one line before printing and suppresses carriage return at the end of the record

Any character other than those listed in Table 8–3 is treated as a space and is deleted from the print line. Note that if you accidentally omit the carriage control character, the first character of the record is not printed.

8.4 Format Specification Separators

When the next value in an I/O list is to be transferred to or from the current record, you use a comma to separate the relevant field descriptor from the preceding one. However, when the next value is to be transferred to or from the next succeeding record, you use a slash (/) to separate the relevant field descriptor from the preceding one. For example, the statements

```
      WRITE (6,40) K,L,M,N,O,P
40    FORMAT (3O6/I6,2F8.4)
```

are equivalent to the following:

```
      WRITE (6,40) K,L,M
40    FORMAT (3O6)
      WRITE (6,50) N,O,P
50    FORMAT (I6,2F8.4)
```

You can use multiple slashes to bypass input records or to output blank records. If n consecutive slashes appear between two field descriptors, (n–1) records are skipped on input or (n–1) blank records are output. The first slash terminates the current record; the second slash terminates the first skipped or blank record; and so on.

However, n slashes at the beginning or end of a format specification result in n skipped or blank records, because the opening and closing parentheses of the format specification are themselves a record initiator and terminator, respectively. For example, the statements

```
      WRITE (6,99)
99    FORMAT ('1',T51,'HEADING LINE'//T51,'SUBHEADING LINE'//)
```

produce the following output:

```
Column 50, top of page

           HEADING LINE
 (blank line)
           SUBHEADING LINE
 (blank line)
 (blank line)
```

8.5 External Field Separators

A field descriptor such as Fw.d specifies that an input statement is to read w characters from an external record. If the data field in the external record contains fewer than w characters, the input statement reads characters from the next data field in the external record, unless you have padded the short field with leading zeros or spaces. When the field descriptor is numeric, you can avoid having to pad the input field by using a comma to terminate the field; the comma overrides the field descriptor's field-width specification. Using a comma to override a field descriptor's field-width specification is called short field termination and is particularly useful when you are entering data from a terminal keyboard. You can use it with the I, O, Z, F, E, D, G, and L field descriptors. For example, if the statements

```
      READ (5,100) I,J,A,B
100   FORMAT (2I6,2F10.2)
```

read the record

```
1,-2,1.0,35
```

the following assignments occur:

```
I = 1

J = -2

A = 1.0

B = 0.35
```

The physical end of the record also serves as a field terminator.

Note that the d part of a w.d specification is not affected by an external field separator. Therefore, you should always include an explicit decimal point in an external field for F, E, D, and G field descriptors.

You can use a comma to terminate fields only when those fields are less than w characters long. If a comma follows a field of w characters or more, the comma is considered part of the next field.

Two successive commas, or a comma after a field of exactly w characters, constitutes a null (zero-length) field. Depending on the field descriptor specified, the resulting value assigned is 0, 0.0, 0.D0, or .FALSE..

You cannot use a comma to terminate a field that is controlled by an A, H, or alphanumeric-literal field descriptor. However, if a record being read under the control of an A, H, or alphanumeric-literal field descriptor reaches its physical end before w characters are read, short-field termination occurs and the characters that have been read are successfully assigned. Trailing spaces are appended as required by the corresponding I/O list element or the field descriptor.

8.6 Run-Time Formats

You can store format specifications in arrays (numeric or character), array elements, character variables, and character substrings to use at run time. These format specifications are called run-time formats and can be constructed or altered during program execution.

Virtual arrays must not be used for storing specifications for run-time formats.

A run-time format in an array has the same form as that of a FORMAT statement, without the word FORMAT and the statement label. The opening and closing parentheses are required. Variable format expressions are not permitted.

Run-time formats are especially useful when you cannot know before execution time exactly which field descriptors will be required. To solve this problem, you can write a program to create a format with field descriptors that depend on the attributes of the relevant data.

The following example demonstrates the use of run-time formats:

```
      REAL TABLE(10,5)
      CHARACTER*26 FMT
      CHARACTER*5 FBIG,FMED,FSML
      DATA FMT(1:1)/'('/,FMT(26:26)/')'/
      DATA
      FMT(6:6)/',',/,FMT(11:11)/',',/,FMT(16:16)/',',/,FMT(21:21)/',',/
      DATA FBIG,FMED,FSML/'F8.2','F9.4','F9.6'/
      DO 10 I=1,10
      DO 15 J=1,5
      TABLE(I,J)=100.
15    CONTINUE
10    CONTINUE
      DO 20 I=1,10
      DO 18 J=1,5
           IF (TABLE(I,J).GE.100) THEN
                FMT(5*(J-1)+2:5*(J-1)+5=FBIG
           ELSEIF(TABLE(I,J).LE.0.1) THEN
                FMT(5*(J-1)+2:5*(J-1)+5)=FMED
           ELSE
                FMT(5*(J-1)+2:5*(J-1)+5)=FSML
           ENDIF
18    CONTINUE
      TYPE *, FMT
      WRITE(6,FMT)(TABLE(I,J), J=1,5)
20    CONTINUE
      END
```

In the above example, the given data is stored in the real array TABLE. The magnitudes of the data stored in the elements of TABLE will not be known until just before output. The format specification is stored in the character variable FMT. A left parenthesis is stored in the first character of FMT, and a right parenthesis is stored in the last character of FMT. A selection of field descriptors is stored in the character variables FBIG, FMED, FSML. The choice of field descriptors to be assigned to FMT is made to depend on the magnitudes of the data in TABLE. Finally, the output statement references FMT instead of a format statement label.

Each time an I/O statement referencing a run-time format is executed, the format is compiled (or recompiled) and assigned a storage location. Data read into that location through use of the H field descriptor is not stored in the array holding the format. At the end of the I/O statement, the data is lost.

8.7 Format Control Interaction with Input/Output Lists

Format control begins with execution of a formatted I/O statement. During format control, the action taken depends on information provided jointly by the next element of the I/O list (if one exists) and the next field descriptor of the format specification. The I/O list and the format specification are correlated from left to right, except when repeat counts are specified.

If the I/O statement contains an I/O list, you must specify at least one I, O, Z, F, E, D, G, L, A, or Q field descriptor in the format, or an error occurs.

On execution, a formatted input statement reads one record from the specified unit and initiates format control. Thereafter, additional records can be read as indicated by the format specification. Format control requires that a new record be read when a slash occurs in a format specification, or when the last closing parenthesis of a format specification is reached before all the elements in the corresponding I/O list have been assigned values. When this new record is read, any remaining characters read from the current record are discarded.

A formatted output statement transmits a record to the specified unit as format control terminates. Records can also be written during format control if a slash appears in the controlling format specification, or if the last closing parenthesis in the controlling format specification is reached and more I/O list elements remain to be transferred.

Each I, O, Z, F, E, D, G, L, A, and Q field descriptor corresponds to one element in an I/O list. No list element corresponds to an H, X, P, T, BN, BZ, $, :, TL, TR, S, SP, SS, or field descriptor. In H and character-constant field descriptors, data transfer occurs directly between an external record and the storage location of the format specification.

In format control, when an I, O, Z, F, E, D, G, L, A, or Q field descriptor is encountered, the I/O list is checked for a corresponding element. If a corresponding element is found, data is transferred and, if appropriate, translated between the external record and the list element. If a corresponding element is not found, format control terminates.

When the last closing parenthesis of the format specification is reached, format control determines whether there are any more I/O list elements remaining to be processed. If there are no more, format control terminates. However, if additional list elements remain, part or all of the format specification is reused in a process called format reversion.

Format reversion consists of the termination of the current record and the starting of a new record. Format control reverts to the group repeat specification whose left parenthesis is complemented by the next-to-last right parenthesis of the format specification. If the format does not contain a group repeat specification, format control returns to the beginning of the format specification.

Examples of format reversion follow.

```
      READ (I,100) A, B, C, D, E, F
100   FORMAT (F8.3, F8.3)
```

In this example, three records containing two fields are read. The first record assigns values to A and B; the second to C and D; and the third to E and F.

```
      READ (4,200) N, A, I1, I3, I2, I4, I11, I13, I12, I14
200   FORMAT (I2, F8.3,2(I2, I4))
```

In this example, format control reverts to the group repeat specification 2(I2, I4), and I/O list elements I11, I13, I12, and I14 are assigned values from the next record.

```
      DIMENSION A(5,5),B(5)
      WRITE (6,10)X,(I,B(I),(A(I,J),J=1,5),I=1,5)
10    FORMAT (E10.3/(I5,E10.3, 5(F8.5)))
```

In this example, format reversion returns to the group repeat specification that begins with I5.

8.8 Summary of Rules for Format Statements

The following sections summarize the rules for constructing and using the format specifications and their components, and for constructing external fields and records. Table 8–4 summarizes the FORMAT codes.

Table 8–4: Summary of FORMAT Codes

Code	Form	Effect
I	Iw[.m]	Specifies transfer of decimal integer values
O	Ow[.m]	Specifies transfer of octal integer values
Z	Zw[.m]	Specifies transfer of hexadecimal integer values

Table 8–4 (Cont.): Summary of FORMAT Codes

Code	Form	Effect
F	Fw.d	Specifies transfer of real or double-precision values in basic real form
E	Ew.d[Ee]	Specifies transfer of real or double-precision values in exponential form
D	Dw.d	Specifies transfer of real or double-precision values in double-precision exponential form with a D instead of an E
G	Gw.d[Ee]	Specifies transfer of real or double-precision values: on input, acts like F code; on output, acts like E code or F code
L	Lw	Specifies transfer of logical data: on input, transfers T, t, .T, .t, F, f, .F, or .f; on output, transfers T or F
A	A[w]	Specifies transfer of character or Hollerith values
H	nHc . . . c	Specifies transfer of Hollerith values between an external record and the format storage location
X	nX	Specifies that n characters are to be skipped on input or that n spaces are to be skipped on output
S	S	Reinvokes optional plus characters in numeric output fields: counters the action of SP and SS
SP	SP	Writes plus characters that would otherwise be optional into numeric output fields
SS	SS	Suppresses optional plus characters in numeric output fields
T	Tn	Specifies the position, in the external record, of the next character to be processed
TL	TLn	Relative tabulation specifier (left)
TR	TRn	Relative tabulation specifier (right)
Q	Q	Specifies the number of characters remaining to be transferred in an input record
$	$	Suppresses carriage return during interactive I/O
:	:	Terminates format control if the I/O list is exhausted
BN	BN	Specifies that embedded and trailing blanks in a numeric input field are to be ignored
BZ	BZ	Specifies that embedded and trailing blanks in a numeric input field are to be treated as zeros

8.8.1 General Rules

- A FORMAT statement must always be labeled.
- In a field descriptor such as rIw or nX, the terms r, w, m, and n must be unsigned integer constants greater than 0. (They cannot be names assigned to constants in PARAMETER statements.) You can omit the repeat count and field width specification.
- In a field descriptor such as Fw.d, the term d must be an unsigned integer constant. If w is specified, then you must specify d in F, E, D, and G field descriptors even if it is 0; and the field width specification (w) must be greater than or equal to d. The decimal point is also required. You must either specify both w and d or omit them both. In a field descriptor such as Ew.dEe, the term e must also be an unsigned integer constant.
- In a field descriptor such as nHc1c2 . . . cn, exactly n characters must follow the H format code. You can use any printing ASCII character in this field descriptor.
- In a scale factor of the form nP, n must be a signed or unsigned integer constant in the range –127 through 127 inclusive. The scale factor affects the F, E, D, and G field descriptors only. Once you specify a scale factor, it applies to all subsequent F, E, D, and G field descriptors in that format specification until another scale factor appears. You must explicitly specify 0P to reinstate a scale factor of zero. Format reversion does not affect the scale factor.
- No repeat count is permitted for BN, BZ, $, :, H, X, T, TR, TL, S, SP, SS, or character constant field descriptors, unless these descriptors are enclosed in parentheses and treated as a group repeat specification.
- If the associated I/O statement contains an I/O list, the format specification must contain at least one field descriptor other than H, X, P, T, or a character constant.
- A run-time format specification must be constructed in the same way as a format specification in a FORMAT statement, including the opening and closing parentheses. The word FORMAT and the statement label only are omitted.
- If a character-constant format includes apostrophes, those apostrophes must be represented by double apostrophes.

8.8.2 Input Rules

- A minus sign must precede a negative value in an external input field; a plus sign is optional before a positive value.
- On input, an external field under I field descriptor control must be an integer constant. It cannot contain a decimal point or an exponent. An external field under O field descriptor control must contain only the numerals 0 through 7. An external field input under Z field descriptor control must contain only the numerals 0 through 9 and the letters A(a) through F(f). An external field under O or Z field descriptor control must not contain a sign, a decimal point, or an exponent. You cannot use octal and hexadecimal constants in the form '777'O or 'AF9'X in external records.
- On input, an external field under F, E, D, or G field descriptor control must be an integer constant or a real or double-precision constant. It can contain a decimal point and/or an E or D exponent field.
- If an external field contains a decimal point, the actual size of the fractional part of the field, as indicated by that decimal point, overrides the d specification of the corresponding real or double-precision field descriptor.
- If an external field contains an exponent, the scale factor (if any) of the corresponding field descriptor has no effect on the conversion of that field.
- The field width specification must be large enough to accommodate both the numeric character string of the external field and any other characters that are allowed (algebraic sign, decimal point, and/or exponent).
- A comma is the only character you can use as an external field separator. It terminates input of numeric fields that are shorter than the number of characters expected. It also designates null (zero-length) fields.

8.8.3 Output Rules

- A format specification cannot specify more output characters than the external record can contain. For example, a line printer record cannot contain more than 133 characters, including the carriage control character.
- The field width specification (w) must be large enough to accommodate all characters that the data transfer can generate, including an algebraic sign, decimal point, and exponent. For example, the field width specification in an E field descriptor should be large enough to contain d+7 characters.
- The first character of a record output to a line printer or terminal is used for carriage control; it is not printed. The first character of such a record should be a space, 0, 1, $, or +. Any other character is treated as a space and is deleted from the record.

Chapter 9

Auxiliary Input/Output Statements

The auxiliary input/output statements perform file management functions. The auxiliary I/O statements and their respective functions are as follows:

- OPEN—establishes a connection between a logical unit and a file or device, and specifies the attributes required for read and write operations
- CLOSE—terminates the connection between a logical unit and a file or device
- REWIND and BACKSPACE—perform file-positioning functions
- DELETE—deletes records in a relative or indexed file
- UNLOCK—frees locked records for other users in a shared-file environment
- ENDFILE—writes a special record that causes an end-of-file condition (and an END= transfer) when an input statement reads the record

See Section 7.2 for a definition of the I/O components of these statements.

9.1 OPEN Statement

An OPEN statement either connects an existing file to a logical unit or creates a new file and connects this new file to a logical unit. In addition, OPEN can specify file attributes that control file creation and/or subsequent processing.

The OPEN statement has the form:

```
OPEN(par[,par]...)
```

par
A parameter, or keyword specification, in one of the forms:

```
kwd
kwd = value
```

kwd
A keyword, as described below.

value
Depends on the keyword, as described below.

Keywords are divided into the following functional categories.

- Keywords that identify the unit and file:

UNIT	–	logical unit number to be used
FILE or NAME	–	file name specification for the file
STATUS or TYPE	–	file existence status at OPEN
DISPOSE	–	file existence status after CLOSE

- Keywords that describe the file processing to be performed:

ACCESS	–	FORTRAN access method to be used
ORGANIZATION	–	logical file structure
READONLY	–	write protection

- Keywords that describe the records in the file:

BLOCKSIZE	–	size of I/O transfer buffer
CARRIAGECONTROL	–	type of printer control
FORM	–	type of FORTRAN record formatting
RECL or RECORDSIZE	–	logical record length
RECORDTYPE	–	logical record structure
BLANK	–	blank interpretation for numeric input
KEY	–	key field definition

- Keywords that describe file storage allocation when a file is created:

INITIALSIZE	–	initial file storage allocation
EXTENDSIZE	–	file storage allocation increment size

- Keywords that provide additional capability for direct access I/O:

ASSOCIATEVARIABLE	–	variable holding the next direct access record number
MAXREC	–	maximum direct access record number

- Optional keywords that provide improved performance or special capabilities:

ERR	–	statement to which control is transferred if an error occurs during execution of the OPEN statement
BUFFERCOUNT	–	number of I/O buffers to use
NOSPANBLOCKS	–	records are not to be split across physical blocks
SHARED	–	other programs can simultaneously access the file
USEROPEN	–	option to provide a user-written external function that controls the opening of the file

NOTE

Not all PDP–11 operating systems support all keywords and options. Consult the *PDP–11 FORTRAN–77 User's Guide* for information on system-specific restrictions.

Table 9–1 lists in alphabetical order the keywords and their possible associated values, including default values.

Table 9–1: OPEN Statement Keyword Values

Keyword	Values[1]	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' 'APPEND' 'KEYED'	Access method	'SEQUENTIAL'
ASSOCIATEVARIABLE	v	Next record number in direct access	No associate variable
BLANK	'NULL' 'ZERO'	Interpretation of blanks	'NULL' (/F77)
BLOCKSIZE	e	Size of I/O buffer	System default
BUFFERCOUNT	e	Number of I/O buffers	System default
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Print control	'FORTRAN' (formatted) 'NONE' (unformatted)
DISPOSE DISP	'SAVE' or 'KEEP' 'PRINT' 'DELETE'	File disposition at close	'SAVE'
ERR	s	Error transfer label	No error transfer
EXTENDSIZE	e	File storage allocation increment	Volume or system default
FILE NAME	c	File name specification	Depends on unit and system
FORM	'FORMATTED' 'UNFORMATTED'	Format type	Depends on ACCESS keyword
INITIALSIZE	e	File storage allocation	No allocation

[1]c is a character constant, array name, variable name, array element name, or a character substring reference.
e is an integer, real, or double-precision expression. The value of this expression is converted to the integer data type before it is used.
k is a key specification.
p is an external function.
s is a statement label.
v is an integer variable name.

Table 9–1 (Cont.): OPEN Statement Keyword Values

Keyword	Values[1]	Function	Default
KEY	(k[,k] . . .)	Indexed file key fields	No default
MAXREC	e	Maximum record number in direct access	No maximum
NOSPANBLOCKS	–	Records do not span blocks	Records can span blocks
ORGANIZATION	'SEQUENTIAL' 'RELATIVE' 'INDEXED'	File structure	'SEQUENTIAL'
READONLY	–	Write protection	No write protection
RECL RECORDSIZE	e	Record length	Depends on TYPE, ORGANIZATION, and RECORDTYPE keywords
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED'	Record structure	Depends on ACCESS and FORM keywords
SHARED	–	File sharing allowed	File sharing not allowed
STATUS TYPE	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'	File status at open	'UNKNOWN' (/F77)
UNIT	e	Logical unit number	No default
USEROPEN	p	User program option	No option

[1] c is a character constant, array name, variable name, array element name, or a character substring reference.
e is an integer, real, or double-precision expression. The value of this expression is converted to the integer data type before it is used.
k is a key specification.
p is an external function.
s is a statement label.
v is an integer variable name.

Keyword specifications can appear in any order. Determining whether they are optional and which ones are required depends upon the type of

file you are establishing or have established, and upon what you plan to do with it.

Some examples follow.

```
OPEN (UNIT=1, ERR=100)
```

This example creates a new sequential formatted file on unit 1 with the default file name.

```
OPEN (UNIT=3, STATUS='SCRATCH', ACCESS='DIRECT',
      INITIALSIZE=50, RECL=64)
```

This example creates a 50-block sequential file to be used with direct access. The file is deleted at program termination.

```
OPEN (UNIT=I, FILE='MTO:MYDATA.DAT', BLOCKSIZE=8192,
      STATUS='NEW', ERR=14, RECL=1024, RECORDTYPE='FIXED')
```

This example creates a file on magnetic tape with a large block size for efficient processing.

```
OPEN (UNIT=I, FILE='MTO:MYDATA.DAT', READONLY, STATUS='OLD',
      RECL=1024, RECORDTYPE='FIXED', BLOCKSIZE=8192)
```

This example opens the file created in the previous example for input.

Example:

```
OPEN(UNIT=1,STATUS='NEW', ORGANIZATION='INDEXED',
     RECL=60,FORM='UNFORMATTED',
     KEY= (1:20, 30:33:INTEGER, 46:57), ACCESS='KEYED')
```

This statement creates a new indexed file specifying three keys: The primary key will be from byte 1 to 20; the first alternate key will be an integer key from byte 30 to 33; and the second alternate key will be from byte 46 to 57.

Sections 9.1.1 through 9.1.26 describe the OPEN statement keywords in detail.

9.1.1 ACCESS

ACCESS specifies the method of locating, reading, or writing records.

There are three access methods: sequential, direct, and, keyed. If you specify 'DIRECT', the file is accessed directly. If you specify 'SEQUENTIAL', the file is accessed sequentially. If you specify 'KEYED', the file is accessed by a specified key. 'APPEND' implies sequential access and positioning after the last record of the file. The default is 'SEQUENTIAL'.

An ACCESS specification has the form:

```
ACCESS= acc
```

acc
One of the character constants 'SEQUENTIAL', 'DIRECT', 'KEYED' or 'APPEND'.

If no ACCESS is specified, the default is 'DEQUENTIAL'.

Table 9–2 shows the valid combinations of ACCESS values and file organizations:

Table 9–2: Allowed Combinations of ACCESS Values and File Organizations

	ACCESS Value			
File Organization	**SEQUENTIAL**	**DIRECT**	**KEYED**	**APPEND**
Sequential	Yes	Yes[1]	No	Yes
Relative	Yes	Yes	No	No
Indexed	Yes	No	Yes	No

[1]Direct access to a sequential file requires that the records in the file be fixed length (see Section 9.1.19).

In sequential access, you must read or write records in sequence from the beginning of the file. (See Section 7.1.4.1.)

In direct access, you specify in an I/O statement the record number of the desired record, and the system selects that record. (See Section 7.1.4.2.)

In keyed access, you specify in an I/O statement the key value of the desired record, and the system selects the record having a matching key. (See Section 7.1.4.3.)

9.1.2 ASSOCIATEVARIABLE

ASSOCIATEVARIABLE specifies the integer variable that, after each direct access I/O operation, contains the record number of the next sequential record in a file. This specifier is ignored for sequential access or keyed access.

An ASSOCIATEVARIABLE specification has the form:

```
ASSOCIATEVARIABLE = asv
```

asv
An integer variable.

9.1.3 BLANK

BLANK specifies either that all blanks in a numeric input field are to be ignored (except if the field is all blanks, in which case it is treated as zero), or that all blanks other than leading blanks are to be treated as zeros. The default value is 'NULL'.

BLANK has the form:

```
BLANK = blnk
```

blnk
A character constant having a value equal to either 'NULL' or 'ZERO'.

If the /NOF77 compiler command qualifier is specified, the default value is 'ZERO'.

9.1.4 BLOCKSIZE

BLOCKSIZE specifies the size (in bytes) of the I/O transfer buffer.

I/O statements appear to transfer records directly between a file and the entities specified in the I/O list; however, the system actually transfers records between a file and an intermediate I/O buffer. BLOCKSIZE affects the size of this buffer.

A BLOCKSIZE specification has the form:

```
BLOCKSIZE = bks
```

bks
An integer expression.

For sequential files, BLOCKSIZE determines the number of disk blocks to transfer (for disk files), or the physical blocking factor (for magtape files). The default is the system default for the device.

For relative and indexed files, BLOCKSIZE determines a file's bucket size. A bucket is the number of disk blocks used as the unit of I/O transfer and as the unit of locking and control information. Each bucket contains control information as well as data.

See the *PDP-11 FORTRAN-77 User's Guide* for more information.

9.1.5 BUFFERCOUNT

BUFFERCOUNT specifies the number of buffers to be associated with a logical unit for multibuffered I/O. BLOCKSIZE, discussed in the previous section, specifies the size of each of these buffers. If you do not specify BUFFERCOUNT, or if you specify 0, the system default is used.

A BUFFERCOUNT specification has the form:

```
BUFFERCOUNT = bc
```

bc
An integer expression.

A specification of BUFFERCOUNT= -1 opens a file for block I/0.

9.1.6 CARRIAGECONTROL

CARRIAGECONTROL determines the kind of carriage control to be used when a file is printed. The default for formatted files is 'FORTRAN'; the default for unformatted files is 'NONE'. 'FORTRAN' specifies normal FORTRAN interpretation of the first character (see Section 8.3); 'LIST' specifies single spacing between records; and 'NONE' specifies no implied carriage control.

A CARRIAGECONTROL specification has the form:

```
CARRIAGECONTROL = cc
```

cc
The character constant 'FORTRAN', 'LIST', or 'NONE'.

9.1.7 DISPOSE

DISPOSE determines the disposition of a file connected to a unit when that unit is closed. If you specify 'SAVE' or 'KEEP', the file is retained after the unit is closed; file retention is the default operation. If you specify 'PRINT', the file is submitted to the system line printer spooler. (On some systems, the file is deleted after printing.) If you specify 'DELETE', the file is deleted. A read-only file (see Section 9.1.18) cannot be printed or deleted, and a scratch file (see Section 9.1.23) cannot be saved or printed.

A DISPOSE specification has the forms:

```
DISPOSE = dis
DISP = dis
```

dis
The character constant 'SAVE', 'KEEP', 'PRINT', or 'DELETE'.

9.1.8 ERR

ERR transfers control to a specified executable statement if an error occurs during execution of the OPEN statement containing it. The ERR specification applies only to the OPEN statement containing the ERR keyword, not to subsequent I/O operations on the specified unit. If an error does occur, no file is opened or created.

An ERR specification has the form:

```
ERR= s
```

s
The label of an executable statement.

9.1.9 EXTENDSIZE

EXTENDSIZE specifies the number of blocks a disk file is to be extended when additional file storage is allocated. If you do not specify EXTENDSIZE, or if you specify 0, the system default for the device is used.

An *EXTENDSIZE* specification has the form:

```
EXTENDSIZE = es
```

es
An integer expression.

9.1.10 FILE

FILE specifies the name of the file to be connected to a unit. The name can be any file specification accepted by the operating system. The *PDP-11 FORTRAN-77 User's Guide* describes default file name conventions.

If the file name is stored in a numeric variable, numeric array, or numeric array element, the name must consist of ASCII characters terminated by an ASCII null character (zero byte). However, if the file name is stored in a character variable, character array, or character array element, it must not contain a zero byte.

A FILE specification has the form:

```
FILE = fln
```

fln
An array name, variable name, array element name, character constant, or a character substring reference. You cannot use the name of a virtual array or virtual array element.

9.1.11 FORM

FORM specifies whether the file being opened is to be read from and written to with formatted or with unformatted I/O statements. For sequential access, 'FORMATTED' is the default. For direct or keyed access, 'UNFORMATTED' is the default. You must not mix formatted and unformatted I/O statements on the same unit.

A FORM specification has the form:

```
FORM = ft
```

ft
The character constant 'FORMATTED' or 'UNFORMATTED'.

9.1.12 INITIALSIZE

INITIALSIZE specifies the number of blocks allocated for a new file on a disk. If you do not specify INITIALSIZE, or if you specify 0, no initial allocation is made.

An INITIALSIZE specification has the form:

```
INITIALSIZE = insz
```

insz
An integer expression.

9.1.13 KEY

KEY designates fields to be used as key fields in an indexed file. These designated key fields must be included in an OPEN statement when an indexed file is created. Thereafter, all key information is available from the file itself. If key parameters are specified for an existing file, they must match the parameters of the existing file or an error occurs.

A KEY specification has the form:

```
KEY = (kspec [,kspec]...)
```

where each kspec has the form:

```
e1 : e2 [:dtn]
```

e1
The first byte position of the key.

e2
The last byte position of the key.

dtn
The data type of the key.

e1,e2
Integer expressions.

dtn
One of the following data-type names:

INTEGER—Integer key
CHARACTER—Character key

If dtn is omitted, the key data type is character.

The key starts at position e1 in a record and has a length of e2–e1+1. The values of e1 and e2 must be such that:

```
1  .LE.  (e1) .LE.  (e2) .LE.  record-length
1  .LE.  (e2-e1+1) .LE.  255
```

If the key type is INTEGER, the key length must be 2 or 4. There must be at least one key specification following KEY=; but there may be up to 255 key specifications. Each key specification defines a key field. The first key specification, kspec 0, defines the primary key. The second key specification, kspec 1, defines the first alternate key, and so on.

The order of a key specification in a list of key specifications (in a KEY specification) determines the key-of-reference number for that key (the key number to be used in subsequent I/O statements). Each key in a file must be specified in a key specification list.

Up to 254 alternate keys may be specified in a key specification list; however, at least one key—the primary key—must be specified.

9.1.14 MAXREC

MAXREC specifies the maximum number of records permitted in a direct access file. The default is no maximum number of records. MAXREC is ignored for other types of files.

A MAXREC specification has the form:

```
MAXREC = mr
```

mr
An integer expression.

9.1.15 NAME

NAME is a nonstandard synonym for FILE. See Section 9.1.10.

9.1.16 NOSPANBLOCKS

NOSPANBLOCKS specifies that records are not to cross disk block boundaries; it is used only for sequential files stored on disk. If any record exceeds the size of a disk block, an error occurs.

A NOSPANBLOCKS specification has the form:

```
NOSPANBLOCKS
```

9.1.17 ORGANIZATION

ORGANIZATION specifies the internal structure of a file. The default organization is 'SEQUENTIAL'. The organization of the file must always be specified for relative and indexed files.

An ORGANIZATION specification has the form:

```
ORGANIZATION= org
```

org
The character constant 'SEQUENTIAL', 'RELATIVE', or 'INDEXED'.

In sequential files, records are stored in the order in which they are written. In relative files, records are stored in fixed-length cells identified by an integer number. In indexed files, records are stored in a system-defined order; indexes or directories are maintained to locate records based on character strings or integer values, called keys, contained in the record.

Table 9–3 shows the valid combinations of ORGANIZATION keywords and access modes:

Table 9–3: Valid Access Modes for ORGANIZATION Keywords

	Access Mode			
File Organization	**Sequential**	**Direct**	**Keyed**	**Append**
SEQUENTIAL	Yes	Yes[1]	No	Yes
RELATIVE	Yes	Yes	No	No
INDEXED	Yes	No	Yes	No

[1]Direct access to a sequential file requires that the records in the file be fixed length (see Section 9.1.18).

For additional information, see the *PDP-11 FORTRAN-77 User's Guide*.

9.1.18 READONLY

READONLY prohibits a program from writing to a file.

A READONLY specification has the form:

```
READONLY
```

9.1.19 RECL

RECL specifies the logical record length.

If a file contains fixed-length records, RECL specifies the size of each record. If a file contains variable-length records, RECL specifies the maximum length for any record.

You must specify RECL when you create a file that is to have fixed-length records or that is to have relative organization.

A RECL specification has the form:

```
RECL = rl
```

rl
An integer expression.

The value of rl depends on the value of FORM (see Section 9.1.11). If the records are formatted, the length is the number of characters; if the records are unformatted, the length is the number of numeric storage units (four bytes).

For existing files, the default is the existing record size.

9.1.20 RECORDSIZE

RECORDSIZE is a nonstandard synonym for RECL. See Section 9.1.19.

9.1.21 RECORDTYPE

RECORDTYPE specifies whether a file has fixed-length records, variable-length records, or segmented records. When you create a file, the default record types for the various file types are as follows:

File Type	Default Record Type
Relative organization	'FIXED'
Indexed organization	'FIXED'
Direct access files	'FIXED'
Formatted sequential access files	'VARIABLE'
Unformatted sequential access files	'SEGMENTED'

Segmented records consist of one or more variable-length records and allow a FORTRAN logical record to span several physical records. However, they can only be used in sequential access, unformatted files with sequential organization. You cannot specify 'SEGMENTED' for any other file type.

NOTE

ASCIZ stream files are not directly supported by PDP-11 FORTRAN-77.

A RECORDTYPE specification has the form:

```
RECORDTYPE = typ
```

typ
The character constant 'FIXED', 'VARIABLE', or 'SEGMENTED'.

If you do not specify RECORDTYPE when you access an existing file, the record type of the file is used, unless the file is a sequential access, unformatted file with sequential organization; this file has a default of 'SEGMENTED'.

If you specify RECORDTYPE, typ must match the record type of the existing file.

In fixed-length record files, if an output statement does not specify a full record, the record is filled with spaces (for a formatted file) or zeros (for an unformatted file).

9.1.22 SHARED

SHARED specifies that a file is to be opened for shared access by more than one program executing simultaneously.

Sequential files may only be shared if they are stored on disk, and only one program may have write access.

Relative and indexed files may be shared with multiple programs having write access.

A SHARED specification has the form:

```
SHARED
```

See the *PDP-11 FORTRAN-77 User's Guide* for additional information on this keyword.

9.1.23 STATUS

STATUS specifies the status of file to be opened. If you specify 'OLD', the file must already exist. If you specify 'NEW', a new file is created. If you specify 'SCRATCH', a new file is created and then is deleted when the file is closed. If you specify 'UNKNOWN', the system will first try 'OLD'; if the file is not found, the system will assume 'NEW' and therefore create a new file. The default is 'UNKNOWN'.

A STATUS specification has the form:

```
STATUS = sta
```

sta
The character constant 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'.

If the /NOF77 compiler command qualifier is specified, the default value is 'NEW'.

You cannot specify STATUS='SCRATCH' for a file on magnetic tape. If you do, at run time your program will terminate with no error message when it encounters the OPEN statement.

NOTE

STATUS is also used in CLOSE statements to specify the status of a file after the file is closed; however, the values it uses are different from those used in OPEN statements.

9.1.24 TYPE

TYPE is a nonstandard synonym for STATUS. See Section 9.1.23.

9.1.25 UNIT

UNIT specifies the logical unit to which a file is to be connected. The UNIT keyword must appear in any OPEN keyword list. When an OPEN statement is executed, another file cannot be connected to the logical unit specified by the UNIT keyword in the OPEN statement.

There must not be a file converted to the logical unit at the time the OPEN statement is executed.

A UNIT specification has the form:

```
[UNIT] = u
```

u
An integer expression.

The optional character string UNIT= can be omitted only when the value of u will occupy the first position in the keyword list containing it.

9.1.26 USEROPEN

USEROPEN specifies a user-written external function that is to be invoked to control the opening of the specified file. USEROPEN allows knowledgeable users to employ features of the file management system not directly available from FORTRAN, yet retain the convenience of writing programs in FORTRAN.

A USEROPEN specification has the form:

```
USEROPEN = p
```

p
An external function name.

The external function name must be declared in an EXTERNAL statement in the program unit.

Consult the *PDP-11 FORTRAN-77 User's Guide* for information on using the USEROPEN keyword.

9.2 CLOSE Statement

The CLOSE statement disconnects a file from a unit.

The CLOSE statement has the form:

```
                  STATUS
CLOSE ([UNIT=]u , [DISPOSE = p] [,ERR=s])
                  DISP
```

u
A logical unit number.

p
A character constant that determines the disposition of the file; its values are 'SAVE', 'KEEP', 'DELETE', and 'PRINT'.

s
The label of an executable statement.

If you specify either 'SAVE' or 'KEEP', the file is retained after the unit is closed. If you specify 'PRINT', the file is submitted to the line printer spooler. (On some systems, the file is deleted after printing.) If you specify 'DELETE', the file is deleted. For scratch files, the default is 'DELETE'; for all other files, the default is 'SAVE'. The disposition specified in a CLOSE statement supersedes the disposition specified in a preceding OPEN statement; however, a file opened as a scratch file cannot be saved or printed, and a file opened for read-only access cannot be printed or deleted.

For example, the statement

```
CLOSE(UNIT=1,DISPOSE='PRINT')
```

closes the file on unit 1 and submits the file for printing. And the statement

```
CLOSE(UNIT=J,DISPOSE='DELETE',ERR=99)
```

closes the file on unit J and deletes it.

9.3 REWIND Statement

The REWIND statement repositions to the beginning of the file a sequential file currently open for sequential or append access.

The REWIND statement has the forms:

```
REWIND u
REWIND ([UNIT=]u[,ERR=s])
```

u
A logical unit number.

s
The label of an executable statement.

The unit number must refer to an open sequential file on disk or magnetic tape.

For example, the statement

```
REWIND 3
```

repositions logical unit 3 to the beginning of a currently open file.

You must not issue a REWIND statement for a file that is open for direct or keyed access or for a relative or indexed file.

9.4 BACKSPACE Statement

The BACKSPACE statement repositions an open sequential file to the beginning of the preceding record. When the next I/O statement for the unit is executed, this preceding record is the one processed.

The BACKSPACE statement has the forms:

```
BACKSPACE u
BACKSPACE ([UNIT=]u[,ERR=s])
```

u
A logical unit number.

s
The label of an executable statement.

The unit number must refer to an open sequential file on disk or magnetic tape.

For example, the statement

```
BACKSPACE 4
```

repositions the open file on logical unit 4 to the beginning of the preceding record.

You must not issue a BACKSPACE statement for a file that is open for direct, keyed, or append access, or for a relative or indexed file.

9.5 DELETE Statement

The DELETE statement deletes records in relative files and in indexed files. Specifically, this cause a record to be marked as deleted; records so marked are not accessible to subsequent READ or REWRITE statements.

The DELETE statement cannot be used with a sequential file.

There are two kinds of DELETE statement: sequential and direct.

9.5.1 Sequential DELETE Statement

The sequential DELETE statement deletes the last record that was read from a logical unit by a READ statement.

The sequential DELETE statement has the form:

```
DELETE ([UNIT=]u[,ERR=s])
```

u
A logical unit number.

s
The label of an executable statement.

For example, the statement

```
DELETE (11)
```

deletes the last record read from the file connected to logical unit 11.

9.5.2 Direct DELETE Statement

The direct DELETE statement deletes a record specified by a record number.

The direct DELETE statement has the forms:

```
DELETE (u'r[,ERR=s])
DELETE ([UNIT=]u,REC=r[,ERR=s])
```

u
A logical unit number.

r
The direct access record number.

s
The label of an executable statement.

For example, the statement

```
DELETE (1'I)
```

deletes the record specified by the value of I, located in the file connected to logical unit 1.

9.6 UNLOCK Statement

The UNLOCK statement unlocks records in a relative or indexed file. When a record is "locked," it cannot be accessed by any other program or logical unit.

A record accessed in a shared-file environment is automatically locked when a READ statement selects the record. The record is unlocked either when another I/O statement is executed on the same logical unit or when an UNLOCK statement is executed.

Attempts to access a locked record result in error messages.

The UNLOCK statement is used in place of an otherwise unnecessary I/O operation.

The UNLOCK statement has the forms:

```
UNLOCK u
UNLOCK ([UNIT=]u [,ERR=s])
```

u
A logical unit number.

s
The label of an executable statement.

The UNLOCK statement frees the locked records on the specified logical unit. If no record is locked, the statement has no effect. Records in a sequential file cannot be locked.

You must not issue an UNLOCK statement on a sequential file.

Consult the *PDP–11 FORTRAN–77 User's Guide* for information on file sharing and record locking.

9.7 ENDFILE Statement

The ENDFILE statement writes an end-file record to the specified unit. The ENDFILE statement has the forms:

```
ENDFILE u
ENDFILE ([UNIT=]u [,ERR=s])
```

u
A logical unit number.

s
The label of an executable statement.

You can write an end-file record only to sequentially accessed sequential files that contain variable-length or segmented records.

For example, the statement

```
ENDFILE 2
```

outputs an end-file record to logical unit 2.

Appendix A

Additional Language Elements

For the purpose of facilitating compatibility with other versions of PDP–11 FORTRAN, PDP–11 FORTRAN–77 includes the statements ENCODE, DECODE, DEFINE FILE, and FIND, and offers alternative syntax for the PARAMETER statement and octal constants. These language elements are discussed in Sections A.1 through Section A.5.

Section A.6 describes the interpretation of the EXTERNAL statement that applies when the /NOF77 compiler command qualifier is used. The FORTRAN–77 interpretation of the EXTERNAL statement (see Section 5.8) is incompatible with the previous ANSI standard and with previous DIGITAL FORTRAN implementations.

A.1 The ENCODE and DECODE Statements

The ENCODE and DECODE statements transfer data between variables or arrays in internal storage and translate that data from internal to character form, or from character to internal form, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential WRITE and READ statements.

The ENCODE and DECODE statements have the forms:

```
ENCODE(c,f,b [,ERR=s])[list]
DECODE(c,f,b [,ERR=s])[list]
```

c
An integer expression. (In the ENCODE statement, c is the number of characters (bytes) to be translated to character form. In the DECODE statement, c is the number of characters to be translated to internal form.)

f
A format identifier. (If more than one record is specified, an error occurs.)

b
The name of an array, array element, variable, or character substring reference. You cannot use the name of a virtual array or virtual array element. (In the ENCODE statement, b receives the characters after translation to external form. In the DECODE statement, b contains the characters to be translated to internal form.)

s
The label of an executable statement.

list
An I/O list. (In the ENCODE statement, the I/O list contains the data to be translated to character form. In the DECODE statement, the list receives the data after translation to internal form.)

The ENCODE statement translates the list elements to character form according to the format specifier and stores the characters in b, as does a WRITE statement. If fewer than c characters are transmitted, the remaining character positions are filled with spaces.

The DECODE statement translates the character data in b to internal (binary) form according to the format specifier and stores the elements in the list, as does a READ statement.

If b is an array, its elements are processed in the order of subscript progression.

The number of characters that the ENCODE or DECODE statement can process depends on the data type of b in that statement. For example, an INTEGER*2 array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array. A character variable or character array element can contain characters equal in number to its length. A character array can contain characters equal in number to the length of each element multiplied by the number of elements.

The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

An example of the ENCODE and DECODE statements follows:

```
      DIMENSION K(3)
      CHARACTER*12 A, B
      DATA A /'123456789012'/
      DECODE (12,100,A) K
100   FORMAT (3I4)
      ENCODE (12,100,B) K(3), K(2), K(1)
```

The DECODE statement translates the 12 characters in A to integer form (specified by statement 100) and stores them in array K, as follows:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

The ENCODE statement translates the values K(3), K(2), and K(1) to character form and stores the characters in the character variable B as follows:

```
B = '901256781234'
```

A.2 DEFINE FILE Statement

The DEFINE FILE statement describes direct access sequential files that are associated with a logical unit number. However, the OPEN statement (Section 9.1) can also be used to describe direct access sequential files, and is the preferred instrument.

The DEFINE FILE statement establishes the size and structure of a direct access file.

The DEFINE FILE statement has the form:

```
DEFINE FILE u (m,n,U,asv) [,u(m,n,U,asv)] ...
```

u
An integer constant or integer variable that specifies the logical unit number.

m
An integer constant or integer variable that specifies the number of records in the file.

n
An integer constant or integer variable that specifies the length, in 16-bit words (2 bytes), of each record.

U
Specifies that the file is unformatted (binary); this is the only acceptable entry in this position.

asv
An integer variable, called the associated variable of the file. At the end of each direct access I/O operation, the record number of the next higher-numbered record in the file is assigned to asv.

DEFINE FILE specifies that a file containing m fixed-length records of n 16-bit words each exists, or is to exist, on logical unit u. The records in the file are numbered sequentially from 1 through m.

DEFINE FILE must be executed before the first direct access I/O statement that refers to the specified file.

DEFINE FILE also establishes the integer variable asv as the associated variable of the file. At the end of each direct access I/O operation, the FORTRAN I/O system places in asv the record number of the record immediately following the one just read or written. Because the associated variable always points to the next sequential record in the file (unless it is redefined by an assignment, input, or FIND statement), direct access I/O statements can perform sequential processing of the file by using the associated variable of the file as the record number specifier.

For example, the statement

```
DEFINE FILE 3 (1000,48,U,NREC)
```

specifies that logical unit 3 is to be connected to a file of 1000 fixed-length records; each record is forty-eight 16-bit words long. The records are numbered sequentially from 1 through 1000 and are unformatted. After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the record just processed.

A.3 FIND Statement

The FIND statement positions a direct access file on a specified unit to a particular record. No data transfer takes place.

The FIND statement has the forms:

```
FIND (u'r[,ERR=s])
FIND ([UNIT=]u[,REC=r][,ERR=s])
```

u
A logical unit number.

r
The direct access record number.

s
The label of an executable statement.

The record number cannot be less than 1 or greater than the number of records defined for the file.

The associated variable of the file, if specified, is set to the direct access record number.

For a relative organization file, the record is locked.

For example, the statement

```
FIND (1'1)
```

positions logical unit 1 to the first record of the file; the file's associated variable is set to 1. And the statement

```
FIND (4'INDX)
```

positions the file to the record identified by the content of INDX; the file's associated variable is set to the value of INDX.

A.4 PARAMETER Statement

This statement assigns a symbolic name to a constant, as does the PARAMETER statement discussed in Section 5.11. However, it differs from the PARAMETER statement discussed in Section 5.11 in that its list is not bounded with parentheses and the form of the constant (rather than the typing of the symbolic name) determines the data type of the variable.

The PARAMETER statement has the following form:

```
PARAMETER  p=c  [,p=c] ...
```

p
A symbolic name.

c
An integer expression.

Each symbolic name (p) becomes a constant and is defined by the value of the constant (c); c can be any valid FORTRAN constant.

Once a symbolic name is defined to be a constant, it can appear any place in a program that a constant is allowable. The effect of using a symbolic name defined to be a constant is the same as if the constant were being used.

The symbolic name of a constant cannot appear as part of another constant; however, it can appear as a real or imaginary part of a complex constant.

The PARAMETER statement applies only to the program unit in which it appears. A symbolic name can appear only once in a PARAMETER statement in the same program unit.

The constant assigned to the symbolic name determines its data type. The initial letter of the constant's name does not affect its type. You cannot specify the constant's type by using the name in an explicit type declaration statement.

Examples of valid PARAMETER statements are:

```
PARAMETER  PI=3.1415927, DPI=3.14159265358979323800
PARAMETER  FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

A.5 Octal Forms of Integer Constants

Octal forms of integer constants are provided for compatibility with PDP–11 FORTRAN IV-PLUS V3.0. The octal form of an integer constant is:

```
"nn
```

nn
A string of digits in the range 0 to 7.

An octal integer constant cannot be negative or greater than "37777777777.

Examples of valid and invalid octal integer constants are:

Valid	Invalid	Explanation
"107	"108	Contains a digit outside the allowed range
"177777	"1377.	Decimal point not allowed
	"17777"	Trailing quotation mark not allowed

Note that these octal forms are not the same as the typeless octal constants discussed in Section 2.3.5. Integer constants in octal form have integer data type and are treated as integers.

A.6 /NOF77 Interpretation of the EXTERNAL Statement

The /NOF77 interpretation of the EXTERNAL statement combines the function of the INTRINSIC statement with that of the EXTERNAL statement discussed in Section 5.8. It is available only if the /NOF77 compiler command qualifier is present.

The /NOF77 EXTERNAL statement allows the programmer to use subprograms as arguments to other subprograms.

The subprograms to be used as arguments can be either user-supplied procedures or FORTRAN library functions.

The /NOF77 EXTERNAL statement has the form:

```
EXTERNAL [*]v [,[*]v]...
```

v
The symbolic name of a subprogram, or the name of a dummy argument associated with the symbolic name of a subprogram.

*
Specifies that a user-supplied function is to be used instead of a FORTRAN library function having the same name. See Section 6.3 for information on FORTRAN library functions.

The EXTERNAL statement declares that each name in the list is an external procedure name. Such a name can then appear as an actual argument to a subprogram; the subprogram can use the associated dummy argument name in a function reference or CALL statement.

Note, however, that a complete function reference used as an argument (for example, SQRT(B) in CALL SUBR(A,SQRT(B),C)) represents a value, not a subprogram name. The function name need not be defined in an EXTERNAL statement.

An example of the EXTERNAL statement is:

Main Program

```
EXTERNAL SIN,COS,SINDEG

   .
   .
CALL TRIG (ANGLE,SIN,SINE)
   .
   .
CALL TRIG (ANGLE,COS,COSINE)
   .
   .
CALL TRIG (ANGLE,SINDEG,SINE)
```

Subprograms

```
SUBROUTINE TRIG (X,F,Y)
EXTERNAL F
Y = F(X)
RETURN
END


FUNCTION SINDEG(X)
SINDEG = SIN (X*3.14159/180)
RETURN
END
```

In the example, SIN and COS are trigonometric functions supplied in the FORTRAN library, and SINDEG is a user-supplied function. The CALL statements pass the name of a function to the subroutine TRIG. The function reference F(X) subsequently invokes the function in the second statement of TRIG. Depending on which CALL statement invoked TRIG, the second statement is equivalent to one of the following:

```
Y = SIN(X)
Y = COS(X)
Y = SINDEG(X)
```

An asterisk (*) may precede a name in the list; the name then identifies a user-supplied function or subprogram, not a FORTRAN library function. Use the asterisk only when a user-supplied function or subprogram has the same name as that of a FORTRAN library function. (See Section 6.3 for additional information on FORTRAN library functions.)

For example, the statement:

```
EXTERNAL *SIN, *COS
```

identifies the names SIN and COS as user-supplied subprograms and not the FORTRAN library functions for the sine and cosine.

Appendix B

Character Sets

B.1 FORTRAN Character Set

The FORTRAN character set consists of:

- The letters A through Z and a through z
- The numerals 0 through 9
- The following special characters:

Character	Name	Character	Name
△	Space or tab	'	Apostrophe
=	Equal sign	"	Quotation mark
+	Plus sign	$	Dollar sign
-	Minus sign	,	Comma
*	Asterisk	!	Exclamation point
/	Slash	:	Colon
(	Left parenthesis	<	Left angle bracket
)	Right parenthesis	>	Right angle bracket
.	Period		

Other printing characters can appear in a FORTRAN statement only as part of a Hollerith constant. Any printing character can appear in a comment. See Table B–1.

B.2 ASCII Character Set

Table B–1 is a table representing the ASCII character set. At the top of the table are hexadecimal digits (0 to 7), and to the left of the table are hexadecimal digits (0 to F). To determine the hexadecimal value of an ASCII character, locate the ASCII character in the table, use the row number as the unit's position digit, and use the column number as the 16's position digit. For example, the hexadecimal value of the equal sign (=) is 3D.

Table B–1: ASCII Character Set

	COLUMNS							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	NUL	DLE	SP	0	@	P		p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M	]	m	
E	SO	RS	.	>	N	^	n	
F	SI	US	/	?	O	_	o	DEL

Table B–1 (Cont.): ASCII Character Set

	COLUMNS							
	0	1	2	3	4	5	6	7
NUL	Null				DLE	Data Link Escape		
SOH	Start of Heading				DC1	Device Control 1		
STX	Start of Text				DC2	Device Control 2		
ETX	End of Text				DC3	Device Control 3		
EOT	End of Transmission				DC4	Device Control 4		
ENQ	Enquiry				NAK	Negative Acknowledge		
ACK	Acknowledge				SYN	Synchronous Idle		
BEL	Bell				ETB	End of Transmission Block		
BS	Backspace				CAN	Cancel		
HT	Horizontal Tabulation				EM	End of Medium		
LF	Line Feed				SUB	Substitute		
VT	Vertical Tab				ESC	Escape		
FF	Form Feed				FS	File Separator		
CR	Carriage Return				GS	Group Separator		
SO	Shift Out				RS	Record Separator		
SI	Shift In				US	Unit Separator		
SP	Space				DEL	Delete		

B.3 RADIX-50 Constants and Character Set

Radix–50 is a special character data representation in which up to 3 characters can be encoded and packed into 16 bits. The Radix–50 character set is a subset of the ASCII character set.

The Radix–50 characters and their corresponding code values are:

Character	ASCII Octal Equivalent	Radix-50 Value (Octal)
Space	40	0
A - Z	101 - 132	1 - 32
$	44	33
	56	34
(Unassigned)		35
0 - 9	60 - 71	36 - 47

Radix–50 values are stored, up to 3 characters per word, by packing them into single numeric values according to the formula:

```
((i * 50 + j) * 50 + k)
```

where i, j, and k represent the code values of three Radix–50 characters.

Thus, the maximum Radix–50 value is:

```
47*50*50 + 47*50 + 47 = 174777
```

A Radix–50 constant has the form:

```
nRc1c2...cn
```

n
An unsigned, nonzero integer constant that states the number of characters to follow.

c
A character from the Radix–50 character set.

The maximum number of characters is 12. The character count must include any spaces that appear in the character string (the space character is a valid Radix–50 character). You can use Radix–50 constants only in DATA statements.

Examples of valid and invalid Radix–50 constants are:

Valid	Invalid
4RABCD	4RDK0:
6R TO	

When a Radix–50 constant is assigned to a numeric variable or array element, the number of bytes that can be assigned depends on the data type of the component (see Table 2-2). If the Radix–50 constant contains fewer bytes than the length of the component, ASCII null characters (0 bytes) are appended on the right. If the constant contains more bytes than the length of the component, the rightmost characters are not used.

Appendix C

Language Summary

C.1 Expression Operators

Table C–1 lists the expression operators in each data type in order of descending precedence.

Table C–1: Expression Operators

Data Type	Operator	Operation	Operates upon:
Arithmetic	**	Exponentiation	Arithmetic expressions
	*,/	Multiplication, division	
	+,–	Addition, subtraction, unary plus and minus	
Relational	.GT. .GE.	Greater than Greater than or equal to	Arithmetic, logical, or character expressions (all relational operators have equal priority)
	.LT.	Less than	
	.LE.	Less than or equal to	
	.EQ.	Equal to	
	.NE.	Not equal to	
Logical	.NOT.	.NOT. A is true if and only if A is false	Logical or integer expressions
	.AND.	A .AND. B is true if and only if A and B are both true	

Table C–1 (Cont.): Expression Operators

Data Type	Operator	Operation	Operates upon:
	.OR.	A .OR. B is true if either A or B or both are true	
	.EQV.	A .EQV. B is true if and only if A and B are both true, or A and B are both false	.EQV. and .XOR. have equal priority
	.XOR.	A .XOR. B is true if and only if A is true and B is false, or B is true and A is false	
	.NEQV.	Same as .XOR.	

C.2 Statements

Table C–2 summarizes the statements available in the PDP–11 FORTRAN–77 language, including the general form of each statement. The statements are listed alphabetically for ease of reference. The "Manual Section" column indicates the section of the manual that describes each statement in detail.

Table C–2: Statements

Form	Effect	Manual Section
ACCEPT	See READ	
Arithmetic/Logical/Character Assignment		3.1, 3.2, 3.3
v=e		
v	is a variable name, an array element name, or a character substring name.	
e	is an expression.	
	Assigns the value of the arithmetic, logical, or character expression to the variable.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
Statement Function		6.2.1
f([p[,p] . . .])=e		
f	is a symbolic name (not data type character).	
p	is a symbolic name.	
e	is an expression.	
	Creates a user-defined function having the variable p as a dummy argument. When referred to, the expression is evaluated using the actual arguments in the function call.	
ASSIGN s TO v		3.4
s	is a label of an executable statement or a FORMAT statement.	
v	is an integer variable name.	
	Associates the statement label s with the integer variable v for later use in an assigned GO TO statement or as a format specifier.	
BACKSPACE u		9.4
BACKSPACE ([UNIT=]u[,ERR=s])		
u	is an integer expression.	
s	is a label of an executable statement.	
	Backspaces one record the currently open file on logical unit u.	
BLOCK DATA [nam]		5.13
nam	is a symbolic name.	
	Specifies the subprogram that follows as a BLOCK DATA subprogram.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
CALL f[([a][,[a]] . . .)]		4.5 6.2
f	is a subprogram name or entry point.	
a	is an expression, an array name, or a procedure name.	
	Calls the subroutine subprogram with the name specified by f, passing the actual arguments a to replace the dummy arguments in the subroutine definition.	
CLOSE ([UNIT=]u[,p][,ERR=s])		9.2
p	is one of the following forms: STATUS = 'SAVE' DISPOSE 'KEEP' DISP 'DELETE' 'PRINT'	
e	is an integer expression.	
s	is a label of an executable statement.	
	Closes the specified file. DISPOSE can be abbreviated DISP.	
COMMON [/[cb]/] nlist [[,]/[cb]/nlist] . . .		5.4
cb	is a common block name.	
nlist	is a list of one or more variable names, array names, or array declarators separated by commas.	
	Reserves one or more blocks of storage space under the name specified to contain the variables associated with that block name.	
CONTINUE		4.4
	Causes no processing.	
DATA nlist/clist/[[,] nlist/clist/] . . .		5.10

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
nlist	is a list of one or more variable names, array names, array element names, or character substring references, separated by commas. Subscript and substring expressions must be constant.	
clist	is a list of one or more constants separated by commas, each optionally preceded by j*, where j is a nonzero, unsigned integer constant.	
	Initially stores elements of clist in the corresponding elements of nlist.	
DECODE (c,f,b[,ERR=s])[list]		A.1
c	is an integer expression.	
f	is a format specifier.	
b	is a variable name, array name, array element name, or character substring reference.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Reads c characters from buffer b and assigns values to the elements in the list, converted according to format specification f.	
DEFINE FILE u(m,n,U,v)[,u(m,n,U,v)] . . .		A.2
u	is an integer variable or integer constant.	
m	is an integer variable or integer constant.	
n	is an integer variable or integer constant.	
v	is an integer variable name.	
	Defines the record structure of a direct access file where u is the logical unit number, m is the number of fixed-length records in the file, n is the length in words of a single record, U is a fixed argument, and v is the associated variable.	
DELETE ([UNIT=]u[,REC=r][,ERR=s])		9.5

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
DELETE (u'r [,ERR=s])		
u	is an integer expression.	
r	is an integer expression.	
s	is a label of an executable statement.	
	Deletes the record on unit u that is specified by r, or the most recently accessed record.	
DIMENSION a(d)[,a(d)] . . .		5.3
a(d)	is an array declarator.	
	Specifies storage space requirements for arrays.	
DO s [,] v = e1,e2[,e3]		4.3
s	is a label of an executable statement.	
v	is a variable name.	
ei,e2,e3	are numeric expressions.	
	Executes the DO loop by performing the following steps:	
	1. Evaluates cnt= INT((e2–e1+e3)/e3)	
	2. Sets v=e1	
	3. If cnt is less than or equal to zero, does not execute the loop	
	4. If cnt is greater than zero, then	
	a. Executes the statements in the body of the loop	
	b. Evaluates v=v+e3	
	c. Decrements the loop count (cnt=cnt–1). If cnt is greater than zero, repeats the loop	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
ELSE		4.2.3
	Defines a block of statements to be executed if logical expressions in previous IF THEN and ELSE IF THEN statements have values of false. See IF THEN.	
ELSE IF (e) THEN		4.2.3
e	is a logical expression.	
	Defines a block of statements to be executed if logical expressions in previous IF THEN and ELSE IF THEN statements have values of false, and the logical expression e has a value of true. See IF THEN.	
ENCODE (c,f,b[,ERR=s])[list]		A.1
c	is an integer expression.	
f	is a format specifier.	
b	is a variable name, array name, array element name, or character substring reference.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Writes c characters into buffer b, which contains the values of the elements of the list, converted according to format specification f.	
END		4.9
	Delimits a program unit.	
ENDFILE u		9.7
ENDFILE ([UNIT=]u,[,ERR=s])		
u	is an integer expression.	
s	is a label of an executable statement.	
	Writes an end-file record on logical unit u.	
END IF		
	Terminates block IF construct. See IF THEN.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
END=s,ERR=s		7.2.1.6
s	is a label of an executable statement.	
	Transfers control on end-of-file or error condition. This is an optional element in each type of I/O statement and allows the program to transfer to statement number s when an end-of-file (END=) or error (ERR=) condition occurs.	
ENTRY nam [([p[,p] . . .])]		6.2.4
nam	is a subprogram name.	
p	is a symbolic name.	
	Defines an alternative entry point within a subroutine or function subprogram.	
EQUIVALENCE (nlist)[,(nlist)] . . .		5.6
nlist	is a list of two or more variable names, array names, array element names, or character substring names separated by commas. Subscript expressions must be constants.	
	Assigns each of the names in nlist the same storage location.	
EXTERNAL v[,v] . . .		
v	is a subprogram name.	
	Defines the names specified as subprograms.	
EXTERNAL *v[,*v] . . .		5.8
v	is a subprogram name.	
	Defines the names specified as user-defined subprograms.	
FIND (u'r[,ERR=s])		A.3

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
FIND ([UNIT=]u,REC=r[,ERR=s])		
u	is an integer expression.	
r	is an integer expression.	
	Positions the file on logical unit u to the record specified by r.	
FORMAT (field specification, . . .)		8.1 - 8.8
	Describes the format in which one or more records are to be transmitted; a statement label must be present.	
[typ] FUNCTION nam[*n][([p[,p] . . .])]		6.2.2
typ	is a data type specifier.	
nam	is a symbolic name.	
*n	is a data type length specifier.	
p	is a symbolic name.	
	Begins a function subprogram, indicating the program name and any dummy argument names (p). An optional type specification can be included.	
GO TO s		4.1.1
s	is a label of an executable statement.	
	Transfers control to statement number s.	
GO TO (slist)[,] e		4.1.2
slist	is a list of one or more statement labels separated by commas.	
e	is an integer expression.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
	Transfers control to the statement specified by the value of e (if e=1, control transfers to the first statement label; if e=2, control transfers to the second statement label, and so forth). If e is less than 1 or greater than the number of statement labels present, no transfer takes place.	
GO TO v [[,](slist)]		4.1.3
v	is an integer variable name.	
slist	is a list of one or more statement labels separated by commas.	
	Transfers control to the statement most recently associated with v by an ASSIGN statement.	
IF (e) s1,s2,s3		4.2.1
e	is an expression.	
s	is a label of an executable statement.	
	Transfers control to statement si depending on the value of e (if e is less than 0, control transfers to s1; if e equals 0, control transfers to s2; if e is greater than 0, control transfers to s3).	
IF (e) st		4.2.2
e	is an expression.	
st	is any executable statement except a DO, END, block IF, or logical IF.	
	Executes the statement if the logical expression has a value of true.	
IF (e1) THEN		4.2.3
block		
ELSE IF (e2) THEN		
block		
ELSE		

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
block		
END IF		
e1,e2	are logical expressions.	
block	is a series of zero or more FORTRAN statements.	
	Defines blocks of statements and conditionally executes them. If the logical expression in the IF THEN statement has a value of true, the first block is executed and control transfers to the first executable statement after the END IF statement.	
	If the logical expression has a value of false, the process is repeated for the next ELSE IF THEN statement. If all logical expressions have values of false, the ELSE block is executed. If there is no ELSE block, control transfers to the next executable statement following END IF.	
IMPLICIT typ (a[,a] . . .)[,typ(a[,a] . . .)] . . .		5.1
typ	is a data type specifier.	
a	is either a single letter or two letters in alphabetical order separated by a hyphen (i.e., X-Y).	
	The element a represents a single (or a range of) letter(s) whose presence as the initial letter of a variable specifies the variable to be of that data type.	
IMPLICIT NONE		5.1
	Overrides all implicit defaults. If IMPLICIT NONE is specified, no other IMPLICIT statement can be included in the program unit.	
INCLUDE 'filespec'		1.5
'filespec'	is a character constant.	
	Includes the source statements in the compilation from the file specified.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
INTRINSIC func[,func] . . .		5.9
func	is an intrinsic function name.	
	Designates symbolic names as intrinsic functions and allows those names to be used as actual arguments.	
OPEN(par[,par] . . .)		9.1
par	is a keyword specification in one of the following forms:	
	kwd kwd = value	
	kwd is a keyword, as described below.	
	value depends on the keyword, as described below.	

Keyword	Values
ACCESS	'SEQUENTIAL'
	'DIRECT'
	'APPEND'
	'KEYED'
ASSOCIATEVARIABLE	v
BLOCKSIZE	e
BLANK	'NULL'
	'ZERO'
BUFFERCOUNT	e
CARRIAGECONTROL	'FORTRAN'
	'LIST'
	'NONE'
DISPOSE	'SAVE' or 'KEEP'
DISP	'PRINT'

Table C–2 (Cont.): Statements

Form	Effect	Manual Section

Keyword	Values
	'DELETE'
ERR	s
EXTENDSIZE	e
FILE	c
FORM	'FORMATTED'
	'UNFORMATTED'
INITIALSIZE	e
KEY	(k[,k] . . .)
MAXREC	e
NAME	(same as FILE)
NOSPANBLOCKS	–
ORGANIZATION	'SEQUENTIAL'
	'RELATIVE'
	'INDEXED'
READONLY	–
RECL	e
RECORDSIZE	(same as RECL)
RECORDTYPE	'FIXED'
	'VARIABLE'
	'SEGMENTED'
SHARED	–
STATUS	'OLD'
	'NEW'
	'SCRATCH'
	'UNKNOWN'
TYPE	(same as STATUS)
UNIT	e
USEROPEN	p

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
c	is an array name, variable name, array element name, or character constant.	
e	is a numeric expression.	
k	is a key specification.	
p	is a program unit name.	
s	is a label of an executable statement.	
v	is an integer variable name.	
	Opens a file on the specified logical unit according to the parameters specified by the keywords.	
PARAMETER (p=c [,p=c] . . .)		5.11
PARAMETER p=c [,p=c] . . .		A.4
p	is a symbolic name.	
c	is a constant.	
	Defines a symbolic name for a constant.	
PAUSE [disp]		4.7
disp	is a decimal digit string containing one to five digits, an octal constant, or an alphanumeric literal.	
	Suspends program execution and prints the display, if one is specified.	
PRINT	See WRITE	
PROGRAM nam		5.12
nam	is a symbolic name.	
	Specifies a name for the main program.	
READ ([UNIT=]u,[FMT=]f[,END=s][,ERR=s])[list]		7.4.1.1
READ f[,list]		7.4.1.1
ACCEPT f[,list]		7.7

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
u	is an integer expression.	
f	is a format specifier.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Reads one or more logical records from unit u and assigns values to the elements in the list. The values are converted according to format specification f.	
READ ([UNIT=]u,REC=r,[FMT=]f[,ERR=s])[list]		7.4.2.1
READ (u'r,[FMT=]f[,ERR=s])[list]		
u	is an integer expression.	
r	is an integer expression.	
f	is a format specifier.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Reads records starting at record r from logical unit u and assigns values to the elements in the list. The values are converted according to format specification f.	
READ([UNIT=]u[,END=s][,ERR=s])[list]		7.4.1.3
u	is an integer expression.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Reads one unformatted record from logical unit u and assigns values to the elements in the list.	
READ([UNIT=]u,REC=r[,ERR=s])[list]		
READ(u'r[,ERR=s])[list]		7.4.2.2
u	is an integer expression.	
r	is an integer expression.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
s	is a label of an executable statement.	
list	is an I/O list.	
	Reads record r from logical unit u and assigns values to the elements in the list.	
READ([UNIT=]u,[FMT=]*[,END=s][,ERR=s])[list]		7.4.1.2 7.4.1.2
READ *[,list]		7.7
ACCEPT *[,list]		
u	is an integer expression.	
*	denotes list-directed formatting.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Reads one or more records from logical unit u and assigns values to the elements in the list. The values are converted according to the data type of the list element.	
READ ([UNIT=]u, { KEY / KEYEQ / KEYGE / KEYGT } =kv[,KEYID=kn][,ERR=s])[list]		7.4.3.2
u	is an integer expression.	
kv	is a key expression.	
kn	is an integer expression.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Reads the record on logical unit u described by the key expression kv and key-of-reference number kn. The values in the record are assigned to the elements in the list.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
READ ([UNIT=]u,[FMT=]f, { KEY / KEYEQ / KEYGE / KEYGT } =kv[,KEYID=kn][,ERR=s])[list]		7.4.3.1
u	is an integer expression.	
f	is a format specifier.	
kv	is a key expression.	
kn	is an integer expression.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Reads the record on logical unit u described by the key expression kv and key-of-reference number kn. The values in the record are converted according to format specification f and assigned to the elements in the list.	
READ ([UNIT=]c,[FMT=]f[,ERR=s][,END=s])[list]		7.4.4
c	is an internal file specifier.	
f	is a format specifier.	
s	is the label of an executable statement.	
list	is an I/O list.	
	Reads one or more internal records into the I/O list in accordance with the format specification.	
RETURN		4.6
	Returns control to the calling program from the current subprogram.	
REWIND u		9.3
REWIND ([UNIT=]u[,ERR=s])		
u	is an integer expression.	
s	is a label of an executable statement.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
	Repositions logical unit u to the beginning of the currently opened file.	
REWRITE ([UNIT=]u,[FMT=]f[,ERR=s]) [list]		7.6.1.1
u	is an integer expression.	
f	is a format specifier.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Rewrites the current record on logical unit u, containing the values of the elements of the list. The values are translated according to format specification f.	
REWRITE ([UNIT=]u[,ERR=s]) [list]		7.6.1.2
u	is an integer expression.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Rewrites the current record on logical unit u, containing the values of the elements of the list.	
SAVE[a[,a] . . .]		5.7
a	is a named common block enclosed in slashes, a variable name, or an array name.	
	Retains the definition status of an entity after the execution of a RETURN or END statement in a subprogram.	
STOP [disp]		4.8
disp	is a decimal digit string containing one to five digits, an octal constant, or an alphanumeric literal.	
	Terminates program execution and prints the display, if one is specified.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
SUBROUTINE nam[([p[,p] . . .])]		6.2.3
nam	is a symbolic name.	
p	is a symbolic name.	
	Begins a subroutine subprogram, indicating the program name and any dummy argument names (p).	
TYPE	See WRITE, Formatted Sequential.	7.8
	See WRITE, List-Directed.	
Type Declaration		5.2
typ v[,v] . . .		
typ	is one of the following data types: BYTE LOGICAL LOGICAL*1 LOGICAL*2 LOGICAL*4 INTEGER INTEGER*2 INTEGER*4 REAL REAL*4 REAL*8 DOUBLE PRECISION COMPLEX COMPLEX*8 CHARACTER CHARACTER*len	
v	is a variable name, array name, function or function entry name, or an array declarator. The name can optionally be followed by a data type length specifier (*n).	
	For character entities, the length specifier can be *len.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
	The symbolic names (v) are assigned the specified data type.	
UNLOCK u		9.6
UNLOCK ([UNIT=]u[,ERR=s])		
u	is an integer expression.	
s	is a label of an executable statement.	
	Unlocks all records currently locked on logical unit u.	
VIRTUAL a(d)[,a(d)] . . .		5.5
a(d)	is an array declarator.	
	Specifies storage space for arrays outside normal program address space.	
WRITE ([UNIT=]u,[FMT=]f[,ERR=s])[list]		7.5.1.1
PRINT f[,list]		7.8
TYPE f[,list]		7.8
u	is an integer expression.	
f	is a format specifier.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Writes one or more records to logical unit u, containing the values of the elements in the list. The values are converted according to format specification f.	
WRITE([UNIT=]u,REC=r,[FMT=]f[,ERR=s])[list]		
WRITE (u'r,[FMT=]f[,ERR=s])[list]		7.5.2.1
u	is an integer expression.	
r	is an integer expression.	
f	is a format specifier.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
s	is a label of an executable statement.	
list	is an I/O list.	
	Writes one or more records on logical unit u, containing the values of the elements of the list starting at record r. The values are converted according to format specification f.	
WRITE ([UNIT=]u[,ERR=s])[list]		7.5.1.3
u	is an integer expression.	
s	is a label of an executable statement label.	
list	is an I/O list.	
	Writes one unformatted record to logical unit u containing the values of the elements in the list.	
WRITE ([UNIT=]u,REC=r[,ERR=s])[list]		
WRITE (u'r[,ERR=s]) [list]		7.5.2.2
u	is an integer expression.	
r	is an integer expression.	
s	is a label of an executable statement label.	
list	is an I/O list.	
	Writes record r to logical unit u containing the values of the elements in the list.	
WRITE([UNIT=]u,[FMT=]*[,ERR=s])[list]		7.5.1.2 7.8
PRINT *[,list]		7.8
TYPE *[,list]		
u	is an integer expression.	
*	denotes list-directed formatting.	
s	is a label of an executable statement.	

Table C–2 (Cont.): Statements

Form	Effect	Manual Section
list	is an I/O list.	
	Writes one or more logical records to logical unit u containing the values of the elements in the list. The values are converted according to the data type of the list element.	
WRITE ([UNIT=]c,[FMT=]f [,ERR=s])[List]		7.5.4
c	is an internal file specifier.	
f	is a format specifier.	
s	is the label of an executable statement.	
list	is an I/O list.	
	Writes elements in the list to the internal file specified by the unit, converting to character strings in accordance with the format specification.	

C.3 Library Functions

Table C–3 lists the PDP–11 FORTRAN–77 generic functions and intrinsic functions (listed in the column headed "Specific Name"). Superscripts in the table refer to notes that follow the table.

Table C–3: Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Square root[1]	1	SQRT	SQRT	Real	Real
			DSQRT	Double	Double
a(1/2)			CSQRT	Complex	Complex
Natural logarithm[2]	1	LOG	ALOG	Real	Real
			DLOG	Double	Double
Log(e)a			CLOG	Complex	Complex
Common logarithm[2]	1	LOG10	ALOG10	Real	Real
			DLOG10	Double	Double
Log(10)a					
Exponential	1	EXP	EXP	Real	Real
e(a)			DEXP	Double	Double
			CEXP	Complex	Complex
Sine[3]	1	SIN	SIN	Real	Real
			DSIN	Double	Double
Sin a			CSIN	Complex	Complex
Cosine[3]	1	COS	COS	Real	Real
			DCOS	Double	Double
Cos a			CCOS	Complex	Complex
Tangent[3]	1	TAN	TAN	Real	Real
			DTAN	Double	Double
Tan a					
Arc sine[4,5]	1	ASIN	ASIN	Real	Real
			DASIN	Double	Double
Arc sin a					
Arc cosine[4,5]	1	ACOS	ACOS	Real	Real
			DACOS	Double	Double
Arc cos a					

[1]The argument of SQRT and DSQRT must be greater than or equal to 0. The result of CSQRT is the principal value with the real part greater than or equal to 0. When the real part is 0, the result is the principal value with the imaginary part greater than or equal to 0.

[2]The argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than 0. The argument of CLOG must not be (0.,0.).

[3]The argument of SIN, DSIN, COS, DCOS, TAN, and DTAN must be in radians. The argument is treated modulo 2*pi.

[4]The absolute value of the argument of ASIN, DASIN, ACOS, and DACOS must be less than or equal to 1.

[5]The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, and DATAN2 is in radians.

Table C–3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Arc tangent[5]	1	ATAN	ATAN	Real	Real
			DATAN	Double	Double
Arc tan a					
Arc tangent[5,6]	2	ATAN2	ATAN2	Real	Real
			DATAN2	Double	Double
Arc tan a(1) /a(2)					
Hyperbolic sine	1	SINH	SINH	Real	Real
			DSINH	Double	Double
Sinh a					
Hyperbolic cosine	1	COSH	COSH	Real	Real
			DCOSH	Double	Double
Cosh a					
Hyperbolic tangent	1	TANH	TANH	Real	Real
			DTANH	Double	Double
Tanh a					
Absolute value[7]	1	ABS	ABS	Real	Real
			DABS	Double	Double
[a]			CABS	Complex	Real
			IIA BS	Integer*2	Integer*2[8]
			JIABS	Integer*4	Integer*4[8]
		IABS	IIABS	Integer*2	Integer*2[8]
			JIABS	Integer*4	Integer*4[8]
Truncation[9]	1	INT	IINT	Real	Integer*2[8]
			JINT	Real	Integer*4
[a]			IIDINT	Double	Integer*2[8]
			JIDINT	Double	Integer*4[8]

[5]The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, and DATAN2 is in radians.

[6]The result of ATAN2 and DATAN2 is 0 or positive when a(2) is less than or equal to 0. The result is undefined if both arguments are 0.

[7]The absolute value of a complex number, (X,Y), is the real value: (X(2)+Y(2))(1/2).

[8]Integer results are selected by compiler switch I2/I4, rather than argument type. For more information, see Section 4.2.4 of the *PDP-11 FORTRAN-77 User's Guide*.

[9][x] is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as that of x. For example [5.7] equals 5. and [–5.7] equals –5.

Table C–3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
		IDINT	IIDINT	Double	Integer*2[8]
			JIDINT	Double	Integer*4[8]
		AINT	AINT	Real	Real
			DINT	Double	Double
Nearest integer[9]	1	NINT	ININT	Real	Integer*2[8]
			JNINT	Real	Integer*4[8]
[a+.5*sign(a)]			IIDNNT	Double	Integer*2[8]
			JIDNNT	Double	Integer*4[8]
		IDNINT	IIDNNT	Double	Integer*2[8]
			JIDNNT	Double	Integer*4[8]
		ANINT	ANINT	Real	Real
			DNINT	Double	Double
Fix[10] (real-to-integer conversion)	1	IFIX	IIFIX	Real	Integer*2[8]
			JIFXI	Real	Integer*4[8]
Float[10] (integer-to-real conversion)	1	FLOAT	FLOATI	Integer*2	Real
			FLOATJ	Integer*4	Real
Double-Precision float[10] (integer-to-double conversion)	1	DFLOAT	DFLOTI	Integer*2	Double
			DFLOTJ	Integer*4	Double
Conversion to single precision[10]	1	SNGL	–	Real	Real
			SNGL	Double	Real
			FLOATI	Integer*2	Real
			FLOATJ	Integer*4	Real

[8]Integer results are selected by compiler switch I2/I4, rather than argument type. For more information, see Section 4.2.4 of the *PDP–11 FORTRAN–77 User's Guide*.

[9][x] is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as that of x. For example [5.7] equals 5. and [–5.7] equals –5.

[10]Functions that cause conversion of one data type to another type provide the same effect as the implied conversion in assignment statements. The function SNGL with a real argument and the function DBLE with a double-precision argument return the value of the argument without conversion.

Table C–3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to double-precision[10]	1	DBLE	DBLE	Real	Double
			–	Double	Double
			–	Complex	Double
			DFLOTI	Integer*2	Double
			DFLOTJ	Integer*4	Double
Real part of complex or conversion to single precision[10]	1	REAL	REAL	Complex	Real
			FLOATI	Integer*2	Double
			FLOATJ	Integer*4	
			SNGL		
			SNGL		
Imaginary part of complex	1	–	AIMAG	Complex	Real
Complex from two reals	2	–	CMPLX	Real	Complex
Conversion to complex or complex from two arguments		CMPLX	–	Integer*2	Complex
			–	Integer*4	Complex
			–	Real	Complex
			CMPLX	Real	Complex
			–	Double	Complex
			–	Complex	Complex
Complex conjugate (if a=(X,Y) CONJG (a)=(X,Y)	1	–	CONJG	Complex	Complex
Double product of reals a(1)*a(2)	2	–	DPROD	Real	Double
Maximum	n	MAX	AMAX1	Real	Real
			DMAX1	Double	Double
max (a(1),a(2), . . . a(n)) (returns the maximum value from among the argument list; there must be at least two arguments)			IMAX0	Integer*2	Integer*2[8]
			JMAX0	Integer*4	Integer*4[8]

[8]Integer results are selected by compiler switch I2/I4, rather than argument type. For more information, see Section 4.2.4 of the *PDP–11 FORTRAN–77 User's Guide.*

[10]Functions that cause conversion of one data type to another type provide the same effect as the implied conversion in assignment statements. The function SNGL with a real argument and the function DBLE with a double-precision argument return the value of the argument without conversion.

Table C–3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
		MAX0	IMAX0	Integer*2	Integer*2[8]
			JMAX0	Integer*4	Integer*4[8]
		MAX1	IMAX1	Real	Integer*2[8]
			JMAX1	Real	Integer*4[8]
		AMXA0	AIMAX0	Integer*2	Real
			AJMAX0	Integer*4	Real
Minimum	n	MIN	AMIN1	Real	Real
			DMIN1	Double	Double
min(a(1),a(2), . . . a(n)) (returns the minimum value among the argument list; there must be at least two arguments)			IMIN0	Integer*2	Integer*2[8]
			JMIN0	Integer*4	Integer*4[8]
		MIN0	IMIN0	Integer*2	Integer*2[8]
			JMIN0	Integer*4	Integer*4[8]
		MIN1	IMIN1	Real	Integer*2[8]
			JMIN1	Real	Integer*4[8]
		AMIN0	AIMIN0	Integer*2	Real
			AJMIN0	Integer*4	Real
Positive difference	2	DIM	DIM	Real	Real
			DDIM	Double	Double
a(1)–(min(a(1,,a(2))) (returns the first argument minus the minimum of the two arguments)			IIDIM	Integer*2	Integer*2[8]
			JIDIM	Integer*4	Integer*4[8]
		IDIM	IIDIM	Integer*2	Integer*2[8]
			JIDIM	Integer*4	Integer*4[8]

[8]Integer results are selected by compiler switch I2/I4, rather than argument type. For more information, see Section 4.2.4 of the *PDP–11 FORTRAN–77 User's Guide*.

Table C–3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Remainder a(1)–a(2)*[a(1)/a(2)] (returns the remainder when the first argument is divided by the second)	2	MOD	AMOD DMOD IMOD JMOD	Real Double Integer*2 Integer*4	Real Double Integer*2[8] Integer*4[8]
Transfer of sign |a(1)*Sign a(2) \Real Double Integer*2 Integer*4)	2	SIGN	SIGN DSIGN IISIGN JISIGN	Real Double Integer*2 Integer*4	
		ISIGN	IISIGN JISIGN	Integer*2 Integer*4	Integer*2[8] Integer*4[8]
Bitwise AND (performs a logical AND on corresponding bits)	2	IAND	IIAND JIAND	Integer*2 Integer*4	Integer*2[8] Integer*4[8]
Bitwise OR (performs an inclusive OR on corresponding bits)	2	IOR	IIOR JIOR	Integer*2 Integer*4	Integer*2[8] Integer*4[8]
Bitwise exclusive OR (performs an exclusive OR on corresponding bits)	2	IEOR	IIEOR JIEOR	Integer*2 Integer*4	Integer*2[8] Integer*4[8]
Bitwise complement (complements each bit)	1	NOT	INOT JNOT	Integer*2 Integer*4	Integer*2[8] Integer*4[8]
Bitwise shift (a(1) logically shifted left a(2) bits)	2	ISHFT	IISHFT JISHFT	Integer*2 Integer*4	Integer*2[8] Integer*4[8]

[8]Integer results are selected by compiler switch I2/I4, rather than argument type. For more information, see Section 4.2.4 of the *PDP-11 FORTRAN-77 User's Guide*.

Table C–3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Random number[11] (returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1)	1	–	RAN	Integer*4	Real
	2	–	RAN	Integer*2	Real
Length (returns length of the character expression)	1	–	LEN	Character	Integer*2
Index (C(1),C(2)) (returns the position of the substring c(2) in the character expression c(1))	2	–	INDEX	Character	Integer*2
ASCII Value (returns the ASCII value of the argument; the argument must be a character expression that has a length of 1)	1	–	ICHAR	Character	Integer*2
Character relationals (ASCII collating sequence)	2	–	LLT	Character	Logical*2
	2	–	LLE	Character	Logical*2
	2	–	LGT	Character	Logical*2
	2	–	LGE	Character	Logical*2

[11]The argument for this function must be an integer variable or integer array element. The argument should initially be set to 0. The RAN function stores a value in the argument that it later uses to calculate the next random number. Resetting the argument to 0 regenerates the sequence. Alternate starting values generate different random-number sequences.

Index

A

B

C

D

E

F

G

H

I

K

L

S

T

U

V

W

X

Z

Reader's Comments

PDP–11 FORTRAN–77
Language Reference Manual
AA–V193B–TK

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

Do Not Tear - Fold Here and Tape

digital™

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line

Reader's Comments

PDP–11 FORTRAN–77
Language Reference Manual
AA–V193B–TK

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

— Do Not Tear - Fold Here and Tape ——————————————————

digital™

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here ——————————————————

Cut Along Dotted Line