PDP-11 FORTRAN IV
Language Reference Manual

Order No. AA-R953A-TK

March 1983

This document describes the fundamentals of the FORTRAN IV language elements, as implemented for the PDP-11 systems.

SUPERSESSION/UPDATE INFORMATION:	This manual contains information concerning FORTRAN IV as of March 1983
SOFTWARE VERSION TO OPERATING SYSTEM AND VERSION:	FORTRAN IV V2.6 • RSTS/E V8.0 • RT-11 V5B • RSX-11M V4.1 • RSX-11M-PLUS V2.1 • VAX/VMS V3.0

digital equipment corporation · maynard, massachusetts

First Printing, June 1974
Revised: December 1974
December 1975
June 1977
September 1979
March 1983

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

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

CONTENTS

CONTENTS

CONTENTS

CONTENTS

Page

FIGURES

TABLES

CONTENTS

PREFACE

MANUAL OBJECTIVES

This manual describes the elements of PDP-11 FORTRAN IV and is designed as a reference, rather than tutorial, document.

This document serves as the FORTRAN language reference manual for several operating systems that run on the PDP-11 family of computers. Therefore, no information specific to an operating system is presented here. For that information, refer to the user's guide for each system.

INTENDED AUDIENCE

Because this is a reference document, readers who have a basic understanding of FORTRAN will derive maximum benefit.

STRUCTURE OF THIS DOCUMENT

This manual contains nine chapters and two appendixes.

- Chapter 1 consists of general information concerning FORTRAN and introduces basic facts needed for writing FORTRAN programs.
- Chapter 2 describes the components of FORTRAN statements, such as symbolic names, constants, and variables.
- Chapter 3 describes assignment statements, which define values used in the program.
- Chapter 4 deals with control statements, which transfer control from one point in the program to another.
- Chapter 5 describes specification statements, which define the characteristics of symbols used in the program, such as data type and array dimensions.
- Chapter 6 discusses subprograms, both user-written and those supplied with PDP-11 FORTRAN IV.
- Chapter 7 covers FORTRAN input/output.
- Chapter 8 describes the FORMAT statements used in conjunction with formatted I/O statements.

- Chapter 9 contains information on auxiliary I/O statements, such as OPEN, CLOSE, and DEFINE FILE.
- Appendix A summarizes the character sets supported by PDP-11 FORTRAN IV.
- Appendix B summarizes the language elements of PDP-11 FORTRAN IV.

ASSOCIATE DOCUMENTS

The following documents are of interest to PDP-11 FORTRAN IV programmers:

- RT-11/RSTS/E FORTRAN IV User's Guide
- RSX, VAX/VMS FORTRAN IV User's Guide

CONVENTIONS USED IN THIS DOCUMENT

The following conventions are used in this manual:

- Uppercase words and letters used in examples indicate that you should type the words and letters as shown.
- Lowercase words and letters used in examples indicate that you are to substitute a word or value of your choice.
- Brackets ([]) enclose optional elements.
- Braces ({}) enclose lists from which one element is to be chosen.
- Ellipses (...) indicate that the preceding item(s) can be repeated one or more times.

In addition, the following characters denote special nonprinting characters:

Tab character	(TAB)
Space character	△

CHAPTER 1

INTRODUCTION TO PDP-11 FORTRAN IV

1.1 LANGUAGE OVERVIEW

The PDP-11 FORTRAN IV language is based on American National Standard (ANS) FORTRAN X3.9-1966 but includes the following enhancements to ANS FORTRAN:

- 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.
- The following data type has been added:

  LOGICAL*1

- The IMPLICIT statement redefines the implied data type of symbolic names.
- The following input/output (I/O) statements have been added:

ACCEPT	
TYPE	Device-oriented I/O
PRINT	
READ (u'r)	
WRITE (u'r	Unformatted direct-access I/O
FIND (u'r)	
OPEN	
CLOSE	File control and attribute
DEFINE FILE	specification
ENCODE	Formatted data conversion
DECODE	in memory

- The specifications END=s and/or ERR=s can be included in READ or WRITE statements in order to transfer control to the statement specified by s when an end-of-file or error condition occurs.
- Alphanumeric literals (strings of characters bounded by apostrophes) can be used in place of Hollerith constants.
- List-directed I/O can be used to perform formatted I/O without a format specification.
- Constants and expressions are permitted in the I/O lists of WRITE, REWRITE, TYPE, and PRINT statements.

- The DO statement increment parameter can have a negative value.
- For readability, you can optionally use a comma following the label in a DO statement.
- A PROGRAM statement can be used in a main program.
- You can include an explanatory comment on the same line as any FORTRAN 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.
- The statement label list in an assigned GO TO statement is optional.
- 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.

PDP-11 FORTRAN IV is simply referred to as FORTRAN throughout the rest of this manual.

1.2 ELEMENTS OF FORTRAN PROGRAMS

FORTRAN programs consist of FORTRAN 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 is terminated by 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, you can continue it 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 so that other statements can transfer control to it or obtain the information it contains. A statement label is an integer number placed in the first five columns of a statement's initial line.

1.2.2 Comments

Comments do not affect program processing in any way. They are merely a documentation aid to the programmer. You can use them freely to describe the actions of the program, to identify program sections and processes, and to provide greater ease in reading the source program listing. The letter C in the first column of a source line identifies that line as a comment. In addition, if you place an exclamation point (!) in the statement portion of a source line, the rest of that line is treated as a comment.

Any printable character can appear in a comment.

1.2.3 FORTRAN Character Set

The FORTRAN 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

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

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

1.3 FORMATTING A FORTRAN LINE

Each FORTRAN line has the following four fields:

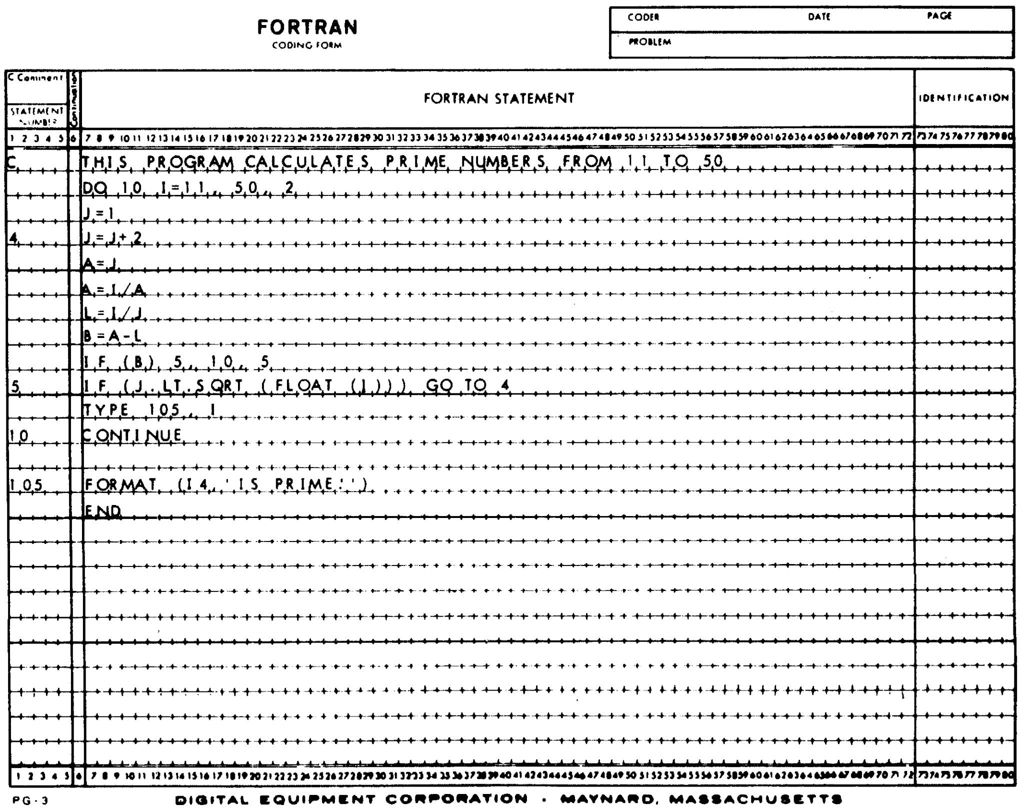
- Statement label field
- Continuation indicator field
- Statement field
- Sequence number field

You can format a FORTRAN line in two ways: 1) by typing one character per column (character-per-column) or 2) by using the tab character (tab-character) 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. However, you can use tab-character formatting, in addition to character-per-column formatting, 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 fields for statement labels, continuation indicators, statement text, and sequence numbers. Sections 1.3.3 through 1.3.6 describe the use of each field.

Each column represents a single character. The columns making up each field follow.


FORTRAN
CODING FORM

CODER DATE PAGE
PROBLEM

```
C     THIS PROGRAM CALCULATES PRIME NUMBERS FROM 11 TO 50
      DO 10 I=11, 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-613-82

Figure 1-1: FORTRAN Coding Form

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

To get from one field to another, type each space individually. For example, in Figure 1-1, enter the first line, type C, press the space bar five times, and begin typing the comment.

1.3.2 Tab-Character Formatting

You can press the tab character to move to the continuation indicator field or the statement field. You cannot move to the sequence number field, however, by pressing the tab. Figure 1-2 compares keystrokes in lines using tab-character formatting and in those using character-per-column formatting.

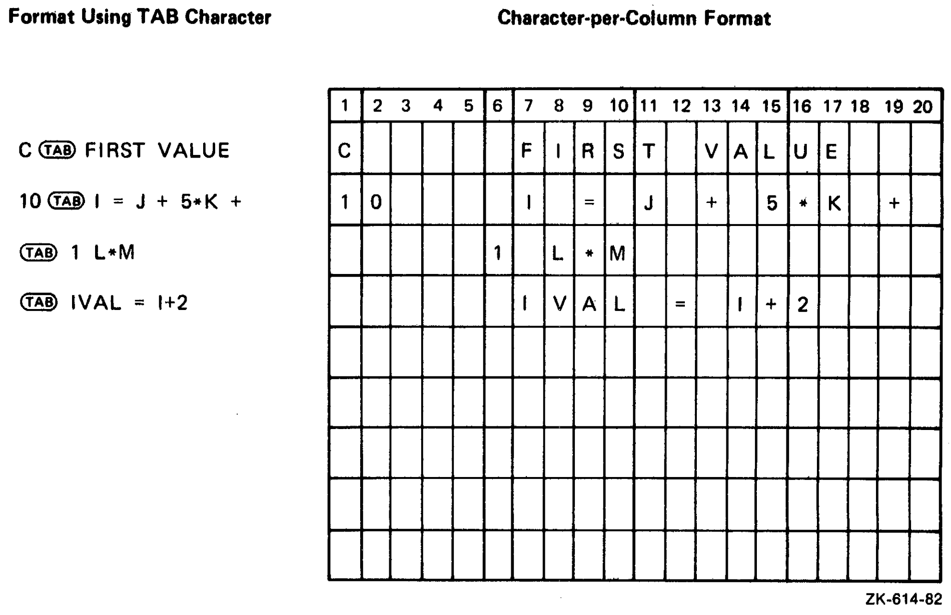


Figure 1-2: Line Formatting Example

The statement label field consists of the characters that you type before the first tab character. The statement label field cannot have more than five characters.

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

To enter the continuation indicator field, type any digit. The statement field then consists of all the characters after the digit to the end of the line.

To enter the statement field without a continuation indicator field, type the statement immediately after the first tab. Note that no FORTRAN statement starts 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 FORTRAN compiler's interpretation of the tab character described above.

You can use the tab character to improve the legibility of a FORTRAN statement. For compilation, spaces are ignored, except those within a Hollerith constant or alphanumeric literal. For legibility, spaces are printed in the source listing. Tabs also are ignored for compilation purposes in a statement field. In the source listing, the tab causes the character that follows to be printed at the next tab stop (located at columns 9, 17, 25, 33, and so forth)

1.3.3 Statement Label Field

A statement label or statement number consists of one 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 hava 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 the 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 below.

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

1.3.3.1 Comment Indicator - The letter C in column 1 indicates that the line is a comment. The compiler prints that line in the source program listing and then ignores the line.

1.3.3.2 Debugging Statement Indicator - The letter D in column 1 designates a debugging statement. The first line of the debugging statement can have a statement label in the remaining columns of the label field. If a debugging statement is continued, every continuation line must have a D in column 1 and a continuation indicator in column 6.

The compiler command specifies whether debugging statements are to be compiled. If you specify debug-statement compilation, debugging statements are compiled as a part of the source program; if you do not specify debug-statement compilation, debugging statements are treated as comments. For a description of compilation commands, refer to the appropriate user's guide.

1.3.4 Continuation Field

A continuation indicator is any character, except 0 or space, in column 6 of a FORTRAN 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 no break occurred at that point. If a continuation indicator is 0, then the compiler considers the line to be the first line of a FORTRAN statement.

Comment lines cannot be continued, but they can occur between a statement's initial line and its continuation line(s) or between successive continuation lines.

1.3.5 Statement Field

The text of a FORTRAN statement is placed in the statement field. Because the compiler ignores the tab character and spaces (except in Hollerith constants or alphanumeric literals), you can space the text in any way desired for maximum 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 and 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 in a FORTRAN program. 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 FORTRAN program unit. In this figure, vertical lines separate statement types that can be interspersed. For example, comment lines, FORMAT statements, DATA statements, and executable statements are allowed alternatives in the body of the program. Horizontal lines indicate statement types that cannot be interspersed. For example, IMPLICIT statements cannot be interspersed with executable statements, a PROGRAM statement, or an END statement because each has a definite order in the program.

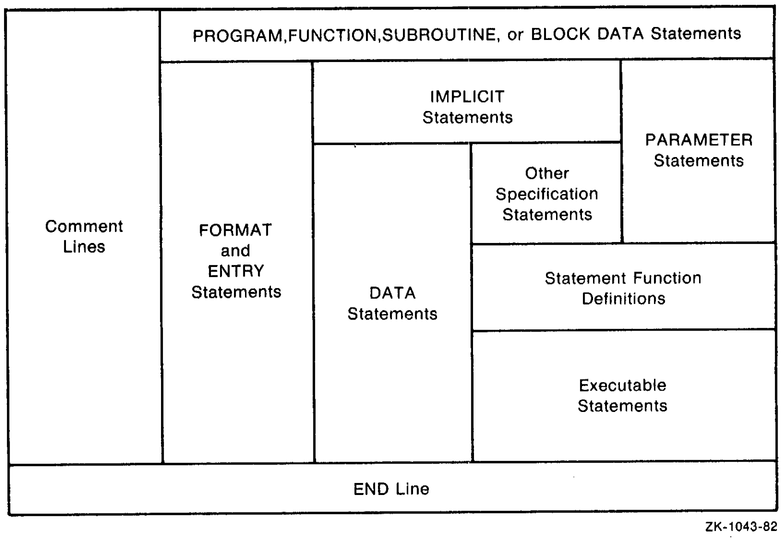

PROGRAM,FUNCTION,SUBROUTINE, or BLOCK DATA Statements
Comment Lines
FORMAT and ENTRY Statements
IMPLICIT Statements
PARAMETER Statements
Other Specification Statements
DATA Statements
Statement Function Definitions
Executable Statements
END Line
ZK-1043-82

Figure 1-3: Required Order of Statements and Lines

CHAPTER 2

FORTRAN STATEMENT COMPONENTS

The basic components of FORTRAN statements are:

- Constants -- fixed values, such as numbers. They cannot be changed by program statements.
- Variables -- symbolic names that represent stored values. The stored 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 just a symbolic name. Individual values are called array elements.
- Function references -- names of functions, optionally followed by lists of arguments. A function is a program unit that performs a specified computation using the arguments, if any: for example, computing the trigonometric sine of the argument. The resulting value is used in place of the function reference.
- Expressions -- constants, variables, array elements, function references or combinations of these components used in conjunction with operators. An operator is a symbol specifying that a certain kind of operation, such as multiplication, is to be performed to obtain a single result.

Variables, arrays, and functions have symbolic names. A symbolic name is a string of characters that identify entities in the program.

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

- Logical
- Integer
- Real
- Double precision
- Complex

The following sections detail the basic components of FORTRAN, with the exception of function references, which are described in Chapter 6. Sections 2.1 and 2.2 on symbolic names and data types, respectively, provide information common to all basic components.

2.1 SYMBOLIC NAMES

Symbolic names identify entities within a FORTRAN program unit. These entities are listed in Table 2-1. The "Typed" column indicates whether the entity has a data type, such as real, integer, and so forth. 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
Processor-defined functions	yes
Function subprograms	yes
Subroutine subprograms	no
Common blocks	no
Main programs	no
Block data subprograms	no
Dummy arguments	yes

A symbolic name is a string of letters and digits totaling a maximum of six characters. The first character must be a letter.

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, you cannot use the same symbolic name to identify two or more entities in the same program unit.

In executable programs consisting of two or more program units, some entities must have unique names throughout all the program units. The entities are:

- Processor-defined functions
- Function subprograms
- Subroutine subprograms
- Common blocks
- Main programs
- Block data subprograms

2.2 DATA TYPES

Each basic component (constants, variables, and so forth) represents one of several data types:

- Integer -- a whole number.

- Real -- a decimal number, that is, a whole number, a decimal fraction, or a combination of the two.
- Double precision -- a real number with more than twice as many maximum significant digits.
- Complex -- a pair of real numbers representing a complex number; the first value represents the real part, the second represents the imaginary part.
- Logical -- true or false.

The data type of a basic component can be specified in one of three ways: it can be inherent in its construction (as in constants); it can be implied by 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 Section 3.1).

ANS FORTRAN specifies that a "numeric storage unit" is the amount of memory needed to store a real, integer, or logical value. Double precision and complex values occupy two numeric storage units. In PDP-11 FORTRAN IV, a numeric storage unit is four bytes of memory.

PDP-11 FORTRAN IV provides additional data types for better control of performance and memory requirements. Table 2-2 lists the data types available and the amount of memory required (in bytes). The form *n appended to a data type name is called a data type length specifier.

Table 2-2: Data Type Storage Requirements

Data Type	Storage Requirements (Bytes)
BYTE	1[2]
LOGICAL	4
LOGICAL*1	1[2]
LOGICAL*4	4
INTEGER	2 or 4[1]
INTEGER*2	2
INTEGER*4	4[3]
REAL	4
REAL*4	4
REAL*8	8
DOUBLE PRECISION	8
COMPLEX	8
COMPLEX*8	8

1. Either two or four bytes are allocated depending on the compiler command qualifier specified. The default allocation is two bytes. Only two bytes are used for computation.
2. The 1-byte storage area can contain the logical values true or false, a single character, or integers in the range -128 to +127.
3. Four bytes are allocated but only two bytes are used for computation.

BYTE and LOGICAL*1 are synonymous.

2.3 CONSTANTS

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

Hollerith constants or alphanumeric literals have no data type. They assume the data type of the context in which they appear (see Section 2.3.6.2).

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 following 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 32767.

Examples of valid and invalid integer constants are:

Valid	**Invalid/Explanation**	
0	99999999999	(too large)
-127	3.14	(decimal point and
+32123	32,767	comma not allowed)

Integer constants can also be specified in octal form. 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 "177777.

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)
	"177777"	(trailing quotation mark not allowed)

2.3.2 Real Constants

A real constant is a number with a decimal point. It can take one of three forms:

- A basic real constant
- A basic real constant followed by a decimal exponent
- 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. Thus, in the constant 0.00001234567, all of the nonzero digits are significant, and none of the 0s are 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 of typically seven decimal digits.

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

Except for algebraic signs, 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. The exponent cannot be omitted, but 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 following 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 of typically 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. Similarly, a minus sign must appear after the letter D for a negative exponent while a plus sign is optional for a positive exponent.

The exponent following the letter D cannot be omitted, but 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; this is a valid real constant)
1D0		

2.3.4 Complex Constants

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

A complex constant has the following 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)	(1,2)	(integers are not allowed)
(+12739E3,0.)	(1.23,)	(second real constant is missing)
	(1.0,1.0D0	(double precision constants are not allowed)

2.3.5 Logical Constants

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

```
.TRUE.

.FALSE.
```

The delimiting periods are a required part of each constant.

2.3.6 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 following 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.

Examples of valid and invalid Hollerith constants are:

Valid	Invalid/Explanation	
16HTODAY'S DATE IS:	3HABCD	(wrong number of characters)
1HB		

2.3.6.1 Alphanumeric Literals - An alphanumeric literal is a string of printable ASCII characters enclosed by apostrophes that represents an alternate form of a Hollerith constant. The form is:

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

c
: A printable character.

Both delimiting apostrophes must be present.

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

The length of the alphanumeric literal is the number of characters between the apostrophes, including spaces and tabs, except that two consecutive apostrophes represent a single apostrophe. A tab is stored as a single character but is displayed as spaces up to the next tab stop. The length must be in the range from 1 to 255.

Examples of valid and invalid literals are:

Valid	**Invalid/Explanation**	
'WHAT?'	'HEADINGS	(must contain trailing apostrophe)
'TODAY''S DATE IS:	''	(must contain at least one character)
HE SAID, "HELLO"'	"NOW OR NEVER"	(quotation marks cannot be used in place of apostrophes)

2.3.6.2 Data Type Rules for Hollerith Constants - 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 + 'XY'	INTEGER*2	2
DOUBLE = 8HABCDEFGH	REAL*8	8

- When a specific data type is required, that 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. 'B'	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.

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.

Variables are classified by data type, as are constants. The data type of a variable indicates the type of data it represents, its precision, and its storage requirements. When data of any type is assigned to a variable, it 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.

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 arguments and dummy arguments in subprogram references.

A variable is considered defined if the storage associated with it contains data of the same type as the 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 the value of one variable is defined (for example, by assignment), the value of the other variable becomes undefined; that is, you cannot predict its value.

2.4.1 Data Type Specification

Type declaration statements (see Section 5.2) specify that given variables are to represent specified data types. For example:

```
COMPLEX VAR1
DOUBLE PRECISION VAR2
```

These statements indicate that the variable VAR1 is to be associated with an 8-byte storage location which will contain complex data, and that the variable VAR2 is to be associated with an 8-byte double precision storage location.

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

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 Type by Implication

In the absence of either IMPLICIT statements or explicit type declaration statements, all variables with names beginning with I, J, K, L, M, or N are assumed to be integer variables. Variables with 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.

An array can have from one to seven dimensions. For example, a column of figures is a one-dimensional array. To refer to a value, you must specify the row number. A table of more than one column of figures is a two-dimensional array. To refer to a value you must specify both row number and column number. A table of figures that covers several pages is a three-dimensional array. To refer to a value in this array, you must specify row number, column number, and page number.

The following FORTRAN 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 demension.

An element of an array is considered defined if the storage 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, and 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 identifies an array within a program unit and indicates the properties of that array.

An array declarator has the following 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 that is an integer constant or integer variable that specifies the upper bound of the array.

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 number of elements in each dimension. IUNIT above contains 1000 elements.

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

Dimension declarators that are not constant can be used in a subprogram 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. Dimension declarators that are not constant are not permitted in a main program.

2.5.2 Subscripts

A subscript qualifies an array name. A subscript is a list of expressions, called subscript expressions, that are enclosed in parentheses and that determine which element in the array is referenced. The subscript is appended to the array name it qualifies.

A subscript has the following form:

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

s
: A subscript expression.

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

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

2.5.3 Array Storage

As discussed earlier in this section, you can think of the dimensions of an array as rows, columns, and levels or planes. However, FORTRAN always stores an array in memory 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 lefmost subscripts vary most rapidly. This is called the "order of subscript progression." For example, Figure 2-1 shows array storage in one, two, and three dimensions.

2.5.4 Data Type of an Array

The data type of an array is specified in the same way as the data type of a variable, 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 that array, it is converted to double precision.

2.5.5 Array References Without Subscripts

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

- Type declaration statements
- COMMON statement
- DATA statement
- EQUIVALENCE statement

- FUNCTION statement
- SUBROUTINE statement
- Input/output statements

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.

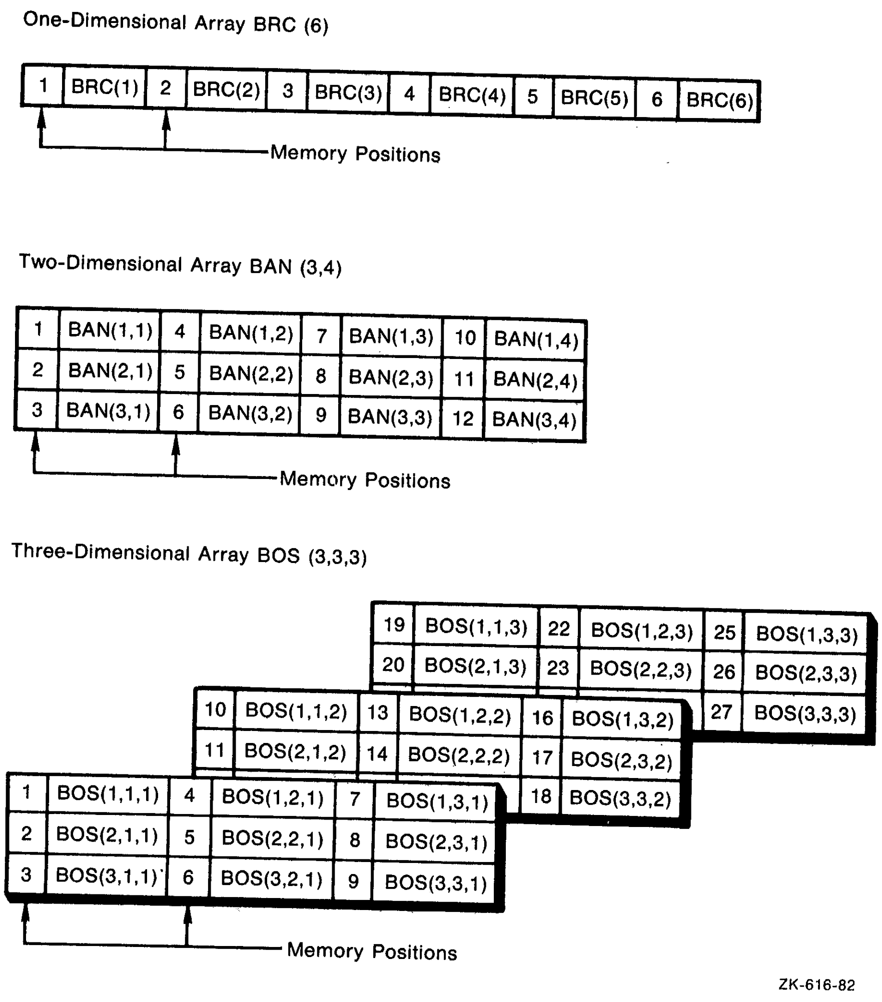

One-Dimensional Array BRC (6)
1 BRC(1) 2 BRC(2) 3 BRC(3) 4 BRC(4) 5 BRC(5) 6 BRC(6)
Memory Positions
Two-Dimensional Array BAN (3,4)
1 BAN(1,1) 4 BAN(1,2) 7 BAN(1,3) 10 BAN(1,4)
2 BAN(2,1) 5 BAN(2,2) 8 BAN(2,3) 11 BAN(2,4)
3 BAN(3,1) 6 BAN(3,2) 9 BAN(3,3) 12 BAN(3,4)
Memory Positions
Three-Dimensional Array BOS (3,3,3)
19 BOS(1,1,3) 22 BOS(1,2,3) 25 BOS(1,3,3)
20 BOS(2,1,3) 23 BOS(2,2,3) 26 BOS(2,3,3)
27 BOS(3,3,3)
10 BOS(1,1,2) 13 BOS(1,2,2) 16 BOS(1,3,2)
11 BOS(2,1,2) 14 BOS(2,2,2) 17 BOS(2,3,2)
18 BOS(3,3,2)
1 BOS(1,1,1) 4 BOS(1,2,1) 7 BOS(1,3,1)
2 BOS(2,1,1) 5 BOS(2,2,1) 8 BOS(2,3,1)
3 BOS(3,1,1) 6 BOS(3,2,1) 9 BOS(3,3,1)
Memory Positions
ZK-616-82

Figure 2-1: Array Storage

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 EXPRESSIONS

An expression represents a single value. It can be a single basic component, such as a constant or variable, or a combination of basic components with one or more operators. Operators specify computations to be performed, using the value(s) of the basic component(s) to obtain a single value.

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

2.6.1 Arithmetic Expressions

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

An arithmetic element can be any of the following:

- A numeric constant
- A numeric variable
- A numeric array element
- An arithmetic expression enclosed in parentheses
- An arithmetic function reference

The term "numeric," as used above, can also be interpreted to include logical data, since logical data is treated as integer data when used in an arithmetic context.

Arithmetic operators specify a computation to be performed using 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. The plus (+) and minus (-) symbols are also unary operators when written immediately preceding an arithmetic element to denote a positive or negative value.

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 data types of base and exponent, and the data type of the result of exponentiation.

Table 2-3: **Result Data Type for Exponentiation**

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

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 the case of a real base and a double precision exponent. The result in this case is double precision.

Arithmetic expressions are evaluated in an order determined by the operators. The operators are ranked in precedence and operations of higher precedence are performed first. The precedence is:

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 A**B**C, B**C is evaluated first and then A is raised to the resulting power.

2.6.1.1 Use of Parentheses - You can use parentheses to force a particular order of evaluation. When part of an expression is enclosed in parentheses, that part is evaluated first and the resulting value 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
  ↑   ↑   ↑   ↑
  3   1   4   2

(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, 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 the evaluation order is often important in high-accuracy computations since evaluation orders that are algebraically equivalent might not be computationally equivalent due to rounding and normalization.

Using parentheses to specify the evaluation order is important, furthermore, in difficult expressions. If any doubt exists as to the resulting value of an expression, use parentheses. Extra parentheses do not affect the result, but lack of sufficient parentheses does.

2.6.1.2 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 that 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, on the following basis:

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 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 can result from division is truncated, not rounded. For example:

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

  The value of this expression is 0, not 1.

- Real operations -- Real operations are performed only on real elements or combinations or real, integer, and logical elements. Any integer elements present are converted to real data type by giving each 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 a real multiplication is performed on that result and X.

- Double precision operations -- Any real or integer element in a double precision operation is converted to double precision by making it the most significant portion of a double precision element. The least significant portion is 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:

  ```
  0.3333333000000000D0
  ```

  not to:

  ```
  0.3333333333333333D0
  ```

- Complex operations -- In an operation on an expression containing a complex element, integer elements are converted to real data type, as previously described. Double precision elements are converted to real data type by rounding the least significant portion. The real element thus obtained is designated as the real part of a complex number; the imaginary part is 0. The expression is then evaluated using complex arithmetic and the resulting value is complex.

2.6.2 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 and 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 first evaluated to obtain their values. These values are then compared to determine whether the relationship stated by the operator exists. For example:

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

This expression 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 that relationship exists, the value of the expression is true; if not, the value of the expression is false.

All relational operators have the same precedence. Arithmetic operators have a higher precedence than relational operators.

As in any other arithmetic expression, you can use parentheses to alter the order of evaluation of the arithmetic expressions in a relational expression. However, since arithmetic operators are evaluated before relational operators, you need not enclose the entire arithmetic expression in parentheses.

You can compare two numeric expressions of different in a relational expression. In this case, the value of the expression with the lower-ranked data type is converted to the higher-ranked data type before the comparison is made.

2.6.3 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.
.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 according to an order of precedence assigned to its operators. The following list gives the order in which all the operators that can appear in a logical expression are evaluated:

Operator	Precedence
**	First (Highest)
*,/	Second
+,-	Third
Relational Operators	Fourth
.NOT.	Fifth
.AND.	Sixth
.OR.	Seventh
.XOR.,.EQV.	Eighth

Operators of equal rank are evaluated from left to right. For example:

```
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., the expression A .AND. (F(X,Y) .GT. 2.0) .AND. B is .FALSE.. The value of the expression can be determined by testing A without evaluating F(X,Y). Thus, the function subprogram F may not be called, and side-effects resulting from the call, such as changing variables in COMMON, cannot 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 for two integer elements. The resulting data type is integer.

Example:

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

In this example, I has the value "165 and K has the value "21.

CHAPTER 3

ASSIGNMENT STATEMENTS

Assignment statements assign a single value to a variable or array element.

The three kinds of assignment statements are:

- Arithmetic assignment statement
- Logical assignment statement
- ASSIGN statement

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 following 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:

```
     KOUNT = KOUNT + 1
```

This statement 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 on the left of the equal sign can be undefined, values must have been previously assigned to all symbolic references in the expression on 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 on 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.

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
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 zero, imaginary part of E is not used
Complex	Append fraction (.) to E and assign to real part of V; imaginary part of V is 0.0	Assign 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/Explanation

`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 following form:

```
v = e
```

v
: A logical variable or array element.

e
: A logical expression.

Note that v must be of logical data type and e must yield a logical value. Otherwise, conversions will be made according to Table 3-1, but the values will not have any logical meaning.

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/Explanation

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

ASSIGN

3.3 ASSIGN STATEMENT

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 following form:

```
ASSIGN s TO v
```

s
: The label of an executable statement in the same program unit as the ASSIGN statement.

v
: An integer variable.

The ASSIGN statement assigns the statement label to the variable. It is similar to an arithmetic assignment statement, except that the variable becomes defined for use as a statement label reference and becomes undefined as an integer variable; that is, the value cannot be used for purposes of output or arithmetic.

The statement label must refer to an executable statement in the same program unit. It may not refer to a FORMAT statement.

The ASSIGN statement must be executed before the assigned GO TO statement(s) in which the assigned variable is to be used. The ASSIGN statement and the assigned GO TO statement(s) must occur in the same program unit.

For example:

```
ASSIGN 100 TO NUMBER
```

This statement associates the variable NUMBER with the statement label 100. Arithmetic operations on the variable, as in the following statement, then become invalid. For example:

```
NUMBER = NUMBER+1
```

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

Assigning the variable a value with the following arithmetic assignment statement:

```
NUMBER=10
```

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

Examples:

Valid

```
ASSIGN 10 TO NSTART

ASSIGN 99999 TO KSTOP
```

Invalid/Explanation

```
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 interrupt normal program flow to transfer control to another section of the program or to a subprogram.

You can use control statements to transfer control to a point within the same program unit or to another program unit. Control statements also govern iterative processing, suspension of program execution, and program termination.

The control statements are:

- GO TO statement -- transfers control within a program unit
- IF statement -- conditionally transfers control or conditionally executes a statement
- 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, giving their forms and examples of the ways in which they are used.

GO TO

4.1 GO TO STATEMENTS

GO TO statements transfer control within a program unit. The three types of GO TO statements are:

- Unconditional GO TO statement
- Computed GO TO statement
- Assigned GO TO statement

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.

```
     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 will be transferred to statement 24. In the second example, if SITU(J,K) has a value of 2, execution will be transferred to statement 340.

4.1.3 Assigned GO TO Statement

The assigned GO TO statement transfers control to a statement label placed in a variable by an ASSIGN statement. Thus, the transfer destination can be changed, depending on the most recently executed ASSIGN statement.

The assigned GO TO statement has the following 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 3.3 on the ASSIGN statement.

The GO TO statement, the associated ASSIGN statement(s), 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.

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 only if a specified condition is met. The two types of IF statements are:

- Arithmetic IF statement
- Logical IF statement

For each type, the decision to transfer control or to execute the 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, based on the value of an arithmetic expression.

The arithmetic IF statement has the following 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 first evaluates the expression (e) in parentheses. Then,

If e is:	Control passes to:
Less than 0	Label s1
Equal to 0	Label s2
Greater than 0	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. Control passes to statement 100 only 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; it transfers control to statement 20 if the value is odd.

4.2.2 Logical IF Statement

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

The logical IF statement has the following 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, or another logical IF statement.

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

Examples of logical IF statements are:

```
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
```

4.3 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 following form:

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

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

v
: An integer variable.

e1,e2,e3
: Integer 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.

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

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

The range of the DO statement is 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.

If the increment parameter (e3) is positive, the terminal parameter (e2) must be greater than or equal to the initial parameter (e1). Conversely, if e3 is negative, e2 must be less than or equal to e1. The increment parameter (e3) cannot be 0.

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

$$\left[\frac{e2 - e1}{e3}\right] + 1$$

where [X] means the greatest integer in X, that is, the greatest integer less than the absolute value of X and with the same sign.

If the iteration count is 0 or negative, the DO 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 transfers 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 terminates.

You can also terminate execution of a DO statement by using a statement within the range that transfers control outside the loop. The control variable of the DO statement remains defined with its current value.

When execution of a DO loop terminates, if other DO loops share its terminal statement, control transfers outward to the next outer DO loop in the DO nesting structure (see Section 4.3.2). If no other DO loop shares the terminal statement, or if this 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 the control variable for other purposes in statements within the range.

The range of a DO statement can contain other DO statements, as long as these nested DO loops 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.

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/Explanation

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

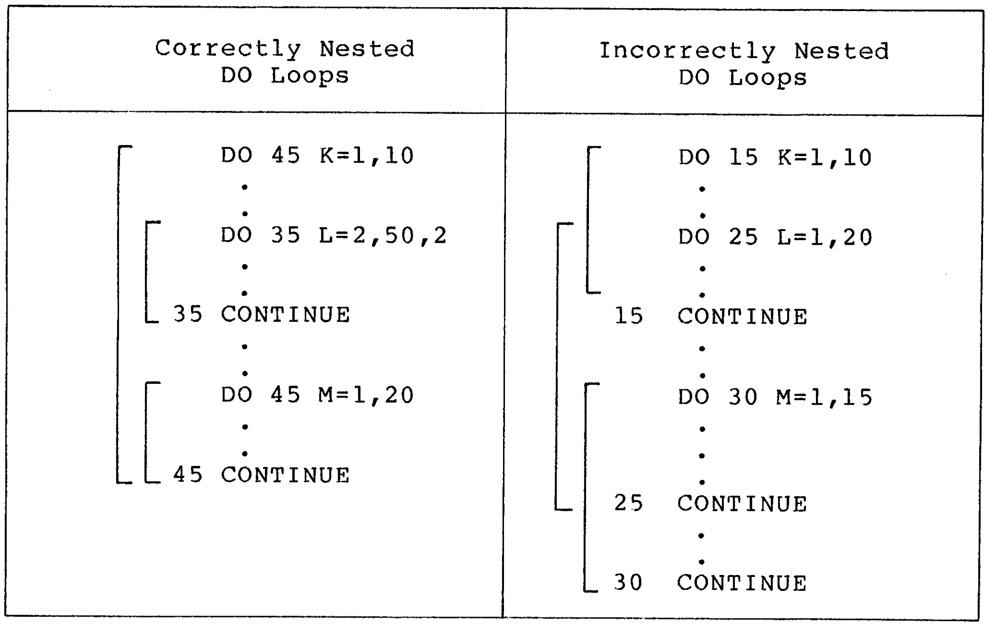
Note that in the last invalid example, the statement

```
     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. The range of an inner-nested DO loop must lie completely within the range of the next outer loop. Nested loops can share a terminal statement. Figure 4-1 illustrates nested loops.



Correctly Nested DO Loops

```
      DO 45 K=1,10
       .
       .
      DO 35 L=2,50,2
       .
       .
   35 CONTINUE
       .
       .
      DO 45 M=1,20
       .
       .
   45 CONTINUE
```

Incorrectly Nested DO Loops

```
      DO 15 K=1,10
       .
       .
      DO 25 L=1,20
       .
       .
   15 CONTINUE
       .
       .
      DO 30 M=1,15
       .
       .
       .
   25 CONTINUE
       .
       .
   30 CONTINUE
```

Figure 4-1: Nested DO Loops

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, a transfer from an outer loop to an inner loop is not permitted.

If two or more nested DO loops share the same terminal statement, you can transfer control to that statement only from within the range of the innermost loop. Since the shared terminal statement is part of the innermost loop, any transfer to the terminal statement from an outer loop is an invalid transfer.

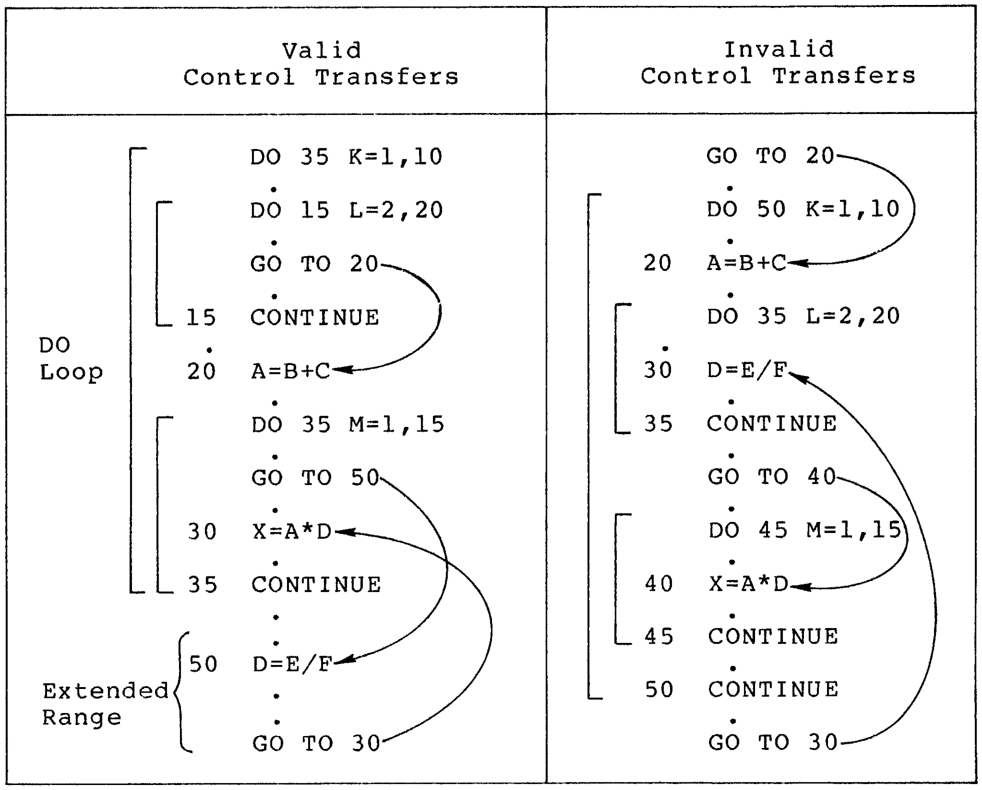
4.3.4 Extended Range

A DO loop has an extended range if it contains a control statement that transfers control out of the loop and if, after execution of one or more statements, another control statement returns control back into the loop. Thus, the range of the loop is extended to include 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 extended range:

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

Figure 4-2 illustrates valid and invalid extended range control transfers.



Valid Control Transfers

```
            DO 35 K=1,10
            .
            DO 15 L=2,20
            .
            GO TO 20
            .
        15  CONTINUE
DO          .
Loop    20  A=B+C
            .
            DO 35 M=1,15
            .
            GO TO 50
            .
        30  X=A*D
            .
        35  CONTINUE
            .
            .
        50  D=E/F
Extended    .
Range       .
            GO TO 30
```

Invalid Control Transfers

```
        GO TO 20
        .
        DO 50 K=1,10
        .
    20  A=B+C
        .
        DO 35 L=2,20
        .
    30  D=E/F
        .
    35  CONTINUE
        .
        GO TO 40
        .
        DO 45 M=1,15
        .
    40  X=A*D
        .
    45  CONTINUE
        .
    50  CONTINUE
        .
        GO TO 30
```

Figure 4-2: Control Transfers and Extended Range

4.4 CONTINUE STATEMENT

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

The CONTINUE statement has the following 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 the greater detail 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. They can be variables, arrays, array elements, constants, expressions, Hollerith constants, alphanumeric literals, 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
```

RETURN

4.6 RETURN STATEMENT

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

PAUSE

4.7 PAUSE

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 following 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 but the program is not suspended.

If the program is running in interactive mode, the contents of disp are displayed at your terminal, followed by the prompt sequence indicating that the program is suspended; you should then enter a control command, after which execution resumes with the first executable statement following the PAUSE. Because the command is specific to the operating system, it is not given here.

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 following form:

```
     STOP [disp]
```

disp
: An alphanumeric literal, 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.

Example 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 following form:

```
     END
```

The END statement must not occur on a continuation line or 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 specfication statements are:

- IMPLICIT statement -- specifies the implied data type 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 two or more entities with the same storage location
- EXTERNAL statement -- declares the specifed symbolic names to be external procedure names
- DATA statement -- assigns initial values to variables, arrays, and array elements before program execution
- 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, giving their forms and examples of their usage.

IMPLICIT

5.1 IMPLICIT STATEMENT

By default, all names beginning with the letters I through N are interpreted as integer data, and all names beginning with any other letter are interpreted as real. The IMPLICIT statement permits you to change these default data typing rules.

The IMPLICIT statement has the following form:

```
     IMPLICIT typ(a[,a]...)[,typ(a[,a]...)]...
```

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

a
: An alphabetic specification in one of two forms: c or cl-c2, where c is an alphabetic character. The latter form specifies a range of letters, from cl through c2, which must occur in alphabetical order.

The IMPLICIT statement assigns the specified data type to all symbolic names that begin with any specified letter, or any letter in a specified range, and which have no explicit data type declaration. For example:

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

These statements specify the default in the absence of any explicit statement.

IMPLICIT statements must precede all other specification statements, and all executable statements.

You cannot label IMPLICIT statements.

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

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

TYPE DECLARATION

5.2 TYPE DECLARATION STATEMENTS

The type declaration statement explicitly gives a data type to specified symbolic names.

The type declaration statement has the following form:

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

typ
: One of the data type specifiers (see Table 2-2).

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

The following rules apply to a type declaration statement:

- A type declaration statement must precede all executable statements.
- You can declare the data type of a symbolic name only once.

- You cannot label a type declaration statement.
- 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.

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:

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

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

DIMENSION

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 following 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 array element in each dimension of each array. The data type of the array determines the length of a storage element. Moreover, the total number of storage elements assigned to an array is equal to the product of its dimension declarators. For example:

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

This statement defines ARRAY as having 16 real elements of 4 bytes each, and defines MATRIX as having 125 integer elements of 2 bytes each.

You can also use array declarators in type declaration, COMMON, and VIRTUAL statements. However, in each program unit, you can use an array name in only one array declarator.

You cannot label DIMENSION statements.

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.

COMMON

5.4 COMMON STATEMENT

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

The COMMON statement has the following 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.

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

When you declare common blocks of the same name in different program units, these names are all associated with the same storage area when the program units are combined into an executable program. For example:

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

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

In the example, BLOCK1 in MAIN and BLOCK1 in GSUB are associated with the same storage area, as are the BLOCK2s.

You can have only one blank common block in an executable program, but you can have any number of 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 they each occur first in the list.

Entities assigned by a COMMON statement in one program unit should agree in data type with entities placed in one-to-one correspondence with them in the same common block by another program unit. 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 into an executable program 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 or (BYTE) 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 block are allocated beginning on a word (even byte) boundary.

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 the 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 Usage

```
LOGICAL*1 CHARS(9)
COMMON/STRING/ILEN,CHARS
```

Invalid Usage

```
LOGICAL*1 CHARS(9)
COMMON/STRING/CHARS,ILEN
```

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

VIRTUAL

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 specifies a virtual array. It specifies the number of dimensions, and the number of elements in each dimension. The VIRTUAL statement has the following 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 space available to user programs executing on a PDP-11 family computer is 64K, or 65,536 bytes. An array can have a maximum of 32,767 elements of from 1 to 8 bytes per element. A maximum LOGICAL*1 array of 1 byte per element would require 32,767 bytes of storage space. 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. Virtual arrays are placed in external main memory without significantly diminishing the 64K of directly-addressable memory available to programs.

NOTE

Virtual arrays are not supported on all PDP-11 operating systems. See the appropriate PDP-11 FORTRAN IV user's guide for more information.

Examples:

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

The above example defines a one-dimensional array named A having 1000 elements, a two-dimensional array named LARG having 32400 elements, and a seven-dimensional array named MULT having 16384 elements.

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

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

5.5.1 Restrictions on the Use of Virtual Arrays

The names of virtual arrays and virtual array elements must not be used in some contexts:

1. A virtual array name must not be used in a COMMON statement (Section 5.4).
2. The name of a virtual array or virtual array element must not be used in an EQUIVALENCE statement (Section 5.6).
3. A virtual array or virtual array element cannot be assigned an initial value by a DATA statement (Section 5.8).
4. Virtual arrays cannot be used to contain run-time format specifications (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.
5. 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 (Section 7.6).
6. 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 (Section 6.1).
7. The name of a virtual array or virtual array element cannot be used to specify the NAME keyword in an OPEN statement (Section 9.1.13).

Examples:

Valid Usage

```
     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 Usage

```
VIRTUAL A(10)
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 declared as a virtual array can only 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 declared as a simple variable (see Section 2.4). In effect, an actual argument that is a virtual array element is treated as an expression. Furthermore, a value must have been assigned to the element before it is used as an actual argument. The subprogram must not alter the value of the corresponding dummy argument.

EXAMPLES:

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)
```

EQUIVALENCE

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 following form:

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

nlist
A list of variables, array elements, and arrays separated by commas. You must specify at least two entities in each list.

The EQUIVALENCE statement allocates all of the entities in each list beginning at the same storage location.

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

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

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, 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 EQUIVALENCE statements are:

Valid Usage

```
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.

Invalid Usage

```
LOGICAL*1 BYTES(10)
EQUIVALENCE (ILEN, BYTS(2))
```

In the above 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 the corresponding elements of the two arrays. Thus, if the first elements of two equal-sized arrays are made equivalent, both arrays share the same storage space. If, for example, 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. Figure 5-1 shows how these statements align the arrays.

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		

Figure 5-1: Equivalence of Array Storage

The following statements also align the two arrays as shown in Figure 5-1:

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

In the EQUIVALENCE statement only, you can identify an array element with a single subscript (that is, the linear element number), even though the array was defined as a multidimensional array. For example, the following statement aligns the two arrays as shown in Figure 5-1:

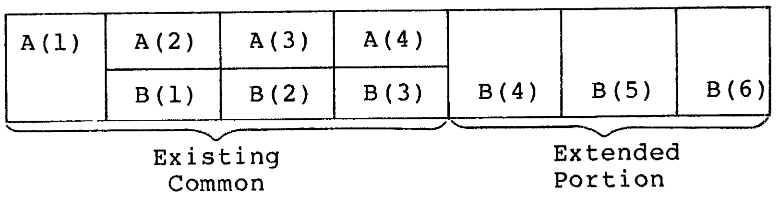
```
     EQUIVALENCE (TABLE(4), TRIPLE(7))
```

5.6.2 Extending Common Blocks

When you make entities equivalent to other 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. But you can extend the common block only beyond the last element of the previously established common block. You cannot extend the common block in such a way as to 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:

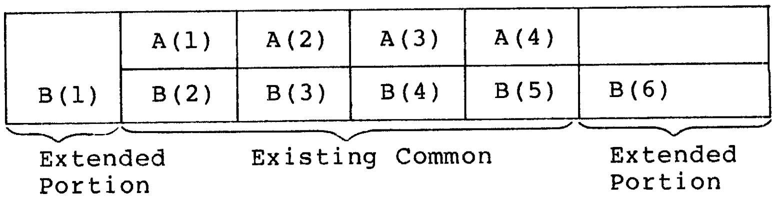
Valid

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



Invalid

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



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

5.7 EXTERNAL STATEMENT

The EXTERNAL statement lets you use external procedure names as actual arguments to other subprograms.

An external procedure can be a user-supplied function or subroutine subprogram (Section 6.2) or a FORTRAN library function (Section 6.3).

The EXTERNAL statement has the following form:

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

v

The symbolic name of a subprogram or the name of a dummy argument associated with a subprogram name.

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)
```

DATA

5.8 DATA STATEMENT

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

The DATA statement has the following form:

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

nlist
A list of one or more variable names, array names, or array element names, separated by commas, to which the values in clist are to be assigned sequentially on a one-for-one basis. Subscript expressions must be 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
Used when you specify clist as n * val. Specifies 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.

The DATA statement assigns the constant values in each clist to the entites in the preceding nlist. Values are assigned one for one in the order in which 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 nlist, values are assigned to every element. The associated constant list must therefore contain enough values to fill the array. Array elements are filled in the order of subscript progression.

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

When a Hollerith constant or alphanumeric literal is assigned to a variable or array element, the number of characters that can be assigned depends on the data type of the component (see Table 2-2). If the 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.

Examples of the DATA statement are:

```
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/
```

In the example, the DATA statements assign 0 to all 10 elements of array A, and the ASCII control character codes are assigned to the byte variables BELL, TAB, LF, FF.

Some other examples are:

```
REAL X(5)
COMPLEX Z
DATA X/2*-3.,4.,2*0.37/,Z/(1.0,-3.0)/
```

PROGRAM

5.9 PROGRAM STATEMENT

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

The PROGRAM statement has the following 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 same as the name of any subprogram, entry, or common block in the same executable program (see Section 2.1).

The PROGRAM statement must not have a statement label.

BLOCK DATA

5.10 BLOCK DATA STATEMENT

The BLOCK DATA statement begins a special type of program unit whose only purpose is to declare common blocks and to define data in common blocks. Therefore, the BLOCK DATA program unit can contain only nonexecutable statements.

The BLOCK DATA statement has the following form:

```
BLOCK DATA [nam]
```

nam
A symbolic name.

You can use only type declaration, IMPLICIT, DIMENSION, COMMON, EQUIVALENCE, and DATA statements following a BLOCK DATA 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. A BLOCK DATA statement must not have a statement label.

If you use a BLOCK DATA program unit to initialize any entity in a common block, 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 in a DATA statement. 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 is:

```
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 the example, enough information is provided to explicitly or implicitly declare 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 one or a group of statements that define a computing procedure. A subprogram is invoked when a statement that references the subprogram is executed. The referencing statement is located, in some cases, in the same program unit and, in other cases, in a different program unit.

Subprograms are of two kinds: either user-written or supplied as part of the FORTRAN system.

User-written subprograms are of three kinds:

- Statement functions
- Functions
- Subroutines

There is one kind of reference to FORTRAN library functions:

- Processor-defined function references

In many cases, the program that references the subprogram passes values, called actual arguments, to the subprogram, which uses the actual arguments to compute the results. The subprogram specifies entities, called dummy arguments, to receive the actual arguments. Values may in turn be passed back to the referencing program.

In the discussion below, 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 which passes a value to or from a subprogram. Actual arguments are specified in the statement referencing the subprogram. Dummy arguments are specified in the definition of the subprogram and are associated with actual arguments on a one-to-one basis when control is transferred to the subprogram. Each dummy argument takes on the value of the corresponding actual argument; and any value assigned to the dummy argument in the subprogram also is assigned to the corresponding actual argument. When the subprogram returns, the association of actual and dummy arguments ends. There is no retention of argument association from one reference of a subprogram to the next.

For example, 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 has a value of 4.

6.1.1 Rules Governing Subprogram Arguments

Actual arguments can be constants, variables, expressions, arrays, array elements, or subprogram names. Dummy arguments as specified in the subprogram definition appear as unsubscripted variable 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 which become associated with variables, arrays, or subprograms defined or declared in other program units. 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 with the attributes of the associated actual argument.

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

A dummy argument declared as 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.

If the actual argument is an array element, the dummy argument array will be associated with elements of the actual argument array starting from the actual argument. In this case, the dummy argument array must not be larger than the number of elements remaining in the actual argument array.

Valid Usage

```
PROGRAM MAIN
DIMENSION A(10), B(5.5)
    .
    .
    .
CALL X(A, B(1,2))
END
SUBROUTINE X(Y,Z)
DIMENSION Y(10), Z(5,2)
END
```

Invalid Usage

```
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
END                   of elements remaining in actual array)
```

6.1.2 Adjustable Arrays

An adjustable array is a dummy argument array declared in a subprogram with dimensions that can be changed or "adjusted" to match the dimensions of the associated actual argument array in the referencing program. An adjustable array declarator contains integer variables, as well as constants, in the dimension declarators.

The following rules govern the use of adjustable arrays:

- The adjustable array must be a dummy argument of the subprogram.
- The adjustable array must become associated with an actual argument that is an array.
- The size of the adjustable array must be less than or equal to the size of the actual array.
- Variables in the adjustable array declarator that represent the adjustable dimensions must be of the integer data type.
- Variables in the adjustable array declarator must be dummy arguments of the subprogram, and the corresponding actual arguments must have a defined value.

For example,

```
      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
```

In the 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.

6.2 USER-WRITTEN SUBPROGRAMS

A user-written subprogram is a FORTRAN statement or group of FORTRAN statements that perform a computing procedure. A computing procedure can be a series of either arithmetic operations or FORTRAN statements. You can use subprograms to perform a computing procedure in several places in your program, and thus avoid having to duplicate the series of operations or statements in each place.

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

Table 6-1: Types of User-Written Subprograms

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

A function reference consists of the function name and the function arguments and is used in an expression. The function returns a value that is used in place of the reference in the expression in which it appears.

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

A subprogram 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 the statement function name, with its arguments, in an expression, the computation defined by the statement function name is performed and the resulting value replaces the statement function name in evaluating the expression. Statement functions are defined and referenced within a single program unit.

The statement function definition statement has the following form:

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

f
: The name of the 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 statement function reference has the following 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 definition. The expression in the definition is then evaluated. The resulting value is used to complete the evaluation of the expression containing the function reference.

The following rules govern the use of statement functions:

- Statement function names must be unique within the same program unit.
- A statement function reference must appear in the same program unit as its definition.
- A statement function definition statement can include a reference to another statement function, which must be defined earlier in the same program unit.
- Statement function definitions must be placed before all executable statements (see Figure 1-3).
- The data type of the resulting value assigned to the name is determined either implicitly by the first letter of the name, or explicitly by a type declaration statement.
- Statement function dummy arguments serve only to indicate order, number, and data type of arguments for the statement function.
- Statement function dummy argument names do not follow the usual rules for uniqueness of symbolic names (see Section 2.1). Statement function dummy arguments must be unique only within each statement function definition. Variables or arrays of the same names as the dummy arguments can be declared and used within the same program unit.
- The data type of statement function dummy arguments is determined either implicitly by the first letter of the name, or explicitly by a type declaration statement.

Examples of statement function definitions 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/Explanation

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

Examples of statement function references follow. The examples below refer to the second statement function definition above.

Valid

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

Invalid/Explanation

```
FINAL = AVG (TEST3,TEST4,LAB2)
```

(An actual argument and its corresponding dummy argument must agree in data type; in this case, LAB2 is integer but C above is real.)

6.2.2 Function Subprograms

A function subprogram is a program unit referenced by a symbolic name. When you reference the function name, with its arguments, in an expression, the program unit defined by the function name is executed; and the resulting value of the function replaces the function name in evaluating the expression. A function subprogram consists of a FUNCTION statement followed by a series of statements that define a computing procedure.

The FUNCTION statement has the following form:

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

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

nam
The name of the function.

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

p
A dummy argument.

A function reference that transfers control to a function subprogram has the following form:

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

nam
The symbolic name of the function.

a
An actual argument.

When the name of the function subprogram is used in an expression, control is transferred to the subprogram; and the values of the actual arguments (if any) in the function reference are associated with the dummy arguments (if any) in the FUNCTION statement. The statements in the subprogram are then executed. A value must be assigned to the name of the function as though it were a variable.

Finally, a RETURN statement is executed in the function and returns control to the calling program unit. An END statement acts as an implied RETURN. The value assigned to the function's name is now used to complete the evaluation of the expression containing the name.

The following rules govern the use of function subprograms:

- The FUNCTION statement must be the first statement of a function subprogram.
- The FUNCTION statement must not have a statement label.
- A function subprogram must not contain these statements: SUBROUTINE, BLOCK DATA, or another FUNCTION statement.
- 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 implicitly or explicitly in the FUNCTION statement or in a type declaration statement.
- The function name must have the same data type in the subprogram and in the referencing program.

An example of a function subprogram is:

```
      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:

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

The caluclation 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 program unit referenced by a symbolic name. When you reference the subroutine name in a CALL statement, the program unit defined by the subroutine name is executed. In contrast to the statement function and function subprogram, no value is returned to the subroutine name. A subroutine subprogram consists of a SUBROUTINE statement followed by a series of statements that define a computing procedure.

The SUBROUTINE statement has the following 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 the subroutine, the values of the actual arguments (if any) in the CALL statement are associated with the corresponding dummy arguments (if any) in the SUBROUTINE statement. The statements in the subprogram are then executed. Finally, a RETURN statement is executed in the subroutine and it returns control to the calling program. An END statement acts as an implied RETURN.

The following rules govern the use of subroutine subprograms:

- The SUBROUTINE statement must be the first statement of a subroutine.
- The SUBROUTINE statement must not have a statement label.
- A subroutine subprogram must not contain a FUNCTION, BLOCK DATA, or another SUBROUTINE statement.
- A subroutine subprogram can reference another subprogram, but it cannot reference itself either directly or indirectly.

Example:

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.

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
```

6.3 FORTRAN LIBRARY FUNCTIONS

FORTRAN library functions are system-supplied subprograms referenced in the same way as user-written function subprograms.

For example:

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

ABS is a FORTRAN library function. As a result of this reference, the absolute value of X-1 is calculated and multiplied by the constant 3.14159; the result is assigned to the variable ^^R.

The FORTRAN library functions are listed in Appendix B, which also gives the data type of each library function and of the actual arguments.

The FORTRAN library functions also are called processor-defined functions. Note that the processor-defined functions include both the Intrinsic Functions and the Basic External Functions described in ANS FORTRAN.

Normally, a name in the table of processor-defined functions 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 appears in a type declaration statement specifying a different data type from that shown in the table.
- The name is used in a function reference with arguments of a different data type from that shown in the table.

Processor-defined function names apply only to the program unit in which they are referenced. Thus, they can be used for other purposes in other program units. In addition, the data type of a processor-defined function does not change because of an IMPLICIT statement.

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.

Each READ or WRITE statement refers to the logical unit to or from which data is to be transferred. A logical unit can be connected to a device or file by the OPEN statement (see Section 9.1).

The ACCEPT, TYPE, and PRINT statements do not refer to logical units; rather, they transfer data between the program and an implicit logical unit. The ACCEPT and TYPE statements are normally connected to the user's terminal and the PRINT statement is normally connected to the system line printer.

Input/output (I/O) statements are grouped into three categories:

- Sequential I/O - transfers records sequentially to and from files, or to and from an I/O device such as a terminal. See Section 7.3.
- Direct Access I/O - transfers records, selected by record number, to and from direct access files. See Section 7.4.
- Internal I/O - ENCODE and DECODE statements translate and transfer data between variables and arrays within the FORTRAN program. See Section 7.6.

I/O statements that contain format specifiers are called formatted I/O statements. Formatted I/O statements are used to translate data between internal (binary) form within the program and external (readable character) form in the records.

I/O statements that do not contain format specifiers are called unformatted I/O statements. Unformatted I/O statements transfer data without translation. Unformatted I/O is generally used when data output by a program will be subsequently input by the same (or a similar) program. Unformatted I/O saves execution time by eliminating the data translation process, preserves greater precision in the external data, and usually conserves file storage space.

I/O statements transfer all data in terms of records. The amount of data that one record can contain, and the way records are separated, depend on how the data is transferred.

In unformatted I/O, the I/O statement specifies the amount of data to be transferred. In formatted I/O, the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.

Executing an input or output statement initiates transfer of a new record. Normally, the data transferred by an I/O statement constitutes one record. However, formatted I/O statements can transfer more than one record.

Section 7.1 describes general FORTRAN input/output concepts. Section 7.2 describes the components of FORTRAN I/O statements. Sections 7.3 through 7.6 describe each category of I/O statement.

7.1 I/O OVERVIEW

The following sections describe in general terms the characteristics of FORTRAN I/O processing: records, files and access modes. See the appropriate PDP-11 FORTRAN IV user's guide for more detail about FORTRAN I/O processing.

7.1.1 Records

A record is a collection of data items, called fields, that are logically related and are processed as a unit. Each FORTRAN I/O statement transfers one record. Formatted I/O statements may transfer additional records.

If an input statement does not use all of the data fields in a record, the remaining fields are ignored. If an input statement requires more data fields than the record contains, an error occurs.

If an output statement attempts to write more data fields than the record can contain, an error condition occurs.

7.1.2 Files

A file is a collection of logically related records, arranged in a sequential order, and treated as a unit. The arrangement of a file is determined when the file is created.

Files can be stored on disk or on magnetic tape. Other peripheral devices such as terminals, card readers, and line printers are treated as sequential files.

In the sequential file organization, records appear in physical sequence. Each record, except the first, has another record preceding it, and each record, except the last, has another record following it. The physical order in which records appear is always identical to the order in which the records were originally written to the file.

7.1.3 Access Modes

Access mode is the method your 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 IV supports two kinds of access modes: sequential and direct.

7.1.3.1 Sequential Access - Sequential access means that records are processed in sequence. For a sequential organization file, the sequence is the physical sequence of the records.

7.1.3.2 Direct Access

7.1.3.2 Direct Access - Direct access means that the program specifies the order of processing by including a direct access record number in each I/O statement. For sequential organization files, the records must be fixed-length.

7.2 I/O STATEMENT COMPONENTS

The following sections describe the components of I/O statements: logical unit numbers, format specifiers, direct access record numbers, key expressions, I/O lists, and parameters specifying the transfer of control if an error or end-of-file condition occurs.

7.2.1 Logical Unit Numbers

A logical unit number is an integer value that refers to a specific file or I/O device. A logical unit number must be an integer constant or variable with a value in the range 1 through 99.

A logical unit number is connected to a file or device in one of two ways:

- Explicitly through an OPEN statement (see Section 9.1).
- Implicitly by the system. The appropriate PDP-11 FORTRAN IV user's guide describes the use of implicit logical unit numbers in greater detail.

7.2.2 Format Specifiers

Format specifiers are used in formatted I/O statements and can be any of the following:

- The statement label of a FORMAT statement.
- The name of an array containing a run-time format (see Section 8.6).

Chapter 8 describes FORMAT statements. Section 8.7 describes the interaction between formats and I/O statements.

In sequential I/O statements, you can use an asterisk instead of a format specifier to denote list-directed formatting. See Sections 7.3.3 and 7.3.4 on list-directed I/O.

7.2.3 Direct Access Record Numbers

A direct access record number is an integer value that specifies the position of the record in a direct access file.

The value must be greater than or equal to 1, and less than or equal to the maximum number of records in the file.

7.2.4 End-of-File Condition and Error Condition Parameters

If an I/O error or end-of-file condition is encountered, any READ, WRITE, REWRITE, ENCODE or DECODE statement can specify that control is to be transferred to a specified statement. The specifiers have the following forms, respectively, for end-of-file and error conditions:

```
END=s

ERR=s
```

s
: The label of an executable statement to which control is to be transferred.

A READ, WRITE, REWRITE, ENCODE or DECODE statement can include either or both of the above specifications, in any order. The specification(s) must follow the unit number, record number, and/or format.

The statement label in the END=s or ERR=s specification must refer to an executable statement within the same program unit as that of 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 9.8) 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 or direct 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 appropriate PDP-11 FORTRAN IV 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 the type of error that occurred.

Examples of 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 in execution of the WRITE statement.

```
READ (1,FORM,ERR=150,END=200) ARRAY
```

This statement transfers control to statement 150 if an error occurs in the execution of the READ statement and to statement 200 if the end-of-file condition occurs.

7.2.5 Input/Output Lists

The I/O list in an input, output, ENCODE, or DECODE statement contains the names of variables, arrays, and array elements from which or to which data will 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 following 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.

7.2.5.1 Simple Lists - A simple I/O list element can be a single variable, array, array element, constant, or expression. 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 and enclosed in parentheses. For example:

```
     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 defines a two-dimensional array:

```
     DIMENSION ARRAY(3,3)
```

if the name ARRAY, with no subscripts, appears in a READ statement, that statement assigns values from the input record(s) 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:

```
          READ (1,1250) J,K,ARRAY(J,K)
     1250 FORMAT (I1,X,I1,X,F6.2)
```

The input record 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, thereby establishing the actual subscript values for ARRAY(J,K). Then the value 721.73 is 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 further I/O operations. For example, an output statement I/O list expression must not refer to a function subprogram that performs I/O.

An input statement I/O list must not contain an expression, except as a subscript expression in an array reference.

7.2.5.2 Implied DO Lists - An implied DO list is an I/O list element that functions as though 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 a sequence different 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

Integer expressions.

The variable i and the parameters e1, e2, and e3 have the same forms and the same 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. Some examples are:

```
WRITE (3,200) (A,B,C, I=1,3)
```

The statement in this example behaves as though you had written:

```
WRITE (3,200) A,B,C,A,B,C,A,B,C,
```

Another example is:

```
WRITE (6) (I,(J,P(I),Q(I,J),J=1,L),I=1,M)
```

The I/O list in this example consists of an implied DO list containing another implied DO list nested within it. The implied DO lists together will write a total of (1+3*L) *M fields, varying the Js for each value of I.

In a series of nested implied DO lists, the parentheses indicate the nesting (see Section 4.3.2). Execution of the innermost lists is repeated most often. For 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 L advances from 1 through 10 for each increment of the first subscript. This is the reverse of the order of subscript progression. In addition, K is incremented by 2, so 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:

```
     READ (5,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

In this example, 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:

```
     READ (3,5555) (BOX(1,J), J=1,10)
```

This statement 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:

```
     WRITE (6,1111) (I, I=1,20)
```

This statement simply prints the integers 1 through 20.

7.3 SEQUENTIAL INPUT/OUTPUT

Sequential I/O statements transfer records sequentially to or from either files or I/O devices.

Formatted sequential I/O statements transfer records using format specifiers to control the translation of data between internal and external form.

List-directed sequential I/O statements transfer formatted records. Instead of using a format specifier, the data types of the I/O list elements control the translation of data between internal and external form. In effect, list-directed sequential I/O statements are a method of obtaining simple formatted input or output without using FORMAT statements. Both formatted sequential and list-directed I/O statements can refer to the same logical unit. When you read files that contain both formatted and list-directed records, you must ensure that each record is read with the correct format.

Unformatted sequential I/O statements transfer records of binary data without translation.

READ
ACCEPT

7.3.1 Formatted Sequential Input Statements

The formatted sequential READ statement transfers data from the specified logical unit. If a formatted sequential READ statement does not have a logical unit number, it uses an implicit logical unit.

The formatted sequential ACCEPT statement is similar to a formatted sequential READ statement except that an implicit logical unit number is always used.

Formatted sequential input statements have the following forms:

```
     READ (u,f[,END=s][,ERR=s])[list]

     READ f[,list]

     ACCEPT f[,list]
```

u
: A logical unit number.

f
: A format specifier.

s
: The label of an executable statement.

list
: An I/O list.

A statement of the following form causes data to be read from a system-defined logical unit:

```
     READ 200, ALPHA,BETA,GAMMA
```

Characters transferred by formatted sequential statements are translated to the internal form specified by the format specifier. The resulting values are assigned to the elements of the I/O list.

If the number of list elements is less than the number of input record fields, the excess portion of the record is ignored.

Usually a single formatted record is transferred by the execution of a formatted sequential input statement. However, the format specifier can specify that more than one record is to be read during execution of a single input statement.

If the FORMAT statement associated with a formatted input statement contains a Hollerith constant, input data is read and stored directly into the storage location of the format specification. See Section 8.1.9.

If no I/O list is present, data transfer occurs only between the record and the storage location of the format specifier. For example:

```
           READ (5,100)
     100   FORMAT (15H DATA GOES HERE)
```

These statements read 15 characters from the next record on logical unit 5. If the 15 characters are:

```
     REVIEW SECTIONS
```

The FORMAT statement becomes:

```
     100 FORMAT (15HREVIEW SECTIONS)
```

Other examples of formatted sequential input statements follow:

```
           READ (1,300) ARRAY
     300   FORMAT (20F8.2)
```

These statements read a record from logical unit 1 and assign fields to ARRAY.

```
          READ 100, ICOUNT,ALPHA,BETA
      100 FORMAT (I5, F8.2, F5.1)
```

These statements read a record from an implicit logical unit, and assign fields to integer variable ICOUNT and real variables ALPHA and BETA.

WRITE
TYPE
PRINT

7.3.2 Formatted Sequential Output Statements

The formatted sequential WRITE statement transfers data to the specified logical unit.

The formatted sequential TYPE and PRINT statements are similar to the formatted sequential WRITE statement, except that output is directed to an implicit logical unit.

The formatted sequential output statements have the following forms:

```
WRITE (u,f[,ERR=s])[list]

TYPE f[,list]

PRINT f[,list]
```

u
A logical unit number.

f
A format specifier.

s
The label of an executable statement.

list
An I/O list.

The I/O list specifies a sequence of values that are converted to characters and positioned as specified by the format specifier. If no I/O list is present, data transfer occurs only between the storage location of the format specifier and the record.

The data transferred by a formatted sequential output statement normally constitutes one formatted record. However, the format can specify that additional records are to be written during execution of a single output statement.

Numeric data output under format control is rounded during the conversion to external format. If such data is input for additional calculations, loss of precision may result. To avoid loss of precision, use unformatted output.

The records transmitted by a formatted WRITE statement must not exceed the length that the specified device can accept. For example, a line printer typically cannot print a record longer than 132 characters.

Examples of formatted sequential output statements follow.

```
          WRITE (6, 650)
     650  FORMAT (' HELLO THERE')
```

These statements write the alphanumeric literal contained in the FORMAT statement to logical unit 6.

```
          WRITE (1,95) AYE,BEE,CEE
     95   FORMAT (3F8.5)
```

These statements write one record, consisting of three fields, to logical unit 1.

```
          WRITE (1,950) AYE,BEE,CEE
     950  FORMAT (F8.5)
```

These statements write three separate records, consisting of one field each, to logical unit 1.

In the last example, the rightmost parenthesis of the FORMAT statement is reached before all elements of the I/O list are output. Each time this occurs, the current record is terminated and a new record is initiated. Thus, three separate records are written. For a more complete explanation, see Section 8.7.

READ
ACCEPT

7.3.3 List-Directed Input Statements

The list-directed READ statement transfers records from the specified logical unit, translates the data from external to internal form, and assigns the resulting values to the elements of the I/O list in the order in which those elements appear, from left to right. The I/O list is required. If a list-directed READ statement does not include a logical unit number, an implicit logical unit number is used. The list-directed ACCEPT statement is similar to a list-directed READ statement except that an implicit logical unit number is always used.

The list-directed input statements have the following forms:

```
READ (u,*[,END=s][,ERR=s]) list

READ *,list

ACCEPT *,list
```

u
: A logical unit number.

*
: Indicates list-directed formatting.

s
: The label of an executable statement.

list
: An I/O list.

The external record must contain a sequence of values and value separators. A value can be:

- A constant
- A null value
- A repetition of constants in the form r*c
- A repetition of null values in the form r*

The following paragraphs describe these values.

Each input constant has the form of a FORTRAN constant (see Section 2.3). 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 is either T or t (true) or F or f (false). Hollerith and octal constants are not permitted. If the data types of a list element and its corresponding constant do not match, conversion is performed according to the rules for arithmetic assignment (see Table 3-1).

A null value is specified by two consecutive commas with no intervening constant. Spaces can occur between the commas. A null value specifies that the corresponding list element remains unchanged. A null value cannot be used for either part of a complex constant, but can represent an entire complex constant.

The form r*c indicates 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* indicates r occurrences of a null value where r is an unsigned integer constant.

A value separator can be:

- One or more spaces or tabs
- A comma, with or without surrounding spaces or tabs
- A slash

The slash value separator terminates processing on the input statement and record; all remaining I/O list elements are unchanged.

Each input statement will read one or more records as required to satisfy the I/O list. If a slash separator occurs or the I/O list is exhausted before all the values in a record are used, the remainder of the record is ignored.

The end of a record is equivalent to a space character.

Spaces at the beginning of a record are ignored.

An example follows.

The program unit includes:

```
     DOUBLE PRECISION T
     COMPLEX D,E
     LOGICAL L,M
     READ (1,*) I,R,D,E,L,M,J,K,S,T,A,B
          .
          .
          .
```

The external record contains:

```
     4 6.3 (3.4E0,4.2e0), (3, 2 ) , t,F,,3*14.6 /
```

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	14.6DO

A, B, and J will be unchanged.

WRITE
TYPE
PRINT

7.3.4 List-Directed Output Statements

The list-directed WRITE statement transfers the elements in the I/O list to the specified logical unit, translating and editing each value according to the data type of the value.

The list-directed TYPE and PRINT statements are similar to the list-directed WRITE statement, except that output is directed to an implicit logical unit.

List-directed output statements have the following forms:

```
     WRITE (u,*[,ERR=s]) list

     TYPE *,list

     PRINT *,list
```

u

A logical unit number.

*

Indicates list-directed formatting.

s

The label of an executable statement.

list

An I/O list.

The values in the I/O list are converted to character form and written in a fixed format according to the data type of the value. Table 7-1 lists the output formats for each data type. The I/O list is required.

Table 7-1: List-Directed Output Formats

Data Type	Output Format
LOGICAL*1	I5
LOGICAL*4	L2
INTEGER*2	I7
INTEGER*4	I12
REAL*4	1PG15.7
REAL*8	1PG25.16
COMPLEX*8	IX,'(',1PG14.7, ',', 1PG14.7,')'
Hollerith	1X,An (n is the length of the Hollerith constant)

List-directed output statements do not produce octal values, null values, slash separators, or repeated forms of values. Alphanumeric literals in the I/O list itself are output without delimiting apostrophes. Note that list-directed output records that contain Hollerith constants may not be input using list-directed formatting.

Each output record begins with a space for carriage control. Each output statement writes one or more complete records (see Section 8.7). Each output value is contained within a single record, except for Hollerith constants that are longer than a record.

Some examples follow:

```
PRINT *, 'THE△ARRAY△Z△IS',Z
TYPE *,'THE△ANSWER△IS',(I,XX(I),I=1,10)
```

If a program unit consists of:

```
DIMENSION A(5)
DATA A/5*3.4/
WRITE (1,*) 'ARRAY△VALUES△FOLLOW'
WRITE (1,*) A,5
```

then the following records will be output:

```
ARRAY△VALUES△FOLLOW
△△△3.400000△△△△△△△3.400000△△△△△△△3.400000△△△△△△△3.400000
△△△3.400000△△△△△△△△△△5
```

READ

7.3.5 Unformatted Sequential Input Statement

The unformatted sequential READ statement transfers one unformatted record from the specified logical unit, and assigns the untranslated fields of the record to the I/O list elements in the order in which they appear, from left to right. The data type of each element determines the amount of data input to the element.

The unformatted sequential READ statement has the following form:

```
     READ (u[,END=s][,ERR=s])[list]
```

u
: A logical unit number.

s
: The label of an executable statement.

list
: an I/O list.

An unformatted sequential READ statement reads exactly one record. If the I/O list does not use all the values in the record (that is, 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 an unformatted sequential READ statement contains no I/O list, one full record is skipped.

The unformatted sequential READ statement must only be used to read records created by unformatted sequential WRITE statements.

Examples of unformatted sequential input statements follow.

```
     READ (1) FIELD1, FIELD2
```

This statement reads one record from logical unit 1 and assigns values to variables FIELD1 and FIELD2.

```
     READ (8)
```

This statement advances logical unit 8 by one record.

WRITE

7.3.6 Unformatted Sequential Output Statements

The unformatted sequential WRITE statement transfers the untranslated values of the elements in the I/O list to the specified logical unit as one unformatted record.

The unformatted sequential WRITE statement has the following form:

```
     WRITE (u[,ERR=s])[list]
```

u
    A logical unit number.

s
    The label of an executable statement.

list
    An I/O list.

If an unformatted WRITE statement contains no I/O list, one null record is output to the specified unit.

Some examples follow.

```
     WRITE (1) (LIST(K),K=1,5)
```

This statement outputs the contents of elements 1 through 5 of array LIST to logical unit 1.

```
     WRITE (4)
```

This statement writes a null record on logical unit 4.

7.4 DIRECT ACCESS INPUT/OUTPUT

Direct access I/O statements transfer records, specified by record numbers, to and from direct access files. Each direct access I/O statement contains a record number. The OPEN statement (Section 9.1) establishes the attributes of the direct access file.

Unformatted direct access I/O statements transfer records of binary data without translation. The DEFINE FILE statement (Section 9.9) may be used to specify the attributes of the direct access file.

READ

7.4.1 Unformatted Direct Access Input Statement

The unformatted direct access READ statement transfers the specified record from the file currently connected to the specified unit, and assigns the untranslated fields of the record to the I/O list elements.

The unformatted direct access READ statement has the following form:

```
     READ (u'r[,ERR=s]) [list]
```

u
    A logical unit number.

r
    The record number.

s
    The label of an executable statement.

list
    An I/O list.

If the I/O list does not use all the fields in the record (that is, there are more fields 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 record fields, an error occurs.

Examples of unformatted direct access input statements follow.

```
     READ (1'10) LIST(1),LIST(8)
```

This statement reads record 10 of a file on logical unit 1, and assigns 2 integer values to specified elements of array LIST.

```
     READ (4'IREC) (RHO(N),N=1,5)
```

This statement reads the record specified by the value of IREC of a file on logical unit 4, and assigns 5 real values to array RHO.

WRITE

7.4.2 Unformatted Direct Access Output Statement

The unformatted direct access WRITE statement transfers the untranslated values of the elements in the I/O list to the specified record of the file currently connected to the specified unit.

The unformatted direct access WRITE statement has the following form:

```
     WRITE (u'r[,ERR=s]) [list]
```

u
: A logical unit number.

r
: The record number.

s
: An executable statement label.

list
: An I/O list.

If the values specified by the I/O list do not fill the record, the unused portion of the record is filled with zeroes.

If the I/O list specifies more data than can fit into the record, an error occurs.

You can use a WRITE statement either to write a new record or to update an existing record.

Example:

```
     WRITE (2'35) (NUM(K),K=1,10)
```

This statement outputs 10 integer values to record 35 of the file connected to logical unit 2.

7.5 ENCODE AND DECODE STATEMENTS

The ENCODE and DECODE statements transfer data according to format specifiers, translating the data from internal to character form, and vice versa. Unlike conventional formatted I/O statements, however, these data transfers take place entirely between variables or arrays in the FORTRAN program.

The ENCODE and DECODE statements have the following 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. It corresponds to the length of a formatted record.

f
: A format specifier. If more than one record is specified, an error occurs.

b
: The name of an array, array element or variable. b corresponds to a formatted record. In the ENCODE statement, b receives the characters after translation to external form. In the DECODE statement, it contains the characters to be translated to internal form.

s
: The label of an executable statement.

list
: An I/O list (see Section 7.2.6). 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.

Similar to a WRITE statement, the ENCODE statement translates the list elements to character form according to the format specifier, and stores the characters in b. If fewer than c characters are transmitted, the remaining character positions are filled with spaces.

Similar to a READ statement, the DECODE statement translates the character data in b to internal (binary) form according to the format specifier, and stores this data in the elements in the list.

If b is an array, its elements are processed in the order of subscript prograssion. The record buffer b must not be the name of a virtual array or a virtual array element (see Section 5.5).

The number of characters that the ENCODE or DECODE statement can process depends on the data type of b in that statement. For example, since an INTEGER*2 array can contain 2 characters per element, the maximum number of characters is twice the number of elements in that array.

The interaction between the format specifier and the I/O list (see Section 8.7) for an ENCODE or DECODE statement is the same as that for a formatted I/O statement.

An example of the ENCODE and DECODE statements follows.

```
          DOUBLE PRECISION INBUF, OUTBUF
          INTEGER*2 A,B,C,D
          DATA INBUF/'12345678'/
          DECODE (8,100,INBUF) A,B,C,D
          ENCODE (8,100,OUTBUF) D,C,B,A
     100  FORMAT (4I2)
```

The DECODE statement translates the 8 characters in INBUF to integer form (specified by statement 100), and stores them in the integer variables A,B,C,D, as follows:

```
     A = 12
     B = 34
     C = 56
     D = 78
```

The ENCODE statement translates the values D,C,B,A to character form and stores the characters in the variable OUTBUF, as follows:

```
     OUTBUF = '78563412'
```

CHAPTER 8

FORMAT STATEMENTS

FORMAT statements are nonexecutable statements used with formatted I/O statements, and with ENCODE and DECODE statements, to specify the editing and formatting of the data. If input is being performed or a DECODE statement is being executed, the format statement describes the format in which the input data is interpreted. If output is being performed or an ENCODE statement is being executed, the format statement describes the format in which the output data will be represented.

Throughout this chapter a distinction is made between "external" 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 following form:

```
     FORMAT (q1f1s1f2s2 ...  fnqn)
```

q
: Zero or more slash (/) record terminators.

f
: A field descriptor or a group of field descriptors enclosed in parentheses.

s
: A field separator.

The entire list of field descriptors, field separators, and record terminators, including the parentheses, is called the format specification. The list must be enclosed in parentheses.

The field separators are comma and slash. A slash is also a record terminator. Section 8.5 describes in detail the functions of the field separators.

A field descriptor has the following form:

```
     [r]c[w[.d]]
```

r
: The number of times the field descriptor is to be repeated (repeat count). If you omit r, it is assumed to be 1.

c
: A field descriptor code (I,O,F,E,D,G,L,A,H,X,T,P,Q,$, or :).

w

The external field width.

d

The number of characters to the right of the decimal point.

The terms r, w, and d must all be unsigned integer constants. r, w, and d must be less than or equal to 255; and r and w must be nonzero. The r term is optional; however, you cannot use it in some field descriptors. The d term is required in some field descriptors and is invalid in others.

The field descriptors are:

- Integer -- Iw, Ow
- Logical -- Lw
- Real, double precision, and complex -- Fw.d, Ew.d, Dw.d, Gw.d
- Character -- Aw
- Editing, and Hollerith constant -- nH, '...', nX, Tn, nP, Q, $, : (n is a number of characters or character positions)

Section 8.1 describes each field descriptor in detail.

The first character in an output record generally contains carriage control information. See Section 8.3 for more information.

During data transfers, the format specification is scanned from left to right. The elements in the I/O list are correlated one-for-one with the corresponding field descriptors. However, the editing and Hollerith constant field descriptors do not require an I/O list element. Section 8.7 describes in detail the interaction between format specifiers and the I/O list.

You use an I, O, or L field descriptor to process integer and logical data. You use an F, E, D, or G field descriptor to process real, double precision, and complex data.

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 DESCRIPTORS

A field descriptor describes the size and format of a data item or of several data items; each data item in the external medium is called an external field. The following sections describe each of the field descriptors in detail. The field descriptors ignore leading spaces in the external field, but treat embedded and trailing spaces as zeroes.

8.1.1 I Field Descriptor

The I field descriptor specifies transfer of decimal integer values. It has the following form:

```
Iw
```

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 read from the external field, interpreted as a decimal integer value, and assigned to the corresponding I/O list element.
- The external data must be an integer constant; it cannot contain a decimal point or exponent field.
- If the external data 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 Example:

Format	**External Field**	**Internal Value**
I4	2788	2788
I3	-26	-26
I9	△△△△△△312	312
I4	2 8	2008

Rules in Effect for Data Output:

- The I field descriptor specifies output of the value of the corresponding I/O list element, right justified, to an external field w characters long, as a decimal integer.
- 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 the value of the list element is negative, the field will have a minus sign as its leftmost, nonblank character. The term w must therefore be large enough to provide for a minus sign, when necessary.
- Plus signs are suppressed.

Output Example:

Format	**Internal Value**	**External Representation**
I3	284	284
I4	-284	-284
I5	174	174
I2	3244	**
I3	-473	***
I7	29.812	Not permitted: error

8.1.2 O Field Descriptor

The O field descriptor specifies transfer of octal integer values. It has the following form:

```
Ow
```

The corresponding I/O list element must be of either integer or logical data type.

Rules in Effect for Data Input:

- The O field descriptor specifies that w characters are read from the 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 Example:

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 output of the octal value of the corresponding I/O list element, right justified, to an external field w characters long, as an octal integer.
- No signs are output; a negative value is transmitted in its octal (2'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.

Output Example:

Format	**Internal (Decimal) Value**	**External (octal) Representation**
O6	32767	△77777
O6	-32767	100001
O2	14261	**
O4	27	△△33
O5	13.52	Not permitted: error

8.1.3 F Field Descriptor

The F field descriptor specifies the transfer of real or double precision values. It has the following form:

```
Fw.d
```

The corresponding I/O list element must be of either 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 read from the 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 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 constants and Section 2.3.3 for double precision constants), 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.
- 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 transfers the value of the corresponding I/O list element, 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.
- 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 point, plus 1 for the point, plus the number of digits to the left of the point, plus 1 for a possible negative sign.

Output Examples::

Format	**Internal Value**	**External Representation**
F8.5	2.3547188	△2.35472
F9.3	8789.7361	△8789.736
F2.1	51.44	**
F10.4	-23.24352	△△-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

8.1.4 E Field Descriptor

The E field descriptor specifies transfer of real or double precision values in exponential form. It has the following form:

Ew.d

The corresponding I/O list element must be of either 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 transfers the value of the corresponding I/O list element, 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, a minus sign if the value is negative, an optional 0, a decimal point, d digits to the right of the decimal point, and a 4-character exponent in one of the following two forms:

  E+nn
  E-nn

  nn
  A 2-digit integer constant.

- Plus signs are suppressed.

- w must be greater than or equal to d+7; that is, the field width must not be stated to be less than the number of digits after the point, plus 1 for the point, plus 1 for the 0 before the point, plus one for a possible negative sign, plus 4 for the exponent.

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	*****

8.1.5 D Field Descriptor

The D field descriptor specifies transfer of real or double precision values in exponential form with a D instead of an E. It has the following form:

Dw.d

The corresponding I/O list element must be of either 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 Input:

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	△△△△△0.363D-01
D23.12	5413.87625793	△△△△△0.541387625793D+04
D9.6	1.2	*********

8.1.6 G Field Descriptor

The G field descriptor specifies transfer of real or double precision values, combining E- or F-type formats according to the size of the number being output. It has the following form:

Gw.d

The corresponding I/O list element must be of either 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 transfers the value of the corresponding I/O list element, 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 Conversion
m < 0.1	Ew.d
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

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.
- w must be greater than or equal to d+7; that is, the field width must not be stated to be less than the number of digits after the point, plus 1 for the point, plus 1 for a possible 0 before the point, plus 1 for a possible negative sign, plus 4 for a possible exponent.

Output Examples:

Format	Internal Value	External Representation
G13.6	0.01234567	△0.123457E-01
G13.6	-0.12345678	-0.123457△△△△
G13.6	1.23456789	△△1.23457△△△△
G13.6	12.34567890	△△12.3457△△△△
G13.6	123.45678901	△△123.457△△△△
G13.6	-1234.56789012	△-1234.57△△△△
G13.6	12345.67890123	△△12345.7△△△△
G13.6	123456.78901234	△△123457.△△△△
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	△△△△△0.012346
F13.6	-0.12345678	△△△△-0.123457
F13.6	1.23456789	△△△△△1.234568
F13.6	12.34567890	△△△△12.345679
F13.6	123.45678901	△△△123.456789
F13.6	-1234.56789012	△-1234.567890
F13.6	12345.67890123	△12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	*************

8.1.7 L Field Descriptor

The L field descriptor specifies transfer of logical data. It has the following 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 read from the external field.
- If the first nonblank character of the field is the letter 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 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 output of either the letter T (if the value of the corresponding I/O list element is .TRUE.), or the letter F (if the value is .FALSE.) to an external field w characters long.
- The 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.	△△△△T
L1	.FALSE.	F

8.1.8 A Field Descriptor

The A field descriptor specifies the transfer of Hollerith values. It has the following form:

```
Aw
```

The corresponding I/O list element can be of any data type, since you can use variables of any data type 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, as follows:

I/O List Element	Maximum Number of Characters
BYTE	1
LOGICAL*1	1
LOGICAL*4	4
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*8	8
DOUBLE PRECISION	8
COMPLEX	8

- 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 that element. Leftmost excess characters are ignored.
- If w is less than the number of characters than 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 Value**	
A6	PAGE△#	#	(LOGICAL*1)
A6	PAGE△#	△#	(INTEGER*2)
A6	PAGE△#	GE△#	(REAL)
A6	PAGE△#	PAGE△#△△	(DOUBLE PRECISION)

Rules in Effect for Output:

- The A field descriptor specifies output of the contents of the corresponding I/O list element 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	△OHMS
A5	VOLTS△△△	VOLTS
A5	AMPERES△	AMPER

8.1.9 H Field Descriptor

The H field descriptor specifies transfer of data between the 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.

ci

An ASCII character.

Rules in Effect for Input:

- The H field descriptor specifies acceptance of n characters from the external field and their assignment to the same storage location as the characters of the H descriptor, which are overlaid by the input data, character for character.

Rules in Effect for Output:

- The H field descriptor specifies output of the n characters following the letter H to the external field.

An example of the 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, placing the input data in the H field descriptor in statement 200. The new characters replace the words TITLE GOES HERE. If the user enters less than 20 characters, the remainder of the H field descriptor is filled with spaces to the right. The H field descriptor may also be specified as an alphanumeric literal.

In an alphanumeric literal, 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.

8.1.10 X Field Descriptor

The X field descriptor specifies skipping character positions. It has the following form:

```
     nX
```

The term n specifies how many character positions are to be skipped. The value of n must be greater than or equal to 1 and less than or equal to 255.

Rules in Effect for Input:

- The X field descriptor specifies that the next n character in the input record are to be skipped.

Rules in Effect for Output:

- The X field descriptor specifies output of n spaces to the external record. For example:

```
       WRITE (6,90) NPAGE
     90  FORMAT (13H1PAGE△NUMBER△,I2,16X,23HGRAPHIC△ANALYSIS,△CONT.)
```

The WRITE statement prints a record similar to:

```
     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.)

8.1.11 T Field Descriptor

The T field descriptor specifies the position of the next character to be treated relative to the start of the external record. It has the following form:

```
     Tn
```

The term n indicates the position in the external record of the next character to be treated. The value of n must be greater than or equal to 1 but not greater than the number of characters allowed in the record.

Rules in Effect for Input:

- In an input statement, the T field descriptor specifies that data is to be input starting with the nth character position. For example:

```
    10  FORMAT (T7,A3,T1,A3)
        READ (5,10) J,K
```

In the example, a 3-character string starting at character position 7 in the external record is read first, followed by a 3-character string starting at character position 1; however, any order can be specified.

Rules in Effect for Output:

- In an output statement, the T field descriptor specifies that the data is to be output starting at the nth character position of the external record. For example:

```
        PRINT 25
    25  FORMAT (T50,'COLUMN△2',T20,'COLUMN△1')
```

These statements will print "COLUMN 1" at position 20 and COLUMN 2" at position 50. The remainder of the line contains blank characters.

8.1.12 Q Field Descriptor

The Q field descriptor specifies assignment to the corresponding variable in the I/O list of the number of characters in the input record remaining to be transferred during a READ operation. It has the following form:

```
Q
```

The corresponding I/O list element must be of integer or logical data type.

For example:

```
         READ (4,1000) XRAY,KK,NCHRS,(ICHR(I),I=1,NCHRS)
    1000 FORMAT (E15.7,I4,Q,80A1)
```

These input statements read two fields into the variables XRAY and KK. The number of characters remaining in the record is stored in NCHRS and exactly that many characters are read into the array ICHR. By placing the Q descriptor first in the format specification, you can determine the actual length of the input record.

In an output statement, the Q field descriptor has no effect except that the corresponding I/O list element is skipped.

8.1.13 Dollar Sign Descriptor

The dollar sign character ($) used as a field descriptor suppresses, on output, a carriage return at the end of the line when the first character of the line is a space or a plus sign (see Section 8.3 on carriage control characters). In an input statement, the $ descriptor is ignored. The $ descriptor is intended primarily for interactive I/O; it leaves the terminal print position at the end of the text (rather than returning it to the left margin) so that a typed response will follow the output on the same line.

Thus, the statements

```
         TYPE 100
     100 FORMAT ('△ENTER△RADIUS△VALUE△',$)
         ACCEPT 200,X
     200 FORMAT (F6.2)
```

produce a message on the terminal in the following form:

```
     ENTER△RADIUS△VALUE
```

Your response (in this case, 12.) can then go on the same line, as follows:

```
     ENTER△RADIUS△VALUE△12.
```

Note that the $ used as a carriage control character accomplishes the same result. The following two formats are equivalent:

```
     200 FORMAT (11H△SIGN△HERE:,$)

     200 FORMAT (11H$SIGN△HERE:)
```

8.1.14 Colon Descriptor

The colon character (:) used as a field descriptor terminates format control if no more items are in the I/O list. The : has no effect if I/O list items remain. For example:

```
          PRINT   100,3
          PRINT   200,4
     100  FORMAT('△I=',I2, '△J=',I2)
     200  FORMAT('△K=',I2,:,'△L=',I2)
```

These statements print the following two lines:

```
     I=△3△J=
     K=△4
```

Section 8.7 describes format control in detail.

8.1.15 Complex Data Editing

A complex value is an ordered pair of real values. Therefore, input or output of a complex value is governed by two real field descriptors, using any combination of the forms Fw.d, Ew.d, Dw.d, or Gw.d.

Rules in Effect for Input:

- In an input statement, the two successive fields are read and assigned to a complex I/O list element as its 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:

- In an output statement, the two parts of a complex value are transferred 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	△2.35472 3.45673
E9.2,'△,△',E5.3	47587.222, 56.123	△0.48E+06△,△*****

8.1.16 Scale Factor

A scale factor is a value used in a format specifier which 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. It 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:

  ```
  10 FORMAT (X, I4, E6.3, 3P, 2A3, 2I2, 2F5.3, E8.5)
  ```

  In the example, the 3P applies to the 2F5.3 and the E8.5 but not to the E6.3 or 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:

  ```
  10 FORMAT (3P2F5.3, E8.5)
  ```

  In the example, 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:

  ```
  10 FORMAT (X, F3.2, E3.2, 2PE4.2, F4.2, 3PE4.2)
  ```

  In this example, suppose two records are read, 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 data by 10**-n and assignment of it 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 transfer to the external record. Thus, a positive scale factor moves the decimal point to the right; 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. Thus, a positive scale factor moves the decimal point to the right and decreases the exponent; a negative scale factor moves the decimal point to the left and increases the exponent.

- A scale factor has no effect on a G field descriptor if the magnitude of the data to be output is within the effective range of the descriptor, because the G field descriptor supplies its own scaling function. Moreover, the G field descriptor functions as an E field descriptor if the magnitude of the data value is outside its range. In this case, the scale factor has the same effect as for the E field descriptor.

Output Examples:

Format	**Internal Value**	**External Representation**
1PE12.3	-270.139	△△-2.701E+02
1PE12.2	-270.139	△△△-2.70E+02
-1PE12.2	-270.139	△△△-0.03E+04

8.1.17 Repeat Counts and Group Repeat Counts

You can apply most field descriptors (except H, T, P, or X) to a number of successive data fields by preceding that field descriptor with an unsigned nonzero integer constant specifying the number of repetitions. This 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 to data fields by enclosing these field descriptors in parentheses and preceding them with an unsigned nonzero integer constant. The integer constant is called a group repeat count. For example, the following two statements are equivalent:

```
50  FORMAT (2I8,3(F8.3,E15.7))
```

$$\texttt{50 FORMAT (I8,I8,}\underbrace{\texttt{F8.3,E15.7}}_{1}\texttt{,}\underbrace{\texttt{F8.3,E15.7}}_{2}\texttt{,}\underbrace{\texttt{F8.3,E15.7}}_{3}\texttt{)}$$

An H or X field descriptor, which could not otherwise be repeated, can be enclosed in parentheses and treated as a group repeat specification. Thus, it could be repeated a desired number of times.

If you do not specify a group repeat count, a default count of 1 is assumed.

8.1.18 Default Field Descriptors

If you write the field descriptors I, O, L, F, E, D, G, or A without specifying a w or w.d, default values are supplied based on the data type of the I/O list element. Note that for F, E, D, and G, you may not specify only w or d; you must specify w.d or nothing.

Table 8-2 lists the default values for w and d.

Table 8-2: Default Field Widths

Field Descriptor	List Element Data Type	w	d
I, O	INTEGER*2	7	
I, O	INTEGER*4	12	
L	LOGICAL	2	
F, E, G, D	REAL, COMPLEX	15	7
F, E, G, D	DOUBLE PRECISION	25	16
A	LOGICAL*1 or BYTE	1	
A	INTEGER*2		
A	LOGICAL*4,INTEGER*4	4	
A	REAL, COMPLEX	4	
A	DOUBLE PRECISION	8	

Notice that for the A field descriptor the default is the length of the corresponding I/O list element.

8.2 CARRIAGE CONTROL CHARACTERS

The first character of every record transferred to a printer is not printed. Instead, it is interpreted as a carriage control character. 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 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.

8.3 FORMAT SPECIFICATION SEPARATORS

Field descriptors in a format specification are generally separated by commas. You can also use the slash (/) record terminator to separate field descriptors. A slash terminates input or output of the current record and initiates a new record. For example:

```
        WRITE (6,40) K,L,M,N,O,P
     40 FORMAT (306/I6,2F8.4)
```

This statement is equivalent to:

```
        WRITE (6,40) K,L,M
     40 FORMAT (306)
        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:

```
        WRITE (6,99)
     99 FORMAT ('1',T51,'HEADING△LINE'//T51,'SUBHEADING△LINE'//)
```

The above statements produce the following output:

```
     Column 50, top of page
                  ↓
                  HEADING LINE
     (blank line)
                  SUBHEADING LINE
     (blank line)
     (blank line)
```

8.4 EXTERNAL FIELD SEPARATORS

A field descriptor such as Fw.d specifies that an input statement is to read w characters from the external record. If the data field in the external record contains less than w characters, the input statement would read characters from the next data field in the external record, unless you padded the short field with leading zeroes 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. This 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, F, E, D, G, and L field descriptors. For example:

```
         READ (5,100) I,J,A,B
     100 FORMAT (2I6,2F10.2)
```

The above statements read the following record:

```
     1,-2,1.0,35
```

On execution, 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 the external field for F, E, D and G field descriptions.

You can use a comma to terminate only fields 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, or 0.DO, 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 the record reaches its physical end before w characters are read, short field termination occurs; and the characters that were input are assigned successfully. Trailing spaces are appended to fill the corresponding I/O list element or the field descriptor.

8.5 RUN-TIME FORMATS

A run-time format is a format stored as Hollerith or alphanumeric data in an array. In the I/O statement referencing the format, you write the name of the array instead of a format statement label (see Section 7.2.2). Virtual arrays must not be used for this purpose.

A run-time format in an array has the same form as a FORMAT statement, without the word FORMAT and the statement label. The opening and closing parentheses are required.

Run-time formats are especially useful when you do not know exactly which field descriptors will be required until execution time. To solve this problem, you can write a program to create a format, the choice of field descriptors being made to depend on the attributes of the data. For example:

```
      REAL TABLE(10,5)
      REAL FMT(11),FBIG,FMED,FSML
      DATA FMT(1)/'('/,FMT(11)/')'/
      DATA FBIG,FMED,FSML/'F8.2','F9.4','F9.6'/
      DATA FMT(3),FMT(5),FMT(7),FMT(9)/4*','/
      DO 20 I=1,10
      DO 18 J=1,5
      FMT(2*J)=FMED
      IF (TABLE(I,J).GE.100) FMT(2*J)=FBIG
      IF (TABLE(I,J).LE.0.1) FMT(2*J)=FSMAL
   18 CONTINUE
      WRITE(6,FMT) (TABLE(I,J),J=1,5)
   20 CONTINUE
      END
```

In the example, the 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 real array FMT. A left parenthesis is stored in FMT (1), a right parenthesis is stored in FMT(11), and commas are stored in other odd-numbered elements of FMT. A selection of field descriptors is stored in the real variables FBIG, FMED, FSML. The choice of field descriptors to be assigned to even-numbered elements of FMT is made to depend on the magnitude 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 will be lost.

8.6 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, F, E, D, G, L, A, or Q field descriptor in the format, or else 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 input when a slash occurs in the format specification, or when the last closing parenthesis of the format specification is reached and I/O list elements remain to be filled. Any remaining characters in the current record are discarded when the new record is read.

On execution, 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 format specification or if the last closing parenthesis is reached and more I/O list elements remain to be transferred.

The I, O, F, E, D, G, L, A, and Q field descriptors each correspond to one element in the I/O list. No list element corresponds to an H, X, P, T, or alphanumeric literal field descriptor. In H and alphanumeric literal field descriptors, data transfer occurs directly between the external record and the storage location of the format specification.

In format control, when an I, O, F, E, D, G, L, A, or Q field descriptor is encountered, the I/O list is checked for a corresponding element. If one is found, data is transferred and, if appropriate, translated between the external record and the list element. If one is not found, format control terminates.

When the last closing parenthesis of the format specification is reached, format control determines whether more I/O list elements are to be processed. If not, 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 is the termination of the current record and the starting of a new one. 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 and continues from that point.

Examples of format reversion are:

```
          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.

```
     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.

8.7 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.

8.7.1 General Rules

1. A FORMAT statement must always be labeled.

2. In a field descriptor such as rIw or nX, the terms r, w, and n must be unsigned integer constants greater than 0. You can omit the repeat count and field width specification.

3. 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.

4. 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.

5. 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.

6. No repeat count is permitted for H, X, T, or alphanumeric literal field descriptors unless these descriptors are enclosed in parentheses and treated as a group repeat specification.

7. 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 an alphanumeric literal.

8. A format specification in an array must be constructed the same as a format specification in a FORMAT statement, including the opening and closing parentheses. The word FORMAT and the statement label only are permitted.

8.7.2 Input Rules

1. A minus sign must precede a negative value in an external input field; a plus sign is optional before a positive value.

2. 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 and must not contain a sign, a decimal point, or an exponent.

3. 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.

4. 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.

5. 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.

6. 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).

7. 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.

8.7.3 Output Rules

1. 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.

2. 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.

3. 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.

Table 8-4: Summary of FORMAT Codes

Code	Form	Effect
I	Iw	Specifies transfer of decimal integer values.
O	Ow	Specifies transfer of octal integer values.
F	Fw.d	Specifies transfer of real or double precision values in basic real form.
E	Ew.d	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	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, F, or f; on output, transfers T or F.
A	Aw	Specifies transfer of alphanumeric or Hollerith values.
H	nHc...c	Specifies transfer of alphanumeric or 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 transmitted (on output).
T	Tn	Specifies the position, in the external record, of the next character to be processed.
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.

CHAPTER 9

AUXILIARY INPUT/OUTPUT STATEMENTS

The auxiliary input/output statements perform file management functions. These statements are:

- 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, BACKSPACE, and FIND -- perform file-positioning functions.
- ENDFILE -- writes a special record that causes an end-of-file condition (and END= transfer) when an input statement reads the record.
- DEFINE FILE -- associates a FORTRAN logical unit with an unformatted, direct access file.

See Section 7.2 for a definition of the I/O statement components of these statements.

OPEN

9.1 OPEN STATEMENT

An OPEN statement either connects an existing file to a logical unit, or creates a new file and connects it to a logical unit. In addition, OPEN can specify file attributes that control file creation and subsequent processing.

The OPEN statement has the following form:

```
OPEN(par[,par]...)
```

par

A keyword specification in one of the following forms:

```
key
key = value
```

key

A keyword, as described below.

value

Depends on the keyword, as described below.

Keywords are divided into several categories based on function:

- Keywords that identify the unit and file.

UNIT	- logical unit number to be used
NAME	- file name specification for the file
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
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
RECORDSIZE	- logical record length

- 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

NOTE

Not all PDP-11 operating systems support all keywords and options. Consult the appropriate PDP-11 FORTRAN IV user's guide for information in 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'	Access method	'SEQUENTIAL'
ASSOCIATEVARIABLE	v	Next record number in direct access	No associate variable
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
FORM	'FORMATTED' 'UNFORMATTED'	Format type	Depends on ACCESS
INITIALSIZE	e	File storage allocation	No allocation
MAXREC	e	Maximum record number in direct access	No maximum
NAME	c	File name specification	Depends on unit and system
NOSPANBLOCKS	-	Records do not span blocks	Records can span blocks
READONLY	-	Write protection	No write protection

(continued on next page)

Table 9-1 (Cont): OPEN Statement Keyword Values

Keyword	Values[1]	Function	Default
RECORDSIZE	e	Record length	Depends on TYPE,
SHARED	-	File sharing allowed	File sharing not allowed
TYPE	'OLD' 'NEW 'SCRATCH' 'UNKNOWN'	File status at open	'NEW'
UNIT	e	Logical unit number	No default

1. c is an alphanumeric literal, array name, variable name, or array element name.
   e is an integer expression.
   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, TYPE='SCRATCH', ACCESS='DIRECT',
           INITIALSIZE=50, RECORDSIZE=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, NAME='MT0:MYDATA.DAT', BLOCKSIZE=8192,
           TYPE='NEW', ERR=14, RECORDSIZE=1024, RECORDTYPE='FIXED')
```

This example creates a file on magnetic tape with a large block size for efficient processing.

```
     OPEN (UNIT=I, NAME='MT0:MYDATA.DAT', READONLY, TYPE='OLD',
           RECORDSIZE=1024, RECORDTYPE='FIXED', BLOCKSIZE=8192)
```

This example opens the file created in the previous example for input.

Sections 9.1.1 through 9.1.18 describe the keywords in detail.

9.1.1 ACCESS Keyword

The ACCESS keyword specifies the method of locating, reading, or writing records. In FORTRAN IV there are two access methods: sequential and direct.

This keyword has the following form:

    ACCESS= acc

acc
: One of the alphanumeric literals, 'SEQUENTIAL', or 'APPEND'. ACCESS= 'APPEND' implies sequential access and positioning after the last record in the file.

If no ACCESS is specified, the default is 'SEQUENTIAL'.

In sequential access, you must read or write records in sequence from the beginning of the file.

Direct access to sequential file requires that the records in the file be fixed-length (see Section 9.1.15.)

In direct access, you specify record number n (Section 7.2.3) in the I/O statement, and the system selects the nth record.

9.1.2 ASSOCIATEVARIABLE Keyword

ASSOCIATEVARIABLE specifies the integer variable (asv) that, after each direct access I/O operation, contains the record number of the next sequential record in the file. This specifier is ignored for sequential access.

This keyword has the following form:

    ASSOCIATEVARIABLE = asv

asv
: An integer variable.

9.1.3 BLOCKSIZE Keyword

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. In fact, the system transfers the records to an intermediate I/O buffer.

This keyword has the following 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 magnetic tape files. The default is the system default for the device.

See the appropriate PDP-11 FORTRAN IV user's guide for more information.

9.1.4 BUFFERCOUNT Keyword

BUFFERCOUNT specifies the number of buffers to be associated with the logical unit for multibuffered I/O. The BLOCKSIZE keyword determines the size of each buffer. If you do not specify BUFFERCOUNT, or if you specify 0, the system default is used. This keyword has the following form:

```
     BUFFERCOUNT = bc
```

bc
An integer expression.

9.1.5 CARRIAGECONTROL Keyword

CARRIAGECONTROL determines the kind of carriage control to be used when printing a file. The default for formatted files is 'FORTRAN', the default for unformatted files, the default is 'NONE'. 'FORTRAN' specifies normal FORTRAN interpretation of the first character (see Section 8.2); 'LIST' specifies no FORTRAN interpretation, but rather single spacing between records; and 'NONE' specifies no implied carriage control.

This keyword has the following form:

```
     CARRIAGECONTROL = cc
```

cc
One of the alphanumeric literals 'FORTRAN', 'LIST', or 'NONE'.

9.1.6 DISPOSE Keyword

DISPOSE determines the disposition of the file connected to the unit when the unit is closed. If you specify 'SAVE' or 'KEEP', the file is retained after the unit is closed; this is the default value. If you specfiy 'PRINT', the file is submitted to the system line printer spooler. On some systems, the file is deleted after printing. If you specfiy 'DELETE', the file is deleted. A read-only file (see Section 9.1.14) cannot be printed or deleted, and a scratch file (see Section 9.1.17) cannot be saved or printed.

This keyword has the following forms:

```
     DISPOSE = dis
     DISP = dis
```

dis
One of the alphanumeric literals 'SAVE', 'KEEP', 'PRINT', or 'DELETE'.

9.1.7 ERR Keyword

ERR transfers control to the executable statement specified by s if an error occurs during execution of the OPEN statement. The ERR specification applies only to the OPEN statement, not to subsequent I/O operations on the unit. If an error does occur, no file is opened or created.

This keyword has the following form:

```
ERR= s
```

s
: The label of an executable statement.

9.1.8 EXTENDSIZE Keyword

EXTENDSIZE specifies the number of blocks by which to extend a disk file 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.

This keyword has the following form:

```
EXTENDSIZE = es
```

es
: An integer expression.

9.1.9 FORM Keyword

FORM specifies whether the file being opened is to be read and written using formatted or unformatted I/O statements. For sequential access, 'FORMATTED' is the default. For direct access, 'UNFORMATTED' is the default. You must not mix formatted and unformatted I/O statements on the same unit.

This keyword has the following form:

```
FORM = ft
```

ft
: The alphanumeric literal 'FORMATTED' or 'UNFORMATTED'.

9.1.10 INITIALSIZE Keyword

INITIALSIZE specifies the number of blocks in the initial allocation of space for a new file on a disk. If you do not specify INITIALSIZE, or if you specify 0, no initial allocation is made.

This keyword has the following form:

```
INITIALSIZE = insz
```

insz
: An integer expression.

9.1.11 MAXREC Keyword

MAXREC specifies the maximum number of records permitted in a direct access file. The default is no maximum number of records. This specifier is ignored for other types of files.

This keyword has the following form:

```
     MAXREC = mr
```

mr
: An integer expression.

9.1.12 NAME Keyword

NAME specifies the name of the file to be connected to the unit. The name can be any file specification accepted by the operating system. The appropriate PDP-11 FORTRAN IV user's guide describes default file name conventions.

If the file name is stored in a variable, array, or array element, the name must consist of ASCII characters terminated by an ASCII null character (0 byte).

This keyword has the following form:

```
     NAME = fln
```

fln
: An array name, variable name, array element name, or alphanumeric literal.

9.1.13 NOSPANBLOCKS Keyword

NOSPANBLOCKS is used for sequential files stored on disk only. It specifies that records are not to cross disk block boundaries. If any record exceeds the size of a disk block, an error occurs.

This keyword has the following form:

```
     NOSPANBLOCKS
```

9.1.14 READONLY Keyword

READONLY prohibits writing by the program to a file.

This keyword has the following form:

```
     READONLY
```

9.1.15 RECORDSIZE Keyword

RECORDSIZE specifies the logical record length.

If the file contains fixed-length records, RECORDSIZE specifies the size of each record. If the file contains variable-length records, RECORDSIZE specifies the maximum length for any record.

You must specify RECORDSIZE when you create a file with fixed-length records.

This keyword has the following form:

```
RECORDSIZE = rl
```

rl

An integer expression.

The value of rl depends on the value of FORM (see Section 9.1.9). 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).

9.1.16 SHARED Keyword

SHARED specifies that the 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.

This keyword has the following form:

```
SHARED
```

See the appropriate PDP-11 FORTRAN IV user's guide for additional information on this keyword.

9.1.17 TYPE Keyword

TYPE 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 it 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 use 'NEW', thereby creating a new file. The default is 'NEW'.

This keyword has the following form:

```
TYPE = typ
```

typ

One of the alphanumeric literals 'OLD', 'NEW', 'SCRATCH', or unknown.

9.1.18 UNIT Keyword

UNIT specifies the logical unit to which a file is to be connected. The unit specification must appear in the list. Another file cannot be connected to the logical unit when the OPEN statement is executed.

This keyword has the following form:

```
UNIT = u
```

u

An integer expression.

CLOSE

9.2 CLOSE STATEMENT

The CLOSE statement disconnects a file from a unit.

The CLOSE statement has the following form:

$$\text{CLOSE (UNIT=u}\left[,\begin{Bmatrix}\text{DISPOSE}\\ \text{DISP}\end{Bmatrix}\text{=p}\right]\text{[,ERR=s])}$$

u
: A logical unit number.

p
: An alphanumeric literal 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 the OPEN statement, except that a file opened as a scratch file cannot be saved or printed, nor can a file opened for read-only access be printed or deleted.

For example:

```
     CLOSE(UNIT=1,DISPOSE='PRINT')
```

This statement closes the file on unit 1 and submits the file for printing.

```
     CLOSE(UNIT=J,DISPOSE='DELETE',ERR=99)
```

This statement closes the file on unit J and deletes it.

REWIND

9.3 REWIND STATEMENT

The REWIND statement repositions an open sequential file at the beginning of the file.

The REWIND statement has the following form:

```
     REWIND u
```

u
: A logical unit number.

The unit number must refer to an open sequential file on disk or magnetic tape. For example:

```
     REWIND 3
```

This statement 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 access.

BACKSPACE

9.4 BACKSPACE STATEMENT

The BACKSPACE statement repositions an open sequential file at the beginning of the preceding record. When the next I/O statement for the unit is executed, that record is available for processing.

The BACKSPACE statement has the following form:

```
     BACKSPACE u
```

u
: A logical unit number.

The unit number must refer to an open sequential file on disk or magnetic tape. For example:

```
     BACKSPACE 4
```

This statement 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 or append access.

FIND

9.5 FIND STATEMENT

The FIND statement positions a direct access file on a specified unit to a particular record. No data transfer takes place.

This statement has the following form:

```
     FIND (u'r)
```

u
: A logical unit number.

r
: The direct access record number.

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.

ENDFILE

9.6 ENDFILE STATEMENT

The ENDFILE statement writes an end-file record to the specified unit. This statement has the following form:

```
ENDFILE u
```

u
: A logical unit number.

You can write an end-file record only to sequentially accessed sequential organization files containing variable-length or segmented records.

For example:

```
ENDFILE 2
```

This statement outputs an end-file record to logical unit 2.

DEFINE FILE

9.7 DEFINE FILE STATEMENT

The DEFINE FILE statement describes direct-access sequential files that are associated with a logical unit number. The OPEN statement (Section 9.1) is the preferred way to do this. The DEFINE FILE statement establishes the size and structure of the direct access file.

The DEFINE FILE statement has the following 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 asociated 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. Since 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:

```
     DEFINE FILE 3 (1000,48,U,NREC)
```

This statement 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.

APPENDIX A

CHARACTER SETS

A.1 FORTRAN CHARACTER SET

The FORTRAN character set consists of:

1. The letters A through Z and a through z
2. The numerals 0 through 9
3. The following special characters:

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

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 A-1.

A.2 ASCII CHARACTER SET

Table A-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 A-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

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

A.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 3 Radix-50 characters.

Thus, the maximum Radix-50 value is:

```
47*50*50 + 47*50 + 47 = 174777
```

A Radix-50 constant has the following 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: (the colon is not a Radix-50 character)
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 B

FORTRAN LANGUAGE SUMMARY

B.1 EXPRESSION OPERATORS

Table B-1 lists the expression operators in each data type in order of descending precedence.

Table B-1: Expression Operators

Data Type	Operator	Operation	Operates Upon:
Arithmetic	**	Exponentiation	Arithmetic expressions
	*,/	Multiplication, division	
	+,-	Addition, subtraction, unary plus and minus	
Relational	.GT.	Greater than	Arithmetic or logical expressions (all relational operators have equal priority)
	.GE.	Greater than or equal to	
	.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	
	.OR.	A.OR.B is true if either A or B or both are true	

(continued on next page)

Table B-1 (Cont.): Expression Operators

Data Type	Operator	Operation	Operates Upon:
Logical (cont.)	.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	

B.2 STATEMENTS

The following list summarizes the statements available in the PDP-11 FORTRAN IV 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.

Form	Effect	Manual Section
ACCEPT	See READ, Formatted Sequential	7.3.1
	See READ, List-Directed	7.3.3
Arithmetic/Logical Assignment		3.1, 3.2

v=e

v A variable name, or an array element name.

e An expression.

Assigns the value of the arithmetic or logical expression to the variable.

Statement Function		6.2.1

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

f A symbolic name.

p A symbolic name.

e 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.

Form		Effect	Manual Section
ASSIGN s TO v			3.3
	s	A label of an executable statement.	
	v	An integer variable name.	
		Associates the statement label s with the integer variable v for later use in an assigned GO TO statement.	
BACKSPACE u			9.4
	u	An integer expression.	
		Backspaces one record the currently open file on logical unit u.	
BLOCK DATA [nam]			
	nam	A symbolic name.	
		Specifies the subprogram that follows as a BLOCK DATA subprogram.	
CALL f[([a][,[a]]...)]			4.5 6.2
	f	A subprogram name or entry point.	
	a	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 (p[,p]...)			9.2

	p	One of the following forms:
		
		UNIT = e
		DISPOSE = 'SAVE'
		DISPOSE = 'KEEP'
		DISPOSE = 'DELETE'
		DISPOSE = 'PRINT'
		ERR = s	

	e	An integer expression.	
	s	A label of an executable statement.	
		Closes the specified file. DISPOSE can be abbreviated DISP.	

Form	Effect	Manual Section
COMMON [/[cb]/] nlist [[,]/[cb]/nlist]...		5.4
cb	A common block name.	
nlist	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/]...		5.8
nlist	A list of one or more variable names, array names, or array element names separated by commas. Subscript expressions must be constant.	
clist	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]		7.6
c	An integer expression.	
f	A format specifier.	
b	A variable name, array name, or array element name.	
s	A label of an executable statement.	
list	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)]...		9.9
u	An integer variable or integer constant.	
m	An integer variable or integer constant.	

Form	Effect	Manual Section
n	An integer variable or integer constant.	
v	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.	
DIMENSION a(d)[,a(d)]...		5.3
a(d)	An array declarator.	
	Specifies storage space requirements for arrays.	
DO s [,] v = e1,e2[,e3]		4.3
s	A label of an executable statement.	
v	A variable name.	
e1,e2,e3	Numeric expressions.	
	Executes the DO loop by performing the following steps: 1. Set v = e1 2. Execute all statements through statement number s 3. Evaluate v = v+e3 4. Repeat steps 2 through 3 for the following iterations: MAX (1, INT((e2 - e1)/e3) + 1)	
ENCODE (c,f,b[,ERR=s])[list]		7.6
c	An integer expression.	
f	A format specifier.	
b	A variable name, array name, or array element name.	
s	A label of an executable statement.	
list	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.	

Form		Effect	Manual Section
END			4.9
		Delimits a program unit.	
ENDFILE u			9.8
	u	An integer expression.	
		Writes an end-file record on logical unit u.	
END=s,ERR=s			7.2.5
	s	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.	
EQUIVALENCE (nlist)[,(nlist)]...			5.6
	nlist	A list of two or more variable names, array names, or array element names, separated by commas. Subscript expressions must be constants.	
		Assigns each of the names in nlist the same storage location.	
EXTERNAL v[,v]...			5.7
	v	A subprogram name.	
		Defines the names specified as subprograms.	
FIND (u'r)			9.5
	u	An integer expression.	
	r	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.	

Form	Effect	Manual Section
[typ] FUNCTION nam[*n][([p[,p]...])]		6.2.2
typ	A data type specifier.	
nam	A symbolic name.	
*n	A data type length specifier.	
p	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	A label of an executable statement.	
	Transfers control to statement number s.	
GO TO (slist)[,] e		4.1.2
slist	A list of one or more statement labels separated by commas.	
e	An integer expression.	
	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, etc.). 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	An integer variable name.	
slist	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	An expression.	

Form	Effect	Manual Section
s	A label of an executable statement.	
	Transfers control to statement s1 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	An expression.	
st	Any executable statement except a DO or logical IF.	
	Executes the statement if the logical expression has a value of true.	
IMPLICIT typ (a[,a]...)[,typ(a[,a]...)]...		5.1
typ	A data type specifier.	
a	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.	
OPEN(par[,par]...)		9.1
par	A keyword specification in one of the following forms: key key = value	
key	A keyword, as described below.	
value	Depends on the keyword, as described below.	

Keyword	Values
ACCESS	'SEQUENTIAL' 'DIRECT' 'APPEND'
ASSOCIATEVARIABLE	v
BLOCKSIZE	e
BUFFERCOUNT	e
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'
DISPOSE DISP	'SAVE' or 'KEEP' 'PRINT' 'DELETE'

Form	Effect	Manual Section

Keyword	Values
ERR	s
EXTENDSIZE	e
FORM	'FORMATTED'
	'UNFORMATTED'
INITIALSIZE	e
MAXREC	e
NAME	c
NOSPANBLOCKS	-
READONLY	-
RECORDSIZE	e
SHARED	-
TYPE	'OLD'
	'NEW'
	'SCRATCH'
	'UNKNOWN'
UNIT	e

c	An array name, variable name, array element name, or alphanumeric literal.	
e	A numeric expression.	
s	A label of an executable statement.	
v	An integer variable name.	
	Opens a file on the specified logical unit according to the parameters specified by the keywords.	
PAUSE [disp]		4.7
disp	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, Formatted Sequential.	7.3.2
	See WRITE, List-Directed.	7.3.4
PROGRAM nam		5.10
nam	A symbolic name.	
	Specifies a name for the main program.	
READ (u,f[,END=s][,ERR=s])[list]		7.3.1
READ f[,list]		
ACCEPT f[,list]		
u	An integer expression.	
f	A format specifier.	

Form	Effect	Manual Section
s	A label of an executable statement.	
list	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(u[,END=s][,ERR=s])[list]		7.3.5
u	An integer expression.	
s	A label of an executable statement.	
list	An I/O list.	
	Reads one unformatted record from logical unit u, and assigns values to the elements in the list.	
READ(u'r[,ERR=s])[list]		7.4.3
u	An integer expression.	
r	An integer expression.	
s	A label of an executable statement.	
list	An I/O list.	
	Reads record r from logical unit u and assigns values to the elements in the list.	
READ (u,*[,END=s][,ERR=s])list		7.3.3
READ *,list		
ACCEPT *,list		
u	An integer expression.	
*	Denotes list-directed formatting.	
s	A label of an executable statement.	
list	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.	

Form		Effect	Manual Section
RETURN			4.6
		Returns control to the calling program from the current subprogram.	
REWIND u			9.3
	u	An integer expression.	
		Repositions logical unit u to the beginning of the currently opened file.	
STOP [disp]			
	disp	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.	
SUBROUTINE nam[([p[,p]...])]			6.2.3
	nam	A symbolic name.	
	p	A symbolic name.	
		Begins a subroutine subprogram, indicating the program name and any dummy argument names (p).	
TYPE		See WRITE, Formatted Sequential. See WRITE, List-Directed.	7.3.2 7.3.4
Type Declaration			5.2
typ v[,v]...			
	typ	One of the following data types: BYTE LOGICAL LOGICAL*1 LOGICAL*4 INTEGER INTEGER*2 INTEGER*4 REAL REAL*4 REAL*8 DOUBLE PRECISION COMPLEX COMPLEX*8	

Form	Effect	Manual Section
v	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).	
	The symbolic names (v) are assigned the specified data type.	
VIRTUAL a(d)[,a(d)]...		5.5
a(d)	An array declarator.	
	Specifies storage space for arrays outside normal program address space.	
WRITE (u,f[,ERR=s])[list]		7.3.2
PRINT f[,list]		
TYPE f[,list]		
u	An integer expression.	
f	A format specifier.	
s	A label of an executable statement.	
list	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 (u[,ERR=s])[list]		7.3.6
u	An integer expression.	
s	A label of an executable statement label.	
list	An I/O list.	
	Writes one unformatted record to logical unit u containing the values of the elements in the list.	
WRITE (u'r[,ERR=s]) [list]		7.4.4
u	An integer expression.	
r	An integer expression.	
s	A label of an executable statement label.	
list	An I/O list.	
	Writes record r to logical unit u containing the values of the elements in the list.	

Form	Effect	Manual Section
WRITE(u,*[,ERR=s])list		7.3.4
PRINT *,list		
TYPE *,list		

u	An integer expression.
*	Denotes list-directed formatting.
s	A label of an executable statement.
list	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.

B.3 LIBRARY FUNCTIONS

Table B-2 lists the FORTRAN IV library functions. Superscripts in the table refer to notes which follow the table.

Table B-2: FORTRAN Library Functions

Form	Definition	Argument Type	Result Type
ABS(X)	Real absolute value	Real	Real
IABS(I)	Integer absolute value	Integer	Integer
DABS(X)	Double precision absolute value	Double	Double
CABS(Z)	Complex to Real, absolute value[1]	Complex	Real
FLOAT(I)	Integer to Real conversion[2]	Integer	Real
IFIX(X)	Real to Integer conversion[2]	Real	Integer
SNGL(X)	Double to Real conversion[2]	Double	Real
DBLE(X)	Real to Double conversion[2]	Real	Double
REAL(Z)	Complex to Real conversion, obtain real part	Complex	Real
AIMAG(Z)	Complex to Real conversion, obtain imaginary part	Complex	Real
CMPLX(X,Y)	Real to Complex conversion CMPLX(X,Y)=X+i*Y	Real	Complex
	Truncation functions return the sign of the argument * largest integer $\leq$ |arg|		
AINT(X)	Real to Real truncation[3]	Real	Real
INT(X)	Real to Integer truncation[3]	Real	Integer
IDINT(X)	Double to Integer truncation[3]	Double	Integer
	Remainder functions return the remainder when the first argument is divided by the second.		

(continued on next page)

Table B-2 (Cont.): FORTRAN Library Functions

Form	Definition	Argument Type	Result Type
AMOD(X,Y)	Real remainder	Real	Real
MOD(I,J)	Integer remainder	Integer	Integer
DMOD(X,Y)	Double precision remainder	Double	Double
	Maximum value functions return the largest value from among the argument list; 2 or more arguments.		
AMAX0(I,J,...)	Real maximum from Integer list	Integer	Real
AMAX1(X,Y,...)	Real maximum from Real list	Real	Real
MAX0(I,J,...)	Integer maximum from Integer list	Integer	Integer
MAX1(X,Y,...)	Integer maximum from Real list	Real	Integer
DMAX1(X,Y,...)	Double maximum from Double list	Double	Double
	Minimum value functions return the smallest value from among the argument list; 2 or more arguments.		
AMIN0(I,J,...)	Real minimum of Integer list	Integer	Real
AMIN1(X,Y,...)	Real minimum of Real list	Real	Real
MIN0(I,J,...)	Integer minimum of Integer list	Integer	Integer
MIN1(X,Y,...)	Integer minimum of Real list	Real	Integer
DMIN1(X,Y,...)	Double minimum of Double list	Double	Double
	The transfer of sign functions return (sign of the second argument) * (absolute value of first argument).		
SIGN(X,Y)	Real transfer of sign	Real	Real
ISIGN(I,J)	Integer transfer of sign	Integer	Integer
DSIGN(X,Y)	Double precision transfer of sign	Double	Double
	Positive difference functions return the first argument minus the minimum of the two arguments.		
DIM(X,Y)	Real positive difference	Real	Real
IDIM(I,J)	Integer positive difference	Integer	Integer
	Exponential functions return the value of e raised to the argument power,		
EXP(X)	e^X	Real	Real
DEXP(X)	e^X	Double	Double
CEXP(Z)	e^Z	Complex	Complex
ALOG(X)	$\log_e(X)$	Real	Real
ALOG10(X)	$\log_{10}(X)$	Real	Real
DLOG(X)	$\log_e(X)$	Double	Double
DLOG10(X)	$\log_{10}(X)$	Double	Double
CLOG(Z)[4]	$\log_e(Z)$	Complex	Complex
SQRT(X)	Square root of Real argument	Real	Real
DSQRT(X)	Square root of Double precision argument	Double	Double
CSQRT(Z)[5]	Square root of Complex argument	Complex	Complex

(continued on next page)

Table B-2 (Cont.): FORTRAN Library Functions

Form	Definition	Argument Type	Result Type
SIN(X)	Real sine	Real	Real
DSIN(X)	Double precision sine	Double	Double
CSIN(Z)[6]	Complex sine	Complex	Complex
COS(X)	Real cosine	Real	Real
DCOS(X)	Double precision cosine	Double	Double
CCOS(Z)[6]	Complex cosine	Complex	Complex
TANH(X)	Hyperbolic tangent	Real	Real
ATAN(X)	Real arc tangent	Real	Real
DATAN(X)	Double precision arc tangent	Double	Double
ATAN2(X,Y)	Real arc tangent of (X/Y)	Real	Real
DATAN2(X,Y)[7,8]	Double precision arc tangent of (X/Y)	Double	Double
CONJG(Z)	Complex conjugate, if Z=X+i*Y CONJG(Z)=X-i*Y	Complex	Complex
RAN(I,J)[9]	Returns a pseudo-random number of uniform distribution over the range of 0 to 1.	Integer	Real

1. The absolute value of a complex number, (X,Y), is the real value:

$(X^2+Y^2)^{1/2}$

2. 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.

3. [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.

4. The argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than 0. The argument of CLOG must not be (0.,0.).

5. 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.

6. The argument of SIN, DSIN, COS, DCOS, TAN, and DTAN must be in radians. The argument is treated modulo 2*pi.

7. The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, and DATAN2 is in radians.

8. 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.

9. 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

INDEX

INDEX

INDEX

PDP-11 FORTRAN IV
Language Reference Manual
AA-R953A-TK

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ____________________

Name ____________________ Date ____________________

Organization ____________________

Street ____________________

City ____________________ State __________ Zip Code or Country __________

— — — Do Not Tear - Fold Here and Tape — — — — — — — — — — — — — — — — — — —

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

— — — Do Not Tear - Fold Here —

Cut Along Dotted Line