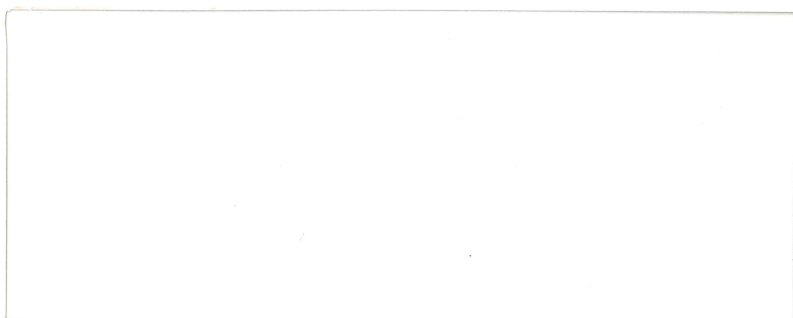


**PDP-11 FORTRAN IV  
Language Reference  
Manual**

digital  
software





# **PDP-11 FORTRAN IV Language Reference Manual**

Order Number: AA-JQ67A-TC

## **Abstract**

This document describes the FORTRAN IV language as implemented for the PDP-11 computer systems.

**Revision/Update Information:** This manual contains information concerning FORTRAN IV as of December, 1986. It supersedes the *PDP-11 FORTRAN IV Language Reference Manual*, AA-R953A-TK.

**Software Version:** FORTRAN IV V2.6, V2.7, and V2.8

**Operating System and Version:** RSTS/E, RSX-11M, RSX-11M-PLUS, IAS (V2.6), VAX-11 RSX (V2.7), RT-11 (V2.8)

**digital equipment corporation maynard, massachusetts**

---

**December, 1986**

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

---

Copyright ©December, 1986 by Digital Equipment Corporation


All Rights Reserved.

Printed in U.S.A.

---

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.

The following are trademarks of Digital Equipment Corporation:

DEC	EduSystem	RSTS
DEC/CMS	FMS-11	RSX
DEC/MMS	FMS-11/RSX	RT-11
DECnet	IAS	UNIBUS
DECsystem-10	MASSBUS	VAX
DECsystem-20	MicroPDP-11	VAXcluster
DECUS	Micro/RSX	VMS
DECtape	PDP	VT
DECwriter	PDT	
DIBOL	Professional	
DIGITAL	RMS	

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by T<sub>E</sub>X, the typesetting system developed by Donald E. Knuth at Stanford University. T<sub>E</sub>X is a trademark of the American Mathematical Society.



---

# Contents

---

## PREFACE

xiii

---

## CHAPTER 1 INTRODUCTION TO PDP-11 FORTRAN IV

1-1

---

### 1.1 LANGUAGE OVERVIEW

1-1

---

### 1.2 ELEMENTS OF FORTRAN PROGRAMS

1-2

#### 1.2.1 Statements

1-2

#### 1.2.2 Comments

1-2

#### 1.2.3 FORTRAN Character Set

1-3

---

### 1.3 FORMATTING A FORTRAN LINE

1-3

#### 1.3.1 Character-per-Column Formatting

1-4

#### 1.3.2 Tab-Character Formatting

1-5

#### 1.3.3 Statement Label Field

1-6

##### 1.3.3.1 Comment Indicator • 1-6

##### 1.3.3.2 Debugging Statement Indicator • 1-6

#### 1.3.4 Continuation Field

1-6

#### 1.3.5 Statement Field

1-6

#### 1.3.6 Sequence Number Field

1-7

---

### 1.4 PROGRAM UNIT STRUCTURE

1-7

---

## CHAPTER 2 FORTRAN STATEMENT COMPONENTS

2-1

---

### 2.1 SYMBOLIC NAMES

2-2

---

### 2.2 DATA TYPES

2-3

---

### 2.3 CONSTANTS

2-4

#### 2.3.1 Integer Constants

2-4

##### 2.3.1.1 Decimal Integer Constants • 2-4

##### 2.3.1.2 Octal Integer Constants • 2-5

#### 2.3.2 Real Constants

2-5

##### 2.3.2.1 Base • 2-5

##### 2.3.2.2 Exponent Specifier • 2-6

##### 2.3.2.3 Examples • 2-6

## Contents

2.3.3	Double Precision Constants	2-6
2.3.3.1	Base • 2-6	
2.3.3.2	Exponent Specifier • 2-7	
2.3.3.3	Examples • 2-7	
2.3.4	Complex Constants	2-7
2.3.5	Logical Constants	2-8
2.3.6	Hollerith Constants	2-8
2.3.6.1	Literal Strings • 2-8	
2.3.6.2	Data Type Rules for Hollerith Constants • 2-9	
<hr/>		
2.4	VARIABLES	2-10
2.4.1	Data Type Specification	2-11
2.4.2	Data Type by Implication	2-11
<hr/>		
2.5	ARRAYS	2-11
2.5.1	Array Declarators	2-12
2.5.2	Subscripts	2-13
2.5.3	Array Storage	2-13
2.5.4	Data Type of an Array	2-13
2.5.5	Array References Without Subscripts	2-14
2.5.6	Adjustable Arrays	2-15
<hr/>		
2.6	EXPRESSIONS	2-15
2.6.1	Arithmetic Expressions	2-15
2.6.1.1	Use of Parentheses • 2-17	
2.6.1.2	Data Type of an Arithmetic Expression • 2-17	
2.6.2	Relational Expressions	2-19
2.6.3	Logical Expressions	2-20
<hr/>		
CHAPTER 3 ASSIGNMENT STATEMENTS		3-1
<hr/>		
3.1	ARITHMETIC ASSIGNMENT STATEMENT	3-1
<hr/>		
3.2	LOGICAL ASSIGNMENT STATEMENT	3-3
<hr/>		
3.3	ASSIGN STATEMENT	3-3



---

**CHAPTER 4 CONTROL STATEMENTS** 4-1


---

<b>4.1</b>	<b>GO TO STATEMENTS</b>	<b>4-1</b>
4.1.1	Unconditional GO TO Statement	4-2
4.1.2	Computed GO TO Statement	4-2
4.1.3	Assigned GO TO Statement	4-3
<b>4.2</b>	<b>IF STATEMENTS</b>	<b>4-3</b>
4.2.1	Arithmetic IF Statement	4-4
4.2.2	Logical IF Statement	4-4
<b>4.3</b>	<b>DO STATEMENT</b>	<b>4-5</b>
4.3.1	DO Iteration Control	4-6
4.3.1.1	DO Loop Termination by Transfer of Control •	4-6
4.3.1.2	DO Loop Termination by Completion •	4-6
4.3.1.3	DO Loop Control Variable and Parameters •	4-6
4.3.2	Nested DO Loops	4-7
4.3.3	Control Transfers in DO Loops	4-8
4.3.4	Extended Range	4-8
<b>4.4</b>	<b>CONTINUE STATEMENT</b>	<b>4-9</b>
<b>4.5</b>	<b>CALL STATEMENT</b>	<b>4-9</b>
<b>4.6</b>	<b>RETURN STATEMENT</b>	<b>4-10</b>
<b>4.7</b>	<b>PAUSE STATEMENT</b>	<b>4-10</b>
<b>4.8</b>	<b>STOP STATEMENT</b>	<b>4-11</b>
<b>4.9</b>	<b>END STATEMENT</b>	<b>4-11</b>

---

**CHAPTER 5 SPECIFICATION STATEMENTS** 5-1


---

<b>5.1</b>	<b>IMPLICIT STATEMENT</b>	<b>5-1</b>
<b>5.2</b>	<b>TYPE DECLARATION STATEMENTS</b>	<b>5-2</b>
<b>5.3</b>	<b>DIMENSION STATEMENT</b>	<b>5-3</b>

## Contents

5.4	COMMON STATEMENT	5-4
5.5	VIRTUAL STATEMENT	5-5
5.5.1	Restrictions on the Use of Virtual Arrays	5-6
5.5.2	Virtual Array References in Subprograms	5-7
5.6	EQUIVALENCE STATEMENT	5-8
5.6.1	Making Arrays Equivalent	5-9
5.6.2	Extending Common Blocks	5-10
5.7	EXTERNAL STATEMENT	5-10
5.8	DATA STATEMENT	5-11
5.9	PROGRAM STATEMENT	5-13
5.10	BLOCK DATA STATEMENT	5-13
<b>CHAPTER 6 SUBPROGRAMS</b>		<b>6-1</b>
6.1	SUBPROGRAM ARGUMENTS	6-1
6.1.1	Rules Governing Subprogram Arguments	6-2
6.1.2	Adjustable Arrays	6-3
6.2	USER-WRITTEN SUBPROGRAMS	6-4
6.2.1	Statement Functions	6-5
6.2.2	Function Subprograms	6-6
6.2.3	Subroutine Subprograms	6-8
6.3	FORTTRAN LIBRARY FUNCTIONS	6-9



**CHAPTER 7 INPUT/OUTPUT STATEMENTS**

7-1

<b>7.1</b>	<b>I/O OVERVIEW</b>	<b>7-2</b>
7.1.1	Records	7-2
7.1.2	Files	7-2
7.1.3	Access Modes	7-2
7.1.3.1	Sequential Access • 7-2	
7.1.3.2	Direct Access • 7-3	
<b>7.2</b>	<b>I/O STATEMENT COMPONENTS</b>	<b>7-3</b>
7.2.1	Logical Unit Numbers	7-3
7.2.2	Format Specifiers	7-3
7.2.3	Direct Access Record Numbers	7-3
7.2.4	End-of-File Condition and Error Condition Parameters	7-4
7.2.5	I/O Lists	7-5
7.2.5.1	Simple Lists • 7-5	
7.2.5.2	Implied DO Lists • 7-6	
<b>7.3</b>	<b>SEQUENTIAL I/O</b>	<b>7-7</b>
7.3.1	Formatted Sequential Input Statements - READ, ACCEPT	7-7
7.3.2	Formatted Sequential Output Statements - WRITE, TYPE, PRINT	7-8
7.3.3	List-Directed Input Statements - READ, ACCEPT	7-10
7.3.3.1	Data Elements • 7-10	
7.3.3.2	Null Elements • 7-11	
7.3.3.3	Element Repeaters • 7-11	
7.3.3.4	Delimiters • 7-11	
7.3.3.5	Example of a List-Directed Input Statement • 7-12	
7.3.4	List-Directed Output Statements - WRITE, TYPE, PRINT	7-12
7.3.5	Unformatted Sequential Input Statement - READ	7-14
7.3.6	Unformatted Sequential Output Statements - WRITE	7-14
<b>7.4</b>	<b>DIRECT ACCESS I/O</b>	<b>7-15</b>
7.4.1	Unformatted Direct Access Input Statement - READ	7-15
7.4.2	Unformatted Direct Access Output Statement - WRITE	7-16
<b>7.5</b>	<b>ENCODE AND DECODE STATEMENTS</b>	<b>7-16</b>

---

**CHAPTER 8 FORMAT STATEMENTS**

**8-1**

<b>8.1</b>	<b>FIELD DESCRIPTORS</b>	<b>8-2</b>
8.1.1	I Field Descriptor _____	8-2
8.1.2	O Field Descriptor _____	8-3
8.1.3	F Field Descriptor _____	8-4
8.1.4	E Field Descriptor _____	8-6
8.1.5	D Field Descriptor _____	8-7
8.1.6	G Field Descriptor _____	8-8
8.1.7	L Field Descriptor _____	8-9
8.1.8	A Field Descriptor _____	8-10
8.1.9	H Field Descriptor _____	8-11
8.1.10	X Field Descriptor _____	8-12
8.1.11	T Field Descriptor _____	8-13
8.1.12	Q Field Descriptor _____	8-13
8.1.13	Dollar Sign Descriptor _____	8-14
8.1.14	Colon Descriptor _____	8-14
8.1.15	Complex Data Editing _____	8-15
8.1.16	Scale Factor _____	8-15
8.1.17	Repeat Counts and Group Repeat Counts _____	8-17
8.1.18	Default Field Descriptors _____	8-17
<b>8.2</b>	<b>CARRIAGE CONTROL CHARACTERS</b>	<b>8-18</b>
8.2.1	Positioning Characters _____	8-18
8.2.2	Output to Other Devices _____	8-19
<b>8.3</b>	<b>FORMAT SPECIFICATION SEPARATORS</b>	<b>8-19</b>
<b>8.4</b>	<b>EXTERNAL FIELD SEPARATORS</b>	<b>8-20</b>
<b>8.5</b>	<b>RUNTIME FORMATS</b>	<b>8-21</b>
8.5.1	Input Using the H Field Descriptor _____	8-21
<b>8.6</b>	<b>FORMAT CONTROL INTERACTION WITH I/O LISTS</b>	<b>8-22</b>
<b>8.7</b>	<b>SUMMARY OF RULES FOR FORMAT STATEMENTS</b>	<b>8-23</b>
8.7.1	General Rules _____	8-23
8.7.2	Input Rules _____	8-24
8.7.3	Output Rules _____	8-24



---

**CHAPTER 9 AUXILIARY INPUT/OUTPUT STATEMENTS** 9-1


---

<b>9.1</b>	<b>OPEN STATEMENT</b>	<b>9-1</b>
9.1.1	ACCESS Keyword	9-4
9.1.2	ASSOCIATEVARIABLE Keyword	9-5
9.1.3	BLOCKSIZE Keyword	9-5
9.1.4	BUFFERCOUNT Keyword	9-6
9.1.5	CARRIAGECONTROL Keyword	9-6
9.1.6	DISPOSE Keyword	9-6
9.1.7	ERR Keyword	9-7
9.1.8	EXTENDSIZE Keyword	9-7
9.1.9	FORM Keyword	9-7
9.1.10	INITIALSIZE Keyword	9-7
9.1.11	MAXREC Keyword	9-8
9.1.12	NAME Keyword	9-8
9.1.13	NOSPANBLOCKS Keyword	9-8
9.1.14	READONLY Keyword	9-8
9.1.15	RECORDSIZE Keyword	9-9
9.1.16	SHARED Keyword	9-9
9.1.17	TYPE Keyword	9-9
9.1.18	UNIT Keyword	9-10
<b>9.2</b>	<b>CLOSE STATEMENT</b>	<b>9-10</b>
<b>9.3</b>	<b>REWIND STATEMENT</b>	<b>9-11</b>
<b>9.4</b>	<b>BACKSPACE STATEMENT</b>	<b>9-11</b>
<b>9.5</b>	<b>FIND STATEMENT</b>	<b>9-12</b>
<b>9.6</b>	<b>ENDFILE STATEMENT</b>	<b>9-12</b>
<b>9.7</b>	<b>DEFINE FILE STATEMENT</b>	<b>9-12</b>

---

## Contents

---

### APPENDIX A CHARACTER SETS A-1

A.1	FORTTRAN CHARACTER SET	A-1
A.2	ASCII CHARACTER SETS	A-1
A.2.1	Hexadecimal	A-1
A.2.2	Octal	A-3
A.3	RADIX-50 CONSTANTS AND CHARACTER SET	A-4

---

### APPENDIX B FORTRAN LANGUAGE SUMMARY B-1

B.1	EXPRESSION OPERATORS	B-1
B.2	STATEMENTS	B-1
B.3	LIBRARY FUNCTIONS	B-10

---

### FIGURES

1-1	FORTTRAN Coding Form	1-4
1-2	Line Formatting Example	1-5
1-3	Required Order of Statements and Lines	1-7
2-1	Array Storage	2-15
4-1	Nested DO Loops	4-8
4-2	Extended Range Control Transfers	4-9

---

### TABLES

2-1	Entities Identified by Symbolic Names	2-2
2-2	Data Type Storage Requirements	2-3
2-3	Result Data Type for Exponentiation	2-16
3-1	Conversion Rules for Assignment Statements	3-2
5-1	Equivalence of Array Storage	5-9
6-1	Types of User-Written Subprograms	6-4
7-1	List-Directed Output Formats	7-13
8-1	Effect of Data Magnitude on G Formats	8-8
8-2	Default Field Widths	8-18
8-3	Carriage Control Character Translation	8-19
8-5	Summary of FORMAT Codes	8-25

9-1	OPEN Statement Keyword Values	9-3
A-1	ASCII Character Set	A-2
A-2	7-Bit ASCII Code	A-3
B-1	Expression Operators	B-1
B-2	Summary of FORTRAN IV Statements	B-2
B-3	FORTRAN Library Functions	B-10



---

## Preface

The *PDP-11 FORTRAN IV Language Reference Manual* is designed as a reference, rather than tutorial, document. It describes the elements of FORTRAN IV common to 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.

---

## 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 I/O.
- 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.

---

### Conventions Used in This Document

The following conventions are used in this manual:

- PDP-11 FORTRAN IV is referred to as FORTRAN throughout 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 Δ

---

### Intended Audience

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

---

### Associated Documents

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

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

# 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:

$\left\{ \begin{array}{l} \text{ACCEPT} \\ \text{TYPE} \\ \text{PRINT} \end{array} \right\}$  Device-oriented I/O

$\left\{ \begin{array}{l} \text{READ (u'r)} \\ \text{WRITE (u'r)} \\ \text{FIND (u'r)} \end{array} \right\}$  Unformatted direct-access I/O

$\left\{ \begin{array}{l} \text{OPEN} \\ \text{CLOSE} \\ \text{DEFINE FILE} \end{array} \right\}$  File control and attribute specification

$\left\{ \begin{array}{l} \text{ENCODE} \\ \text{DECODE} \end{array} \right\}$  Formatted data conversion in memory

- The specifications END=s and/or ERR=s can be included in READ or WRITE statements to transfer control to the statement specified by s when an end-of-file or error condition occurs.
- Literal strings (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, TYPE, and PRINT statements.
- The DO statement increment parameter can have a negative value.
- For readability, you have the option to use a comma following the label in a DO statement.
- A PROGRAM statement can be used in a main program.



- You can include a trailing comment on the same line as a 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.

---

## 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
Δ 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 literal string (see Appendix A for a list of printable characters).

Except in Hollerith constants and literal strings, 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 Type one character per column (character-per-column)
- 2 Use the tab character (tab-character) to get from field to field

# INTRODUCTION TO PDP-11 FORTRAN IV

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.

### Figure 1-1 FORTRAN Coding Form

FORTRAN		CODER	DATE	PAGE
CODING FORM		PROBLEM		
C CONTINUED		FORTRAN STATEMENT		IDENTIFICATION
1	2 3 4 5	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80	
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=1/A		
		K=1/J		
		B=A-L		
		IF (B) 3, 10, 5		
5		IF (J.LT.SQRT (FLOAT (I))) GO TO 4		
		TYPE 105, I		
10		CONTINUE		
105		FORMAT (I4, ' IS PRIME:')		
		END		
1 2 3 4 5		7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80	
PG-3				
DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS				

ZK-613-82

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 the letter C, press the space bar five times, and begin typing the comment.



### 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 zeros are ignored. An all-zero statement label is invalid.

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

You can use two special indicators in the first column of 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 the letter 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 zero, after the first tab character. 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 zero, 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 literal strings), you can space the text any way you like for maximum legibility. The use of tab characters for spacing is discussed in Section 1.3.2.

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.

Figure 1-3 Required Order of Statements and Lines

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

BU-2717





# 2

## FORTRAN STATEMENT COMPONENTS

---

The basic components of FORTRAN statements follow.

- 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 identifies an entity 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. (Function references are not included here. They are described in Chapter 6.) Sections 2.1 on symbolic names and 2.2 on data types 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 "Data Type?" 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	Data Type?
Variables	Yes
Arrays	Yes
Statement functions	Yes
Processor-defined functions	Yes
Function subprograms	Yes
Subroutine programs	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 follow:

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. These entities follow.

- 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) assumes 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 <sup>2</sup>
LOGICAL	4
LOGICAL * 1	1 <sup>2</sup>
LOGICAL * 4	4
INTEGER	2 or 4 <sup>1</sup>

<sup>1</sup> Either two or four bytes are allocated, depending on the compiler command qualifier specified. The default allocation is two bytes. Regardless of the number of bytes allocated, only two bytes are used for computation, arithmetic assignments, and operations of any other sort, including I/O.

<sup>2</sup> The 1-byte storage area can contain the logical values true or false, a single character, or integers in the range -128 to +127. `BYTE` and `LOGICAL * 1` are synonymous.

**Table 2-2 (Cont.) Data Type Storage Requirements**

<b>Data Type</b>	<b>Storage Requirements (Bytes)</b>
INTEGER*2	2
INTEGER*4	4 <sup>3</sup>
REAL	4
REAL*4	4
REAL*8	8
DOUBLE PRECISION	8
COMPLEX	8
COMPLEX*8	8

<sup>3</sup>Four bytes are allocated, but only two bytes are available for FORTRAN IV computations, arithmetic assignments, and operations of any other sort, including I/O.

## 2.3 CONSTANTS

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

Hollerith constants and literal strings 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. Integer constants can be specified in either decimal or octal form.

#### 2.3.1.1 Decimal Integer Constants

A decimal integer constant has the form:

snn

where:

s = An optional sign

nn = A string of numeric characters

Leading zeros, 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 follow.

Valid	Invalid/Explanation
0	9999999999/Too large
-127	3.14/Decimal point not allowed
+32123	32,767/Comma not allowed

## 2.3.1.2 Octal Integer Constants

Octal integer constants have the form:

"nn

where:

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 follow.

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 interpreted as a real number, that is, a number with whole and fractional parts. Occupying four bytes of storage space, it is typically accurate to seven digits. The absolute value of a non-zero real constant must fall between (approximately)  $0.29 \times (10 \times \times -38)$  and  $1.7 \times (10 \times \times 38)$ .

A FORTRAN IV real constant is expressed as a base followed by an optional exponent specifier, as described in Section 2.3.2.1.

### 2.3.2.1 Base

The base of a real constant has four general forms:

[s]nn.  
[s]nn.nn  
[s].nn  
[s]nn

where:

s = A sign

nn = A series of one or more decimal digits

The latter form, containing no decimal point, must be followed by an exponent specifier.

For nonzero bases, if s is a minus sign (-), a negative value results; if s is a plus sign (+), or if s is omitted, a positive value results.

## FORTRAN STATEMENT COMPONENTS

Although the number of digits in a base is not limited, only the leftmost seven digits, excluding leading zeros, are significant. For example, in the real constant 0.00001234567, only the nonzero digits are significant.

### 2.3.2.2 Exponent Specifier

The exponent specifier of a real constant has the form:

$E[s]nn$

where:

$s$  = A sign

$nn$  = A series of one or more decimal digits

When present, the exponent specifier indicates a power of 10 by which the base should be multiplied. If  $s$  is a minus sign (-), a negative power results; if  $s$  is a plus sign (+), or if  $s$  is omitted, a positive power results.

For example, the real constant 1.0E6 is evaluated by multiplying the base 1.0 by  $10^{**6}$ ; the real constant 2.3E-11 is evaluated by multiplying the base 2.3 by  $10^{*(-11)}$ .

### 2.3.2.3 Examples

Examples of valid and invalid real constants follow.

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, like a real constant, is interpreted as a real number. However, it occupies eight rather than four bytes of storage space, and is typically accurate to sixteen rather than seven digits. The absolute value of a nonzero double precision constant must fall between (approximately) 0.29D-38 ( $0.29 * (10^{**-38})$ ) and 1.7D38 ( $1.7 * (10^{**38})$ ).

A FORTRAN IV double precision constant is expressed as a base followed by an exponent specifier, as described in Section 2.3.3.1.

### 2.3.3.1 Base

The base of a double precision constant has four general forms:

$[s]nn.$   
 $[s]nn.nn$   
 $[s].nn$   
 $[s]nn$

where:

$s$  = A sign

$nn$  = A series of one or more decimal digits

For nonzero bases, if *s* is a minus sign (-), a negative value results; if *s* is a plus sign (+), or if *s* is omitted, a positive value results.

Although the number of digits in *nn* is not limited, only the leftmost sixteen digits, excluding leading zeros, are significant.

## 2.3.3.2 Exponent Specifier

A double precision constant must contain an exponent specifier, of the form:

*D*[*s*]*nn*

where:

*s* = A sign

*nn* = A series of one or more decimal digits

The exponent specifier indicates a power of 10 by which the base should be multiplied. If *s* is a minus sign (-), a negative power results; if *s* is a plus sign (+), or if *s* is omitted, a positive power results.

## 2.3.3.3 Examples

Examples of valid and invalid double precision constants follow.

Valid	Invalid/Explanation
1234567890D+5	1234567890D45/Too large
+2.71828182846182D00	1234567890.0D-89/Too small
-72.5D-15	+2.7182812846182/No Snn 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 form:

(*OK*,*OK*)

where:

*OK* = 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.



## FORTRAN STATEMENT COMPONENTS

Examples of valid and invalid complex constants follow.

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 form:

$$nHc_1c_2c_3\dots c_n$$

where:

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

$c_i$  = A printable character

The maximum number of characters is 255.

Examples of valid and invalid Hollerith constants follow.

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

#### 2.3.6.1 Literal Strings

A literal string is a string of printable ASCII characters enclosed by apostrophes that represents an alternate form of a Hollerith constant. The form is:

$$'c_1c_2c_3\dots c_n'$$

where:

$c_i$  = A printable character

Both delimiting apostrophes must be present.

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

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

Examples of valid and invalid literals follow.

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:

- As the following example shows, 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. Note that Y in the example below refers to an array (not a function call).

Statement	Data Type of Constant	Length of Constant
INTEGER*2		
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. In the following example, Y is an array.

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 (9HABCDEFGH)	none	9

## FORTTRAN STATEMENT COMPONENTS

- 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 to be 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 the letters 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.

Unlike simple variables, arrays are not allocated without explicit declaration.

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

An element of an array is considered defined if the storage 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 form:

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

where:

`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. The elements in each dimension are numbered with ascending integers starting from 1. In the example above, each dimension of IUNIT consists of 10 elements, each numbered from 1 to 10.

The number of elements in an array is equal to the product of the number of elements in each dimension. IUNIT 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 form:

(s[,s]...)

where:

s = A subscript expression

A subscript expression can be a constant, variable, or arithmetic expression. If the value of a subscript is not 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). Each subscript expression must refer to an element in its corresponding dimension according to the convention for numbering elements described in Section 2.5.1.

## 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 leftmost subscripts vary most rapidly. This is called the "order of subscript progression." 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

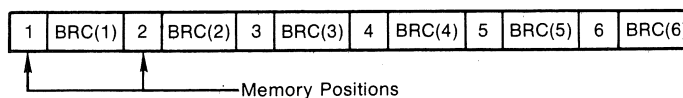
In the following types of statements, you can specify an array name without a subscript to indicate that the entire array is to be used (or defined):

- Type declaration statements
- COMMON statement
- DATA statement
- EQUIVALENCE statement
- FUNCTION statement
- SUBROUTINE statement
- I/O statements

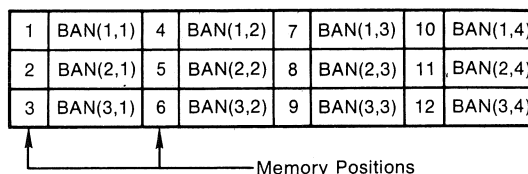
You can also use unsubscripted array names as actual arguments in references to external procedures. Unsubscripted array names are not permitted in any other type of statement.

**Figure 2-1 Array Storage**

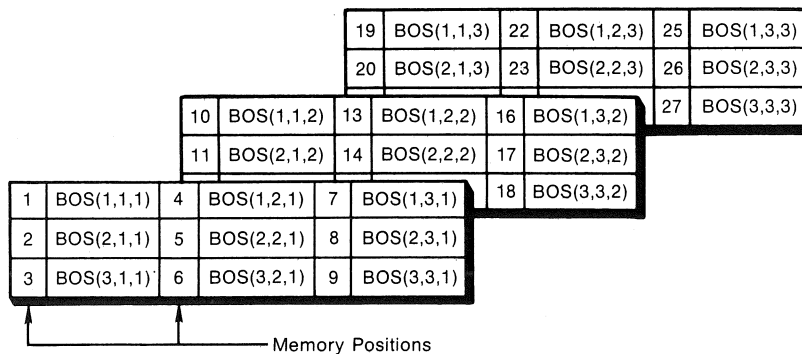
One-Dimensional Array BRC (6)



Two-Dimensional Array BAN (3,4)



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



ZK-616-82

### 2.5.6 Adjustable Arrays

Adjustable arrays allow subprograms to manipulate arrays of variable dimensions. To use an adjustable array in a subprogram, 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 expressions produce logical values; logical expressions produce either logical or integer 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
- 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 follow.

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.



## FORTRAN STATEMENT COMPONENTS

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 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

## FORTRAN STATEMENT COMPONENTS

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 of 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 follow.

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 data types 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 follow.

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction: the expression is true 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 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 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 cannot 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 "65; J has the value "165; K has the value "21.



# 3

## ASSIGNMENT STATEMENTS

---

Assignment statements assign a single value to a variable or array element. The three kinds of assignment statements are:

- Arithmetic
- Logical
- ASSIGN

### 3.1

## ARITHMETIC ASSIGNMENT STATEMENT

---

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

The arithmetic assignment statement has the form:

$$v = e$$

where:

$v$  = 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:

$$\text{KOUNT} = \text{KOUNT} + 1$$

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."

At the time the statement is executed,  $v$  need not be defined. However, if any symbolic reference in expression  $e$  is undefined, then  $v$  will also become undefined.

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.



## ASSIGNMENT STATEMENTS

**Table 3-1 Conversion Rules for Assignment Statements**

Var.or Array Elem. <sup>1</sup>	Source			
	Integer or Log.	Real	Double Precision	Complex
Integer or Logical <sup>2</sup>	Assign E <sup>3</sup> 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 <sup>2</sup>	Append fraction (.0) to E and assign to V	Assign E to V	Assign MS <sup>4</sup> portion of E to V; LS <sup>5</sup> portion of E is rounded	Assign real part of E to V; imaginary part of E is not used
Double Precision <sup>2</sup>	Append fraction (.0) to E and assign to MS <sup>4</sup> portion of V; LS <sup>5</sup> portion of V is 0	Assign E to MS <sup>5</sup> portion of V; LS <sup>5</sup> portion of V is 0	Assign E to V	Assign real part of E to MS <sup>4</sup> portion of V; LS <sup>5</sup> portion of V is zero, imaginary part of E is not used
Complex <sup>2</sup>	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 <sup>4</sup> portion of E to real part of V; LS <sup>5</sup> portion of E is rounded; imaginary part of V is 0.0	Assign E to V

<sup>1</sup>V (throughout table)

<sup>2</sup>Destination

<sup>3</sup>Expression (throughout table)

<sup>4</sup>Most significant (high order)

<sup>5</sup>Least significant (low order)

Examples of valid and invalid assignment statements follow.

Valid:

```
BETA = -1./(2.*X)+A*A/4.*(X*X)
PI = 3.14159
SUM = SUM + 1
```

Invalid:

Invalid Statement	Explanation
3.14 = A-B	Entity on the left must be a variable or array element
-J = I <sup>4</sup>	Entity on the left must be a variable or array element
ALPHA = ((X+6) *B *B/(X-Y)	Entity on the right is an invalid expression because the parentheses are not balanced

### 3.2

## LOGICAL ASSIGNMENT STATEMENT

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

The logical assignment statement has the form:

$$v = e$$

where:

$v$  = 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.

Whether or not  $v$  is defined before execution of the statement, it will become undefined if any symbolic reference in expression  $e$  is undefined when the statement is executed.

Examples of logical assignment statements follow.

Valid:

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

Invalid:

- $X = .TRUE.$  (The entity on the left must be logical)

### 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 form:

$$\text{ASSIGN } s \text{ TO } v$$

where:

$s$  = 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 cannot refer to a FORMAT statement.

## ASSIGNMENT STATEMENTS

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 of valid and invalid assign statements follow.

Valid:

- ASSIGN 10 TO NSTART
- ASSIGN 99999 TO KSTOP

Invalid:

Invalid Statement	Explanation
ASSIGN 250 TO ERROR	Variable must be an integer
ASSIGN I TO NLABEL	s must be a constant
ASSIGN I TO NLABEL(K)	v cannot be an array element

# 4

## CONTROL STATEMENTS

---

Statements are normally executed in the order in which they are written. However, you can 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.

### 4.1

## GO TO STATEMENTS

---

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

- Unconditional
- Computed
- Assigned

## CONTROL STATEMENTS

### 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

where:

s = A statement label

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

Examples:

- GO TO 7734
- GO TO 99999

### 4.1.2 Computed GO TO Statement

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

GO TO (slist)[,] e

where:

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 of 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 form:

GO TO v[[,] (slist)]

where:

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 for more information 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. Slist has no effect on the assigned GO TO statement.

Examples of assigned GO TO statements follow.

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 form:

IF (e)  $s_1, s_2, s_3$

where:

e = An arithmetic expression

$s_1, s_2, s_3$  = Labels of executable statements in the same program unit

All three labels ( $s_1, s_2, s_3$ ) 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 $s_1$
Equal to 0	Label $s_2$
Greater than 0	Label $s_3$

The following examples illustrate IF statements.

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 form:

IF (e) st

where:

e = A logical expression

st = A complete FORTRAN statement, which 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) will be treated as a logical value, regardless of the data type it assumes when it is evaluated.

Examples of logical IF statements follow.

```
IF (J .GT. 4 .OR. J .LT. 1) GO TO 250
IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K) * (-1.5DO)
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 form:

DO s[,] v=e<sub>1</sub>,e<sub>2</sub>[,e<sub>3</sub>]

where:

s = The label of an executable statement. The statement must physically follow in the same program unit.  
v = An integer variable  
e<sub>1</sub>,e<sub>2</sub>,e<sub>3</sub> = Integer expressions

The variable v is called the control variable; e<sub>1</sub>, e<sub>2</sub>, and e<sub>3</sub> are the initial, terminal, and increment parameters, respectively. If you omit the increment parameter, a default increment value of 1 is used.

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

- A GO TO statement
- An arithmetic IF statement
- 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 range of the 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.

The DO statement first evaluates the expressions e<sub>1</sub>, e<sub>2</sub>, and e<sub>3</sub> 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 until a condition exists for termination of the loop.



### 4.3.1 DO Iteration Control

Execution of a DO loop can be terminated either by transfer of control outside the range of the loop or completion of the loop.

#### 4.3.1.1 DO Loop Termination by Transfer of Control

You can 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.

#### 4.3.1.2 DO Loop Termination by Completion

After each execution of the DO range, first the value of the increment parameter is algebraically added to the control variable and then a test is made to determine whether or not the loop has been completed. The result of the test depends on the values of the control variable, and the initial, terminal, and increment parameters ( $e_1$ ,  $e_2$ , and  $e_3$ , respectively) as follows:

- If  $e_1$  is less than  $e_2$ , and  $e_3$  is positive, completion of the DO loop is established when the value of the control variable exceeds the value of  $e_2$ .
- If  $e_1$  is greater than  $e_2$ , and  $e_3$  is negative, completion is established when the value of the control variable is less than the value of  $e_2$ .

The number of iterations for a DO loop that completes by satisfying these conditions can be predicted to be:

$$\left[ \frac{e_2 - e_1}{e_3} \right] + 1$$

where  $[X]$  is the greatest integer less than  $X$ .

In all other cases, the DO loop is completed after one iteration of its range.

When execution of a DO loop completes, 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.

#### 4.3.1.3 DO Loop Control Variable and Parameters

The control variable and parameters of a DO loop can be referenced within the range of the loop. However, since iteration control depends on the values of these quantities, their alteration within the range of a loop might produce unpredictable or undesirable results. Examples such as those below defeat the purpose of the DO statement.

```
DO 10 I=1,10
.
.
I=I+1
.
10 CONTINUE
```

Normally, this DO loop should have 10 iterations, but alteration of its control variable in the statement "I=I+1" will cause it to terminate after only five iterations.

```

      J=10
      DO 10 I=1, J
      .
      .
      J=J+1
      .
10    CONTINUE

```

Normally, the range of this DO loop should execute 10 times, but alteration of its terminal parameter in the statement "J=J+1" will permanently prevent it from meeting the requirements for termination.

Examples of DO statements follow.

Valid	Explanation
<i>DO 100 K=1,50,2</i>	This statement specifies 25 iterations; K = 49 during the final iteration.
<i>DO 350 J=50,-2,-2</i>	This statement specifies 27 iterations; J = -2 during the final iteration.
<i>DO 25 IVAR=1,5</i>	This statement specifies 5 iterations; IVAR = 5 during the final iteration.
Invalid	Explanation
<i>DO NUMBER=5,40,4</i>	Statement label is missing
<i>DO 40 M=2.10</i>	Decimal point has been typed for a comma; note that this statement will not produce a syntax error, but will be interpreted as the following valid assignment statement: DO40M = 2.10

### 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.

Table 4-1 illustrates correctly and incorrectly nested DO loops.

## CONTROL STATEMENTS

Figure 4-1 Nested DO Loops

Correctly Nested DO Loops	Incorrectly Nested DO Loops
<pre>DO 45 K=1, 10   .   .   . DO 35 L=2, 50, 2   .   .   . 35 CONTINUE   .   .   . DO 45 M=1, 20   .   .   . 45 CONTINUE</pre>	<pre>DO 15 K=1, 10   .   .   . DO 25 L=1, 20   .   .   . 15 CONTINUE   .   .   . DO 30 M=1, 15   .   .   . 25 CONTINUE   .   .   . 30 CONTINUE</pre>

BU-2718

### 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

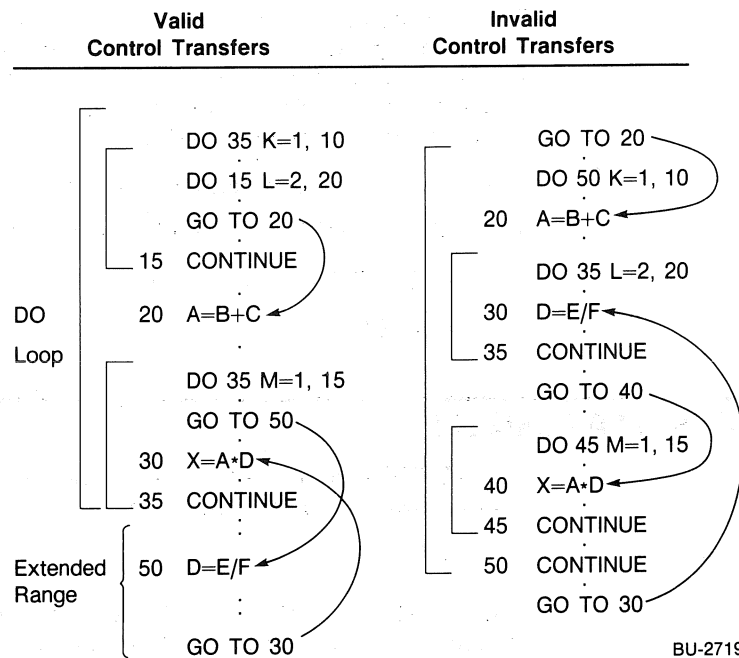
A DO loop has an extended range if it contains a control statement that transfers control out of the loop and if, after execution of one or more statements, another control statement returns control back into the loop. Thus, the range of the loop is extended to include all executable statements between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

Figure 4-2 Extended Range Control Transfers



## 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.

This statement has the form:

CONTINUE

## 4.5

**CALL STATEMENT**

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

The CALL statement has the form:

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

where:

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

a = An actual argument (section 6.1 describes actual arguments)

## CONTROL STATEMENTS

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, literal strings, or subprogram names. An unsubscripted array name in the argument list refers to the entire array.

Examples of CALL statements follow.

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

---

### 4.6 RETURN STATEMENT

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

RETURN

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

An example of a RETURN statement follows.

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

---

### 4.7 PAUSE STATEMENT

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

The PAUSE statement has the form:

PAUSE [disp]

where:

disp = A literal string, 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 on 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` statement. Because the command is specific to the operating system, it is not given here.

Examples of `PAUSE` statements follow.

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

---

### 4.8 STOP STATEMENT

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

The `STOP` statement has the form:

`STOP [disp]`

where:

`disp` = A literal string, a decimal digit string of one to five digits, or an octal constant

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

Examples of `STOP` statements follow.

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

---

### 4.9 END STATEMENT

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

The `END` statement has the form:

`END`

The `END` statement must not occur on a continuation line 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.



---

## SPECIFICATION STATEMENTS

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

The specification statements are:

- **IMPLICIT** statement—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 specified 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 can be assigned to entities contained in common blocks

The following sections describe these statements, giving their forms and examples of their usage.

---

### 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 form:

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

where:

`typ` = One of the data type specifiers (see Table 2-2 in Chapter 2)



## SPECIFICATION STATEMENTS

a = An alphabetic specification in one of two forms: c or c<sub>1</sub>-c<sub>2</sub>, where c is an alphabetic character. The latter form specifies a range of letters, from c<sub>1</sub> through c<sub>2</sub>, 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)
```

---

## 5.2 TYPE DECLARATION STATEMENTS

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

The type declaration statement has the form:

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

where:

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:

- 1 A type declaration statement must precede all executable statements.
- 2 You can declare the data type of a symbolic name only once.
- 3 You cannot label a type declaration statement.
- 4 You can use a type declaration statement to declare an array by appending a dimension 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 follow.

```

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

```

## 5.3 DIMENSION STATEMENT

The DIMENSION statement specifies for one or more arrays the number of dimensions in each array and the number of elements in each dimension.

The DIMENSION statement has the form:

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

where:

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, obtained in the same way as for a simple variable, 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. For example: *INTEGER VALUES DIMENSION VALUES(100)*

These statements define an array value with 100 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 follow.

```

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

```

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

## 5.4 COMMON STATEMENT

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

The COMMON statement has the form:

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

where:

cb = A symbolic name, called a common block name. The symbolic name 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/ICLK
.
.
CALL GSUB
.
.
END

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

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

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 might 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	Subprogram
COMMON HEAT,X/BLK1/KILO,Q	SUBROUTINE FIGURE
.	COMMON /BLK1/LIMA,R/ /ALFA,BET
.	.
CALL FIGURE	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 example of invalid usage, the integer variable ILEN is allocated on an odd byte address.

## 5.5 VIRTUAL STATEMENT

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

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

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

where:

## SPECIFICATION STATEMENTS

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 (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 byte limit on directly-addressable memory. Virtual arrays are placed in external main memory without significantly diminishing the 64K bytes 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.

For example:

```
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 the following 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 runtime 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 7.5).
- 6 When a virtual array element is used as an actual argument to a subprogram, its value will not be altered by an assignment to its corresponding dummy argument within the subprogram.
- 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).

- 8 Until any virtual array element is defined by an assignment statement, its value is indeterminate.

The following examples illustrate valid and invalid usage of virtual arrays.

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:

Example	Explanation
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. That is, when a virtual array element is used as an actual argument to a subprogram, its value will not be altered by an assignment to its corresponding dummy argument within the subprogram. Therefore, if not previously defined by means of an assignment statement, such an argument will continue to be of indeterminate value after the subprogram has terminated. Furthermore, until defined within the subprogram, the value of the corresponding dummy argument will also be indeterminate. An example of valid and invalid usage follows.

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

```

## SPECIFICATION STATEMENTS

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 element)
END
```

### 5.6 EQUIVALENCE STATEMENT

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

The EQUIVALENCE statement has the form:

EQUIVALENCE (nlist) [(nlist)]...

where:

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 can not be used in an EQUIVALENCE 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 valid and invalid EQUIVALENCE statements follow.

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

### 5.6.1 Making Arrays Equivalent

When you make an element of one array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between 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.

Examples of the use of the EQUIVALENCE statement follow.

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

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

**Table 5-1 Equivalence of Array Storage**

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

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

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



## SPECIFICATION STATEMENTS

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 Table 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))

A(1)	A(2)	A(3)	A(4)			
	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)
Existing Common				Extending Portion		

Invalid:

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

	A(1)	A(2)	A(3)	A(4)	
B(1)	B(2)	B(3)	B(4)	B(5)	B(6)
Extended Portion	Existing common				Extended Portion

If you assign two entities 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 form:

EXTERNAL v [,v]...

where:

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 for a main program and subprograms follows.

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)
```

## 5.8

### DATA STATEMENT

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

The `DATA` statement has the form:

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

## SPECIFICATION STATEMENTS

where:

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

where:

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 entities in the preceding nlist. Values are assigned on a one-for-one basis 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 cannot be initialized in DATA statements.

When a Hollerith constant or literal string 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 follow.

```
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 follow.

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

## 5.9

**PROGRAM STATEMENT**

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

The PROGRAM statement has the form:

PROGRAM nam

where:

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.

## 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 form:

BLOCK DATA [nam]

where:

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 follows.

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

## SPECIFICATION STATEMENTS

In this 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.

# 6

---

## SUBPROGRAMS

A subprogram is one statement or a group of statements, that defines 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: user-written, or supplied as part of the FORTRAN system.

There are three kinds of user-written subprograms:

- 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 can in turn be passed back to the referencing program.

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 that 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 can 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.

An actual argument is passed to a subprogram in either of two ways:

- For an actual argument that is a variable name, array name, array element, subprogram name, or constant, the address of the datum is supplied to the subprogram.
- For an actual argument that is a function call, virtual array element, or expression containing at least one operator, the datum is stored in a temporary location, and the address of the temporary location is supplied to the subprogram.

An actual argument passed to a subprogram by the first method will be defined or redefined if its corresponding dummy argument is defined or redefined within the subprogram; the contrary is true for an actual argument passed by the second method. An important consequence of this fact is that a constant used as an actual argument will be corrupted if its associated dummy argument is 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. An example of valid usage and two examples of invalid usage follow.

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(1,2,))
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.)

PROGRAM CORRPT
CALL DOUBLE(1) (The constant 1 will be corrupted by this call.)
TYPE *, 1
STOP
END

SUBROUTINE DOUBLE(I)
INTEGER I
I=I*2
RETURN
END

```

### 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

```



## SUBPROGRAMS

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

For more information on array declarators, see Section 2.5.1.

## 6.2 USER-WRITTEN SUBPROGRAMS

A user-written FORTRAN IV subprogram specifies with FORTRAN IV statements the steps of a computing procedure. Subprograms can be used to promote modularity and structure in a program. They also can promote space-efficiency, since a subprogram can be used to repeat the procedure it defines without physical duplication of its code.

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, which is used in an expression, consists of the function name and function arguments. 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 form:

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

where:

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 form:

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

where:

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.
- References to any statement function can appear only in the program unit in which that statement function is defined.
- A statement function definition statement can include a reference to another statement function, which must be defined earlier in the same program unit.
- Standard statement ordering requires that statement function definitions must be placed before all executable statements in a program unit, as illustrated in Figure 1-3. However, PDP-11 FORTRAN IV requires only that the definition of a statement function physically precede all references to that function.
- 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.

## SUBPROGRAMS

- 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. If a dummy argument has the same name as a variable or array declared in a type declaration statement in the same program unit, that dummy argument will assume the type specified by the declaration.

Examples of valid and invalid statement function definitions follow.

Valid:

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

Invalid:

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

Examples of valid and invalid statement function references follow. These examples refer to the second statement function definition.

Valid:

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

Invalid:

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 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 form:

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

where:

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 form:

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

where:

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; 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 statement. 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:

- 1 The FUNCTION statement must be the first statement of a function subprogram.
- 2 The FUNCTION statement must not have a statement label.
- 3 A function subprogram must not contain these statements: SUBROUTINE, BLOCK DATA, or another FUNCTION statement.
- 4 A function subprogram can reference another subprogram but it cannot reference itself either directly or indirectly.
- 5 The data type of a function name can be specified either implicitly or explicitly in the FUNCTION statement or in a type declaration statement.
- 6 The function name must have the same data type in the subprogram and in the referencing program.

An example of a function subprogram follows.

```
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
```

## SUBPROGRAMS

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:

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

The calculation is repeated until the difference between  $X_i$  and  $X_{i+1}$  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 form:

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

where:

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 statement.

The following rules govern the use of subroutine subprograms:

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

For 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 example:

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

Subroutine example:

```
SUBROUTINE PLYVOL
COMMON NFACES,EDGE,VOLUME
CUBED = EDGE**3
GOTO (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,5),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:

$$R = 3.14159 * \text{ABS}(X-1)$$

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.

## SUBPROGRAMS

By default, each symbol in the table of FORTRAN library functions (Table B-2, Section B.3) refers to the FORTRAN library function named with that symbol. If a program uses any such symbol to identify a variable, array, or user-written subprogram, then that symbol's default association with a FORTRAN library function is overridden. Furthermore, the library function named with that symbol becomes unavailable for use in all program units for which the symbol is redefined.

# 7

## INPUT/OUTPUT STATEMENTS

FORTRAN programs use:

- READ and ACCEPT statements for input
- WRITE, TYPE, and PRINT statements for output
- ENCODE and DECODE statements for I/O-free translation of data

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.

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.
- I/O-free translation - ENCODE and DECODE statements translate data between internal form and external form without performing any I/O. They use variables and arrays within the FORTRAN program for storage of the data. See Section 7.5.

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.



## INPUT/OUTPUT STATEMENTS

Section 7.1 describes general FORTRAN I/O 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 information 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 can 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**

Direct access, also referred to as random 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 runtime 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

where:

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 9.6) 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 I/O 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 form:

$s[,s]...$

where:

$s$  = 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, expression, or simple I/O list enclosed in parentheses. 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.

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 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 can 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 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:

$(list, i = e_1, e_2 [, e_3])$

where:

list = An I/O list

i = An integer variable

$e_1, e_2, e_3$  = Integer expressions

The variable  $i$  and the parameters  $e_1$ ,  $e_2$ , and  $e_3$  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. Since the mechanism of control for implied DO loops is analogous to that for DO loops, the consequences of modifying  $i$  (the control variable) within the range of the implied DO loop are also analogous. Some examples follow.

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

The statement in this example produces the same result as:

```
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  $J$ s 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 I/O

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.

### 7.3.1 Formatted Sequential Input Statements - READ, ACCEPT

A 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.

A formatted sequential ACCEPT statement is similar, except that it always has an implicit logical unit number.

Formatted sequential input statements have the forms:

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

where:

u = A logical unit number  
f = A format specifier  
s = The label of an executable statement

## INPUT/OUTPUT STATEMENTS

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 indicate 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 there is no I/O list, 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)
```

In the following example, the statements read a record from logical unit 1 and assign fields to ARRAY.

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

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

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

### 7.3.2 Formatted Sequential Output Statements - WRITE, TYPE, PRINT

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 forms:

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

```
PRINT f[,list]
```

where:

- 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 might 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 literal string 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.



## 7.3.3 List-Directed Input Statements - READ, ACCEPT

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.

List-directed input statements have the forms:

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

where:

u = A logical unit number  
 \* = Indicates list-directed formatting  
 s = The label of an executable statement  
 list = An I/O list

The list-directed input statement obtains values for assignment to I/O list elements from a perceived input record. The perceived input record is constructed from one or more physical records which are read until either all I/O list elements are filled or a slash delimiter is encountered. Should either of these events occur when a physical record is only partially consumed, the remainder of that record is ignored.

The perceived input record must contain one or more entries separated by delimiters. An entry can be a data element, a null element, or an element repeater.

### 7.3.3.1 Data Elements

A data element can be one of the following data types:

- Integer
- Real
- Double precision
- Complex
- Logical

Integer, real, and double precision data elements are specified with FORTRAN syntax for decimal integer, real, and double precision constants, respectively, as described Section 2.3. Integer data elements cannot be less than -32768 or greater than 32767.

A complex data element has the form:

```
( [ x], [y] )
```

where x and y represent the real and imaginary components of the complex datum. Each component is specified with FORTRAN syntax for decimal integer, real, or double precision constants, and each has a default value of 0.0. Spaces or tab characters surrounding either component are ignored.

A logical data element is specified with a series of non-delimiting characters beginning with either T or t (evaluated as .TRUE.), or F or f (evaluated as .FALSE.).

When the data types of a data element and its corresponding I/O list element differ, conversion is performed according to the rules for arithmetic assignment (see Table 3-1). If a real, double precision, or complex data element is thus converted to integer, the conversion must produce a value that is not less than -32768 or greater than 32767.

## 7.3.3.2 Null Elements

A null element is an empty entry in the perceived record, delimited by commas with or without surrounding spaces or tab characters. A null element suppresses modification of the I/O list element to which it corresponds. Note that null elements are treated differently from omitted components of complex data elements, which assume a default value of 0.0.

## 7.3.3.3 Element Repeaters

The form of an element repeater is:

$$r*[d]$$

where r is a positive integer and d is a data element. If d is present, then the element repeater indicates r occurrences of d; otherwise it indicates r null elements.

## 7.3.3.4 Delimiters

The characters which comprise delimiters in a perceived record are the space, the tab character, the comma, and the slash. The end of a physical record within the perceived record is equivalent to a space.

Spaces or tab characters preceding the first entry in the perceived record are ignored.

Data elements and element repeaters can be delimited by:

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

Null elements must be delimited by commas, with or without surrounding spaces or tab characters.

When a slash delimiter is encountered, processing of the record and input statement is terminated; all characters in the record following the slash are ignored, and all I/O list elements not yet processed remain unchanged.

## INPUT/OUTPUT STATEMENTS

### 7.3.3.5 Example of a List-Directed Input Statement

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 input 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
E	(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.

### 7.3.4 List-Directed Output Statements - WRITE, TYPE, PRINT

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 forms:

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

where:

```
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 <sup>1</sup>
REAL*4	1PG15.7
REAL*8	1PG25.16
COMPLEX*8	IX,'(,1PG14.7, ',', 1PG14.7,')'
Hollerith	1X,An <sup>2</sup>

<sup>1</sup> Although the INTEGER\*4 data type is output in a wider field than is the INTEGER\*2 data type, the magnitudes of values that can be output with either data type are the same. Whereas four bytes are allocated to INTEGER\*4 variables, all operations, including I/O operations, use only two of those four bytes.

<sup>2</sup> 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. Literal strings in the I/O list itself are output without delimiting apostrophes. Note that list-directed output records that contain Hollerith constants cannot 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.

Examples:

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

If a program unit consists of:

```
DIMENSION A(5)
DATA A/5*3.4/
WRITE (1,*) 'ARRAY ΔVALUES ΔFOLLOW'
WRITE (1,*) A,5
```

then the following records will be output:

```
ARRAY ΔVALUES ΔFOLLOW
ΔΔΔ3.400000 ΔΔΔΔΔΔΔ3.400000 ΔΔΔΔΔΔΔ3.400000 ΔΔΔΔΔΔΔ3.400000
ΔΔΔ3.400000 ΔΔΔΔΔΔΔΔΔΔ5
```

### 7.3.5 Unformatted Sequential Input Statement - READ

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 form:

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

where:

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:

```
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.

### 7.3.6 Unformatted Sequential Output Statements - WRITE

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 form:

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

where:

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.

Examples:

```
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 I/O

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) can be used to specify the attributes of the direct access file.

### 7.4.1 Unformatted Direct Access Input Statement - READ

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 form:

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

where:

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:

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

This statement reads record 10 of a file on logical unit 1, and assigns two 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 five real values to array RHO.

### 7.4.2 Unformatted Direct Access Output Statement - WRITE

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 form:

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

where:

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 zeros.

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.

For 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 forms:

```
ENCODE(c,f,b[,ERR=s])[list]  
DECODE(c,f,b[,ERR=s])[list]
```

where:

c = In the ENCODE statement, the total length, in bytes, of the data to be translated to character form; in the DECODE statement, the number of characters to be translated to internal form  
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, b 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 progression. 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 two 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 eight 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'
```





# 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 manner in which the input data is interpreted. If output is being performed or an ENCODE statement is being executed, the format statement describes the manner 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 form:

FORMAT (q<sub>1</sub>f<sub>1</sub>s<sub>1</sub> f<sub>2</sub>s<sub>2</sub> ... f<sub>n</sub>q<sub>n</sub>)

where:

- 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 form:

[r]c[w[.d]]

where:

- r = The number of times the field descriptor is to be repeated (repeat count); if you omit r, it is assumed to be one
- 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 less than or equal to 255; 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

## FORMAT STATEMENTS

- 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 8.2 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.

Use an I, O, or L field descriptor to process integer and logical data. 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 runtime format instead of a FORMAT statement. Section 8.5 describes runtime 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 form:

Iw

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

Rules in effect for data input:

- 1 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.
- 2 The external data must be an integer constant; it cannot contain a decimal point or exponent field.
- 3 If the external data value exceeds the maximum allowed magnitude of the corresponding list element, an error occurs.
- 4 If the first nonblank character of the external field is a minus sign, the field is treated as a negative value.

- 5 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.
- 6 An all-blank field is treated as a value of 0.

Input examples:

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

Rules in effect for data output:

- 1 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.
- 2 If the value does not fill the field, leading spaces are inserted.
- 3 If the value exceeds the field width, the entire field is filled with asterisks.
- 4 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.
- 5 Plus signs are suppressed.

Output examples:

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 form:

Ow

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

## FORMAT STATEMENTS

Rules in effect for data input:

- 1 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.
- 2 The external field can contain only the numerals 0 through 7; it cannot contain a sign, a decimal point, or an exponent field.
- 3 An all-blank field is treated as a value of 0.
- 4 If the value of the external data exceeds the allowed size of the corresponding list element, an error occurs.

Input examples:

Format	Internal (Decimal) Value	External (Octal) Representation
05	77777	32767
04	31274	1623
06	15 ΔΔΔΔ	53248
03	97 Δ	Not permitted: error

Rules in effect for data output:

- 1 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.
- 2 No signs are output; a negative value is transmitted in its octal (2's complement) form.
- 3 If the value does not fill the field, leading spaces are inserted.
- 4 If the value exceeds the field width, the entire field is filled with asterisks.

Output examples:

Format	Internal (Decimal) Value	External (Octal) Representation
06	32767	Δ77777
06	-32767	100001
02	14261	**
04	27	ΔΔ33
05	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 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:

- 1 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  $w$ .
- 2 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  $w$ .
- 3 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.
- 4 If the first nonblank character of the external field is a minus sign, the field is treated as a negative value.
- 5 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.
- 6 An all-blank field is treated as a value of 0.
- 7  $w$  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:

- 1 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.
- 2 If the value does not fill the field, leading spaces are inserted.
- 3 If the value exceeds the field width, the entire field is filled with asterisks.
- 4 Plus signs are suppressed.
- 5  $w$  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.

## FORMAT STATEMENTS

Output examples:

Format	Internal Value	External Representation
F8.5	2.3547188	Δ2.35472
F9.3	8789.7361	Δ8789.736
F2.1	51.44	**
F10.4	-23.24352	ΔΔ-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 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.

Rule 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	ΔΔ1022.43E-6	0.102243E-2
E15.3	52.3759663 ΔΔΔΔΔ	0.523759E + 2
E12.5	210.5271D + 10	0.2105271E + 13

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:

- 1 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.
- 2 If the value does not fill the w characters, leading spaces are inserted.
- 3 If the value exceeds the w characters, the entire field is filled with asterisks.
- 4 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

where:

nn = A 2-digit integer constant

- 5 Plus signs are suppressed.
- 6 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	Δ0.48E+06
E12.5	475867.222	Δ0.47587E+06
E12.3	0.00069	ΔΔΔ0.690E-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 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.

Rule 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 ΔΔΔΔΔ	0.1234500000D+8
D10.2	ΔΔ123.45 ΔΔ	0.1234500000D+3
D15.3	367.4981763D-04	0.3674981763D-1

Rule in effect for output:

There is only one difference between the D and E descriptors on output: if you use the D descriptor, the letter D is output instead of the letter E.



## FORMAT STATEMENTS

Output examples:

Format	Internal Value	External Value
D14.3	0.0363	ΔΔΔΔ0.363D-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 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.

Rule 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:

- 1 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.
- 2 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
$0.1 \leq m < 10.0$	F(w-4).d, 'ΔΔΔΔ' <sup>1</sup>
$1.0 \leq m < 10.0$	F(w-4).1, 'ΔΔΔΔ'
.	.
.	.
.	.
$10d-2 \leq m < 10d-1$	F(w-4).1, 'ΔΔΔΔ'
$10d-1 \leq m < 10d$	F(w-4).0, 'ΔΔΔΔ'
$m \geq 10d$	Ew.d

<sup>1</sup>The 'ΔΔΔΔ' in the second column specifies that four spaces are to follow the numeric data representation.

- 3 Plus signs are suppressed.

- 4 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	Δ0.123457E-01
G13.6	-0.12345678	-0.123457 ΔΔΔΔ
G13.6	1.23456789	ΔΔ1.23457 ΔΔΔΔ
G13.6	12.34567890	ΔΔ12.3457 ΔΔΔΔ
G13.6	123.45678901	ΔΔ12.3457 ΔΔΔΔ
G13.6	-1234.56789012	Δ-1234.57 ΔΔΔΔ
G13.6	12345.67890123	ΔΔ12345.7 ΔΔΔΔ
G13.6	123456.78901234	ΔΔ123457. ΔΔΔΔ
G13.6	-1234567.89012345	-0.123457E + 07

Compare the above examples with the following examples, which show the same values output with an equivalent F field descriptor.

Format	Internal Value	External Representation
F13.6	0.01234567	ΔΔΔΔ0.012346
F13.6	-0.12345678	ΔΔΔΔ0.123457
F13.6	1.23456789	ΔΔΔΔ1.234568
F13.6	12.34567890	ΔΔΔΔ12.345679
F13.6	123.45678901	ΔΔΔ123.456789
F13.6	-1234.56789012	Δ-1234.567890
F13.6	12345.67890123	Δ12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	*****

### 8.1.7 L Field Descriptor

The L field descriptor specifies transfer of logical data. It has the form:

Lw

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

Rules in effect for input:

- 1 The L field descriptor specifies that w characters are read from the external field.
- 2 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.
- 3 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.

## FORMAT STATEMENTS

- 4 Any other value in the external field produces an error.

Rules in effect for output:

- The L field descriptor specifies output of the letter T or t (if the value of the corresponding I/O list element is `.TRUE.`), or the letter F or f (if the value is `.FALSE.`) to an external field w characters long.
- The letter T (t) or F (f) is in the rightmost position of the field, preceded by w-1 spaces.

Output examples:

Format	Internal Value	External Representation
L5	.TRUE.	ΔΔΔΔT
L1	.FALSE.	F

### 8.1.8 A Field Descriptor

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

- 1 The A field descriptor transfers w characters from the external record and assigns them to the corresponding I/O list element.
- 2 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

- 3 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.

- 4 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 $\Delta\#$	# (LOGICAL*1)
A6	PAGE $\Delta\#$	$\Delta\#$ (INTEGER*2)
A6	PAGE $\Delta\#$	GE $\Delta\#$ (REAL)
A6	PAGE $\Delta\#$	PAGE $\Delta\# \Delta\Delta$ (DOUBLE PRECISION)

Rules in effect for output:

- 1 The A field descriptor specifies output of the contents of the corresponding I/O list element to an external field  $w$  characters long.
- 2 If  $w$  is greater than the size of the list element, the data appears in the field, right-justified, with leading spaces.
- 3 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	$\Delta$ OHMS
A5	VOLTS $\Delta\Delta\Delta$	VOLTS
A5	AMPERES $\Delta$	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:

$$nHc_1c_2c_3 \dots c_n$$

where:

- $n$  = The number of characters to be transferred
- $c_i$  = An ASCII character

Rule 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.

Rule in effect for output:

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

## FORMAT STATEMENTS

An example of the H field descriptor usage follows.

```
TYPE 100
100 (41H ΔENTER ΔPROGRAM ΔTITLE, ΔUP ΔTO Δ20 ΔCHARACTERS)
ACCEPT 200
200 FORMAT (20H ΔTITLE ΔGOES ΔHERE ΔΔΔ)
```

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 can also be specified as a literal string.

In a literal string, the apostrophe is written as two apostrophes. For example:

```
50 FORMAT ('TODAY''S ΔDATE ΔIS: Δ',12,'/',12,'/',12)
```

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 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.

Rule in effect for input:

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

Rule 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 ΔNUMBER Δ,12,16X,23HGRAPHIC ΔANALYSIS, ΔCONT.)
```

The WRITE statement prints a record similar to:

```
PAGE NUMBER nn                GRAPHIC ANALYSIS, CONT.
```

where:

nn = The current value of the variable NPAGE

Note that the number 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 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.

Rule 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 this 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.

Rule 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 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, 14, 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 dollar sign (\$) 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 (' ΔENTER ΔRADIUS ΔVALUE Δ', $)
      ACCEPT 200, X
      FORMAT (F6.2)
200
```

produce a message on the terminal in the form:

```
ENTER ΔRADIUS ΔVALUE
```

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

```
ENTER ΔRADIUS ΔVALUE Δ12.
```

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

```
200  FORMAT (11H ΔSIGN ΔHERE:; $)

200  FORMAT (11H$SIGN ΔHERE:)
```

## 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 colon descriptor (:) has no effect if I/O list items remain. For example:

```
PRINT 100,3
PRINT 200,4
100  FORMAT (' Δ = ', I2, ' ΔJ = ', I2)
      FORMAT (' ΔK = ', I2, ' ΔL = ', I2)
200
```

These statements print the following two lines:

```
I = Δ3 ΔJ =
K = Δ4
```

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.

Rule 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	Δ2.35472 3.45673
E9.2,' Δ, Δ',E5.3	47587.222, 56.123	Δ0.48E+06 Δ, Δ* * * *

### 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

where:

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 input and output:

- 1 If you do not use a scale factor, a default scale factor of 0P applies.
- 2 The scale factor is set to 0P at the start of every I/O statement.
- 3 A scale factor applies to all subsequent F, E, D, or G field descriptors, until a new scale factor is specified.



- 4 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 this example, 3P applies to 2F5.3 and E8.5 but not to E6.3 or the X, I, or A descriptors.

- 5 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 this example, 3P applies to both 2F5.3 and E8.5.

- 6 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, 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.

- 7 A scale factor of 0P can be reinstated only by an explicit 0P specification in the format.

Additional rules in effect for input:

- 1 If the external field contains an exponent, the scale factor has no effect.
- 2 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	ΔΔΔ37.614Δ	.037614
3PE10.5	ΔΔ37.614E2	3761.4
-3PE10.5	ΔΔΔΔ37.614	37614.0

Additional rules in effect for output:

- 1 Scale factors apply only to data output. The values of the I/O list variables do not change.
- 2 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.
- 3 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.

- 4 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	ΔΔ-2.701E+02
1PE12.2	-270.139	ΔΔΔ-2.70E+02
-1PE12.2	-270.139	ΔΔΔ0.03E+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))
```

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

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 value, default values are supplied based on the data type of the I/O list element. Note that for F, E, D, and G, you cannot specify only the w or d value; you must specify the w.d value 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

Before and after the transfer of a record to a printer or terminal, additional characters can be transferred to position the carriage or cursor. In general, positioning characters that precede records advance the carriage or cursor downward; those that follow records return the carriage or cursor to the left margin.

A record can be output with any one of five combinations of positioning characters. For each record, the desired combination is specified in the first character of the record, called the carriage control character. During output, this character is translated to its associated combination of positioning characters and then discarded; it is never output with the record.

### 8.2.1 Positioning Characters

Table 8-3 illustrates the mapping of carriage control characters to positioning character combinations. If any character other than '0', '1', '\$', or '+' is supplied as a carriage control character, it is treated as ' '. Note that if the intended carriage control character is inadvertently omitted from the record, the first character present will still be used for carriage control purposes and then deleted from the record.

Table 8-3 Carriage Control Character Translation

Carriage Control Character	Positioning Characters Preceding Record	Positioning Characters Following Record
' '	Line Feed	Carriage Return
'0'	Two Line Feeds	Carriage Return
'1'	Form Feed	Carriage Return
'\$'	Line Feed	None
'+'	None	Carriage Return

### 8.2.2 Output to Other Devices

Carriage control characters are always translated into positioning characters during output to a printer or terminal. However, depending on the operating environment, they might or might not undergo translation during output to other devices. Refer to the appropriate FORTRAN IV user's guide for information on the treatment of carriage control characters during output to devices other than printers and terminals.

Any character not listed in Table 8-4 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 (306/16,2F8.4)

```

This statement is equivalent to:

```

      WRITE (6,40) K,L,M
40    FORMAT (306)
      WRITE (6,50) N,O,P
      FORMAT (16,2F8.4)
50

```

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:

## FORMAT STATEMENTS

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

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 in a w.d specification, the d 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 literal string 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 RUNTIME FORMATS

A runtime 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 runtime 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.

Runtime 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 depends 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 this 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.

### 8.5.1 Input Using the H Field Descriptor

Each time an I/O statement referencing a runtime format is executed, the current format information is interpreted and given a temporary storage location. A difference between runtime and compiled formats when data is input using the H field descriptor is ascribed to this use of temporary storage space.

As explained in Section 8.1.9, data input using an H field descriptor in a compiled format statement overwrites the original Hollerith data given with that field descriptor. The new Hollerith data can subsequently be output using the same format statement.

However, Hollerith data input with a runtime format is stored in a temporary location distinct from the array containing the format specification. Therefore, any Hollerith data present in a runtime format will not be changed by input performed under the control of that format, and subsequent access to the Hollerith data input by the operation is not possible.

### 8.6 **FORMAT CONTROL INTERACTION WITH I/O 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 an error will occur.

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 literal string field descriptor. In H and literal string 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 follow.

```
      READ (I,100) A, B, C, D, E, F
100  FORMAT (F8.3, F8.3)
```

In this example, three records containing two fields are read. The first record assigns values to A and B; the second to C and D; and the third to E and F.

```
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 8-5 summarizes the FORMAT codes.

### 8.7.1 General Rules

- 1 A FORMAT statement must always be labeled.
- 2 In a field descriptor such as rX 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 nHc<sub>1</sub>c<sub>2</sub> ... c<sub>n</sub>, 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 literal string field descriptors unless these descriptors are enclosed in parentheses and treated as a group repeat specification.



## FORMAT STATEMENTS

- 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 a literal string.
- 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.
- 4 Treatment of the first character of a record output to a device other than a printer or terminal depends on the host operating system. Refer to the appropriate FORTRAN IV user's guide for more information.

**Table 8-5 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 exponential 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
:	:	Terminates format control if the I/O list is exhausted



---

## AUXILIARY INPUT/OUTPUT STATEMENTS

The five auxiliary I/O statements perform file management functions.

- 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.

---

### 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 form:

OPEN(par[,par]...)

where:

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

## AUXILIARY INPUT/OUTPUT STATEMENTS

- 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	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' 'APPEND'	Access method	'SEQUENTIAL'
ASSOCIATEVARIABLE	v <sup>4</sup>	Next record number in direct access	No associate variable
BLOCKSIZE	e <sup>2</sup>	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 <sup>3</sup>	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 <sup>1</sup>	File name specification	Depends on unit and system
NOSPANBLOCKS	-	Records do not span blocks	Records can span blocks
READONLY	-	Write protection	No write protection

<sup>1</sup>c = a literal string, array name, variable name, or array element name

<sup>2</sup>e = an integer expression

<sup>3</sup>s = a statement label

<sup>4</sup>v = an integer variable name

## AUXILIARY INPUT/OUTPUT STATEMENTS

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

Keyword	Values	Function	Default
RECORDSIZE	e	Record length	Depends on FORM
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

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.

Examples:

```
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=1, NAME='MTO:MYDATA.DAT', BLOCKSIZE=8192,  
      TYPE='NEW', ERR=14, RECORDSIZE=1024)
```

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

```
OPEN (UNIT=1, NAME='MTO:MYDATA.DAT', READONLY, TYPE='OLD',  
      RECORDSIZE=1024, 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 form:

```
ACCESS = acc
```

where:

acc = One of the literal strings, 'SEQUENTIAL', 'DIRECT', 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

The ASSOCIATEVARIABLE keyword 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 form:

ASSOCIATEVARIABLE = asv

where:

asv = An integer variable

## 9.1.3 BLOCKSIZE Keyword

The BLOCKSIZE keyword 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 form:

BLOCKSIZE = bks

where:

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**

The BUFFERCOUNT keyword 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 form:

BUFFERCOUNT = bc

where:

bc = An integer expression

### 9.1.5 **CARRIAGECONTROL Keyword**

The CARRIAGECONTROL keyword 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 form:

CARRIAGECONTROL = cc

where:

cc = One of the literal strings 'FORTRAN', 'LIST', or 'NONE'

### 9.1.6 **DISPOSE Keyword**

The DISPOSE keyword 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 specify 'PRINT', the file is submitted to the system line printer spooler. On some systems, the file is deleted after printing. If you specify 'DELETE', the file is deleted. A read-only file (see Section 9.1.14) cannot be printed or deleted, and a scratch file (see Section 9.1.17) cannot be saved or printed.

This keyword has the forms:

DISPOSE = dis  
DISP = dis

where:

dis = One of the literal strings 'SAVE', 'KEEP', 'PRINT', or 'DELETE'

---

### 9.1.7 ERR Keyword

The ERR keyword 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 form:

ERR = *s*

where:

*s* = The label of an executable statement

---

### 9.1.8 EXTENDSIZE Keyword

The EXTENDSIZE keyword specifies the number of blocks by which a disk file is extended when additional file storage is allocated. If you do not specify EXTENDSIZE, or if you specify 0, the system default for the device is used.

This keyword has the form:

EXTENDSIZE = *es*

where:

*es* = An integer expression

---

### 9.1.9 FORM Keyword

The FORM keyword 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 form:

FORM = *ft*

where:

*ft* = The literal string 'FORMATTED' or 'UNFORMATTED'

---

### 9.1.10 INITIALSIZE Keyword

The INITIALSIZE keyword 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 form:

INITIALSIZE = *insz*

where:

*insz* = An integer expression

---

### 9.1.11 MAXREC Keyword

The MAXREC keyword 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 form:

MAXREC = mr

where:

mr = An integer expression

---

### 9.1.12 NAME Keyword

The NAME keyword 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 form:

NAME = fln

where:

fln = An array name, variable name, array element name, or literal string

---

### 9.1.13 NOSPANBLOCKS Keyword

The NOSPANBLOCKS keyword 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 form:

NOSPANBLOCKS

---

### 9.1.14 READONLY Keyword

The READONLY keyword prohibits writing by the program to a file.

This keyword has the form:

READONLY

---

### 9.1.15 RECORDSIZE Keyword

The RECORDSIZE keyword 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 form:

RECORDSIZE = rl

where:

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

The SHARED keyword specifies that the file is to be opened for shared access by more than one program executing simultaneously.

Sequential files can be shared only if they are stored on disk, and only one program can have write access.

This keyword has the form:

SHARED

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

---

### 9.1.17 TYPE Keyword

The TYPE keyword specifies the status of the 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 form:

TYPE = typ

## AUXILIARY INPUT/OUTPUT STATEMENTS

where:

typ = One of the literal strings 'OLD', 'NEW', 'SCRATCH', or unknown

---

### 9.1.18 UNIT Keyword

The UNIT keyword 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 form:

UNIT = u

where:

u = An integer expression

---

## 9.2 CLOSE STATEMENT

The CLOSE statement disconnects a file from a unit.

The CLOSE statement has the form:

CLOSE (unit=u [ , { DISPOSE  
DISP } =p ] [,ERR=s])

where:

u = A logical unit number

p = A literal string 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, and a file opened for read-only access cannot 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.

---

### 9.3 REWIND STATEMENT

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

The REWIND statement has the form:

REWIND u

where:

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.

---

### 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 form:

BACKSPACE u

where:

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.

---

### 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.

The FIND statement has the form:

FIND (u'r)

where:

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.

---

### 9.6 ENDFILE STATEMENT

The ENDFILE statement writes an end-file record to the specified unit.

The ENDFILE statement has the form:

ENDFILE u

where:

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.

---

### 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 form:

DEFINE FILE u (m,n,U,asv) [u(m,n,U,asv)] ...

where:

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

## AUXILIARY INPUT/OUTPUT STATEMENTS

U = Specifies that the file is unformatted (binary); this is the only acceptable entry in this position  
asv = An integer variable, called the associated variable of the file; at the end of each direct access I/O operation, the record number of the next higher-numbered record in the file is assigned to asv

DEFINE FILE specifies that a file containing m fixed-length records of n 16-bit words each exists, or is to exist, on logical unit u. The records in the file are numbered sequentially from 1 through m.

DEFINE FILE must be executed before the first direct-access I/O statement that refers to the specified file.

DEFINE FILE also establishes the integer variable asv as the associated variable of the file. At the end of each direct access I/O operation, the FORTRAN I/O system places in asv the record number of the record immediately following the one just read or written. 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.





# 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
Δ 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 SETS

#### A.2.1

##### Hexadecimal

Table A-1 presents the ASCII character set. At the top of the table are hexadecimal digits 0 to 7 (columns), and to the left of the table are hexadecimal digits 0 to F (rows). 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.

## CHARACTER SETS

**Table A-1 ASCII Character Set**

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	{	`	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

**Key to Table A-1:**

NUL = Null

SOH = Start of Heading

STX = Start of Text

ETX = End of Text

EOT = End of Transmission

ENQ = Enquiry

ACK = Acknowledge

BEL = Bell

BS = Backspace

HT = Horizontal Tabulation

LF = Line Feed

VT = Vertical Tab

FF = Form Feed

CR = Carriage Return

SO = Shift Out

SI = Shift In

SP = Space

DLE = Data Link Escape

DC1 = Device Control 1

DC2 = Device Control 2

DC3 = Device Control 3

DC4 = Device Control 4

NAK = Negative Acknowledge

SYN = Synchronous Idle

ETB = End of Transmission Block

CAN = Cancel

EM = End of Medium

SUB = Substitute

ESC = Escape

FS = File Separator

GS = Group Separator

RS = Record Separator

US = Unit Separator

DEL = Delete

## A.2.2 Octal

Table A-2 shows 7-bit ASCII characters and their octal code.

**Table A-2 7-Bit ASCII Code**

Char	Octal Code	Char	Octal Code	Char	Octal Code	Char	Octal Code
NUL	000	SP	040	@	100	`	140
SOH	001	!	041	A	101	a	141
STX	002	"	042	B	102	b	142
ETX	003	#	043	C	103	c	143
EOT	004	\$	044	D	104	d	144
ENQ	005	%	045	E	105	e	145
ACK	006	&	046	F	106	f	146
BEL	007	'	047	G	107	g	147
BS	010	(	050	H	110	h	150
HT	011	)	051	I	111	i	151
LF	012	*	052	J	112	j	152
VT	013	+	053	K	113	k	153
FF	014	,	054	L	114	l	154
CR	015	-	055	M	115	m	155
SO	016	.	056	N	116	n	156
SI	017	/	057	O	117	o	157
DLE	020	0	060	P	120	p	160
DC1	021	1	061	Q	121	q	161
DC2	022	2	062	R	122	r	162
DC3	023	3	063	S	123	s	163
DC4	024	4	064	T	124	t	164
NAK	025	5	065	U	125	u	165
SYN	026	6	066	V	126	v	166
ETB	027	7	067	W	127	w	167
CAN	030	8	070	X	130	x	170
EM	031	9	071	Y	131	y	171
SUB	032	:	072	Z	132	z	172
ESC	033	;	073	[	133	{	173
FS	034	<	074	\	134		174
GS	035	=	075	]	135	}	175
RS	036	>	076	^	136	~	176
US	037	?	077	_	137	DEL	177

## 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 = 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 form:

$$nRc_1c_2...c_n$$

where:

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 follow.

Valid	Invalid/Explanation
4RABCD	4RDK0: /The colon is not a Radix-50 character
6RΔTOΔΔΔ	

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.

# 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	
	.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

Table B-2 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.

# FORTRAN LANGUAGE SUMMARY

**Table B-2 Summary of FORTRAN IV Statements**

Form	Effect—Explanation	Manual Section
ACCEPT	See READ, Formatted Sequential See READ, List-Directed	7.3.1 7.3.3
Arithmetic/Logical Assignment v = e	Assigns the value of the arithmetic or logical expression to the variable  v - A variable name or an array element name e - An expression	3.1 3.2
Statement Function f([p[,p]...]) = e	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.  f - A symbolic name p - A symbolic name e - An expression	6.2.1
ASSIGN s TO v	Associates the statement label s with the integer variable v for later use in an assigned GO TO statement  s - A label of an executable statement v - An integer variable name	3.3
BACKSPACE u	Backspaces one record the currently open file on logical unit u  u - An integer expression	9.4
BLOCK DATA [nam]	Specifies the subprogram that follows as a BLOCK DATA subprogram  nam - A symbolic name	5.10
CALL f([a[,a]...])	Calls the subroutine subprogram with the name specified by f, passing the actual arguments a to replace the dummy arguments in the subroutine definition  f - A subprogram name or entry point a - An expression, an array name, or a procedure name	4.5 6.2
CLOSE (p[,p]...)	Closes the specified file. DISPOSE can be abbreviated DISP.  p - one of the following forms:	9.2

**Table B-2 (Cont.) Summary of FORTRAN IV Statements**

Form	Effect—Explanation	Manual Section
	UNIT = e DISPOSE = 'SAVE' DISPOSE = 'KEEP' DISPOSE = 'DELETE' DISPOSE = 'PRINT' ERR = s e - An integer expression s - A label of an executable statement	
COMMON <i>[/cb]/</i> nlist <i>[[,]/[cb]/nlist]...</i>	Reserves one or more blocks of storage space under the name specified to contain the variables associated with that block name cb - A common block name nlist - A list of one or more variable names, array names, or array declarators separated by commas	5.4
CONTINUE	Causes no processing	4.4
DATA nlist/clist <i>[[,] nlist/clist]...</i>	Initially stores elements of clist in the corresponding elements of nlist 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.	5.8
DECODE (c,f,b[,ERR = s])[list]	Reads c characters from buffer b and assigns values to the elements in the list, converted according to format specification f 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	7.5
DEFINE FILE u(m,n,U,v)[,u(m,n,U,v)]...	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 u - An integer variable or integer constant	9.7



# FORTRAN LANGUAGE SUMMARY

**Table B-2 (Cont.) Summary of FORTRAN IV Statements**

Form	Effect—Explanation	Manual Section
	<p>m - An integer variable or integer constant</p> <p>n - An integer variable or integer constant</p> <p>v - An integer variable name</p>	
DIMENSION a(d)[,a(d)]...	<p>Specifies storage space requirements for arrays</p> <p>a(d) - An array declarator</p>	5.3
DO s [,] v = e <sub>1</sub> ,e <sub>2</sub> [,e <sub>3</sub> ]	<p>Executes the DO loop by performing the following steps:</p> <ol style="list-style-type: none"> <li>1 Set v = e<sub>1</sub></li> <li>2 Execute all statements through statement number s</li> <li>3 Evaluate v = v + 3e</li> <li>4 Repeat steps 2 through 3 for the following iterations:</li> </ol> <p style="text-align: center;">MAX (1, INT((e<sub>2</sub> - e<sub>1</sub>)/e<sub>3</sub>) + 1)</p> <p>s - A label of an executable statement</p> <p>v - A variable name</p> <p>e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub> - Numeric expressions</p>	4.3
ENCODE (c,f,b[,ERR = s])[list]	<p>Writes c characters into buffer b, which contains the values of the elements of the list, converted according to format specification f</p> <p>c - An integer expression</p> <p>f - A format specifier</p> <p>b - A variable name, array name, or element name</p> <p>s - A label of an executable statement</p> <p>list - An I/O list</p>	7.5
ENDFILE u	<p>Writes an end-file record on logical unit u</p> <p>u - An integer expression</p>	9.6
END = s, ERR = s	<p>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 numbers when an end-of-file (END =) or error (ERR =) condition occurs.</p> <p>s - A label of an executable statement</p>	7.2.4

**Table B-2 (Cont.) Summary of FORTRAN IV Statements**

<b>Form</b>	<b>Effect—Explanation</b>	<b>Manual Section</b>
EQUIVALENCE (nlist)[,(nlist)]...	Assigns each of the names in nlist the same storage location  nlist - A list of two or more variable names, array names, or array element names, separated by commas. Subscript expressions must be constants.	5.6
EXTERNAL v[,v]...	Defines the names specified as subprograms  v - A subprogram name	5.7
FIND (u'r)	Positions the file on logical unit u to the record specified by r  u - An integer expression r - An integer expression	9.5
FORMAT (field specification,...)	Describes the format in which one or more records are to be transmitted; a statement label must be present	8.1 - 8.7.3
[typ] FUNCTION nam[*n][[(p[,p]...)]]	Begins a function subprogram, indicating the program name and any dummy argument names (p). An optional type specification can be included.  typ - A data type specifier nam - A symbolic name *n - A data type length specifier p - A symbolic name	6.2.2
GO TO s	Transfers control to statement number s	4.1.1
GO TO (slist)[,]e	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.  slist - A list of one or more statement labels separated by commas e - An integer expression	4.1.2
GO TO v [[,](slist)]	Transfers control to the statement most recently associated with v by an ASSIGN statement  v - An integer variable name	4.1.3

# FORTRAN LANGUAGE SUMMARY

**Table B-2 (Cont.) Summary of FORTRAN IV Statements**

Form	Effect—Explanation	Manual Section
	<p>slist - A list of one or more statement labels separated by commas</p>	
IF (e) s <sub>1</sub> ,s <sub>2</sub> ,s <sub>3</sub>	<p>Transfers control to statement s<sub>1</sub> depending on the value of e (if e is less than 0, control transfers to s<sub>1</sub>; if e equals 0, control transfers to s<sub>2</sub>; if e is greater than 0, control transfers to s<sub>3</sub>).</p> <p>e - An expression</p> <p>s - A label of an executable statement</p>	4.2.1
IF (e) st	<p>Executes the statement if the logical expression has a value of true</p> <p>e - An expression</p> <p>st - Any executable statement except a DO or logical IF</p>	4.2.2
IMPLICIT typ (a[,a]...)[,typ(a[,a]...)]...	<p>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</p> <p>typ - A data type specifier</p> <p>a - Either a single letter, or two letters in alphabetical order separated by a hyphen (i.e., X-Y)</p>	5.1
OPEN(par[,par]...)	<p>Opens a file on the specified logical unit according to the parameters specified by the keywords</p> <p>par - A keyword specification in one of the following forms:</p> <p>key</p> <p>key = value</p> <p>key - A keyword, as described below</p> <p>c - An array name, variable name, array element name, or literal string</p> <p>e - A numeric expression</p> <p>s - A label of an executable statement</p> <p>v - Opens a file on the specified logical unit according to the parameters specified by the keywords</p> <p>value - Depends on the keyword, as described below</p>	9.1

**Table B-2 (Cont.) Summary of FORTRAN IV Statements**

Form	Effect—Explanation	Manual Section
PAUSE[disp]	Suspends program execution and prints the display if one is specified  disp - A decimal digit string containing one to five digits, an octal constant, or a literal string	4.7
PRINT	See WRITE, Formatted Sequential See WRITE, List-Directed	7.3.2 7.3.4
PROGRAM nam	Specifies a name for the main program  nam - A symbolic name	5.9
READ (u,f[,END = s][,ERR = s])[list] READ f[,list] ACCEPT f[,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.  u - An integer expression f - A format specifier s - A label of an executable statement list - An I/O list	7.3.1
READ(u[,END = s][,ERR = s])[list]	Reads one unformatted record from logical unit u and assigns values to the elements in the list  u - An integer expression s - A label of an executable statement list - An I/O list	7.3.5

## Keyword Values for OPEN Statement

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

# FORTRAN LANGUAGE SUMMARY

**Table B-2 (Cont.) Summary of FORTRAN IV Statements**

<b>Form</b>	<b>Effect—Explanation</b>	<b>Manual Section</b>
READ(u'r[,ERR = s])[list]	Reads record r from logical unit u and assigns values to the elements in the list u - An integer expression r - An integer expression s - A label of an executable statement list - An I/O list	7.4.1
READ (u,*[,END = s][,ERR = s])list READ *,list ACCEPT *,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. u - An integer expression * - Denotes list-directed formatting s - A label of an executable statement list - An I/O list	7.3.3
RETURN	Returns control to the calling program from the current subprogram	4.6
REWIND u	Repositions logical unit u to the beginning of the currently opened file u - An integer expression	9.3
STOP [disp]	Terminates program execution and prints the display if one is specified disp - A decimal digit string containing one to five digits, an octal constant, or a literal string	4.8
SUBROUTINE nam([(p[,p]...)])	Begins a subroutine subprogram, indicating the program name and any dummy argument names (p) nam - A symbolic name p - A symbolic name	6.2.3
TYPE	See WRITE, Formatted Sequential See WRITE, List-Directed	7.3.2 7.3.4

**Table B-2 (Cont.) Summary of FORTRAN IV Statements**

<b>Form</b>	<b>Effect—Explanation</b>	<b>Manual Section</b>
Type Declaration 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  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.	5.2
VIRTUAL a(d)[,a(d)]...	Specifies storage space for arrays outside normal program address space  a(d) - An array declarator	5.5
WRITE (u,f[,ERR = s])[list] PRINT f[,list] TYPE f[,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.  u - An integer expression f - A format specifier s - A label of an executable statement list - An I/O list	7.3.2
WRITE (u[,ERR = s])[list]	Writes one unformatted record to logical unit u containing the values of the elements in the list  u - An integer expression s - A label of an executable statement label list - An I/O list	7.3.6
WRITE (u'r[,ERR = s])[list]	Writes record r to logical unit u containing the values of the elements in the list  u - An integer expression	7.4.2

# FORTRAN LANGUAGE SUMMARY

**Table B-2 (Cont.) Summary of FORTRAN IV Statements**

Form	Effect—Explanation	Manual Section
	r - An integer expression	
	s - A label of an executable statement label	
	list - An I/O list	
WRITE (u,*,[ERR = s])list PRINT *,list TYPE *,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. u - An integer expression * - Denotes list-directed formatting s - A label of an executable statement list - An I/O list	7.3.4

## B.3

## LIBRARY FUNCTIONS

Table B-3 lists the FORTRAN IV library functions. Superscripts in the table refer to notes which follow the table.

**Table B-3 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 <sup>1</sup>	Complex	Real
FLOAT(I)	Integer to Real conversion <sup>2</sup>	Integer	Real
IFIX(X)	Real to Integer conversion <sup>2</sup>	Real	Integer
SNGL(X)	Double to Real conversion <sup>2</sup>	Double	Real
DBLE(X)	Real to Double conversion <sup>2</sup>	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 $CMPLX(X,Y) = X + i \cdot Y$	Real	Complex

Truncation functions return the sign of the argument \*  
largest integer  $\leq |arg|$

<sup>1</sup>The absolute value of a complex number, (X,Y), is the real value:  $(X^2 + Y^2)^{1/2}$

<sup>2</sup>Functions that cause conversion of one data type to another type provide the same effect as the implied conversion in assignment statements. The function SNGL with a real argument and the function DBLE with a double precision argument return the value of the argument without conversion.

**Table B-3 (Cont.) FORTRAN Library Functions**

Form	Definition	Argument Type	Result Type
AIN(X)	Real to Real truncation <sup>3</sup>	Real	Real
INT(X)	Real to Integer truncation <sup>3</sup>	Real	Integer
IDINT(X)	Double to Integer truncation <sup>3</sup>	Double	Integer
Remainder functions return the remainder when the first argument is divided by the second.			
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; two 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.			
AMINO(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

<sup>3</sup>[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.



# FORTRAN LANGUAGE SUMMARY

**Table B-3 (Cont.) FORTRAN Library Functions**

Form	Definition	Argument Type	Result Type
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) <sup>4</sup>	$\log_e(Z)$	Complex	Complex
SQRT(X)	Square root of Real argument	Real	Real
DSQRT(X)	Square root of Double precision argument	Double	Double
CSQRT(Z) <sup>5</sup>	Square root of Complex argument	Complex	Complex
SIN(X)	Real sine	Real	Real
DSIN(X)	Double precision sine	Double	Double
CSIN(Z) <sup>6</sup>	Complex sine	Complex	Complex
COS(X)	Real cosine	Real	Real
DCOS(X)	Double precision cosine	Double	Double
CCOS(Z) <sup>6</sup>	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) <sup>7 8</sup>	Double precision arc tangent of (X/Y)	Double	Double

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

<sup>5</sup>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.

<sup>6</sup>The argument of SIN, DSIN, COS, DCOS, TAN, and DTAN must be in radians. The argument is treated modulo  $2\pi$ .

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

<sup>8</sup>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.

**Table B-3 (Cont.) FORTRAN Library Functions**

Form	Definition	Argument Type	Result Type
CONJG(Z)	Complex conjugate, if $Z = X + i \cdot Y$ , $\text{CONJG}(Z) = X - i \cdot Y$	Complex	Complex
RAN(I,J) <sup>9</sup>	Returns a pseudo-random number of uniform distribution over the range of 0 to 1	Integer	Real

---

<sup>9</sup>The argument for this function must be an integer variable or integer array element. The argument should initially be set to 0. The RAN function stores a value in the argument that it later uses to calculate the next random number. Resetting the argument to 0 regenerates the sequence. Alternate starting values generate different random number sequences.

---



---

# Index

---

## A

---

Absolute value  
  ABS • 6-9  
  integer constant • 2-4  
  non-zero double precision constant • 2-6  
  non-zero real constant • 2-5  
ACCEPT statement • 7-7  
  formatted sequential • 7-7  
ACCESS keyword • 9-4  
Access mode  
  direct • 7-3  
  sequential • 7-2  
Access modes • 7-2  
Adjustable arrays • 2-15, 6-3  
A field descriptor • 8-10  
All-blank field  
  value • 8-4, 8-5  
All-zero statement label • 1-6  
American National Standard FORTRAN X3.9-1966  
  • 1-1  
Apostrophe character • 2-9  
Area  
  1-byte storage • 2-3  
Argument  
  actual • 2-9, 2-10  
  lists • 2-1  
Arguments  
  actual • 2-14, 5-10, 6-1, 6-3, 6-9  
  dummy • 2-2, 4-10, 5-8, 6-2, 6-3  
    statement function • 6-5  
  subprogram • 2-12, 2-15, 6-1  
    rules governing • 6-2  
Arithmetic elements • 2-15  
Arithmetic expression  
  data type • 2-17  
Arithmetic operators • 2-15  
Array • 2-11  
  data type • 2-13  
  element • 2-11  
  name • 2-11  
  virtual • 1-2, 5-5  
Array declaration • 2-11  
Array declarator • 5-3  
Array declarators • 2-12  
Array elements  
  order • 5-12

Array names  
  unsubscripted • 2-14  
Array reference  
  subscripted • 2-13  
Array references  
  unsubscripted • 2-14  
Arrays  
  adjustable • 2-12, 6-3  
  array elements • 2-1  
  making equivalent • 5-9  
Array storage • 2-13  
Array subscript • 1-1  
Assignments  
  arithmetic • 2-3  
Assignment statement  
  arithmetic • 3-1  
  logical • 3-3  
Assignment statements • 3-1  
ASSIGN statement • 3-3  
ASSOCIATEVARIABLE keyword • 9-5  
Asterisk • 1-3

---

## B

---

BACKSPACE statement • 9-11  
Base  
  double precision constant • 2-6  
  real constant • 2-5  
BLOCK DATA statement • 5-13  
BLOCKSIZE keyword • 9-5  
BUFFERCOUNT keyword • 9-6

---

## C

---

C (letter) • 1-2  
CALL statement • 4-9  
  argument list • 4-9, 4-10  
  arguments • 4-10  
  Hollerith constants • 4-10  
  literal strings • 4-10  
Carriage control characters • 8-18  
CARRIAGECONTROL keyword • 9-6  
Character-per-Column  
  formatting • 1-4  
Character set • A-1

# Index

## Character set (cont'd.)

- ASCII • A-1
  - hexadecimal • A-1
  - octal • A-3
- FORTTRAN • 1-3, A-1
- Radix-50 • A-4
- CLOSE statement • 9-10
- Colon descriptor • 8-14
- Comment
  - indicator • 1-6
- Comments • 1-2
  - ! • 1-2
  - indicator • 1-2
  - line • 1-2
- Common blocks
  - extending • 5-10
- COMMON statement • 5-4
- Complex constants • 2-7
- Constant • 2-13
- Constants • 2-4
  - complex • 2-7
  - decimal integer • 2-4
  - double precision • 2-6
  - Hollerith • 2-4, 2-8
  - integer • 2-4
  - logical • 2-8
  - octal integer • 2-5
  - real • 2-5
- Continuation field • 1-6
- CONTINUE statement • 4-1, 4-9
- Control statements • 4-1

---

## D

- Data editing
  - complex • 8-15
- Data elements • 7-10
- DATA statement • 5-11
  - examples • 5-12
- Data type • 2-9
  - by implication • 2-11
  - complex • 2-3
  - double precision • 2-3
  - integer • 2-3
  - logical • 2-3
  - real • 2-3
- Data types • 2-1, 2-3
- Data type specification • 2-11
- Debugging statement
  - indicator • 1-6
- Decimal integer constants • 2-4

- DECODE statement • 7-1
- DECODE statements • 7-16
- DEFINE FILE statement • 9-12
- Delimiters • 7-11
- D field descriptor • 8-7
- DIMENSION statement • 5-3
- Direct access • 7-3
- Direct access I/O • 7-15
- Direct access record numbers • 7-3
- DISPOSE keyword • 9-6
- DO iteration control • 4-6
- DO lists
  - implied • 7-6
- Dollar sign descriptor • 8-14
- DO loop
  - control variable and parameters • 4-6
- DO loops
  - control transfers • 4-8
  - extended range • 4-8
  - nested • 4-7
- DO loop termination
  - completion • 4-6
  - transfer of control • 4-6
- DO statement • 4-1, 4-5
- Double precision constants • 2-6
- Dummy arguments • 5-8

---

## E

- E field descriptor • 8-6
- Element
  - array • 2-16
  - variable • 2-16
- Element repeaters • 7-11
- ENCODE statement • 7-1
- ENCODE statements • 7-16
- ENDFILE statement • 9-12
- End-of-file condition
  - parameters • 7-4
- END statement • 4-1, 4-11
- EQUIVALENCE statement • 5-8
- ERR keyword • 9-7
- Error condition
  - parameters • 7-4
- Exponent specifier
  - double precision constant • 2-7
  - real constant • 2-6
- Expression
  - arithmetic • 1-1, 2-13
  - subscript • 2-13

Expression operators • B-1  
 Expressions • 2-15  
   arithmetic • 2-15  
   logical • 2-15, 2-20  
   relational • 2-15, 2-19  
 EXTENDSIZE keyword • 9-7  
 EXTERNAL statement • 5-10

---

## F

---

F field descriptor • 8-4  
 Field descriptor  
   form • 8-1  
 Field descriptors • 8-1, 8-2  
   character • 8-1  
   complex • 8-1  
   default • 8-17  
   double precision • 8-1  
   editing and Hollerith constant • 8-1  
   format specification separators • 8-19  
   integer • 8-1  
   logical • 8-1  
   real • 8-1  
 Field separator  
   comma • 8-1  
   slash • 8-1  
 Field separators  
   external • 8-20  
 Field terminator  
   end of record • 8-20  
 Files • 7-2  
 FIND statement • 9-12  
 Format specification • 8-1  
 Format specification separators • 8-19  
 Format specifier  
   scale factor • 8-15  
 Format specifiers • 7-3  
 Format statement rules  
   general • 8-23  
   input • 8-24  
   output • 8-24  
   summary • 8-23  
 FORMAT statements • 8-1  
   external • 8-1  
   internal • 8-1  
 Formatting • 1-3  
 FORM keyword • 9-7  
 FORTRAN character set • A-1  
 FORTRAN coding form • 1-4  
 FORTRAN IV library functions • B-10

FORTRAN IV statements • B-1  
 FORTRAN library functions • 6-9  
 FORTRAN line  
   formatting • 1-3  
 FORTRAN programs  
   elements • 1-2  
 Function references  
   processor-defined • 6-1  
 Function subprograms • 6-6

---

## G

---

G field descriptor • 8-8  
 GO TO  
   computed • 1-2  
 GO TO statement • 4-1  
   assigned • 4-3  
   computed • 4-2  
   unconditional • 4-2

---

## H

---

Hexadecimal character set • A-1  
 H field descriptor • 8-11  
   input using • 8-21  
 Hollerith constants • 2-4, 2-8  
   data type rules • 2-9

---

## I

---

I/O  
   overview • 7-2  
 I/O lists • 7-5  
   format control interaction with • 8-22  
 I/O statement  
   formatted sequential • 7-7  
   list-directed • 7-7  
   sequential • 7-7  
   unformatted sequential • 7-7  
 I/O statement components • 7-3  
 I field descriptor • 8-2  
 IF statement • 4-1, 4-3  
   arithmetic • 4-3, 4-4  
   logical • 4-3, 4-4  
 IF statements  
   examples • 4-4  
 IMPLICIT statement • 5-1

## Index

INITIALSIZE keyword • 9-7

Input

using H field descriptor • 8-21

Input statement

ACCEPT • 7-1

READ • 7-1

unformatted direct access

READ • 7-15

Input statements

formatted sequential

ACCEPT • 7-7

READ • 7-7

list-directed

ACCEPT • 7-10

READ • 7-10

Integer constant

absolute value • 2-4

Integer constants • 2-4

---

## L

L field descriptor • 8-9

Library functions • 6-9

FORTTRAN • 6-9

Lines • 1-2

List-directed input statement

example • 7-12

List-directed output statement

PRINT • 7-12

TYPE • 7-12

WRITE • 7-12

Literal string

apostrophe • 8-12

Literal strings • 1-1, 1-3, 1-5, 1-6, 2-4, 2-8

Logical constants • 2-8

Logical operator

.AND. • 2-20

.EQV. • 2-20

.NOT. • 2-20

.OR. • 2-20

.XOR. • 2-20

Logical unit numbers • 7-3

---

## M

MAXREC keyword • 9-8

---

## N

NAME keyword • 9-8

Names

symbolic • 2-1, 2-2

NOSPANBLOCKS keyword • 9-8

Null elements • 7-11

---

## O

Octal character set • A-3

Octal integer constants • 2-5

O field descriptor • 8-3

OPEN statement • 9-1

Operators

arithmetic • 2-15

binary • 2-16

Output statement

PRINT • 7-1

TYPE • 7-1

unformatted direct access

WRITE • 7-16

WRITE • 7-1

Output statements

formatted sequential

PRINT • 7-8

TYPE • 7-8

WRITE • 7-8

---

## P

Parentheses

use of • 2-17

PAUSE statement • 4-1, 4-10

Positioning characters • 8-18

PROGRAM statement • 5-13

Program unit structure • 1-7

---

## Q

Q field descriptor • 8-13

---

## R

---

Radix-50  
     character set • A-4  
     constants • A-4  
 READONLY keyword • 9-8  
 READ statement  
     formatted sequential • 7-7  
 Record numbers  
     direct access • 7-3  
 Records • 7-2  
 RECORDSIZE keyword • 9-9  
 Relational operators • 2-19  
 Repeat counts • 8-17  
     group • 8-17  
 RETURN statement • 4-1, 4-10  
 REWIND statement • 9-11  
 Runtime format • 8-21

---

## S

---

Scale factor • 8-15  
 Sequence number field • 1-7  
 Sequential access • 7-2  
 Sequential I/O • 7-7  
 SHARED keyword • 9-9  
 Simple lists • 7-5  
 Statement  
     label • 1-6  
     number • 1-6  
 Statement field • 1-6  
 Statement functions • 6-5  
 Statement label  
     field • 1-6  
 Statements • 1-2  
 STOP statement • 4-1, 4-11  
 Storage elements • 5-3  
 Strings • 1-2  
     literal • 2-8  
 Subprogram arguments • 6-1  
 Subprograms • 6-1  
     functions • 6-1  
     statement functions • 6-1  
     subroutines • 6-1  
     supplied • 6-1  
     user-written • 6-1, 6-4  
 Subroutine subprograms • 6-8  
 Subscripts • 2-13  
 Symbolic names • 2-1, 2-2

---



---

## T

---

Tab-character  
     formatting • 1-5  
 T field descriptor • 8-13  
 Type declaration statement • 5-2  
 TYPE keyword • 9-9  
     default • 9-9  
     NEW • 9-9  
     OLD • 9-9  
     SCRATCH • 9-9  
     UNKNOWN • 9-9

---

## U

---

Unformatted sequential input statement  
     READ • 7-14  
 Unformatted sequential output statement  
     WRITE • 7-14  
 UNIT keyword • 9-10  
 Unit numbers  
     logical • 7-3

---

## V

---

Variable • 2-13  
 Variables • 2-10  
 Virtual array • 5-5  
     references  
         subprograms • 5-7  
         restrictions • 5-6  
 VIRTUAL statement • 5-5

---

## X

---

X field descriptor • 8-12

---







