

FORTTRAN IV

Part I. Language

Part II. Object Time System

Part III. Science Library

ADVANCED SOFTWARE SYSTEM Programmer's Reference Manual

Order No. DEC-9A-AF4B-D from Program Library, Maynard, Mass. Price: \$2.50
Direct comments concerning this manual to Software Quality Control, Maynard, Mass.

Copyright 1968 by Digital Equipment Corporation

PREFACE

This manual describes the FORTRAN IV language and compiler system for the PDP-9 computer. It provides the user with the necessary information for writing FORTRAN programs for compilation and execution with the PDP-9 Advanced Software System. The manual is divided into three parts: Basic FORTRAN Language (Part I), FORTRAN Object Time System (Part II), and the FORTRAN Science Library (Part III).

Part I, Basic FORTRAN IV Language, is divided into chapters as follows:

- Chapter 1. Introduction
- Chapter 2. Elements of the FORTRAN Language
- Chapter 3. Arithmetic Statements
- Chapter 4. Control Statements
- Chapter 5. Input/Output Statements
- Chapter 6. Specification Statements
- Chapter 7. Subprograms

Part I is intended to familiarize the user with specific PDP-9 FORTRAN coding procedures. Several excellent texts are available for a more elementary approach to FORTRAN programming. "A Guide to FORTRAN IV Programming," by Daniel D. McCracken (published by John Wiley and Sons, Inc.) is recommended.

Part II, FORTRAN IV Object Time System, describes the group of subprograms that process compiled FORTRAN statements, particularly I/O statements, at execution time.

Part III, PDP-9 Science Library, provides detailed descriptions of the intrinsic functions, external functions, subfunctions, and arithmetic routines in the PDP-9 Science Library.

PDP-9 FORTRAN IV is essentially the language specified by the United States of America Standards Institute (X3.9 - 1966) with the exceptions noted in Appendix 2 at the end of Part I of this manual.

CONTENTS

	<u>Page</u>
PART I	
CHAPTER 1	
INTRODUCTION	
1.1	FORTRAN 1-1
1.2	Source Program Format 1-1
1.2.1	Card Format (IBM Model 029 Keypunch Codes) 1-2
1.2.2	Paper Tape Format 1-2
CHAPTER 2	
ELEMENTS OF THE FORTRAN LANGUAGE	
2.1	Constants 1-5
2.1.1	Integer Constants 1-5
2.1.2	Real Constants 1-5
2.1.3	Double-Precision Constants 1-6
2.1.4	Logical Constants 1-6
2.1.5	Hollerith Constants 1-7
2.2	Variables 1-7
2.2.1	Variable Types 1-7
2.2.2	Integer Variables 1-7
2.2.3	Real Variables 1-8
2.2.4	Double-Precision and Logical Variables 1-8
2.3	Arrays and Subscripts 1-8
2.3.1	Arrangement of Arrays in Storage 1-8
2.3.2	Subscript Expressions 1-9
2.3.3	Subscripted Variables 1-9
2.4	Expressions 1-9
2.4.1	Arithmetic Expressions 1-9
2.4.2	Relational Expressions 1-11
2.4.3	Logical Expressions 1-12
2.5	Statements 1-13
CHAPTER 3	
ARITHMETIC STATEMENTS 1-14	

CONTENTS (Cont)

PageCHAPTER 4
CONTROL STATEMENTS

4.1	Unconditional GO TO Statements	I-17
4.2	ASSIGN Statement	I-17
4.3	Assigned GO TO Statement	I-17
4.4	Computed GO TO Statement	I-18
4.5	Arithmetic IF Statement	I-18
4.6	Logical IF Statement	I-18
4.7	DO Statement	I-19
4.8	CONTINUE Statement	I-21
4.9	PAUSE Statement	I-21
4.10	STOP Statement	I-21
4.11	END Statement	I-21

CHAPTER 5
INPUT/OUTPUT STATEMENTS

5.1	General I/O Statements	I-24
5.1.1	Input/Output Argument Lists	I-24
5.1.2	READ Statement	I-24
5.1.3	WRITE Statement	I-25
5.2	FORMAT Statements	I-26
5.2.1	Specifying FORMAT	I-26
5.2.2	Conversion of Numeric Data	I-28
5.2.3	P-Scale Factor	I-30
5.2.4	Conversion of Alphanumeric Data	I-31
5.2.5	Logical Fields, L Conversion	I-31
5.2.6	Blank Fields, X Conversion	I-31
5.2.7	FORTRAN Statements Read in at Object Time	I-32
5.2.8	Printing of a Formatted Record	I-32
5.3	Auxiliary I/O Statements	I-33
5.3.1	BACKSPACE Statement	I-33
5.3.2	REWIND Statement	I-33
5.3.3	ENDFILE Statement	I-33

CONTENTS (Cont)

	<u>Page</u>
CHAPTER 6 SPECIFICATION STATEMENTS	
6.1	I-35
6.2	I-35
6.3	I-36
6.4	I-37
6.5	I-37
6.6	I-38
CHAPTER 7 SUBPROGRAMS	
7.1	I-39
7.2	I-40
7.3	I-42
7.4	I-44
7.5	I-45
7.5.1	I-45
APPENDIX 1 SUMMARY OF PDP-9 FORTRAN IV STATEMENTS	
	I-47
APPENDIX 2 A NOTE ON PDP-9 FORTRAN IV	
	I-49
APPENDIX 3 FORTRAN IV AND MACRO-9 LINKAGE	
	I-51
APPENDIX 4 CHAINING FORTRAN IV PROGRAMS	
	I-53
APPENDIX 5 FORTRAN IV ERROR LIST	
	I-55
ILLUSTRATION	
I-1	I-3
TABLES	
I-1	I-41
I-2	I-43

PDP-9 FORTRAN IV

CONTENTS

Page

PART II FORTRAN IV OBJECT - TIME SYSTEM

Introduction	II-1
OTS Binary Coded Input/Output	II-3
OTS Binary Input/Output	II-6
OTS Auxiliary Input/Output	II-8
OTS IOPS Communication	II-10
OTS Calculate Array Element Address	II-12
OTS Computed GOTO	II-14
OTS Stop	II-15
OTS Pause	II-16
OTS Octal Print	II-17
OTS Errors	II-18
File Commands	II-19

TABLE

		<u>Page</u>
II-1	OTS Errors	II-2
III-1	PDP-9 Science Library	III-5

PART III PDP-9 SCIENCE LIBRARY

Page

Introduction	III-1
Intrinsic Functions	III-1
External Functions	III-1
External Functions	III-1
Sub-Functions	III-1
The Arithmetic Package	III-1
Accumulators	III-2
A-Register	III-2
Floating Accumulator	III-2
Held Accumulator	III-3
Calling Sequences	III-3
PDP-9 Science Library Algorithm Descriptions	III-15

CHAPTER 1 INTRODUCTION

1.1 FORTRAN

Each type of digital computer is designed to respond to certain machine language codes. The codes are different for each type of computer. FORTRAN makes it unnecessary for the scientist or engineer to learn the machine language for specific computers. Using FORTRAN, he can write programs in a simple language that adapts easily to scientific usage. The FORTRAN language is composed of mathematical-like statements, constructed in accordance with precisely formulated rules. A FORTRAN program consists of meaningful sequences of FORTRAN statements that direct the computer to perform specific operations and calculations. A program written using FORTRAN statements is called a source program. It must be translated by the FORTRAN compiler program before execution. The translated version of the program is referred to as an object program. It is in a binary-coded form that the machine can understand.

1.2 SOURCE PROGRAM FORMAT

The FORTRAN character set consists of the 26 letters:

A, B, C, D, E, F, G, H, I, J, K, L, M,
N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

the 10 digits:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

and 11 special characters:

Blank	
Equals	=
Plus	+
Minus	-
Asterisk	*
Slash	/
Left Parenthesis	(
Right Parenthesis	)
Comma	,
Decimal Point	.
Dollar Sign	\$

1.2.1 Card Format (IBM Model 029 Keypunch Codes)

The FORTRAN source program is written on a standard FORTRAN coding sheet (Figure 1-1) which consists of the following fields: statement number field, line continuation field, statement field, and identification field.

The FORTRAN statement is written in columns 7-72. If the statement is too long for one line, it can be continued in the statement field of as many lines as necessary if column 6 of each continuation line contains any character other than blank or zero. There are two exceptions to this rule: (1) the DO statement must be on one line; and (2) the equal sign (=) of an assignment statement must appear on the first line.

For one statement to be referenced by another, a statement number is placed in columns 1 through 5 of the first line of that statement. This number is made up of digits only, and may contain from one to five digits. Leading zeros and all blanks in this field are ignored. The statement numbers are used for identification only, and may be assigned in any order.

The FORTRAN compiler ignores the last eight columns (columns 73 through 80) which may be used for program identification, sequencing, or any other purpose desired by the user. Comments may be included in the program by putting a "C" in column 1 of each line containing a comment (or continuation of a comment). The compiler ignores these comments except for printing them.

Blanks may be used to aid readability of a FORTRAN statement, except where indicated in this manual.

1.2.2 Paper Tape Format

When FORTRAN source program statements are prepared on paper tape, the sequence of characters is exactly the same as for card input, and each line is terminated with a carriage return, line feed sequence.

A statement number (all digits) may be written as the first five characters, or a "C" may be the first character to indicate a comment line or a continuation of a comment line. For statement continuation lines, any numeric character other than blank or zero is written as the sixth character. The seventh character begins the statement and must be alphabetic. Each line is terminated with a carriage return, line feed.

The TAB key can increase the speed of writing FORTRAN statements on paper tape. A TAB followed by an alphabetic character begins the statement in column 7. A TAB followed by a digit places the digit in column 6, indicating a statement continuation line. A statement number less than five digits, followed by a TAB, places the next character in column 6 if it is a digit, or in column 7 if it is a letter.

CODER	DATE	PAGE
PROBLEM		

[illegible]

I-3

PDP-9 FORTRAN IV

If it is desired to have a title at the beginning of the tape for listing purposes, it must be entered as a comment line.

CHAPTER 2 ELEMENTS OF THE FORTRAN LANGUAGE

2.1 CONSTANTS

There are five types of constants allowed in the FORTRAN source program: integer, real, double-precision, logical, and Hollerith.

2.1.1 Integer Constants

An integer constant is a number written without a decimal point, consisting of one to six decimal digits. A + or - sign preceding the number is optional. The magnitude of the constant must be less than or equal to $131071 (2^{17} - 1)$.

Example:

```
+97
0
-2176
576
```

If the magnitude $> 2^{17} - 1$, an error message will be output. Negative numbers are represented in 2's complement notation.

2.1.2 Real Constants (6-decimal-digit accuracy)

A real constant is an integer, fraction, or mixed format number and may be written in the following forms:

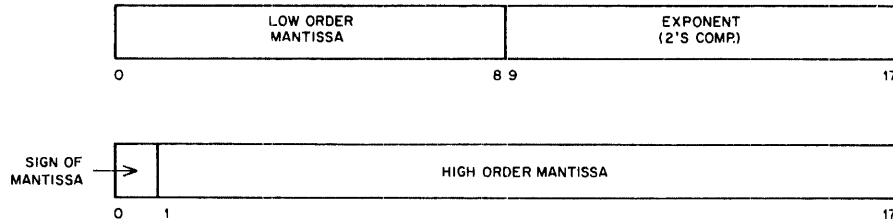
- a. A number consisting of one to six significant decimal digits with a decimal point included someplace within the constant. A + or - sign preceding the number is optional.
- b. A number followed by the letter E, indicating a decimal exponent, and a 1- or 2-digit constant with magnitude less than 76* indicating the appropriate power of 10. A + or - sign may precede the scale factor. The decimal point is not necessary in real constants having a decimal exponent.

Example:

```
352.
+12.03
-.0054
5.E-3
+5E7
```

*If the adjusted magnitude exceeds 75, an error results. .999999E75 is legal, but 999.999E73 is illegal.

Real constants are stored in two words in the following format:



Negative mantissae are indicated with a change of sign.

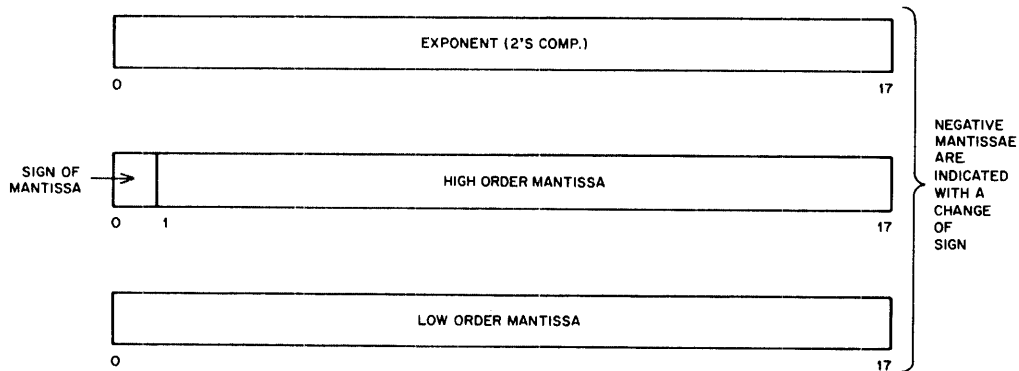
2.1.3 Double-Precision Constants (9-decimal-digit accuracy)

A double-precision constant is written as a real number followed by a decimal exponent, indicated by the letter D and a 1- or 2-digit constant with magnitude not greater than 76. A + or - sign may precede the constant and may also precede the scale factor. A decimal point within the constant is optional. A double-precision constant is interpreted identically to a real constant, the only difference being that the degree of accuracy is greater.

Example:

```
-3.0D0
987.6542D15
32.123D+7
```

Double-precision constants are stored in three PDP-9 words:



2.1.4 Logical Constants

The two logical constants are the words TRUE and FALSE, each both preceded and followed by a decimal point.

```
.TRUE.  777777
.FALSE.  0
```

2.1.5 Hollerith Constants

A Hollerith constant is written as an unsigned integer constant, whose value, n , must be equal to or greater than one and less than or equal to five, followed by the letter H, followed by exactly n characters, which are the Hollerith data. Any FORTRAN character, including blank, is acceptable. The Hollerith constants are used only in CALL and DATA statements and must be associated with real variable names. The Hollerith constants are packed in 7-bit ASCII, five, per two words of storage with the rightmost bit always zero.

Examples:

```
1HA
4HA$C
```

2.2 VARIABLES

A variable is a symbolic representation of a numeric quantity whose values may change during the execution of a program either by assignment or by computation. The symbol's representation, or name of the FORTRAN variable consists of from one to six alphanumeric (alphabetic and numeric) characters, the first of which must be alphabetic.

Example:

```
X = Y + 10.    Both X and Y are variables; X by computation, and Y by
               assignment in some previous statement.
```

```
TEST
GAMMA
X12345
```

NOTE

If three or less characters are used for each symbol, considerable core space can be saved.

2.2.1 Variable Types

Variables in FORTRAN may represent one of the following types of quantities: integer, real, double-precision, or logical. This corresponds to the type of constant the variable is supposed to represent.

2.2.2 Integer Variables

Variable names beginning with the letters I, J, K, L, M, or N are considered to be integer variables. If the first letter is not one of the above letters, it is an integer variable only if it was named in a previous integer type specification statement.

2.2.3 Real Variables

Variable names beginning with letters other than I, J, K, L, M, or N are considered to be real variables. If the first character is one of the above letters, it is a real variable only if it was named in a previous real type specification statement.

2.2.4 Double-Precision and Logical Variables

A type specification statement is the only way to assign a variable value to one of these two types. This is done with either a double precision statement or a logical statement.

2.3 ARRAYS AND SUBSCRIPTS

An array is an ordered set of data identified by a symbolic name. Each individual quantity in this set of data is referred to in terms of its position within the array. This identifier is called a subscript. For example,

A (3)

represents the third element in a one-dimensional array named A. To generalize further, in an array A with n elements, A (I) represents the Ith element of the array A where $I = 1, 2, \dots, n$.

FORTRAN allows for one, two, and three-dimensional arrays, so there can be up to three subscripts for the array, each subscript separated from the next by a comma. For example,

B (1, 3)

represents the value located in the first row and the third column of a two-dimensional array named B. A dimension statement defining the size of the array (i.e., the maximum values each of its subscripts can attain) must precede the array in the source program.

2.3.1 Arrangement of Arrays in Storage

Arrays are stored in column order in ascending absolute storage locations. The array is stored with the first of its subscripts varying most rapidly and the last varying least rapidly. For example, a three-dimensional array A, defined in a DIMENSION statement as A (2,2,2) will be stored sequentially in this order:

A(1,1,1)	↓ ascending absolute storage locations
A(2,1,1)	
A(1,2,1)	
A(2,2,1)	
A(1,1,2)	
A(2,1,2)	
A(1,2,2)	
A(2,2,2)	

2.3.2 Subscript Expressions

Subscripts may be written in any of the following forms:

$$\begin{array}{l} V \\ C \\ V + k \\ V - k \\ C * V \\ C * V + k \\ C * V - k \end{array}$$

where C and k represent unsigned integer constants and V represents an unsigned integer variable.

Example:

$$\begin{array}{l} I \\ I3 \\ I\text{MOST} + 3 \\ I\text{LAST} - 1 \\ 5 * I\text{FIRST} \\ 2 * J + 9 \\ 4 * M1 - 7 \end{array}$$

2.3.3 Subscripted Variables

A subscripted variable is a variable name followed by a pair of parentheses which contain one to three subscripts separated by commas.

Example:

$$\begin{array}{l} A(I) \\ B(I, J - 3) \\ \text{BETA}(5 * J + 9, K + 7, 6 * \text{JOB}) \end{array}$$

2.4 EXPRESSIONS

An expression is a combination of elements (constants, subscripted or nonsubscripted variables, and functions) each of which is related to another by operators and parentheses. An expression represents one single value which is the result of the calculations specified by the values and operators that make up the expression. The FORTRAN language provides two kinds of expressions: arithmetic and logical.

2.4.1 Arithmetic Expressions

An arithmetic expression consists of arithmetic elements joined by the arithmetic operators +, -, *, /, and **, which denote addition, subtraction, multiplication, division, and exponentiation, respectively. An expression may consist of a single element (meaning a constant, a variable, or a

function name). An expression enclosed in parentheses is considered a single element. Compound expressions use arithmetic operators to combine single elements.

2.4.1.1 Mode of an Expression - The type of quantities making up an expression determines its mode; i.e., a simple expression consisting of an integer constant or an integer variable is said to be in the integer mode. Similarly, real constants or variables produce a real mode of expression, and double-precision constants or variables produce a double-precision mode. The mode of an arithmetic expression is important because it determines the accuracy of the expression.

In general, variables or constants of one mode cannot be combined with variables or constants of another mode in the same expression. There are, however, exceptions to this rule.

a. The following examples show the modes of the valid arithmetic expressions involving the use of the arithmetic operators +, -, *, and /. I, R, and D indicate integer, real, and double-precision variables or constants. A + is used to indicate any one of the four operators:

I + I	Integer result
R + R	Real result
R + D	} Double-precision result
D + R	
D + D	

b. When raising a value to a power, the mode of the power may be different than that of the value being raised. The following examples show the modes of the valid arithmetic expressions using the arithmetic operator **. As above, I, R, and D indicate integer, real, and double-precision.

I**I	Integer result
R**I	} Real result
R**R	
R**D	} Double-precision result
D**I	
D**R	
D**D	

The subscript of a subscripted variable, which is always an integer quantity, does not affect the mode of the expression.

2.4.1.2 Hierarchy of Operations - The order in which the operations of an arithmetic expression are to be computed is based on a priority rating. The operator with the highest priority takes precedence over other operators in the expression. Parentheses may be used to determine the order of computation. If no parentheses are used, the order is understood to be as follows:

- a. Function reference
- b. ******(Exponentiation)
- c. Unary minus evaluation
- d. ***** and **/**(multiplication and division)
- e. **+** and **-**(addition and subtraction)

Within the same priority, operations are computed from left to right.

Example:

$$\text{FUNC} + \text{A} * \text{B} / \text{C} - \text{D}(\text{I}, \text{J}) + \text{E} ** \text{F} * \text{G} - \text{H}$$

interpreted as,

$$\text{FUNC} + ((\text{A} * \text{B}) / \text{C}) - \text{D}(\text{I}, \text{J}) + (\text{E}^{\text{F}} * \text{G}) - \text{H}$$

2.4.1.3 Rules for Constructing Arithmetic Expressions -

- a. Any expression may be enclosed in parentheses.
- b. Expressions may be preceded by a **+** or **-** sign.
- c. Simple expressions may be connected to other simple expressions to form a compound expression, provided that:
 - (1) No two operators appear together.
 - (2) No operator is assumed to be present.
- d. Only valid mode combinations may be used in an expression (described under Mode of an Expression, Section 2.4.1.1).
- e. The expression must be constructed so that the priority scheme determines the order of operation desired (described in Section 2.4.1.2, Hierarchy of Operations).

Examples of arithmetic expressions follow:

```

3
A(I)
B + 7.3
C*D
A + (B*C) - D**2 + E/F

```

2.4.2 Relational Expressions

A relational expression is formed with the arithmetic expressions separated by a relational operator. The result value is either true or false depending upon whether the condition expressed by the relational operator is met or not met. The arithmetic expressions may both be of the integer mode or they may be a combination of real and/or double-precision. No other mode combinations are legal. The relational operators must be preceded by and followed by a decimal point. They are:

.LT.	Less than (<)
.LE.	Less than or equal to ($\leq$)
.EQ.	Equal to (=)
.NE.	Not equal to ($\neq$)
.GT.	Greater than (>)
.GE.	Greater than or equal to ($\geq$)

Examples:

```

N .LT. 5
DELTA + 7.3 .LE. B/3E7
(KAPPA + 7/5 .NE. IOTA
1.736D-4 .GT. BETA
X.GE. Y*Z**2

```

2.4.3 Logical Expressions

A logical expression consists of logical elements joined by logical operators. The value is either true or false. The logical operator symbols must be preceded by and followed by a decimal point.

They are:

.NOT.	Logical negation. Reverses the state of the logical quantity that follows.
.AND.	Logical AND generates a logical result (TRUE or FALSE) determined by two logical elements as follows: T .AND. T generates T T .AND. F generates F F .AND. T generates F F .AND. F generates F
.OR.	Logical OR generates a logical result determined by two logical elements as follows: T .OR. T generates T T .OR. F generates T F .OR. T generates T F .OR. F generates F

2.4.3.1 Rules for Construction Logical Expression -

- A logical expression may consist of a logical constant, a logical variable, a reference to a logical function, a relational expression, or a complex logical expression enclosed in parentheses.
- The logical operator .NOT. need only be followed by a logical expression, while the logical operators .AND. and .OR. must be both preceded by and followed by a logical expression for more complex logical expressions.
- Any logical expression may be enclosed in parentheses. The logical expression following the logical operator .NOT. must be enclosed in parentheses if it contains more than one quantity.

d. No two logical operators may appear in sequence, not separated by a comma or parenthesis unless the second operator is `.NOT.` In addition, no two decimal points may appear together, not separated by a comma or parenthesis, unless one belongs to a constant and the other to a relational operator.

2.4.3.2 Hierarchy of Operations - Parentheses may be used as in normal mathematical notation to specify the order of operations. Within the parentheses, or where there are no parentheses, the order in which the operations are performed is as follows:

- a. Evaluation of functions
- b. `**`(Exponentiation)
- c. Evaluation of unary minus quantities
- d. `*` and `/` (multiplication and division)
- e. `+` and `-` (addition and subtraction)
- f. `.LT.`, `.LE.`, `.EQ.`, `.NE.`, `.GT.`, `.GE.`
- g. `.NOT.`
- h. `.AND.` and `.OR.`
- i. `=` Replacement operator

Unlike an arithmetic expression where sequence of elements of the same priority (i.e., operations being performed from left to right) is important for the end result of the expression, the order of operation within the same priority in logical and relational expressions is unimportant.

2.5 STATEMENTS

Statements specify the computations required to carry out the processes of the FORTRAN program. There are four categories of statements provided for by the FORTRAN language:

- a. Arithmetic statements define a numerical calculation.
- b. Control statements determine the sequence of operation in the program.
- c. Input/output statements are used to transmit information between the computer and related input/output devices.
- d. Specification statements define the properties of variables, functions, and arrays appearing in the source program. They also enable the user to control the allocation of storage.

CHAPTER 3 ARITHMETIC STATEMENTS

An arithmetic statement is a mathematical equation written in the FORTRAN language which defines a numerical or logical calculation. It directs the assignment of a calculated quantity to a given variable. An arithmetic statement has the form

$$V = E$$

where V is a variable (integer, real, double-precision, or logical, subscripted or unsubscripted) or any array element name; = means replacement rather than equivalence, as opposed to the conventional mathematical notation; and E is an expression.

In some cases, the mode of the variable may be different from that of the expression. In such cases an automatic conversion takes place. The rules for the assignment of an expression E to a variable V are as follows:

<u>V Mode</u>	<u>E Mode</u>	<u>Assignment Rule</u>
Integer	Integer	Assign
Integer	Real	Fix and assign
Integer	Double-precision	Fix and assign
Real	Integer	Float and assign
Real	Real	Assign
Real	Double-precision	Double-precision evaluate and real assign
Double-precision	Integer	Double-precision float and assign
Double-precision	Real	Double-precision evaluate and assign
Double-precision	Double-precision	Assign
Logical	Logical	Assign

Mode conversions involving logical quantities are illegal unless the mode of both V and E is logical. Examples of an assignment statement:

```
ITEM = ITEM + 1
A(I) = B(I) = ASSIN (C (I) )
V = .FALSE.
X = A.GT.B.AND.C .LE. G
A = B
```


CHAPTER 4
CONTROL STATEMENTS

The statements of a FORTRAN program normally are executed as written. However, it is frequently desirable to alter the normal order of execution. Control statements give the FORTRAN user this capability. This section discusses the reasons for control statements and the ways in which they may be used.

4.1 UNCONDITIONAL GO TO STATEMENTS

The form of the unconditional GO TO statement is

GO TO n

where n is a statement number. Upon the execution of this statement, control is transferred to the statement identified by the statement number, n, which is the next statement to be executed.

Example:

GO TO 17

4.2 ASSIGN STATEMENT

The general form of an ASSIGN statement is

ASSIGN n TO i

where n is a statement number and i is a nonsubscripted integer variable name which appears in a subsequently executed assigned GO TO statement. The statement number, n, is the statement to which control will be transferred after the execution of the assigned GO TO statement.

Example:

ASSIGN 27 TO ITEST

4.3 ASSIGNED GO TO STATEMENT

Assigned GO TO statements have the form

GO TO i, (n₁, n₂, ..., n_m)

where i is a nonsubscripted integer variable reference appearing in a previously executed ASSIGN statement, and n₁, n₂, ..., n_m are the statement numbers which the ASSIGN statement may legally assign to i. Examples:

ASSIGN 13 TO KAPPA
GO TO KAPPA, (1, 13, 72, 100, 35)

There is no object time checking to ensure that the assignment is one of the legal statement numbers.

4.4 COMPUTED GO TO STATEMENT

The format of a computed GO TO statement is

$$\text{GO TO } (n_1, n_2, \dots, n_m), i$$

where $n_1, n_2, \dots, n_m$ are statement numbers and i is an integer variable reference whose value is greater than or equal to 1 and less than or equal to the number of statement numbers enclosed in parentheses. If the value of i is out of this range, the statement is effectively a CONTINUE statement.

Example:

$$\text{GO TO } (3, 17, 25, 50, 66), \text{ITEM}$$

If the value of ITEM is 2 at the time this GO TO statement is executed, the statement to which control is transferred is the statement whose number is second in the series, i.e., statement number 17.

4.5 ARITHMETIC IF STATEMENT

The form of the arithmetic IF statement is

$$\text{IF } (e) \ n_1, n_2, n_3$$

where e is an arithmetic expression and n_1, n_2, n_3 are statement numbers. The IF statement evaluates the expression in parentheses and transfers control to one of the referenced statements. If the value of the expression (e) is less than, equal to, or greater than zero, control is transferred to n_1, n_2 , or n_3 respectively.

Example:

$$\text{IF } (\text{AUB}(\text{I}) - \text{B}^*\text{D}) \ 10, 7, 23$$

4.6 LOGICAL IF STATEMENT

The general format of a logical IF statement is

$$\text{IF } (e) \ s$$

where e is a logical expression and s is any executable statement other than a DO statement or another logical IF statement. The logical expression is evaluated, and different statements are executed depending upon whether the expression is true or false. If the logical expression e is true, statement s is executed and control is then transferred to the following statement (unless the statement is a GO TO statement or an arithmetic IF statement, in which cases control is transferred as indicated; or the statement s is a CALL statement, in which case control is transferred to the next statement after return from the subprogram). If the logical expression e is false, statement s is ignored and control is transferred to the statement following the IF statement.

Example:

```
IF (L1) I = I + 1
IF (L.LE.k) GO TO 17
IF (LOG.AND. (.NOT.LOG1) ) IF (X) 3,5,5
```

4.7 DO STATEMENT

The DO statement is a command used to execute repeatedly a specified series of statements. The general format of the DO statement is

$$\text{DO } n \text{ } i = m_1, m_2, m_3$$

or

$$\text{DO } n \text{ } i = m_1, m_2$$

where n is a statement number representing the terminal statement or the end of the "range"; i is a non-subscripted integer variable known as the "index"; and m_1 , m_2 , and m_3 are unsigned nonzero integer constants or nonsubscripted integer variables, which represent the "initial," "final," and "increment" values of the index. If m_3 is omitted, as in the second form of the DO statement, its value is assumed to be 1.

The DO statement is a command used to execute repeatedly a group of statements following it, up to and including statement n . The initial value of i is m_1 (m_1 must be less than or equal to m_2). Each succeeding time the statements are operated, i is increased by the value of m_3 . When i is greater than m_2 , control passes to the statement following statement number n .

The range of a DO statement is a series of statements to be executed repeatedly. It consists of all statements immediately following the DO, up to and including statement n . Any number of statements may appear between the DO and statement n . The terminal statement (statement n) may not be a GO TO (of any form), an arithmetic IF, a RETURN, a STOP, a PAUSE, or a DO statement, or a logical IF statement containing any of these forms.

The index of a DO is the integer variable i which is controlled by the DO statement in such a way that its initial value is set to m_1 , and is increased each time the range of statements is executed by m_3 , until a further incrementation would cause the value of m_2 to be exceeded. Throughout the range of the DO, the index is available for computation either as an ordinary integer variable or as the variable of a subscript. However, the index may not be changed by any statement within the DO range.

The initial value is the value of the index at the time the range is executed for the first time.

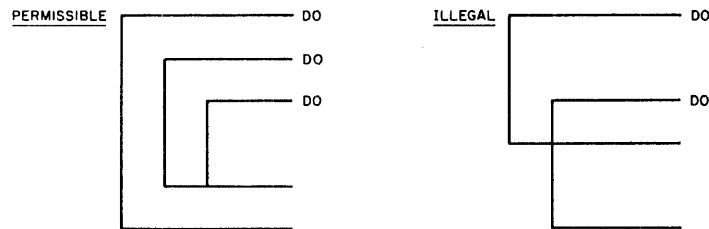
The final value is the value which the index must not exceed. When the condition is satisfied the DO is completed and control passes to the first executable statement following statement n .

The increment is the amount by which the index is to be increased after each execution of the range. If the increment is omitted, a value of 1 is implied.

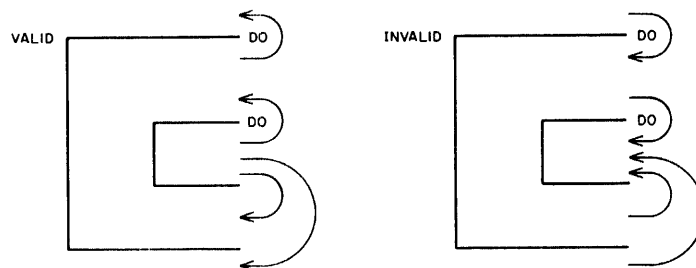
Example:

```
DO 72 I = 1, 10, 2
DO 15K = 1, 5
DO 23 I = 1, 11, 4
```

Any FORTRAN statement may appear within the range of a DO statement, including another DO statement. When such is the case, the range of the second DO must be contained entirely within the range of the first; i.e., it is not permissible for the ranges of DOs to overlap. A set of DOs satisfying this rule is called a nest of DOs. It is possible for a terminal statement to be the terminal statement for more than one DO statement. The following configuration, where brackets are used to represent the range of the DOs, indicates the permissible and illegal nesting procedures.



Transfer of control from within the range of a DO statement to outside its range is permitted at any time. However, the reverse is not true; i.e., control cannot be transferred from outside the range of a DO statement to inside its range. The following examples show both valid and invalid transfers.



4.8 CONTINUE STATEMENT

The CONTINUE statement causes no action and generates no machine coding. It is a dummy statement which is used for terminating DO loops when the last statement would otherwise be an illegal terminal statement (i.e., GO TO, arithmetic IF, RETURN, STOP, PAUSE, or DO, or a logical IF containing any of these forms). The form consists of the single word

CONTINUE

4.9 PAUSE STATEMENT

A PAUSE statement is a temporary halt of the program at run time. The PAUSE statement has one of the two forms

PAUSE

or

PAUSE n

where n is an octal integer whose value is less than 777777_8 . The integer n is typed out on the console Teletype for the purpose of determining which of several PAUSE statements was encountered. Program execution is resumed, by typing control P ($\uparrow$ P), starting with the first statement following the PAUSE statement.

4.10 STOP STATEMENT

The STOP statement is of one of the forms

STOP

or

STOP n

where n is an octal integer whose value is less than 777777_8 . The STOP statement is placed at the logical end of a program and causes the computer to type out on the console Teletype, the integer n and then to exit back to the Monitor. There must be at least one STOP statement per main program, but none are allowed in subprograms.

4.11 END STATEMENT

The END statement is placed at the physical end of a program or subprogram. The form consists of the single word

END

The END statement is used by the compiler and generates no code. It signals the compiler that the processing of the source program is complete.

A control transfer type statement, a STOP statement, or a RETURN statement must immediately precede END. This will be checked by the compiler.

CHAPTER 5 INPUT/OUTPUT STATEMENTS

The input/output (I/O) statements direct the exchange of data between the computer and I/O devices. The information thus transmitted by an I/O statement is defined as a logical record, which may be formatted or unformatted. A logical record, or records, may be written on a device as one or more physical records. This is a function of the size of the logical record(s) and the physical device used.

The definition of the data which comprises a user's optimum physical record varies for each I/O device, as follows:

Unit or Device	Formatted Physical Record Definition	Unformatted (Binary) Physical Record Definition
Typewriter (input and output)	One line of type is terminated by a carriage return. Maximum of 72 printing characters per line	Undefined
Line printer	One line of printing. Maximum of 120 characters per line	Undefined
Cards (input and output)	One card. Maximum of 80 characters	50 words
Paper tape (input and output)	One line image of 72 printing characters	50 words
Magnetic tape	One line image of 630 characters	252 words
Disc/drum/ DECtape	One line image of 630 characters	252 words

Each I/O device is identified by an integer constant which is associated with a device assignment table in the PDP-9 Monitor. This table may be modified at system generation time, or just before run time. For example, the statement

READ (u,f) list

requests one logical record from the device associated with slot u in the device assignment table.

The statement descriptions in this section use u to identify a specific I/O unit, f as the statement number of the FORMAT statement describing the type of data conversion, and list as a list of arguments to be input or output.

5.1 GENERAL I/O STATEMENTS

These statements cause the transfer of data between the computer and I/O devices.

5.1.1 Input/Output Argument Lists

An I/O statement which calls for the transmission of information includes a list of quantities to be transmitted. In an input statement this list consists of the variables to which the incoming data is to be assigned; in an output statement the list consists of the variables whose values are to be transmitted to the given I/O device. The list is ordered, and the order must be that in which the data words exist (input) or are to exist (output) in the I/O device. Any number of items may appear in a single list. The same statement may transmit integer and real quantities. If the data to be transmitted exceeds the items in the list, only the number of quantities equal to the number of items in the list are transmitted. The remaining data is ignored. Conversely, if the items in the list exceed the data to be transmitted, succeeding superfluous records are transmitted until all items specified in the list have been transmitted.

5.1.1.1 Simple Lists - The list uses the form

$$C_1, C_2, \dots, C_n$$

where each C_i is a variable, a subscripted variable, or an array identifier. Constants are not allowed as list items. The list reads from left to right. When an array identifier appears in the list, the entire array is to be transmitted before the next item in the list. Examples of Simple Lists:

Y, Y, Z
A, B (3), C, D (I + 1, 4)

5.1.1.2 DO-Implied Lists - Indexing similar to that of the DO statement may be used to control the number of times a group of simple lists is to be repeated. The list elements thus controlled, and the index control itself, are enclosed in parentheses, and the contents of the parentheses are regarded as a single item of the I/O list.

Example:

W, X (3), (Y (I), Z (I, K), I = 1, 10)

5.1.2 READ Statement

The READ statement is used to transfer data from any input device to the computer. The general READ statement can be used to read either BCD or binary information. The form of the statement determines what kind of input will be performed.

5.1.2.1 Formatted READ - The formatted READ statements have the general form

```
READ (u,f) list
```

or

```
READ (u,f)
```

Execution of this statement causes input from device *u* to be converted as specified by format statement *f*, the resulting values to be assigned to the items specified by *list*, if any.

Examples:

```
READ (3,13) A,B,C
READ (2,10) A, (B(I), I = 1,5)
READ (1,3)
```

5.1.2.2 Unformatted READ - An unformatted READ statement has the general form

```
READ (u) list
```

or

```
READ (u)
```

Execution of this statement causes input from device *u*, in binary format, to be assigned to the items specified by *list*. If no *list* is given, one record will be read, but ignored. If the record contains more information words than the *list* requires, that part of the record is lost. If more elements are in the *list* than are in one record, additional records are read until the *list* is satisfied.

Example:

```
READ (5) I,J,K
READ (8)
```

5.1.3 WRITE Statement

The WRITE statement is used to transmit information from the computer to any I/O device. The WRITE statement closely parallels the READ statement in both format and operation.

5.1.3.1 Formatted WRITE - The formatted WRITE statement has the general form

```
WRITE (u,f) list
```

or

```
WRITE (u,f)
```

Execution of this statement causes the *list* elements, if any, to be converted according to format statement *f*, and output into device *u*.

5.1.3.2 Unformatted WRITE - The unformatted WRITE statement has the general form

WRITE (u) list

Execution of this statement causes output onto device u, in binary format, of all words specified by the list. If the list elements do not fill the record, the remaining part of the record is filled with blanks. If the list elements more than fill one record, successive records are written until all elements of the list are satisfied, the last record padded with blanks if necessary. Examples of WRITE:

```
WRITE (1,10) A, (B (1), (C (1,J), J=2,10,2), I=1,5)
WRITE (2,7) A,B,C
WRITE (5) W,X(3), Y(I + 1,4), Z
```

5.2 FORMAT STATEMENTS

These statements are used in conjunction with the general I/O statements. They specify the type of conversion which is to be performed between the internal machine language and the external notation. FORMAT statements are not executed. Their function is to supply information to the object program.

5.2.1 Specifying FORMAT

The general form of the FORMAT statement is

FORMAT (S₁, S₂, ..., S_n)

where S₁...S_n are data field descriptions. Breaking this format down further, the basic data field descriptor is written in the form

nkw.d

where n is a positive unsigned integer indicating the number of successive fields for which the data conversion will be performed according to the same specification. This is also known as the repeat count. If n is equal to 1, it may be omitted. The control character k indicates which type of conversion will be performed. This character may be I, E, F, D, P, L, A, H, or X. The nonzero integer constant w specifies the width of the field. The integer constant d indicates the number of digits to the right of the decimal point.

Six of the nine control characters listed above provide for data conversion between internal machine language and external notation.

<u>Internal</u>	<u>Type</u>	<u>External</u>
Integer variable	I	Decimal integer
Real variable	E	Floating-point, scaled
Real variable	F	Floating-point, mixed

<u>Internal</u>	<u>Type</u>	<u>External</u>
Real variable	G	Floating-point, mixed/scaled
Double-precision variable	D	Floating-point, scaled
Logical variable	L	Letter T or F
Alphanumeric	A	Alphanumeric (BCD) characters

The other three control types are special purpose control characters:

<u>Type</u>	<u>Purpose</u>
P	Used to set a scale factor for use with E, F, and D conversions.
X	Provides for skipping characters in input or specifying blank characters in output.
H	Designates Hollerith fields.

FORMAT statements are not executed and therefore may be placed anywhere in the source program. Because they are referenced by READ or WRITE statements, each FORMAT statement must be given a statement number.

Commas (,) and slashes (/) are used as field separators. The comma is used to separate field descriptors, with the exception that a comma need not follow a field specified by an H or X control character. The slash is used to specify the termination of formatted records. A series of slashes is also a field separator. Multiple slashes are the equivalent of blank records between output records, or records skipped for input records. If the series of n slashes occurs at the beginning or the end of the FORMAT specifications, the number of input records skipped or blank lines inserted in output is n. If the series of n slashes occurs in the middle of the FORMAT specifications, this number is n-1. A comma may precede and/or follow a slash, but is not necessary. An integer value cannot precede a slash.

For all field descriptors (with the exception of H and X), the field width must be specified. For those descriptors of the w.d type (see next page), the d must be specified even if it is zero. The field width should be large enough to provide for all characters (including decimal point and sign) necessary to constitute the data value as well as blank characters needed to separate it from other data values. Since the data value within a field is right justified, if the field specified is too small, the most significant characters of the value will be lost.

Successive items in the I/O list are transmitted according to successive descriptors in the FORMAT statement, until the entire I/O list is satisfied. If the list contains more items than descriptors in the FORMAT statement, a new record must be begun. Control is transferred to the preceding left parenthesis where the same specifications are used again until the list is complete.

Field descriptors (except H and X) are repeated by preceding the descriptor with an unsigned integer constant (the repeat count). A group repeat count is used to enable the repetition of a group of field descriptors or field separators enclosed in parentheses. The group count is placed to the left of the parenthesis. Two levels of parentheses (not including those enclosing the FORMAT specification) are permitted.

The field descriptors in the FORMAT must be the same type as the corresponding item in the I/O list; i.e., integer quantities require integer (I) conversion; real quantities require real (E or F) conversion, etc.

Example:

```
FORMAT (I7,F10.3)
FORMAT (I3, I7/E10.4,E10.4)
FORMAT (2I4, 3(I5,D10.3))
```

5.2.2 Conversion of Numeric Data

5.2.2.1 I-Type Conversion - Field descriptor: Iw or nIw

The number of characters specified by w is converted as a decimal integer.

On input, the number in the input field by w is converted to a binary integer. A minus sign indicates a negative number. A plus sign, indicating a positive number, is optional. The decimal point is illegal. If there are blanks, they must precede the sign or first digit. All imbedded blanks are interpreted as zero digits.

On output, the converted number is right justified. If the number is smaller than the field w allows, the leftmost spaces are filled with blanks. If an integer is too large, the most significant digits are truncated and lost. Negative numbers have a minus sign just preceding their most significant digit if sufficient spaces have been reserved. No sign indicates a positive number.

Examples (b indicates blank):

<u>Format Descriptor</u>	<u>Input</u>	<u>Internal</u>	<u>Output</u>
I5	bbbbbb	+00000	bbbb0
I3	-b5	-05	b-5
I8	bbb12345	+12345	bbb12345

5.2.2.2 E-Type Conversion - Field descriptor: Ew.d or nEw.d

The number of characters specified by w is converted to a floating-point number with d spaces reserved for the digits to the right of the decimal point. The w includes field d, spaces for a sign, the decimal point, plus four spaces for the exponent (written $E \pm XX$) in addition to space for optional sign and one digit preceding the decimal point.

The input format of an E-type number consists of an optional sign, followed by a string of digits containing an optional decimal point, followed by an exponent. Input data can be any number of digits in length, although it must fall within the range of 0 to $\pm 10^{\pm 39}$.

E output consists of a minus sign if negative (blank if positive), the digit 0, a decimal point, a string of digits rounded to d significant digits, followed by an exponent of the form E $\pm$ XX.

Examples:

<u>Format Descriptor</u>	<u>Input</u>	<u>Internal</u>	<u>Output</u>
E10.4	00.2134E03	213.4	0.2134E+03
E9.4	0.2134E02	21.34	.2134E+02
E10.3	bb-23.0321	-23.0321	-0.230E+02

5.2.2.3 F-Type Conversion - Field descriptor: Fw.d or nFw.d

The number of characters specified by w is converted as a floating-point mixed number with d spaces reserved for the digits to the right of the decimal point.

Input for F-type conversion is basically the same as that for E-type conversion, described above.

The output consists of a minus sign if the number is negative (blank if positive), the integer portion of the number, a decimal point, and the fractional part of the number rounded to d significant digits.

Examples:

<u>Format Descriptor</u>	<u>Input</u>	<u>Internal</u>	<u>Output</u>
F6.3	b13457	13.457	13.457
F6.3	313457	313.457	13.457
F9.2	-21367.	-21367.	-21367.00
F7.2	-21367.	-21367-	1367.00

5.2.2.4 G-Type Conversion - Field descriptor: Gw.d or nGw.d

The external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real number.

Input for G-type conversion is basically the same as that for E-type conversion, described in paragraph 5.2.2.2.

The form of the G-type output depends upon the magnitude of the internal floating-point number. Comparison is made between the exponent (e) of the internal value and the number of significant digits (d) specified by the format descriptor. If e is greater than d, the E-type conversion is used. If e is less than or equal to d, the F-type conversion is used, but modified by the following formula:

$$F(w-4).(d-e),4X$$

The 4X represents four blank spaces that are always appended to the value. If the value to be represented is less than .1, the E-type conversion is always used.

Examples:

<u>Format Descriptor</u>	<u>Internal</u>	<u>Output</u>
G14.6	.12345678 $\times 10^{-1}$	0.12345678E-01
G14.6	.12345678 $\times 10^0$	bb0.123456bbbb
G14.6	.12345678 $\times 10^4$	bbb1234.56bbbb
G14.6	.12345678 $\times 10^8$	bb0.123456E+08

5.2.2.5 D-Type Conversion - Field descriptor: Dw.d or nDw.d

The number of characters specified by w is converted as a double-precision floating-point number with the number of digits specified by d to the right of the decimal point.

The input and output are the same as those for E-type conversion except that a D is used in place of the E in the exponent.

Examples:

<u>Format Descriptor</u>	<u>Input</u>	<u>Internal</u>	<u>Output</u>
D12.6	bb+21345D 03	21.345	0.213450D+02
D12.6	b+3456789012	3456.789012	0.345678D+04
D12.6	-12345.6D-02	-123.456	0.123456D+03

5.2.3 P-Scale Factor - Field descriptor: nP or -nP

This scale factor n is an integer constant. The scale factor has effect only on E, F, G, and D-type conversions. Initially, a scale factor of zero is implied. Once a P field descriptor has been processed, the scale factor established by n remains in effect for all subsequent E, F, and D descriptors within the same FORMAT statement until another scale factor is encountered.

For E, F, G, and D input conversions (when no exponent exists in the external field) the scale factor is defined as external quantity = internal quantity $\times 10^n$.

The scale factor has no effect if there is an exponent in the external field.

The definition of scale factor for F output conversion is the same as it is for F input. For E and D output, the fractional part is multiplied by 10^n and the exponent is reduced by n.

Examples:

<u>Format Descriptor</u>	<u>Input</u>	<u>Scale Factor</u>	<u>Internal</u>	<u>Output</u>
-3PF6.3	123456	-3	+123456.	23.456
-3PE12.4	123456	-3	+12345.6	bb0.0001E+08
1PD10.4	12.3456	+1	+1.23456	1.2345D+00

5.2.4 Conversion of Alphanumeric Data

5.2.4.1 A-Type Conversion (7-Bit ASCII, Handled As REAL Variables) - Field descriptor: Aw or nAw

The number of alphanumeric characters specified by w is transmitted according to list specifications.

If the field width specified for A input is greater than or equal to five (the number of characters representable in two machine words), the rightmost five characters are stored internally. If w is less than five, 5-w trailing blanks are added.

For A output, if w is greater than five, w-5 leading blanks are output followed by five alphanumeric characters. If w is less than or equal to five, the leftmost w characters are output.

5.2.4.2 H-Field Descriptor (7-Bit ASCII) - Field descriptor: nHa₁a₂a₃...a_n

The number of characters specified by n immediately following the H descriptor are transmitted to or from the external device. Blanks may be included in the alphanumeric string. The value of n must be greater than 0.

On Hollerith input, n characters read from the external device replace the n characters following the letter H.

In output mode, the n characters following the letter H, including blanks, are output.

Examples:

```
3HABC
17H THIS IS AN ERROR
16H JANUARY 1, 1966
```

5.2.5 Logical Fields, L Conversion - Field descriptor: Lw or nLw

The external format of a logical quantity is T or F. The internal format of a logical quantity is T or F. The internal format is 777777₈ for T or 0 for F.

On L input, the first nonblank character must be a T or F. Leading blanks are ignored. A nonblank character is illegal.

For L output, if the internal value is 0, an F is output. Otherwise a T is output. The F or T is preceded by w-1 leading blanks.

5.2.6 Blank Fields, X Conversion - Field descriptor: nX

The value of n is an integer number greater than 0. On X input, n characters are read but ignored. On X output, n spaces are output.

5.2.7 FORTRAN Statements Read in at Object Time

FORTRAN provides the facility of including the formatting data along with the input data. This is done by using an array name in place of the reference to a FORMAT statement label in any of the formatted I/O statements. For an array to be referenced in such a manner, the name of the variable FORMAT specification must appear in a DIMENSION statement, even if the size of the array is 1. The statements have the general form:

```

      READ (u, name)
      READ (u, name) list
      WRITE (u, name)
      WRITE (u, name) list

```

The form of the FORMAT specification which is to be inserted into the array is the same as the source program FORMAT statement, except that the word FORMAT is omitted and the nH field descriptor may not be used. The FORMAT specification may be inserted into the array by using a data initialization statement, or by using a READ statement together with an A format.

For example, this facility can be used to specify at object time, the format of a deck of cards to be read. The first card of the deck would contain the format statement,

```

      1      10
      (17,F10.3)

```

the subsequent cards would contain data in the general form,

```

      7      17
      xx      xxxx
      DIMENSION AA (10)
13  FORMAT (10A5)
      READ (3,13) (AA(I), I=1,10)
      :
      READ (3,AA) JJ, BOB

```

With the card reader assigned to device number 3, the first READ places the format statement from the first card into the array AA, and the second READ statement causes data from the subsequent cards to be read into JJ and BOB with format specifications I7 and F10.3, respectively.

5.2.8 Printing of a Formatted Record

When formatted records are prepared for printing, the first character of the record is not printed. The first character is used instead to determine vertical spacing as follows:

<u>Character</u>	<u>Vertical Spacing Before Printing</u>
Blank	One line
0	Two lines
1	Skip to first line of next page
+	No advance

Output of formatted records to other devices considers the first character as an ordinary character in the record.

5.3 AUXILIARY I/O STATEMENTS

These statements manipulate the I/O file oriented devices. The *u* is an unsigned integer constant or integer variable specifying the device.

5.3.1 BACKSPACE Statement

The BACKSPACE statement has the general form

BACKSPACE *u*

Execution of this statement causes the I/O device identified by *u*, to be positioned so that the record which had been the preceding record becomes the next record. If the unit *u* is positioned at its initial point, execution of this statement has no effect.

5.3.2 REWIND Statement

The REWIND statement has the general form

REWIND *u*

Execution of this statement causes the I/O device identified by *u* to be positioned at its initial point.

5.3.3 ENDFILE Statement

The ENDFILE statement has the general form

ENDFILE *u*

Execution of this statement causes an endfile record to be written on the I/O device identified by *u*.

and another program which is called later contains the statement,

```
COMMON/N/XX,YY,ZZ
```

the latter program will find the values 3, 4, and 5 in its variables XX, YY, and ZZ, respectively, since variables in the same relative positions in COMMON statements share the same registers in memory.

6.4 EQUIVALENCE STATEMENT

The EQUIVALENCE statement is used to permit two or more entities of the same size and type to share the same storage location. The general format of the EQUIVALENCE statement is:

```
EQUIVALENCE (k1), (k2), ..., (kn)
```

where each k represents a list of two or more variables or subscripted variables separated by commas. Each element in the list is assigned the same memory storage location.

An EQUIVALENCE statement may lengthen the size of a COMMON block. The size can only be increased by extending the COMMON block beyond the last assignment for that block made directly by a COMMON statement. A variable cannot be made equivalent to an element of an array if it causes the array to extend past the beginning of the COMMON block.

When two variables or array elements share the same storage location because of the use of an EQUIVALENCE statement, they may not both appear in COMMON statements within the same program.

Example:

```
EQUIVALENCE (A,B), (C(10), D(10), E(15) )
```

6.5 EXTERNAL STATEMENT

An EXTERNAL statement is used to pass a subprogram name on to another subprogram. The general form of an EXTERNAL statement is:

```
EXTERNAL y,z,...
```

Example:

```
EXTERNAL ISUM, I SUB
      :
      CALL DEBUG (ISUM, A, B)
      :
      CALL DEBUG (ISUB, A, B)
      :
END
```

CHAPTER 6

SPECIFICATION STATEMENTS

Specification statements are nonexecutable because they do not generate instructions in the object program. They provide the compiler with information about the nature of the constants and variables used in the program. They also supply the information required to allocate locations in storage for certain variables and/or arrays. All SPECIFICATION statements must appear before any executable code generating statement. They must appear in this order: type statements, DIMENSION statements, COMMON statements, and EQUIVALENCE statements. EXTERNAL and DATA statements may appear anywhere after all type statements and before the executable code generating statements.

6.1 TYPE STATEMENTS

The type statements are of the forms

```

      INTEGER a,b,c
      REAL a,b,c
      DOUBLE PRECISION a,b,c
      LOGICAL a,b,c
  
```

where a, b, and c are variable names which may be dimensional or function names. A type statement is used to inform the compiler that the identifiers listed are variables or functions of a specified type, i.e., INTEGER, REAL, etc. It overrides any implicit typing; i.e., identifiers which begin with the letters I, J, K, L, M, or N are implicitly of the INTEGER mode; those beginning with any other letter are implicitly of the REAL mode. The type statement may be used to supply dimension information. The variable or function names in each type statement are defined to be of that specific type throughout the program; the type may not change.

Examples:

```

      INTEGER ABC,IJK,XYZ
      REAL A(2,4), I,J,K
      DOUBLE PRECISION ITEM, GROUP
      LOGICAL TRUE, FALSE
  
```

6.2 DIMENSION STATEMENT

The DIMENSION statement is used to declare arrays and to provide the necessary information to allocate storage for them in the object program.

The general form of the DIMENSION statement is:

```

      DIMENSION V (i1), V2(i2), ... Vn(in)
  
```

where each V is the name of an array and each i is composed of one, two, or three unsigned integer constants separated by commas. The number of constants represents the number of dimensions the array contains; the value of each constant represents the maximum size of each dimension. If the dimension information for the variable is given in a type statement or a COMMON statement, it must not be included in a DIMENSION statement.

Example:

```
DIMENSION ITEM (150), ARRAY (50,50)
```

When arrays are passed to subprograms, they must be redeclared in the subprogram. The mode, number of dimensions, and size of each dimension must be the same as that declared by the calling program.

6.3 COMMON STATEMENT

The COMMON statement provides a means of sharing memory storage between a program and its subprograms. The general form of the COMMON statement is:

```
COMMON /x1/a1/x2/a2/ ... /xn/an
```

where each x is a variable which is a COMMON block name, or it can be blank. If x_1 is blank, the first two slashes are optional. Each a represents a list of variables and arrays separated by commas. The list of elements pertaining to a block name ends with a new block name, with a blank COMMON block designation (two slashes), or the end of the statement.

The elements of a COMMON block, which are listed following the COMMON block name (or the blank name), are located sequentially in order of their appearance in the COMMON statement. An entire array is assigned in sequence. Block names may be used more than once in a COMMON statement, or may be used in more than one COMMON statement within the program. The entries so assigned are strung together in the given COMMON block in order of their appearance. Labeled COMMON blocks with the same name appearing in several programs or subprograms executed together must contain the same number of total words. The elements within the blocks, however, need not agree in name, mode, or order. A blank COMMON may be any length.

Examples:

```
COMMON A,B,C/XX/X,Y,Z
COMMON/A/X(3,3), Y(2,5)//Z(5,10,15)
```

The COMMON statement is a means of transferring data between programs. If one program contains the statements,

```
COMMON/N/AA,BB,CC
AA=3
BB=4
CC=5
```

and another program which is called later contains the statement,

```
COMMON/N/XX,YY,ZZ
```

the latter program will find the values 3, 4, and 5 in its variables XX, YY, and ZZ, respectively, since variables in the same relative positions in COMMON statements share the same registers in memory.

6.4 EQUIVALENCE STATEMENT

The EQUIVALENCE statement is used to permit two or more entities of the same size and type to share the same storage location. The general format of the EQUIVALENCE statement is:

```
EQUIVALENCE (k1), (k2), ..., (kn)
```

where each k represents a list of two or more variables or subscripted variables separated by commas. Each element in the list is assigned the same memory storage location.

An EQUIVALENCE statement may lengthen the size of a COMMON block. The size can only be increased by extending the COMMON block beyond the last assignment for that block made directly by a COMMON statement. A variable cannot be made equivalent to an element of an array if it causes the array to extend past the beginning of the COMMON block.

When two variables or array elements share the same storage location because of the use of an EQUIVALENCE statement, they may not both appear in COMMON statements within the same program.

Example:

```
EQUIVALENCE (A,B), (C(10), D(10), E(15) )
```

6.5 EXTERNAL STATEMENT

An EXTERNAL statement is used to pass a subprogram name on to another subprogram. The general form of an EXTERNAL statement is:

```
EXTERNAL y,z,...
```

Example:

```
EXTERNAL ISUM, ISUB
      :
      :
CALL DEBUG (ISUM, A,B)
      :
      :
CALL DEBUG (ISUB, A,B)
      :
      :
END
```

```
SUBROUTINE DEBUG (X,Y,Z)
```

```
  ⋮
```

```
  RETURN
```

```
  END
```

6.6 DATA STATEMENT

The DATA statement is used to set variables or array elements to initial values at the time the object program is loaded. The general form of the DATA initialization statement is:

$$\text{DATA } k_1/d_1/, k_2/d_2/, \dots, k_n/d_n/$$

where each k is a list of variables or array elements (with constant subscripts) separated by commas, and each d is a corresponding list of constants with optional signs. The k list may not contain dummy arguments. There must be a one-to-one correspondence between the name list and the data list, except where the data list consists of a sequence of identical constants. In such a case, the constant need be written only once, preceded by an integer constant indicating the number of repeats and an asterisk. A Hollerith constant may appear in the data list.

Variable or array elements appearing in a DATA statement may not be in blank COMMON. They may be in a labeled COMMON block and initially defined only in a BLOCK DATA subprogram. Example:

```
DATA A,B,C/3*2.0/
DATA X(1), X(2), X(3), X(4)/0.0, 0.1, 0.2, 0.3/, Y(1), Y(2)
2 DATA Y(3), Y(4)/1.0E2, 1.0E-2, 1.0E4, 1.0E-4/
```

CHAPTER 7
SUBPROGRAMS

A subprogram is a series of instructions which another program uses to perform complex or frequently used operations. Subprograms are stored only once in the computer, regardless how many times they are referred to by another program.

There are five categories of subprograms:

- a. Statement Functions
- b. Intrinsic or Library Functions
- c. External Functions
- d. External Subroutines
- e. Block Data Subprograms

The first three categories of subprograms are referred to as functions. The fourth category is referred to as subroutines. Functions and subroutines differ in the following two respects. Functions can return only a single value to the calling program; subroutines can return more than one value. Functions are called by writing the name of the function and an argument list in a standard arithmetic expression; subroutines are called by using a CALL statement. The last category is a special purpose subprogram used for data initialization purposes.

7.1 STATEMENT FUNCTIONS

A statement function is defined by a single statement similar in form to that of an arithmetic assignment statement. It is defined internally to the program unit by which it is referenced. Statement functions must follow all specification statements and precede any executable statements of the program unit of which they are a part. The general format of a statement function is:

$$f(a_1, a_2, \dots, a_n) = e$$

where f is a function name; the a 's are unsubscripted variables, known as dummy arguments, which are to be used in evaluating the function; and e is an expression.

The value of a function is a real quantity unless the name of the function begins with I, J, K, L, M, or N; in which case it is an integer quantity, or the function type may be defined by using the appropriate specification statement.

Since the arguments are dummy variables, their names are unimportant, except to indicate mode, and may be used elsewhere in the program, including within the expression on the right side of the statement function.

The expression of a statement function, in addition to containing unsubscripted dummy arguments, may only contain:

- a. Non-Hollerith constants
- b. Variable references
- c. Intrinsic function references
- d. References to previously defined statement functions
- e. External function references

A statement function is called any time the name of the function appears in any FORTRAN arithmetic expression. The actual arguments must agree in order, number, and type with the corresponding dummy arguments.

Execution of the statement function reference results in the computations indicated by the function definition. The resulting quantity is used in the expression which contains the function reference.

Examples:

```
A(X) = 3.2+SQRT (5.7* X**2)
SUM (A,B,C) = A+B+C
FUNC (A,B) = 2.*A/B**2.+Z
```

7.2 INTRINSIC OR LIBRARY FUNCTIONS

Intrinsic or library functions are predefined subprograms that are a part of the FORTRAN system library. The type of each intrinsic function and its arguments are predefined and cannot be changed.

An intrinsic function is referenced by using its function name with the appropriate arguments in an arithmetic statement. The arguments may be arithmetic expressions, subscripted or simple variables, constants, or other intrinsic functions (see table I-1).

Examples:

```
X = ABS (A)
I = INT (X)
J = IFIX (R)
```

Table I-1.
Intrinsic Functions

Intrinsic Functions	Definition	No. of Arguments	Symbolic Name	Type of Argument	Type of Function
Absolute value	$ a $	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
Truncation	Sign of a times largest integer $\leq a $	1	AINT	Real	Real
			INT	Real	Integer
			IDINT	Double	Integer
Remaindering*	$a_1 \text{ (mod } a_2)$	2	AMOD	Real	Real
			MOD	Integer	Integer
Choosing largest value	Max ($a_1, a_2, \dots$)	2	AMAX0	Integer	Real
			AMAX1	Real	Real
			MAX0	Integer	Integer
			MAX1	Real	Integer
			DMAX1	Double	Double
Choosing smallest value	Min ($a_1, a_2, \dots$)	2	AMIN0	Integer	Real
			AMIN1	Real	Real
			MIN0	Integer	Integer
			MIN1	Real	Integer
			DMIN1	Double	Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of sign	Sign of a_2 times $ a_1 $	2	SIGN	Real	Real
			ISIGN	Integer	Integer
			DSIGN	Double	Double
Positive difference	$a_1 - \text{Min}(a_1, a_2)$	2	DIM	Real	Real
			IDIM	Integer	Integer
Obtain most significant part of double precision argument		1	SNGL	Double	Real
Express single precision argument in double precision form		1	DBLE	Real	Double

*The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2] a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x .

7.3 EXTERNAL FUNCTIONS

An external function is an independently written program which is executed whenever its name appears in another program. The general form in which an external function is written is:

```

t  FUNCTION NAME (a1, a2, ..., an)
    (FORTRAN statements)
    :
    :
    NAME = final calculation
    RETURN
    END

```

where t is either INTEGER, REAL, DOUBLE PRECISION, LOGICAL, or is blank; NAME is the symbolic name of the function to be defined; and the a's are dummy arguments which are unsubscripted variable names, array names, or other external function names.

The first letter of the function name implicitly determines the type of function. If that letter is I, J, K, L, M, or N, the value of the function is INTEGER. If it is any other letter, the value is REAL. This can be overridden by preceding the word FUNCTION with the specific type name.

The symbolic name of a function is one to six alphanumeric characters, the first of which must be the alphabetic name and must not appear in any nonexecutable statement of the function subprogram except in the FUNCTION statement where it is named. The function name must also appear at least once as a variable name within the subprogram. During every execution of the subprogram, the variable must be defined before leaving the function subprogram. Once defined, it may be referenced or redefined. The value of this variable at the time any RETURN statement in the subprogram is encountered is called the value of the function.

There must be at least one argument in the FUNCTION statement. There must be unsubscripted variable names. If a dummy argument is an array name, an appropriate DIMENSION statement is necessary. The dummy argument names may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram.

The function subprogram may contain any FORTRAN statements with the exception of a BLOCK DATA, SUBROUTINE, or another FUNCTION statement. It, of course, cannot contain any statement which references itself, either directly or indirectly.

A function subprogram must contain at least one RETURN statement. The general form is:

```

RETURN

```

This signifies the logical end of the subprogram and returns control and the computed value to the calling program.

An END statement, described in section 4.11, signals the compiler that the physical end of the subprogram has been reached.

An external function is called by using its function name, followed by an actual argument list enclosed in parentheses, in an arithmetic or logical expression. The actual arguments must correspond in number, order, and type to the dummy arguments. An actual argument may be one of the following:

- a. A variable name
- b. An array element name
- c. An array name
- d. Any other expression
- e. The name of an external function or subroutine

Example:

```

      DIMENSION A(100), B(100)
      :
      :
      RSLT = SUM (A,B)**2
      :
      :
      END

      FUNCTION SUM (X,Y)
      DIMENSION X (100), Y(100)
      SUM = X(1) + Y(1)
      DO 10 K = 2, 100
      SUM = SUM + X(K) + Y (K)
      RETURN
      END

```

Main Program

Function Subprogram

Table I-2.
External Functions

Basic External Function	Definition	No. of Arguments	Symbolic Name	Type of Argument	Type of Function
Exponential	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
Natural logarithm	$\log_e (a)$	1	ALOG	Real	Real
		1	DLOG	Double	Double
Common logarithm	$\log_{10} (a)$	1	ALOG10	Real	Real
		2	DLOG10	Double	Double
Trigonometric sine	$\sin (a)$	1	SIN	Real	Real
		1	DSIN	Double	Double
Trigonometric cosine	$\cos (a)$	1	COS	Real	Real
		1	DCOS	Double	Double
Hyperbolic tangent	$\tanh (a)$	1	TANH	Real	Real
Square root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double

Table 1-2. (Cont)
External Functions

Basic External Function	Definition	No. of Arguments	Symbolic Name	Type of Argument	Type of Function
Arctangent	$\arctan(a)$	1	ATAN	Real	Real
		1	DATAN	Double	Double
	$\arctan(a_1/a_2)$	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Remaindering*	$a_1 \pmod{a_2}$	2	DMOD	Double	Double

*The function DMOD (a_1, a_2) is defined as $a_1 - \left[a_1/a_2 \right] a_2$, where $\left[x \right]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x .

7.4 SUBROUTINES

A subroutine is defined externally to the program unit which references it. It is similar to an external function in that both contain the same sort of dummy arguments, and both require at least one RETURN statement and an END statement. A subroutine, however, may have multiple outputs. The general form of a subroutine is:

SUBROUTINE NAME ($a_1, a_2, \dots, a_n$)

or

SUBROUTINE NAME

where NAME is the symbolic name of the subroutine subprogram to be defined; and the a 's are dummy arguments (there need not be any) which are unsubscripted variable names, array names, or the dummy name of another subroutine or external function.

The name of a subroutine consists of one to six alphanumeric characters, the first of which is alphabetic. The symbolic names of the subroutines cannot appear in any statement of the subroutine except the SUBROUTINE statement itself.

The dummy variables represent input and output variables. Any arguments used as output variables must appear on the left side of an arithmetic statement or an input list within the subprogram. If an argument is the name of an array, it must appear in a DIMENSION statement within the subroutine. The dummy argument names may not appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram.

The subroutine subprogram may contain any FORTRAN subprograms with the exception of FUNCTION, BLOCK DATA, or another SUBROUTINE statement.

The logical termination of a subroutine is a RETURN statement. The physical end of the subroutine is an END statement.

A subroutine is referenced by a CALL statement, which has the general form

CALL NAME ($a_1, a_2, \dots, a_n$)

or

CALL NAME

where NAME is the symbolic name of the subroutine subprogram being referenced, and the a's are the actual arguments that are being supplied to the subroutine. The actual arguments in the CALL statement must agree in number, order, and type with the corresponding arguments in the SUBROUTINE subprogram. The array sizes must be the same. An actual argument in the CALL statement may be one of the following:

- a. A Hollerith constant
- b. A variable name
- c. An array element name
- d. An array
- e. Any other expression
- f. The name of an external function or subroutine

7.5 BLOCK DATA SUBPROGRAM

The BLOCK DATA subprogram is a special subprogram used to enter data into a COMMON block during compilation. A BLOCK DATA statement takes the form

BLOCK DATA

This special subprogram contains only DATA, COMMON, EQUIVALENCE, DIMENSION, and TYPE statements. It cannot contain any executable statements. It can be used to initialize data only in a labeled COMMON block area; not in a blank COMMON block area.

All elements of a given COMMON block must be listed in the COMMON statement, even if they do not all appear in a DATA statement. Data may be entered in more than one COMMON block in a single BLOCK DATA subprogram.

An END statement signifies the termination of a BLOCK DATA subprogram.

7.5.1 Example of BLOCK DATA Subprogram

```

BLOCK DATA
DIMENSION X(4), Y(4)
COMMON/NAME/A, B, C, I, J, X, Y
DATA A, B, C/3*2.0/
DATA X(1), X(2), X(3), X(4)/0.0, 0.1, 0.2, 0.3/Y(1), Y(2)
2 DATA Y(3), Y(4)/1.0E2, 1.0E-2, 1.0E4, 1.0E-4/
END

```


APPENDIX 1

SUMMARY OF PDP-9 FORTRAN IV STATEMENTS

CONTROL STATEMENTS

ASSIGN n to i	I-15
CALL name (a ₁ , a ₂ , ...)	I-42
CONTINUE	I-19
DO n i=m ₁ , m ₂ , m ₃	I-17
GO TO n	I-15
GO TO i, (n ₁ , n ₂ , ...)	I-15
GO TO (n ₁ , n ₂ , ...), i	I-16
IF (e ₁) n ₁ , n ₂ , n ₃	I-16
IF (e) s	I-16
PAUSE	I-19
PAUSE n	I-19
STOP	I-19
STOP n	I-19
END	I-19
RETURN	I-39
SUBROUTINE NAME (a ₁ , a ₂ , ... a _n)	I-41
FUNCTION NAME (a ₁ , a ₂ , ... a _n)	I-39

INPUT/OUTPUT STATEMENTS

BACKSPACE u	I-31
END FILE u	I-31
FORMAT (s ₁ , s ₂ , ..., s _n)	I-24
READ (u, f)	I-23
READ (u, f) list	I-23
READ (u)	I-23
READ (u) list	I-23
READ (u, name)	I-30
READ (u, name) list	I-30
REWIND u	I-31
WRITE (u, f) list	I-23

WRITE (u, f)	I-23
WRITE (u) list	I-24
WRITE (u, name)	I-30
WRITE (u, name) list	I-30

SPECIFICATION STATEMENTS

COMMON $/x_1/a_1/x_2/a_2/\dots/x_n/a_n$	I-33
DATA $k_1/d_1/,k_2/d_2/, \dots, k_n/d_n/$	I-35
DIMENSION $V_1(i_1), V_2(i_2), \dots, V_n(i_n)$	I-32
DOUBLE PRECISION a, b, c	I-32
EQUIVALENCE $(k_1), (k_2) \dots (k_n)$	I-34
EXTERNAL $y, z, \dots, /$	I-34
INTEGER a, b, c	I-32
LOGICAL a, b, c	I-32
REAL a, b, c	I-32
BLOCK DATA	I-42

APPENDIX 2
A NOTE ON PDP-9 FORTRAN IV

The FORTRAN language used in this manual is essentially the language of USASI Standard FORTRAN (X3.9-1966) with the exception of the following features which are modified to allow the compiler to operate in 8192 words of core storage:

- a. All references to complex arithmetic are illegal.
- b. The size of arrays in subprograms is not adjustable to the size specified by the calling program.
- c. Blank COMMON is treated as name COMMON.
- d. The implied DO feature is not legal in a DATA statement.

There are two versions of the FORTRAN IV compiler: F4 and F4A. F4 is the basic compiler, and F4A is an abbreviated version of the compiler that allows DECtape input and output in an 8K system. F4A operates under control of the Keyboard Monitor only, and is called by typing F4A rather than F4 on the Teletype. The F4A version does not provide for the following options available in the F4 version:

O	Object code listing
S	Symbol table printout
L	Source listing

In paper tape systems, the FORTRAN compiler along with necessary I/O device handlers and an appropriate version of the I/O Monitor are punched on a tape in absolute format, referred to as a "system tape." At the beginning of the system tape is a Bootstrap Loader. The system tape can be loaded by setting the starting address of the Loader (17720 for 8K systems, 37720 for 16K) on the console address switches, pressing I/O RESET, and then pressing the READIN switch. (Refer to the I/O Monitor Guide for Paper Tape Systems, DEC-9A-NGAA-D.)

In larger systems with a bulk storage device such as DECtape, the Keyboard Monitor accepts direct keyboard commands to load the compiler in a device-independent environment. (Refer to Keyboard Monitor Guide, DEC-9U-NGAA-D.) This feature enables use of READ (I,f) or READ (I) statements where the value of I is undefined at compile and load times. If such statements are used, it is important to clear unused positive .DAT slots before loading to avoid loading device handlers that are not required.

Either DDT-9 or the Linking Loader must be used to load user object programs for execution. Refer to the appropriate Monitor Guide (I/O Monitor or Keyboard Monitor) for operating procedures.

Since A is a single-precision, floating-point number, two machine words are required and must be accounted for in the subprogram. Thus MIN1 and MIN2 (which contain the addresses of A and B) must be incremented to get to the second word of each number. FORTRAN expands the CALL statement as follows:

```

      CALL MIN (A,B)
00013      JMS*      MIN      (Exit to MACRO-9 subprogram)
00014      JMP      $00014    (Entry from MACRO subprogram)
00015      .DSA      A
00016      .DSA      B
$00014=00017

```

When the program is loaded, the address (plus relocation factor) of A is stored in location 00015 (plus relocation factor) and the address of B in 00016 (plus relocation factor). When .DA is called from the MACRO-9 subprogram, it stores these addresses in MIN1 and MIN2 (plus relocation factor). Thus MIN1 must be referenced indirectly to get the value of A (a direct reference would get the address of A).

2. Linking MACRO-9 Programs With FORTRAN IV Subprograms:

There are two forms of FORTRAN IV subprograms: subroutines and external functions. The main difference between the two is the method of returning arguments to the calling program: subroutines return the argument directly to the calling program, while functions return arguments through accumulators.

The MACRO-9 program set-up for a FORTRAN IV subroutine is basically that described in Part III of this manual for FORTRAN IV Science Library routines. The name of the subroutine to be called must be declared as a global, there must be a jump around the argument addresses, and the number and type (integer, real, double precision) of arguments must agree from the calling program to the subroutine.

An example of a calling routine:

```

      TITLE
      .GLOBL      SUBROT
      JMS*      SUBROT
      JMP      .+N+1
      .DSA      ADDR OF ARG1      /+400000 if indirect
      .DSA      ADDR OF ARG2      /+400000 if indirect
      :
      :
      .DSA      ADDR OF ARGN      /+400000 if indirect
      :
      :

```

APPENDIX 3 FORTRAN IV AND MACRO-9 LINKAGE

1. Linking FORTRAN IV Programs With MACRO-9 Subprograms

There are two essential elements of a MACRO subprogram that is linked to FORTRAN IV. One is the declaration of the name of the subprogram (as used in the F4 program) in a .GLOBL statement within the subprogram. The second is leaving open registers in the subprogram for the transfer vectors of the arguments used in the FORTRAN calling sequence. The number of open registers must agree with the number of arguments given in the calling sequence.

As an illustrative example, consider a FORTRAN program and a MACRO-9 subprogram which read, negate, and write a number. One positive, single-precision floating-point number is read by the FORTRAN program, negated in the MACRO-9 subprogram, and written out from the FORTRAN program.

FORTRAN IV PROGRAM:

```

C          TEST MACRO SUBPROGRAM
C          READ A NUMBER (A)
1          READ (1,100) (A)
100        FORMAT (E12.4)
C          NEGATE THE NUMBER AND PUT IT IN B
C          CALL MIN (A,B)
C          WRITE OUT THE NUMBER (B)
C          WRITE (2,100) B
C          STOP
C          END

```

MACRO-9 SUBPROGRAM:

```

MIN          .TITLE MIN
              .GLOBL MIN, .DA
0            /ENTRY/EXIT
JMS*         .DA      /USE THE F4 GENERAL GET ARGUMENT
              /SUBPROGRAM TO LOAD THE ARGUMENTS
JMP          .+3      /JUMP AROUND REGISTERS LEFT FOR
              /ARGUMENT ADDRESSES
MIN1         .DSA      0      /ARG 1
MIN2         .DSA      0      /ARG2
LAC*         MIN1      /PICK UP FIRST WORD OF A
DAC*         MIN2      /STORE IN FIRST WORD OF B
ISZ          MIN1      /BUMP THE POINTER TO SECOND WORD
ISZ          MIN2      /OF A AND B
LAC*         MIN1      /PICK UP SECOND WORD OF A
TAD          (400000)   /SIGN BIT = 1
DAC*         MIN2      /STORE IN SECOND WORD OF B
JMP*         MIN       /EXIT
              .END

```

When the FORTRAN IV subroutine is compiled, the compiler will generate code for .DA, the General Get Argument Routine, which will transfer the arguments from the MACRO-9 calling program to the FORTRAN IV subroutine. .DA expects to find the calling sequence just described for the calling program. The following is an example of an expansion of the beginning of a FORTRAN IV subroutine.

C	TITLE SUBROT	
	SUBROUTINE SUBROT (A, B)	
000000	CAL	0
000001	JMS*	.DA
000002	JMP	\$000002
000003	.DSA	A
000004	.DSA	B
\$000002=000005		

The simplest method of passing arguments between the main program and the subroutine is to use one of the calling arguments as output. For example, if the value of D is to be calculated in the subroutine, use D as one of the calling arguments. "D=" will generate a "DAC* D", which will store the value calculated for D by the subroutine in location D in the calling program.

The MACRO-9 program set-up for a FORTRAN IV External Function is identical to that for linkage with subroutines, except that some provision must be made for storage of the values calculated and stored in the accumulator. In the case of integers, the value is returned in the A-register, and in the floating accumulator for real and double precision numbers. The simplest method of storing the values is to use the FORTRAN IV routines furnished in the library for this purpose. .AH store real values, and .AP stores double precision values. Since the A-register is the standard hardware accumulator, a DAC instruction will store integer values.

3. Linking MACRO-9 Programs With FORTRAN IV Library Routines

Refer to Part III of this manual , PDP-9 Science Library, for a complete description of the linkage to library routines.

APPENDIX 4

CHAINING FORTRAN IV PROGRAMS

Chaining is a method of program segmentation that allows for multiple core overlap of executable code and certain types of data areas. FORTRAN programs can thus be divided into segments and executed separately, with intersegment communication of data accomplished through common storage. Common areas of core are reserved by means of the blank COMMON statement.

Transfer of control from one chain segment to another can be specified in a FORTRAN source program with the statement

CALL CHAIN (N)

where N is the segment number to be called. The chain number (N) is established at chain-build time (refer to the CHAIN section of the Keyboard Monitor Guide, DEC-9U-NGAA-D). N can be greater than or less than but not equal to the current chain number. A STOP statement must immediately precede the END statement in the main program for each chain. Only variables and arrays named in blank COMMON statements are retained from one chain segment to another. Blank common size should be the same for all chain segments.

```

C          TEST CHAIN PROGRAM
C
C          CHAIN JOB SEGMENT 1
C          COMMON A,B,C
C          DIMENSION ARRAY (10,10)
C          READ (4,5) ARRAY
C          :
C          CALL CHAIN (2)
C          STOP
C          END
C
C          CHAIN JOB SEGMENT 2
C          COMMON A,B,C
C          DIMENSION TABLE (30)
C          :
C          CALL CHAIN (3)
C          :
C          STOP
C          END
C
C          CHAIN JOB SEGMENT 3
C          COMMON A,B,C
C          DIMENSION A LIST (5,5)
C          :

```

```
WRITE (4,6) ALIST  
6 FORMAT (E10.3)  
STOP  
END
```

APPENDIX 5 FORTRAN IV ERROR LIST

These letter-coded error messages apply to F4 V2A and all versions of F4 thereafter and all versions of F4A. Refer to page II-2 of this manual for a list of object-time errors.

	<u>Error Code</u>	<u>Cause</u>
X	Syntax error	Statement cannot be recognized as a properly constructed FORTRAN IV statement.
V	Variable/constant mode error	Illegal mode mixing. Missing constant, variable or exponent, or illegal matching of constants or variables in a DATA statement.
N	Statement number error	Phase error, number more than 5 digits, no statement number where one is required, statement shouldn't be labeled or doubly defined statement numbers.
S	Argument/subscript error	Missing argument or subscript, illegal use of subscripts, illegal construction of subscripted variable, more than 3 subscripts or stated number of subscripts does not agree with declared number.
F	FORMAT statement error	Illegal FORMAT specification or illegal construction of FORMAT statement.
I	Character/statement/term error	Illegal character, unrecognizable statement, illegal statement for program type, statement out of order or improper statement preceding END statement.
D	DO loop error	Illegal DO construction or illegal statement terminating DO LOOP.
T	Table overflow	Symbol/constant/arg (I)/OP(I) table limits exceeded.
L	Nesting error	Illegal nesting or DO nesting too deep.
M	Magnitude error	Program exceeds 8190 words, maximum number of dummy arguments or EQUIVALENCE classes exceeded, or constant/variable exceeds specified limits.
C	COMMON/EQUIVALENCE/DIMENSION/DATA statement error	Illegal construction of statement, illegal EQUIVALENCE relationships, illegal COMMON declaration or non-common storage declared in BLOCK DATA subprogram.
E	FUNCTION/SUBROUTINE/EXTERNAL/CALL statement error	Illegal use of FUNCTION/SUBROUTINE name, out of order, or illegal variable for EXTERNAL declaration.
H	Hollerith error	Hollerith data illegal in this statement or illegal of Hollerith constant.

PDP-9 FORTRAN IV

APPENDIX 5A FORTRAN IV ERROR LIST

This list of error messages applies of all versions of F4 prior of V2A.

<u>Error Code</u>	<u>Cause</u>
001	Improper statement preceding END statement.
002	FUNCTION/SUBROUTINE name not used or used improperly in a subprogram.
003	Variable required.
004	Positive non-zero constant required.
005	Symbol-constant table limits exceeded.
006	Statement number has more than 5 digits.
007	Unsigned simple integer argument required.
008	Integer value greater than $(2^{17} - 1)$.
009	Magnitude of number (ignoring decimal point and/or exponent) greater than $(2^{35} - 1)$.
010	Array element (function reference) and next argument not separated by comma.
012	Exponent missing from numeric field.
014	Open parenthesis in subscript.
015	Binary operator used in unary sense or missing argument.
016	Additional grouping parenthesis not allowed.
017	Subscript list terminated before last argument obtained.
018	Too many right parenthesis.
019	Argument follows converted argument (no operator separating them).
021	E or D in numeric field after exponent has been processed.
023	Format descriptor character used in non-format statement.
024	Hollerith data illegal this statement.
025	Non-integer constant precedes Hollerith constant.
026	Length of Hollerith constant exceeds range $(0 < x < 6)$.
027	Hollerith constant contains statement termination character.
030	Symbolic name exceeds 6 characters.
031	Limit on number of nested functions exceeded.
032	Simple variable delineated by left parenthesis.
033	Subscripted variable used as subscript.
034	Unrecognizable logic term.
036	Logical .NOT. used as binary operator or logical constant used as a binary operator.
037	Illegal character.

PDP-9 FORTRAN IV

<u>Error Code</u>	<u>Cause</u>
040	Adjusted floating point exponent exceeds 76.
045	Subscript expression not delimited by ",", " or ")"
050	Arg (I), op (I) table overflow.
051	Unsubscripted array reference used as a variable.
052	Function name used as a variable.
053	Comma used other than as parameter separator.
054	Function or array name used in an expression representing a function parameter.
055	Expression used as an assignment variable.
056	Expression contains uneven number of parenthesis.
060	Illegal mode mixing.
063	Signed assignment variable.
069	Doubly defined statement numbers or phase error.
070	Illegal statement for BLOCK DATA program.
071	Statement not a DO or DO illegal as True statement for logical IF.
072	Illegal logical IF true statement.
074	Unrecognizable statement (first 3 characters).
075	Unrecognizable statement (4-n characters).
076	Statement out of order.
077	Statement shouldn't be labeled.
078	First character following READ or WRITE not a left parenthesis.
079	Illegal format specification in READ or WRITE.
080	Binary WRITE has no list.
081	Illegal list element separator.
082	Illegal implied DO construction.
086	DO nesting too deep.
087	Missing DO parameter.
088	Illegal DO parameter construction.
089	Illegal statement type terminated DO.
090	Improper DO nesting.
091	Illegal character following I/O unit specification.
092	Illegal character following FORMAT specification.
095	Name previously appeared in a specification statement.
096	Statement improperly delimited.
097	Illegal array declaration - DIMENSION statement.

PDP-9 FORTRAN IV

<u>Error Code</u>	<u>Cause</u>
098	Array size greater than 78192 - DATA specification statement.
100	Improper subscript list delimiter in a DIMENSION statement.
101	More than 3 subscripts (dimensions).
102	Integer argument not a constant.
103	Integer constant not less than 8192.
104	Integer constant negative.
105	FORMAT statement has no statement number.
106	Illegal FORMAT specification.
114	COMMON block name has illegal delimiter (not /).
115	Name declared as COMMON Block name previously declared otherwise.
116	COMMON block size exceeds 8190 words.
117	Declared COMMON variable is a dummy, function, or already in COMMON.
118	ASSIGN statement missing "TO".
120	Computed GOTO statement number list members not delimited.
121	Computed GOTO statement number list not delimited.
123	Assigned GOTO statement number list not preceded by left parenthesis.
124	Assigned GOTO statement number list not delimited.
126	DATA/EQUIVALENCE variable is a dummy variable.
127	DATA/EQUIVALENCE variable is a function name.
128	DATA/EQUIVALENCE simple variable subscripted by more than one number.
129	DATA/EQUIVALENCE array element greater than 8192.
130	Missing right or redundant left parenthesis in FORMAT statement.
131	Stated number of subscripts does not agree with declared number.
132	EQUIVALENCE class not started with left parenthesis.
136	EQUIVALENCE class improperly delimited (no right parenthesis).
137	Maximum number of EQUIVALENCE classes exceeded.
138	Illegal EQUIVALENCE relationships.
139	Illegal extension of common block caused by EQUIVALENCE relationship.
140	Illegal variable for EXTERNAL declaration.
142	Program size exceeds 8190 words.
143	Non-common storage declared in BLOCK DATA subroutine.
144	IF expression not terminated by closing parenthesis.
145	No comma separating statement numbers - IF statement.
147	No statement number where one is required.

PDP-9 FORTRAN IV

<u>Error Code</u>	<u>Cause</u>
148	Referenced statement number does not appear as a statement label.
149	Illegal statement function name (used twice or is external).
150	Delimiter following statement function dummy argument list not "=".
151	Function statement out of order (or more than one).
152	FUNCTION name not followed by argument list.
153	Dummy variable previously declared as something else.
154	Dummy argument list not enclosed in parenthesis.
155	Maximum number of dummy arguments exceeded.
156	STOP/PAUSE statement constant contains a non-octal digit.
157	STOP/PAUSE statement constant contains too many digits.
158	DATA statement variable is "common" but data not "block data".
159	DATA statement is "block data" but variable is not "common".
160	DATA statement variables not delimited by a slash.
161	DATA statement argument not a constant.
162	Mode of variable and corresponding constant disagree in a DATA statement.
163	DATA statement constants not separated by a comma (may indicate more variables than constants).
164	Two successive asterisks used as operators in a DATA statement.
165	Constant preceding asterisk is not integer in a DATA statement.
167	More constants than variables in a DATA statement.
168	RETURN statement in main-body program unit (not subroutine).
169	CALL statement - name is not a function name.

FORTRAN IV OBJECT - TIME SYSTEM

INTRODUCTION

Part II describes the subprograms included in the PDP-9 FORTRAN IV Object Time System. The Object Time System is a group of subprograms that process compiled FORTRAN IV statements, particularly I/O statements, at execution time. The compiler outputs calls in the form of globals to various subprograms, depending upon the content of the FORTRAN program. When the compiled program is loaded via the Linking Loader, the Loader attempts to satisfy these globals by searching the FORTRAN library. As it finds the required object time subprograms, it brings them into core and sets up the necessary linkages.

Included in the package are programs for processing formatted and unformatted READ and WRITE statements, BACKSPACE, REWIND and ENDFILE statements, the index of computed GO TO statements, STOP and PAUSE statements, and File commands. There are eight error messages output by the object time system which are described in Table II-1.

The following information is given for each program:

- a. Class
- b. Purpose
- c. Calling sequence
- d. External calls
- e. Size
- f. Error conditions

Table II-1.
OTS Errors

<u>Error Number</u>	<u>Error Description</u>	<u>Library Routines* That May Cause Error</u>
00 -04	Not used	
05	Negative REAL Square Root Argument	SQRT
06	Negative DOUBLE PRECISION Square Root Argument	DSQRT
07	Illegal Index in Computed GO TO	.GO
10	Illegal I/O Device Number	.FR, .FW, .FS, .FX, .FR, .FA, .FE, .FF, .FS,
11	Bad input data - IOPS Mode Incorrect	.FR, .FA, .FE, .FF, .FS,
12	Bad FORMAT	.FA, .FE, .FF
13	Negative or Zero REAL Logarithmic Argument	.BC, .BE, ALOG
14	Negative or Zero DOUBLE PRECISION Logarithmic Argument	.BD, .BF, .BG, .BH, DLOG, DLOG10

*Only those routines whose calls are generated by the compiler are listed.

BCDIO

1. Class: Object - Time System

2. Purpose: The BCD input/output object-time package is designed to process the formatted READ and WRITE statements in FORTRAN IV programs and subprograms. The FORTRAN IV compiler generates all the necessary object-time subroutine calls to perform input and output operations on a character-to-character basis under the control of a FORMAT statement. To permit FORMAT statements to be altered or read at execution time, the FORMAT statements are interpreted by BCDIO at execution time rather than at compile-time. This has two advantages:

- 1) It provides a greater flexibility to the FORTRAN programmer,
- 2) It provides the ability to utilize fully the capabilities of BCDIO in machine-language programs.

In demonstrating this capability, an illustrative MACRO-9 language program is given below, which reads 8 floating point numbers into memory with F-conversion and writes them on an output device using the E-conversion.

Example:

	.TITLE		
	.GLOBL	.FP, .FR, .FE, .EE, .FW	
	.IODEV	3,4	
ENTRY	JMS*	.FP	/Initialize I/O device status table.
	JMS*	.FR	/Initialize device 3 for input
	.DSA	(3)	/under control of FORMAT statement
	.DSA	FRMT1	/FRMT1 and read first record into line
			/buffer.
	LAW	-1Ø	/Set loop counter to 8.
	DAC	COUNT	
	LAC	(ARRAY)	/Set element address to first word
	DAC	ARG1	/in the array.
LOOP1	JMS*	.FE	/Convert next line buffer field from
ARG1	Ø		/BCD to floating point binary and
			/store in ARRAY.
	ISZ	ARG1	/Increment ARRAY address by two.
	ISZ	ARG1	
	ISZ	COUNT	/Check the counter and
	JMP	LOOP1	/if not done, repeat loop.
	JMS*	.FF	/Otherwise, terminates reading.
	JMS*	.FW	/Initialize device 4 for output
	.DSA	(4)	/under control of FORMAT
	.DSA	FRMT2	/statement FRMT2.
	LAW	-1Ø	/Set loop counter to 8.
	DAC	COUNT	

OTS Binary Coded Input/Output

	LAC	(ARRAY)	/Set element address to first
	DAC	ARG2	/word in the array.
LOOP2	JMS*	.FE	/Convert floating-point binary word
ARG2	Ø		/pair to BCD and store in line-buffer.
	ISZ	ARG2	/Increment ARRAY address by 2.
	ISZ	ARG2	/
	ISZ	COUNT	/Check count.
	JMP	LOOP2	/If not done, go to LOOP 2.
	JMS*	.FE	/if done, output last line-buffer
			/and terminates writing.
	HLT		
ARRAY	.BLOCK	2Ø	
FRMT1	.ASCII	'(8F10.5)'	
FRMT2	.ASCII	'(8E12.5)'	
COUNT	Ø		
	.END		

3. Calling Sequences:

a. To initialize a device for BCD input (output):

JMS*	.FR (.FW)
.DSA	address of slot number.
.DSA	address of first word of FORMAT statement or array.

b. To input (output) a data element:

JMS*	.FE
.DSA	address of element (first word)

c. To input (output) an entire FORTRAN array:

JMS*	.FA
.DSA	address of last word in the Array Descriptor Block.

d. To terminate the current logical record:

JMS*	.FF
------	-----

All BCDIO routines utilize the FIOPS object-time package to perform all I/O data transfers between devices and the FIOPS line buffer. Device level communication is never employed.

e. External Calls:

FIOPS, OTSER, REAL ARITHMETIC

f. Size: 2773 octal locations

g. Error Conditions:

OTS ERROR 10 - Illegal I/O Device Number
OTS ERROR 11 - Bad Input Data (IOPS Mode Incorrect)
OTS ERROR 12 - Illegal FORMAT

BINIO

1. Class: Object - Time System

2. Purpose: The Binary Input/Output Object-Time package is designed to process the unformatted READ and WRITE statements in FORTRAN IV programs and subprograms. A FORMAT statement is not required and the data transfer is on a word-to-word basis instead of on character-to-character basis, regardless of data type.

The size of the physical data record is always the standard line buffer size provided by IOPS.

Logical data records are comprised of one or more physical records, the number of which is determined by the length of the I/O list associated with the WRITE statements that generates the logical record.

Each WRITE statement generates one logical record.

Each READ statement reads one logical record, regardless of the length of its I/O list. For this reason, it is the responsibility of the FORTRAN programmer to ensure that I/O lists for WRITE and READ statements are compatible.

3. Calling Sequences:

a. To initialize a device for binary input (output):

```
JMS*      .FS (.FW)
.DSA      DEVICE
```

b. To input (or output) an integer data element:

```
JMS*      .FI
.DSA      address of the element
```

c. To input (or output) an integer data element:

```
JMS*      .FJ
.DSA      address of the element (first word)
```

d. To input (or output) a double precision data element:

```
JMS*      .FK
.DSA      address of the element (first word)
```

e. To input (or output) a logical data element:

```
JMS*      .FL
.DSA      address of the element
```

f. To input (or output) an entire FORTRAN array:

```
JMS*      .FB
.DSA      address of the last word in the Array Descriptor Block.
```

g. To terminate the current logical record:

JMS* .FG

The third word of each physical record contains a record of ID numbers starting with ZERO for the first record. Then ID is incremented by one as each physical record is generated until the last record in the logical record has bit 0 set.

A typical WRITE statement may generate the following record for ID:

	000000	
	000001	
LOGICAL	000002	PHYSICAL RECORD
RECORD	000003	FOR ID (OCTAL)
	000004	

4. External Calls:

FIOPS, OTSER

5. Size: 244 octal locations

6. Error Conditions:

OTS ERROR 10 - Illegal I/O Device Number

OTS ERROR 11 - Illegal Input Data (IOPS Mode Incorrect)

AUXIO

1. Class: Object - Time System

2. Purpose: Auxiliary Input/Output consists of the processors for the three auxiliary I/O statements in FORTRAN IV: BACKSPACE, REWIND, and ENDFILE.

These statements are normally used to control Magnetic Tape Transports which are being used by unformatted READ and WRITE statements (BINIO).

1) BACKSPACE (.FT):

Repositions the tape at a point just prior to the first physical record associated with the current logical record.

Example:

```
WRITE (7) A,B,C
BACKSPACE (7)
READ (7) D,E,F
```

These three instructions as shown in the above order cause the data of A, B, and C to be transferred to D, E, and F.

2) REWIND (.FU)

Causes the specified device to be positioned at its initial (load) point.

3) ENDFILE (.FV)

Issues an IOPS command to close the current file on the specified device.

In the case of Magnetic Tape, this writes a file mark.

3. Calling Sequences:

a. To backspace one logical record:

```
JMS*      .FT
.DSA      DEVICE
```

b. To position a device at its initial point:

```
JMS*      .FU
.DSA      DEVICE
```

c. To end (close) a file:

```
JMS*      .FV
.DSA      DEVICE
```

4. External Calls:

FIOPS

OTS Auxiliary Input/Output

5. Size: 64 octal locations

6. Error Conditions:

OTS ERROR 1Ø - Illegal I/O Device Number

FIOPS

1. Class: Object - Time System

2. Purpose: FIOPS provides the necessary calls to IOPS required by all FORTRAN input and output statements.

Slot numbers are initialized by the .FC routine (Initialize I/O Device). Initialization of all slots is maintained in the device status table. The first time that .FC is called for any device, the appropriate .INIT call is made to IOPS. The buffer size and input/output flag are stored in the status word table. Then all subsequent calls to .FC for the same device suppress another .INIT unless the input/output flag has changed.

One life buffer is used by all FORTRAN programs. Data transfers between the line buffer and I/O devices are performed by the .FQ routine, which performs a .READ if the input/output flag (.FH) is "ZERO" or a .WRITE if .FH is "ONE." A .WAIT is always performed.

The .FP routine is called at the beginning of all FORTRAN main programs. This routine sets all words in the device status table to zero, indicating that all devices are uninitialized.

3. Calling Sequences:

a. To initialize the I/O device status table:

JMS* .FP

b. To specify input:

DZM* .FH

c. To specify output:

LAC (1)
DAC* .FH

d. To select device:

LAC DEVICE (address of slot number)
JMS* .FC

e. To input or output the line buffer:

LAC address of .DAT slot number (bits 9-17) and IOPS mode (bits 6-8)
JMS* .FQ

f. Notes:

- 1) DEVICE is a cell containing the slot number.
- 2) The line buffer is in locations .FN to .FN+377₈.

OTS IOPS Communication

- 3) The standard line buffer size (for the device currently selected) is in location .FM.
- 4) On output, IOPS header words (.FN and .FN + 1) must be prepared by the user.

4. External Calls:

OTSER

5. Size: 530 octal locations

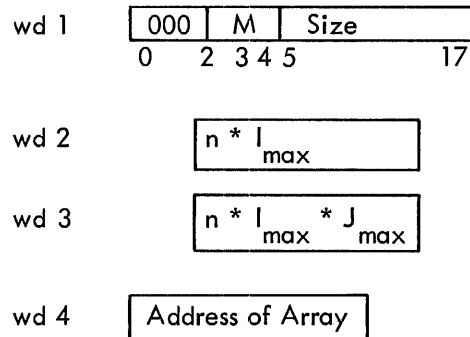
6. Error Conditions:

OTS ERROR 1Ø - Illegal I/O Device Number

OTS Calculate Array Element Address

.SS

1. Class: Object - Time System
2. Purpose: To calculate the array element address. The Array Descriptor Block is constructed as follows:



M: data type - translates to number of words per array element (n)

M	N	Type
00	01	integer
01	02	real
10	03	double precision
11	01	logical

Subscript calculation formula (for declared A (I,J,K) and specified A (i,j,k))

Addr of A (i,j,k) =

$$\underbrace{A + (i-1) * n}_{\text{wd 4}} + \underbrace{(j-1) * I * n}_{\text{wd 2}} + \underbrace{(k+1) * I * J * n}_{\text{wd 3}}$$

1-dimension array
 2-dimension array
 3-dimension array

3. Calling Sequence:

JMS*	.SS	
.DSA	ARRAY	/Address (indirect) of array
		/Descriptor Block
LAC	S1	/Subscript 1 (I)
LAC	S2	/Subscript 2 (J)
LAC	S3	/Subscript 3 (K)
DAC		/Store array element address

OTS Calculate Array Element Address

4. External Calls:

INTEGER and REAL ARITHMETIC

5. Size: 57 octal locations

6. Error Conditions:

None.

GOTO (.GO)

1. Class: Object - Time System
2. Purpose: To compute the index of a computed GO TO
3. Calling Sequence:

LAC	V	/Index value in A-register
JMS*	.GO	
-N		/Number of statement addresses
STMT ADDR (1)		
STMT ADDR (2)		
⋮		
STMT ADDR (N)		

4. External Calls:

OTSER

5. Size: 26 octal locations
6. Error Conditions:

OTS ERROR 7 if the index is illegal (equal to or less than zero).

STOP (.ST)

1. Class: Object - Time System
2. Purpose: To process the STOP statement and return control to the monitor.
3. Calling Sequence:

LAC	(Octal number to be printed)
JMS*	.ST

4. External Calls:

SPMSG (.SP)

5. Size: 13 octal locations
6. Error Conditions:

None

PAUSE (.PA)

1. Class: Object - Time System
2. Purpose: To process the PAUSE statement. After receiving a ↑ P (Control P) from the keyboard, control is returned to the program.
3. Calling Sequence:

LAC	(Octal number to be printed)
JMS*	.PA
4. External Calls:

SPMSG (.SP)

5. Size: 14 octal locations
6. Error Conditions:

None

SPMSG (.SP)

1. Class: Object - Time System
2. Purpose: To print the octal number coded with STOP and PAUSE. If no number is given, zero (0) is assumed.

3. Calling Sequence:

LAC	(Octal integer to be printed)
JMS*	.SP
.DSA	(Control return) /pause only
LAC	1st Character
LAC	2nd Character
LAC	3rd Character
LAC	4th Character
LAC	5th Character
LAC	6th Character

4. External Calls:

None

5. Size: 74 octal locations

6. Error Conditions:

None

OTSER (.ER)

1. Class: Object - Time System

2. Purpose:

a. To announce an error on the teletype:

JMS*	.ER
.DSA	Error number

- b. If bit 0 of the error number is a 1, the error is recoverable and program control is returned to the calling program at the first location following the error number.
- c. If bit 0 of the error number is a 0, the error is unrecoverable and program control is transferred to the monitor by means of the .EXIT function.
- d. In the case of recoverable errors, the AC and link are restored to their original contents prior to returning control to the caller.
- e. If the error is a bad format statement (unrecoverable), the current 5/7 ASCII word pair of the erroneous format statement is printed in addition to the error number.

3. Calling Sequence:

	JMS*	.ER
	.DSA	Error number, octal
ERROR #12	LAC	Note word 1
only	LAC	Note word 2

Words 1 and 2 are the current 5 characters (in 5/7 ASCII of the bad format statement (ERROR #12)

4. External Calls:

None

5. Size: 117 octal locations

6. Error Conditions:

None

FILE

1. Class: External Subroutine
2. Purpose: To provide the device-independent .IOPS commands SEEK, ENTER, and CLOSE. These commands are used to allow the FORTRAN IV Object Time System to communicate with .IOPS file-oriented devices.
 - a. SEEK finds and opens a named input file.
 - b. ENTER initiates and opens a named output file.
 - c. CLOSE terminates an input or an output file and must be used if SEEK or ENTER has been used.

NOTE

BACKSPACE, REWIND, and ENDFILE commands should never be used with a device that is operating in the file-oriented mode using the above subroutines.

3. Calling Sequences:

- a. To seek a named file:

CALL SEEK (N,A)

where N = device number

A = array name containing the 9-character 5/7 ASCII
file name and extension.

- b. To enter a named file:

CALL ENTER (N,A)

where N and A are the same as for SEEK.

- c. To close a named file:

CALL CLOSE (N)

where N is the same as for SEEK.

4. External Calls:

FIOPS, .DA, .SS.

5. Size: 174 octal locations

6. Error Conditions:

OTS Error 10 if I/O device number is illegal.

PDP-9 SCIENCE LIBRARY

INTRODUCTION

All mathematical routines in the PDP-9 Science Library are described in Part III. Most of the descriptive material is listed in Table III-1; in cases where detailed calculations or algorithms are involved, a reference (Δ) is made in column 1 to detailed descriptions following the table. Information given in Table III-1 for each routine includes the routine name, mnemonic, calling sequence, function, mode, errors, accuracy and timing (where available), storage requirements, and external calls. Routines are categorized by Intrinsic Functions, External Functions, Sub-Functions, or part of the Arithmetic Package and are listed in the table accordingly.

Intrinsic Functions

Intrinsic Functions are predefined subprograms that are part of the FORTRAN library. The type of each Intrinsic Function and its arguments is predefined and cannot be changed. Intrinsic Functions are referenced in a FORTRAN program by writing the function name along with the desired arguments in an appropriate FORTRAN statement.

Example:

$$X = \text{ABS}(A)$$
External Functions

External Functions are independently written programs that are executed each time their name appears in a FORTRAN program. Each External Function accepts one or more numerical arguments and computes a single result. SIN, COS, and ALOG are examples of external functions. All basic External Functions supplied with the FORTRAN system are described in Table III-1.

Sub-Functions

Sub-Functions are called by Intrinsic and External Functions, but are not directly accessible to the user via FORTRAN. For example, the Sub-Function .EB is called by the External Function SIN, and performs the actual computation of the sine.

The Arithmetic Package

The Arithmetic Package contains all arithmetic routines required for integer, real, and double precision arithmetic. Both EAE and non-EAE versions are available, depending upon the hardware.

Accumulators

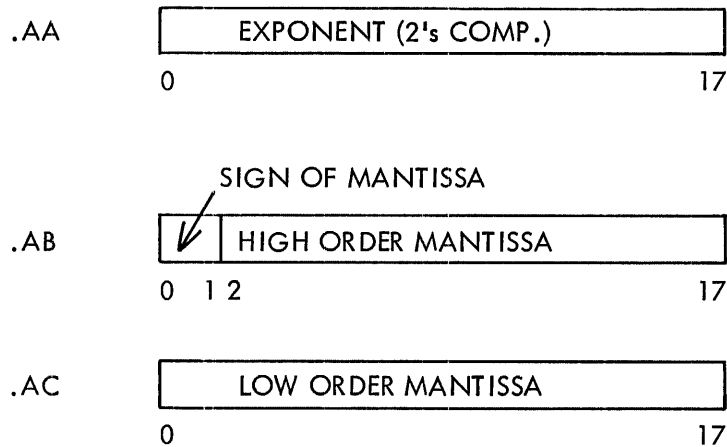
There are three accumulators referred to in the CALLING SEQUENCE column of the table. These include the A-register, the floating accumulator, and the held accumulator.

A-Register

The A-register is the standard hardware accumulator and is used in some of the computations that involve integer values. *occasionally*

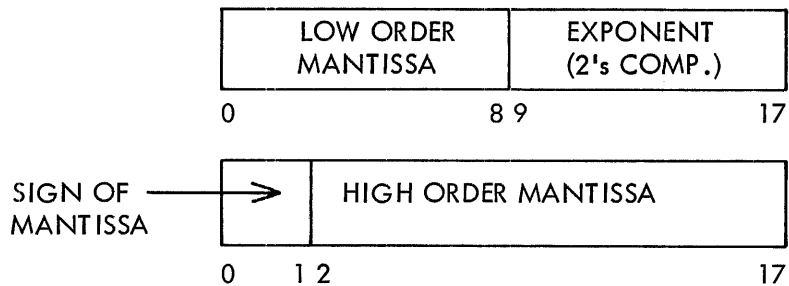
Floating Accumulator

The floating accumulator is a software accumulator that is included in the REAL ARITHMETIC package. It is a 3-word accumulator, .AA being the label of the first word, .AB the second, and .AC the third. Numbers are stored in this accumulator in the following format:



Negative mantissae are indicated with a change of sign.

Used by both the single and double precision routines, this format is also that of double precision numbers. Single precision numbers have a different format and must be converted before and after use in the floating accumulator. The format of single precision numbers is:



Held Accumulator

The held accumulator has the same format as the floating accumulator and is used as temporary storage by some routines. The labels of the three words are CE01, CE02, and CE03.

are not global

Calling Sequences

BUT = .CE-1, .CE-2, .CE-3

The MACRO-9 calling sequences, given in the third column of Table 3-1, assume in some cases where there are two arguments, that the appropriate accumulator has been loaded with the first argument. If the first argument is an integer value, it can be loaded into the A-register with a LAC instruction. If the first argument is a real or double precision value, the routines .AG and .AO, respectively, should be used to load the floating accumulator. The DAC instruction may be used to store the result of routines that return with an integer value in the A-register. The routines .AH and .AP should be used to store the result of routines that return with real or double precision values in the floating accumulator.

In calling sequences that use the .DSA pseudo operation to define the symbolic address of arguments, 400000 must be added to the address field if indirect addressing is involved.

FORTRAN library routines that are used in MACRO-9 programs must be declared with a .GLOBL pseudo operation in the MACRO-9 program. There must be agreement in the number and type of arguments between the calling program and the FORTRAN library routine.

The following example shows a section of a MACRO-9 main program that uses the FORTRAN External Function SIN.

```

      .TITLE
      .GLOBL      SIN, .AH
      :
      :
      JMS*        SIN
      JMP         .+2           /JUMP AROUND ARGUMENT
      .DSA        A            /+400000 IF INDIRECT
      JMS*        .AH          /STORE IN REAL FORMAT AT X
      .DSA        X
      :
      :
X      .DSA        0
      .DSA        0

```

When the above MACRO-9 program is loaded, the Linking Loader will attempt to satisfy the globals by searching the Science Library. The External Function SIN and the REAL ARITHMETIC package will be loaded. The references to these routines in the MACRO-9 program must be indirect (as indicated in the example) since only the transfer vectors are given in the main program.

Table III-1
PDP-9 Science Library

ROUTINE NAME	Mnemonic	Calling Sequence	Function	Mode	Errors	Accur. Bits	Timing		Storage (Octal)	External Calls
							Non- EAE	EAE		
INTRINSIC FUNCTIONS										
Exponentiation:										
Integer Base, Integer Exponent	.BB	LAC ARG1 (base) JMS* .BB LAC ARG2 (exp) JMS* SUBR .DSA ADDR of ARG2 (exp.)	I**K	I=I**I	None	N.A.	Note 1		45	INTEGER
Real Base, Integer Exponent	.BC		A**K	R=R**I	#13, if base ≤ 0	26	23.2 ms		44	.EE, .EF, REAL
DP Base, Integer Exponent	.BD		A**K	D=D**I	#14, if base ≤ 0	32	27.8 ms		46	.DE, .DF, DOUBLE
Real Base, Real Exponent	.BE		A**B	R=R**R	#13, if base ≤ 0	26	23.0 ms		20	.EE, .EF, REAL
Real Base, DP Exponent	.BF		A**B	D=R**D	#13, if base ≤ 0	26	27.6 ms		21	.EE, .DF, DOUBLE
DP Base, Real Exponent	.BG		A**B	D=D**R	#14, if base ≤ 0	32	27.6 ms		22	.DE, .DF, DOUBLE
DP Base, DP Exponent	.BH		A**B	D=D**D	#14, if base ≤ 0	32	26.6 ms		21	.DE, .DF, DOUBLE
Absolute Value:										
Real Absolute Value	ABS	JMS* SUBR JMP .+2 .DSA ADDR of ARG	A	R=ABS(R)	None	N.A.	120 μ s		16	.DA, REAL
Integer Absolute Value	IABS		I	I=IABS(I)	None	N.A.	64 μ s		14	.DA
DP Absolute Value	DABS		A	D=DABS(D)	None	N.A.	120 μ s		16	.DA, DOUBLE
Truncation:										
Real to Real Truncation	AIN	Sign of A times largest integer $\leq A$	R=AIN(R)	None	None	N.A.	365 μ s		15	.DA, REAL
Real to Integer Truncation	INT		I=INT(R)	None	None	N.A.	180 μ s		13	.DA, REAL
DP to Integer Truncation	IDINT		I=IDINT(D)	None	None	N.A.	180 μ s		13	.DA, REAL, DOUBLE
Remaindering:										
Real Remaindering	AMOD		Note 2	R=AMOD(R, R)	None	N.A.	3015 μ s		27	.DA, REAL
Integer Remaindering	MOD		Note 2	I=MOD(I, I)	None	N.A.	477 μ s		24	.DA, INTEGER
DP Remaindering	DMOD		Note 2	D=DMOD(D, D)	None	N.A.	3335 μ s		30	.DA, DOUBLE
Transfer of Sign:										
Real Transfer of Sign	SIGN	Sign of A1 ↓ Sign of A2	R=SIGN(R, R)	None	None	N.A.	198 μ s		26	.DA, REAL
Integer Transfer of Sign	ISIGN		I=SIGN(I, I)	None	None	N.A.	81 μ s		20	.DA
DP Transfer of Sign	DSIGN		D=SIGN(D, D)	None	None	N.A.	192 μ s		26	.DA, DOUBLE
Positive Difference:										
Real Positive Difference	DIM	A1-MIN(A1, A2) I1-MIN(I1, I2)	A1-MIN(A1, A2)	R=DIM(R, R)	None	N.A.	794 μ s		22	.DA, REAL
Integer Positive Difference	IDIM		I1-MIN(I1, I2)	I=IDIM(I, I)	None	N.A.	85 μ s		15	.DA, INTEGER
Conversion:										
Integer to Real Conversion	FLOAT	JMS* SUBR JMP .+2 .DSA ADDR of ARG	A $\leftarrow$ I	R=FLOAT(I)	None	N.A.	246 μ s		11	.DA, REAL
Real to Integer Conversion	IFIX		I $\leftarrow$ A	I=IFIX(R)	None	N.A.	180 μ s		13	.DA, REAL
DP to Real Conversion	SNGL		A $\leftarrow$ B	R=SNGL(D)	None	N.A.	144 μ s		27	.DA, DOUBLE
Real to DP Conversion	DBLE		A $\leftarrow$ B	D=DBLE(R)	None	N.A.	115 μ s		11	.DA, REAL

NOTES: [†] Timing indicated in this column is estimated unless indicated to be otherwise with a dagger (†). The dagger indicates actual, average-to-worst-case times based on arbitrarily chosen values.
1. Timing is dependent upon the size of the exponent, but is approximately equal to 335 μ s times n, where n is the largest power of 2 in the exponent.
2. Remaindering is defined as $A1 - [A1/A2] A2$, where $[A1/A2]$ is the integer whose magnitude does not exceed the magnitude of $A1/A2$ and whose sign is the same as $A1/A2$.

Table III-1 (cont)
PDP-9 Science Library

ROUTINE NAME	Mnemonic	Calling Sequence	Function	Mode	Errors	Accur. Bits	Timing		Storage (Octal)	External Calls
							Non- EAE	EAE		
INTRINSIC FUNCTIONS (Cont)										
Maximum/Minimum Value:										
Integer Maximum/Minimum	IMNMX	JMS* MAX0,MIN0, AMAX0, or AMIN0 JMP .+n+1 .DSA ADDR of ARG1 .DSA ADDR of ARG2	Max. Value	I=MAX0(I1,...,In)	None	N.A.	Note 3		106	INTEGER, REAL
Integer to Integer Max.	MAX0									
Integer to Integer Min.	MIN0		Min. Value	I=MIN0(I1,...,In)	None	N.A.	Note 3			
Integer to Real Max.	AMAX0	.DSA ADDR of ARGn	Max. Value	R=AMAX0(I1,...In)	None	N.A.	Note 4			
Integer to Real Min.	AMIN0		Min. Value	R=AMIN0(I1,...In)	None	N.A.	Note 4			
Real Maximum/Minimum	RMNMX	JMS* AMAX1,AMIN1, MAX1, or MIN2 JMP .+n+1 .DSA ADDR of ARG1 .DSA ADDR of ARG2	Max. Value	R=AMAX1(R1,...Rn)	None	N.A.	Note 5		117	INTEGER, REAL
Real to Real Max	AMAX1									
Real to Real Min.	AMIN1		Min. Value	R=AMIN1(R1,...Rn)	None	N.A.	Note 5			
Real to Integer Max.	MAX1	.DSA ADDR of ARGn	Max. Value	I=MAX1(R1,...Rn)	None	N.A.	Note 6			
Real to Integer Min.	MIN1		Min. Value	I=MIN1(R1,...Rn)	None	N.A.	Note 6			
DP Maximum/Minimum	DMNMX	JMS* DMAX1 or DMIN1 JMP .+n+1 .DSA ADDR of ARG1 .DSA ADDR of ARGn	Max. Value	D=DMAX1(D1,...Dn)	None	N.A.	Note 7		105	DOUBLE
DP Maximum	DMAX1									
DP Minimum	DMIN1		Min. Value	D=DMIN1(D1,...Dn)	None	N.A.	Note 7			
EXTERNAL FUNCTIONS										
Square Root:										
Real Square Root 1	SQRT		$X^{1/2}$	R=SQRT(R)	#5, ARG < 0	26	† 6.657ms	† 3.584 ms	66	.DA,.ER,REAL
DP Square Root 1	DSQRT		$X^{1/2}$	D=DSQRT(D)	#6, ARG < 0	34	† 8.191 ms	† 4.094 ms	66	.DA,.ER,DOUBLE
Exponential:										
Real Exponential 2	EXP	JMS* SUBR	e^X	R=EXP(R)	#13, ARG ≤ 0	26	† 15.489 ms	† 4.672 ms	13	.DA,.EF,.ER,REAL
DP Exponential 2	DEXP		e^X	D=DEXP(D)	#14, ARG ≤ 0	34	† 17.664 ms	† 7.223 ms	13	.DA,DF,.ER,DOUBLE
Natural Logarithm:										
Real Natural Logarithm 3	ALOG	JMP .+2 .DSA ADDR of ARG	$\log_e X$	R=ALOG(R)	#13, ARG < 0	26	† 8.197 ms	† 4.092 ms	20	.DA,.EE,.ER,REAL
DP Natural Logarithm 3	DLOG		$\log_e X$	D=DLOG(D)	#14, ARG < 0	32	† 15.489 ms	† 4.095 ms	21	.DA,.DE,.ER,DOUBLE
Common Logarithm:										
Real Common Logarithm 3	ALOG10		$\log_{10} X$	R=ALOG10(R)	#13, ARG < 0	26	† 8.197 ms	† 4.094 ms	20	.DA,.EE,.ER,REAL
DP Common Logarithm 3	DLOG10		$\log_{10} X$	D=DLOG10(D)	#14, ARG < 0	32	† 11.7 ms		21	.DA,.DE,.ER,DOUBLE
Sine:										
Real Sine 4	SIN		Sin (X)	R=SIN(R)	None	26	† 10.368 ms	† 4.094 ms	13	.DA,.EB,REAL
DP Sine 4	DSIN		Sin (X)	D=SIN(D)	None	34	† 16.383 ms	† 5.632 ms	13	.DA,DB,DOUBLE
Cosine:										
Real Cosine 4	COS		Cos (X)	R=COS(R)	None	26	† 11.025 ms	† 4.901 ms	20	.DA,.EB,REAL
DP Cosine 4	DCOS		Cos (X)	D=COS(D)	None	34	† 16.383 ms	† 6.145 ms	21	.DA,.DB,DOUBLE

NOTES: 3. 57 μ s + 40 μ s for each argument.
4. 242 μ s + 40 μ s for each argument.
5. 168 μ s + 624 μ s for each argument.
6. 233 μ s + 624 μ s for each argument.
7. 163 μ s + 607 μ s for each argument.

Table III-1 (cont)
PDP-9 Science Library

ROUTINE NAME	Mnemonic	Calling Sequence	Function	Mode	Errors	Accur. (Bits)	Timing		Storage (Octal)	External Calls
							Non- EAE	EAE		
EXTERNAL FUNCTIONS (Cont)										
Arctangent:										
Real Arctangent $\triangle 5$	ATAN	JMS* ATAN or DATAN JMP .DSA ADDR of ARG	$\tan^{-1}(a)$	R=ATAN(2)	None	26	16.352 ms	5.632 ms	13	.DA, .ED, REAL
DP Arctangent $\triangle 5$	DATAN	JMS* ATAN or DATAN JMP .DSA ADDR of ARG	$\tan^{-1}(a)$	D=DATAN(D)	None	34	14.6 ms		13	.DA, .DD, DOUBLE
Real Arctangent (x/y) $\triangle 5$	ATAN2	JMS* ATAN2 or DATAN2 JMP .+3	$\tan^{-1}(x/y)$	R=ATAN2(R, R)	None	26	12.4 ms		17	.DA, .ED, REAL
DP Arctangent (x/y) $\triangle 5$	DATAN2	JMS* ATAN2 or DATAN2 JMP .+3 .DSA ADDR of ARG1 .DSA ADDR of ARG2	$\tan^{-1}(x/y)$	D=DATAN2(D, D)	None	34	16.2 ms		17	.DA, .DD, DOUBLE
Hyperbolic Tangent $\triangle 6$	TANH	JMS* TANH JMP .+2 .DSA ADDR OF ARG	$\tanh(a)$	R=TANH(R)	None	26	16.383 ms	7.233 ms	47	.DA, .EF, REAL
SUB-FUNCTIONS										
Sine Computation:										
Real Sine $\triangle 4$	.EB		Sin (a)	R=.EB(R)	None	19	9.3 ms		100	.EC, .REAL
DP Sine $\triangle 4$	.DB		Sin (a)	D=.DB(D)	None	28	10.8 ms		116	.DC, .DOUBLE
Arctangent Computation:										
Real Arctangent $\triangle 5$	.ED	JMS* SUBR	$\tan^{-1}(a)$	R=.ED(R)	None	26	11.0 ms		65	.EC, .REAL
DP Arctangent $\triangle 5$	.DD	NOTE	$\tan^{-1}(a)$	D=.DD(D)	None	34	14.5 ms		144	.DC, .DOUBLE
Logarithm (Base 2) Computation:										
Real Log $\triangle 7$	.EE	Enter with argument in floating accumulator. Returns with result in floating accumulator.	$\log_2 a$	R=.EE(R)	#13, ARG ≤ 0	26	9.0 ms		71	.ER, .REAL
DP Log			$\log_2 a$	D=.DE(D)	#14, ARG ≤ 0	32	10.7 ms		101	.ER, .DOUBLE
Exponential Computation:										
Real Exponential $\triangle 2$	.EF		e^X	R=.EF(R)	None	26	12.2 ms		116	REAL
DP Exponential $\triangle 2$	.DF		e^X	D=.DF(D)	None	34	15.0 ms		137	DOUBLE
Polynomial Evaluation:										
Real Polynomial Evaluation $\triangle 8$	.EC	JMS* .EC or .DC CAL PLIST : :	$x = \sum_{i=0}^n C_{2i+1} Z^{2i+1}$	$R = .EC(R_2, R_1, \dots, R_n)$	None	N.A.	Note 8		44	REAL
DP Polynomial Evaluation $\triangle 8$	.DC	PLIST -N /-No. of terms +1 C _n /last term C _{n-1} /next to last : C ₁ /2nd term C ₀ /1st term	$x = \sum_{i=0}^n C_{2i+1} Z^{2i+1}$	$D = .DC(D_2, D_1, \dots, D_n)$	None	N.A.	Note 8		47	DOUBLE

NOTES: 8. 2.0 ms + 1.3 ms for each coefficient.

Table III-1 (cont)
PDP-9 Science Library

ROUTINE NAME	Mnemonic	Calling Sequence	Function	Mode	Errors	Accur. Bits	Timing †		Storage (Octal)	External Calls
							Non-EAE	EAE		
SUB-FUNCTIONS (Cont)										
General Get Argument	.DA	Routine that calls Calling Routine JMS* SUBR SUBR CAL 0 JMP .+n+1 JMS* .DA .DSA ARG1 JMP .+n+1 .DSA ARG2 (address of ARG1) ⋮ ⋮ (address of ARG2) DSA ARGn ⋮ (address of ARGn)	N.A.	N.A.	None	N.A.	Note 9		46	None
ARITHMETIC PACKAGE										
Integer Arithmetic:										
	INTEGE	ARG1 A-Register ARG2							Note 11	
Multiplication	.AD	Multiplicand Multiplier	I*J	I=I*I	None		† 281 μs	† 48 μs		
Division	.AE	Dividend Divisor	I/J	I=I/I	None		† 352 μs	† 55 μs		
Reverse Division	.AF	Divisor Dividend	J/I	I=I/I	None					
Subtraction	.AY	Minuend Subtrahend	I-J	I=I-I	None					
Reverse Subtraction	.AZ	Subtrahend Minuend	J-I	I=I-I	None					
Double Precision Arithmetic:										
	DOUBLE	ARG1 FL.ACC. ARG2							142	REAL
Load	.AO	Address	N.A.	D=.AO(D)	None	N.A.	† 70 μs			
Store	.AP	Value Address	N.A.	D=.AP(D)	None	N.A.	† 72 μs			
Add	.AQ	Augend Addend	A+B	D=D-D	None		† 255 μs			
Subtract	.AR	Minuend Subtrahend	A-B	D=D-D	None		† 324 μs			
Reverse Subtract	.AU	Subtrahend Minuend	B-A	D=D-D	None					
Multiply	.AS	Multiplicand Multiplier	A*B	D=D*D	None		† 1.937 ms	† 264 μs		
Divide	.AT	Dividend Divisor	A/B	D=D/D	None		† 1.327 ms	† 324 μs		
Reverse Divide	.AV	Divisor Dividend	B/A	D=D/D	None					
Real Arithmetic (Includes Floating):										
	REAL	ARG1 FL.ACC. ARG2							Note 12	
Load	.AG	Address	N.A.	R=.AG(R)	None	N.A.	† 67 μs			
Store	.AH	Value Address	N.A.	R=.AH(R)	None	N.A.	† 70 μs			
Add	.AI	Augend Addend	A+B	R=R+R	None		† 280 μs			
Subtract	.AJ	Minuend Subtrahend	A-B	R=R-R	None		† 385 μs			
Reverse Subtract	.AM	Subtrahend Minuend	B-A	R=R-R	None					
Multiply	.AK	Multiplicand Multiplier	A*B	R=R*R	None		† 2.047 ms	† 272 μs		
Divide	.AL	Dividend Divisor	A/B	R=R/R	None		† 1.537 ms	† 352 μs		
Reverse Divide	.AN	Divisor Dividend	B/A	R=R/R	None					

- NOTES: 9. 37 μs + 15 μs for each argument.
10. The sign of the result (the exclusive OR of the sign bits of .AB and CE02) is stored in .CE. The sign of .AB is saved in CE05.
11. 130₈ for EAE, 164₈ for non EAE.
12. 764₈ for EAE, 733 for non EAE.

Table III-1 (cont)
PDP-9 Science Library

	Mnemonic	Calling Sequence		Function	Mode	Errors	Accur. Bits	Timing		Storage (Octal)	External Calls	
								Non- EAE	EAE			
ARITHMETIC PACKAGE (Cont)												
Floating Arithmetic												
Float	.AW	A-Register	FL.ACC.	JMS* SUBR	A ← I	R=.AW(I)	None	N.A.	185 μs			
Fix	.AX	Integer	F.P. No.		I ← A	I=.AX(R)	None	N.A.	65 μs			
Negate	.BA		F.P. No.		A ← -A	R=.BA(R)	None	N.A.	10 μs			
Multiply	.CA	FL.ACC.	HELD ACC.	JMS* SUBR	A*B	R=R*R	None		774 μs			
Divide	.CI	Multiplicand	Multiplier		A/B	R=R/R	None		1124 μs	(avg)		
		Divisor	Dividend						1444 μs	(real)		
Add	.CC	Augend	Addend		A+B	R=R+R	None		300 μs	(DP)		
Normalize	.CD	Value			N.A.	R=.CD(R)	None	N.A.	160 μs	(avg)		
Hold	.CF	Value			N.A.	R=.CF(R)	None	N.A.	16 μs			
Round & Sign	.CH	Value		N.A.	R=.CH(R)	None	N.A.	30 μs				
Sign Control	.CG	Value	Value	Note 10	R=.CG(R)	None	N.A.	30 μs				
Short Get Argument	.CB	{ CAL 0 JMS* .CB CAL 0 DSA 0 }			N.A.	R=.CB(R)	None	N.A.	28 μs			

PDP-9 SCIENCE LIBRARY ALGORITHM DESCRIPTIONS

1. SQUARE ROOT (SQRT, DSQRT)

A first-guess approximation of the square root of the argument is obtained as follows.

If the exponent (EXP) of the argument is odd:

$$P_0 = .5^{\left(\frac{\text{EXP}-1}{2}\right)} + \text{ARG}^{\left(\frac{\text{EXP}-1}{2}\right)}$$

If the exponent (EXP) of the argument is even:

$$P_0 = .5^{\left(\frac{\text{EXP}}{2}\right)} + \text{ARG}^{\left(\frac{\text{EXP}}{2} - 1\right)}$$

Newton's iterative approximation is then applied three times.

$$P_{i+1} = \frac{1}{2} \left(P_i + \frac{\text{ARG}}{P_i} \right)$$

boncha damn morous!
 $P_{i+1} = \frac{1}{2} (P_i + \frac{\text{ARG}}{P_i})$

2. EXPONENTIAL (EXP, DEXP, .EF, .DF)

e^x is calculated as

$$\left(2^{x \log_2 e} \right) \left(2^{F \log_2 e} \right)$$

WHAT'S THIS?

and will have an integral portion (I) and a fractional portion (F). Then

$$e^x = (2^I) (2^F)$$

where $2^F = \left(\sum_{i=0}^n C_i F^i \right)^2$ and $n = 6$.

The values of C are:

$$C_0 = 1.0$$

$$C_1 = 0.34657359$$

$$C_2 = 0.06005663$$

$$C_3 = 0.00693801$$

$$C_4 = 0.00060113$$

$$C_5 = 0.00004167$$

$$C_6 = 0.00000241$$

3. NATURAL AND COMMON LOGARITHMS (ALOG, ALOG10, DLOG, DLOG10)

The exponent of the argument is saved as one greater than the integral portion of the result. The fractional portion of the argument is considered to be a number between 1 and 2. Z is computed as follows.

$$Z = \frac{X - \sqrt{2}}{X + \sqrt{2}}$$

$$\text{Then } \log_2 X = \frac{1}{2} + \left(\sum_{i=0}^n C_{2i+1} Z^{2i+1} \right)$$

where $n = 2$ for ALOG, and $n = 3$ for DLOG. The values of C are as follows.

<u>ALOG & ALOG10</u>	<u>DLOG & DLOG10</u>
$C_1 = 2.8853913$	$C_1 = 2.8853900$
$C_3 = 0.96147063$	$C_3 = 0.96180076$
$C_5 = 0.59897865$	$C_5 = 0.57658434$
	$C_7 = 0.43425975$

Finally,

$$\log_e X = (\log_2 X) (\log_e 2), \text{ for ALOG \& DLOG}$$

and

$$\log_{10} X = (\log_2 X) (\log_{10} 2), \text{ for ALOG10 \& DLOG10.}$$

4. SINE AND COSINE (SIN, COS, DSIN, DCOS, .EB, .DB)

The argument is converted to quarter circles by multiplying by $2/\pi$. The low two bits of the integral portion determine the quadrant of the argument and produce a modified value of the fractional portion (Z) as follows.

<u>Low 2 Bits</u>	<u>Quadrant</u>	<u>Modified Value (Z)</u>
00	I	F
01	II	1-F
10	III	-F
11	IV	-(1-F)

Z is then applied to the following polynomial expression.

$$\sin X = \left(\sum_{i=0}^n C_{2i+1} Z^{2i+1} \right)$$

where $n=4$ for REAL routines and $n=6$ for DP routines. The values of C are as follows.

REAL ROUTINES

$$\begin{aligned}
C_1 &= 1.570796318 \\
C_3 &= -0.6459637111 \\
C_5 &= 0.7968967928 \\
C_7 &= -0.00467376557 \\
C_9 &= 0.00015148419
\end{aligned}$$

DP ROUTINES

$$\begin{aligned}
C_1 &= 1.5707932680 \\
C_3 &= -0.6459640975 \\
C_5 &= 0.06969262601 \\
C_7 &= -0.004681752998 \\
C_9 &= 0.00016043839964 \\
C_{11} &= -0.000003595184353 \\
C_{13} &= 0.00000005446285
\end{aligned}$$

The argument for COS and DCOS routines is adjusted by adding $\pi/2$. The sin sub-function is then used to compute the cosine according to the following relationship:

$$\cos x = \sin \left(\frac{\pi}{2} + x \right)$$

5. ARCTANGENT (ATAN, DATAN, ATAN2, DATAN2, .ED, .DD)

For X less than or equal to 1, $Z = X$, and:

$$\text{arctangent } X = \left(\sum_{i=0}^n C_{2i+1} Z^{2i+1} \right)$$

where $n = 8$ for REAL routines and $n = 3$ for DP routines. For X greater than 1, $Z = 1/X$, and:

$$\text{arctangent } X = \frac{\pi}{2} - \left(\sum_{i=0}^n C_{2i+1} Z^{2i+1} \right)$$

where $n = 8$ for REAL routines and $n = 3$ for DP routines. The values of C are as follows.

REAL ROUTINES

$$\begin{aligned}
C_1 &= 1.570796327 \\
C_3 &= 0.9999993329 \\
C_5 &= -0.3332985605 \\
C_7 &= 0.1994653599 \\
C_9 &= -0.1390853351 \\
C_{11} &= 0.0964200441 \\
C_{13} &= -0.0559098861 \\
C_{15} &= 0.0218612288 \\
C_{17} &= -0.0040540580
\end{aligned}$$

DP ROUTINES

$$\begin{aligned}
C_1 &= 0.9992150 \\
C_3 &= -0.3211819 \\
C_5 &= 0.1462766 \\
C_7 &= -0.0389929
\end{aligned}$$

6. HYPERBOLIC TANGENT (TANH)

$$\tanh |X| = \left(1 - \frac{2}{1 + e^{2|X|}} \right)$$

e^X , calculated as $\left(2^{X \log_2 e} \right)$ will have an integral portion (I) and a fractional portion (F), then:

$$e^X = (2^I) (2^F)$$

where

$$2^F = \left(\sum_{i=0}^n C_i F^i \right)^2 \quad \text{and } n = 6$$

The values of C are as follows.

$$\begin{aligned} C_1 &= 1.0 \\ C_2 &= 0.34657359 \\ C_3 &= 0.06005663 \\ C_4 &= 0.00693801 \\ C_5 &= 0.00060113 \\ C_6 &= 0.00004167 \\ C_7 &= 0.00000241 \end{aligned}$$

7. LOGARITHM, BASE 2 (.EE, .DE)

The exponent of the argument is saved as one greater than the integer portion of the result. The fractional portion of the argument is considered to be a number between 1 and 2. Z is computed as follows.

$$Z = \frac{X - \sqrt{2}}{X + \sqrt{2}}$$

Then

$$\log_2 X = \frac{1}{2} + \left(\sum_{i=0}^n C_{2i+1} Z^{2i+1} \right)$$

where $n = 2$ for .EE and $n = 3$ for .DE. The values of C are as follows.

<u>.EE</u>	<u>.DE</u>
$C_1 = 2.8853913$	$C_1 = 2.8853900$
$C_3 = 0.96147063$	$C_3 = 0.96180076$
$C_5 = 0.59897865$	$C_5 = 0.57658434$
	$C_7 = 0.43425975$

8. POLYNOMIAL EVALUATOR (.EC, .DC)

The polynomial is evaluated as follows.

$$X = Z (C_0 + Z^2 (C_1 \dots + Z^2 (C_n Z^2 + C_{n-1})))$$