DEC-9A-AF40-D

FORTRAN IV

ADVANCED
SYSTEM
SOFTWARE

PDP-9

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

DEC-9A-AF4Ø-D

FORTRAN IV LANGUAGE MANUAL

PDP-9

ADVANCED SYSTEM SOFTWARE

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

PREFACE

This manual describes the FORTRAN IV language for the PDP-9. It provides users with the necessary information for writing FORTRAN programs for compilation and execution in the PDP-9 ADVANCED Software System.

Operating instructions will be given in later publications. In paper tape systems, the compiler along with an Input/Output Monitor is loaded from paper tape. In larger systems with a bulk storage device such as DECtape, the Keyboard Monitor accepts direct console commands to load the compiler in an I/O device independent environment.

Several excellent texts are available for a more elementary approach to FORTRAN programming. "A Guide to FORTRAN Programming," by Daniel D. McCracken (published by John Wiley and Sons, Inc.) is recommended.

This is essentially the language of USA Standard FORTRAN (X3.9-1966) with the exceptions noted in appendix 2.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

TABLES

CHAPTER 1
INTRODUCTION

1.1 FORTRAN

FORTRAN (for FORmula TRANslation) is the most widely used programming language for engineering and scientific applications. It consists of precise procedures for expressing numerical compilations.

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	$

1.1.1 Card Format (IBM Model 029 Keypunch codes)

The FORTRAN source program is written on a standard FORTRAN coding sheet, 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 1-5 of the first line of that statement. This number is made up of digits only, and may contain one to five digits. Leading zeros and all blanks in this field are ignored. The statement numbers are used for identification only, and they may be assigned in any order.

The FORTRAN compiler ignores the last eight columns (columns 73-80) 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.

1.1.2 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 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.

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:

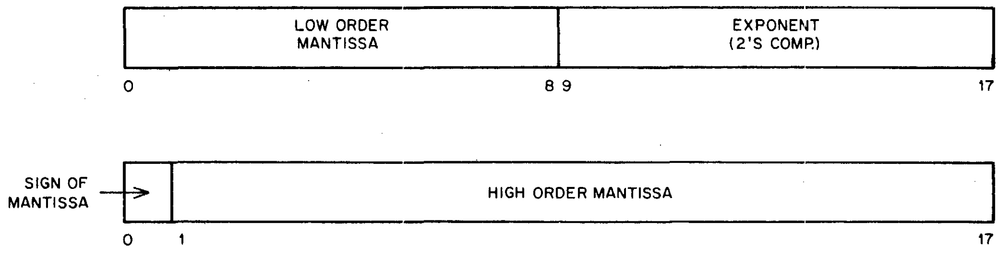
a. A number consisting of one to six significant decimal digits with a decimal point included someplace within the constant. A + or − sign preceding the number is optional.

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:



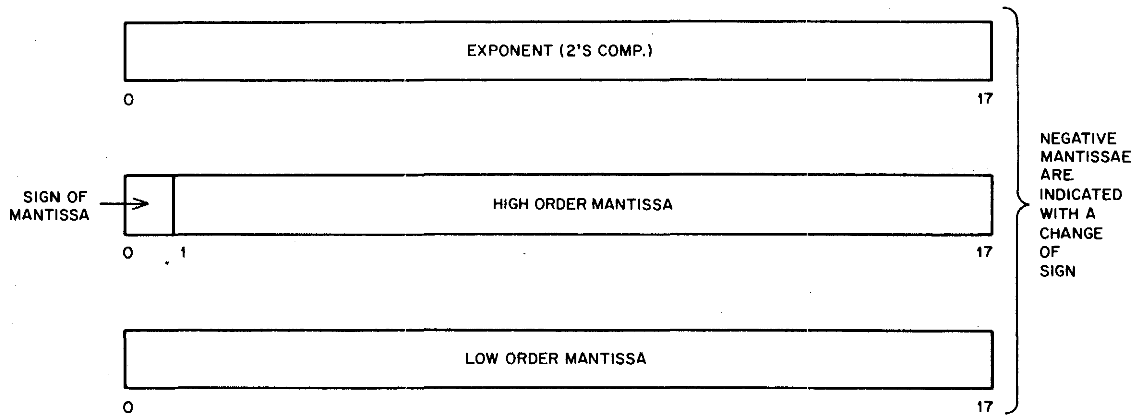
Negative mantissae are indicated with a change of sign.

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 righmost bit always zero. Examples:

```
1HA
4H A$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 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
```

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,..., 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)   |
A(2,1,1)   |
A(1,2,1)   |
A(2,2,1)   |     ascending absolute
A(1,1,2)   |     storage locations
A(2,1,2)   |
A(1,2,2)   |
A(2,2,2)   v
```

2.3.2 Subscript Expressions

Subscripts may be written in any of the following forms:

```
V
C
V + k
V – k
C * V
C * V + k
C * V – k
```

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

```
I
13
IMOST + 3
ILAST – 1
5 * IFIRST
2 * J + 9
4 * M1 – 7
```

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:

```
A (I)
B (I, J – 3)
BETA (5 * J + 9, K + 7, 6 * JOB)
```

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 determine 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 D + R D + D	Double-precision result

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

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:

FUNC + A*B/C−D(I,J) + E**F*G−H

interpreted as,

FUNC+ ((A*B)/C) − D(I,J) + (E^F*G)−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 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 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 of 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. 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.

b. The logical operator .NOT. need only be followed by a logical expression, while the logical operators .AND. and .OR. must be both preceded and followed by a logical expression for form more complex logical expressions.

c. 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 nonsubscripted) 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:

V Mode	E Mode	Assignment Rule
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 (n_1, n_2, ..., n_m)

where i is an nonsubscripted integer variable reference appearing in a previously executed ASSIGN statement, and n_1, n_2, ..., 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

GO TO (n_1, n_2, ..., n_m), i

where n_1, n_2, ..., 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:

```
GO TO (3, 17, 25, 50, 66), 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

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:

```
IF (AUB (I) - B*D) 10, 7, 23
```

4.6 LOGICAL IF STATEMENT

The general format of a logical IF statement is

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

DO n i = m_1, m_2, m_3

or

DO n i = m_1, m_2

where n is a statement number representing the terminal statement or the end of the "range"; i is a nonsubscripted 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 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.

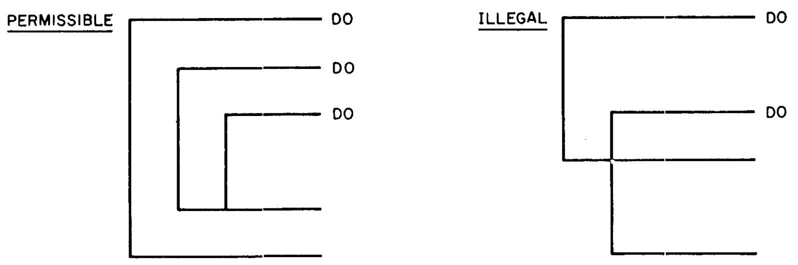
The initial value is the value of the index at the time the range is executed for the first time.

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.

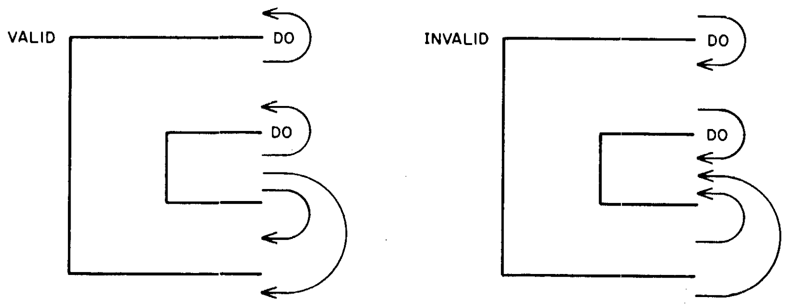
The increment is the amount by which the index is to be increased after each execution of the range. If the increment is omitted, a value of 1 is implied. Example:

```
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 operator intervention, 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 must 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

$C_1, C_2, \ldots, C_n$

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.

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 of READ:

```
READ (3,13) A,B,C
READ (2,10) A, (B (I), I = 1,5)
READ (1,3)
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 (I), (C (I,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 (S_1, S_2, ..., S_n)

where S_1.... 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.

Internal	Type	External
Integer variable	I	Decimal integer
Real variable	E	Floating-point, scaled
Real variable	F	Floating-point
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:

Type	Purpose
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):

Format Descriptor	Input	Internal	Output
I5	bbbbb	+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 ± 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 ± XX. Examples:

Format Descriptor	Input	Internal	Output
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:

Format Descriptor	Input	Internal	Output
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 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:

Format Descriptor	Input	Internal	Output
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-, 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 F, E, and D input conversions (when no exponent exists in the external field) the scale factor is defined as external quantity = internal quantity x 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:

Format Descriptor	Input	Scale Factor	Internal	Output
F6.3	123456	−3	+123456.	23.456
E12.4	123456	−3	+12345.6	bb0.0001E+08
D10.4	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: $nHa_1 a_2a_3 \ldots a_n$

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 is 777777_8 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

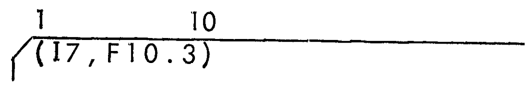
FORTRAN provides the facility of including the formatting data along with the input data. This is done by using an array name in place of the reference to a FORMAT statement label in any of the formatted I/O statements. For an array to be referenced in such a manner, the name of the variable FORMAT specification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

The statements have the general form:

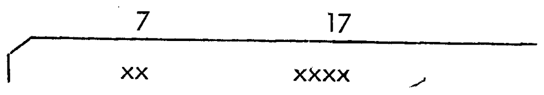
```
READ (u, name)
READ (u, name) list

WRITE (u, name)
WRITE (u, name) list
```

The form of the FORMAT specification which is to be inserted in 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 in 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,



the subsequent cards would contain data in the general form,



```
   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:

Character	Vertical Spacing Before Printing
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.

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, with the exception of the DATA statement, must appear before any executable code generating statement. They must appear in this order: type statements, DIMENSION statements, COMMON statements, and EQUIVALENCE statements. EXTERNAL statements may appear anywhere after all type statements and before the executable code generating statements. The DATA statements may appear anywhere in the source program.

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 (i_1), V_2 (i_2), ... $V_n(i_n)$

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 $/x_1/a_1/x_2/a_2/ \ldots /x_n/a_n$

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
```

then 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 (k_1), (k_2), ..., (k_n)

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

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:

DATA $k_1/d_1/$, $k_2/d_2/$, ... $k_n/d_n/$

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     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 exectuable statements of the program unit of which they are a part. The general format of a statement function is:

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

where f is a function name; the a's are nonsubscripted 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 nonsubscripted 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) = 3.*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 1).

TABLE 1 INTRINSIC FUNCTIONS

Intrinsic Functions	Definition	No. of Argu-ments	Symbolic Name	Type of Argument	Type of Function
Absolute value	$\lvert a \rvert$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
Truncation	Sign of a times largest integer $\leq \lvert a \rvert$	1	AINT	Real	Real
			INT	Real	Integer
			IDINT	Double	Integer
Remaindering*	a_1 (mod a_2)	2	AMOD	Real	Real
			MOD	Integer	Integer
Choosing largest value	Max (a_1, a_2, ...)	2	AMAXO	Integer	Real
			AMAXI	Real	Real
			MAXO	Integer	Integer
			MAXI	Real	Integer
			DMAXI	Double	Double
Choosing smallest value	Min (a_1, a_2, ...)	2	AMINO	Integer	Real
			AMINI	Real	Real
			MINO	Integer	Integer
			MINI	Real	Integer
			DMINI	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 $\lvert a_1 \rvert$	2	SIGN	Real	Real
			ISIGN	Integer	Integer
			DSIGN	Double	Double
Positive difference	a_1 - 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 (a_1, a_2, ..., a_n)
(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 nonsubscripted 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 nonsubscripted 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 subroutine 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

Table 2 contains the basic external functions supplied by the FORTRAN System.

TABLE 2 EXTERNAL FUNCTIONS

Basic External Function	Definition	No. of Argu-ments	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
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 $a_1 - [a_1/a_2]\, a_2$, where [x] is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as 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, ..., 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 nonsubscripted 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, ..., 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 don't 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     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	15
CALL name (a_1, a_2, ...)	42
CONTINUE	19
DO n i=m_1, m_2, m_3	17
GO TO n	15
GO TO i, (n_1, n_2, ...)	15
GO TO (n_1, n_2, ...), i	16
IF (e_1) n_1, n_2, n_3	16
IF (e) s	16
PAUSE	19
PAUSE n	19
STOP	19
STOP n	19
END	19
RETURN	40
SUBROUTINE NAME (a_1, a_2, ...a_n)	42
FUNCTION NAME (a_1, a_2, ...a_n)	40

INPUT/OUTPUT STATEMENTS

BACKSPACE u	30
END FILE u	31
FORMAT (s_1, s_2, ...,s_n)	24
READ (u, f)	23
READ (u, f) list	23
READ (u)	23
READ (u) list	23
READ (u, name)	29
READ (u, name) list	30
REWIND u	30
WRITE (u, f) list	23

APPENDIX 2
A NOTE ON USA STANDARD FORTRAN IV

The FORTRAN language used in this manual is essentially the language of USA 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 named COMMON.

d. The implied DO feature is not legal in a DATA statement.

digital

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

Printed in U.S.A. 1/67