FORTRAN II

BASIC SYSTEM

PDP-9
PROGRAMMING MANUAL

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

Page Missing From Original
Document

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

ILLUSTRATIONS

TABLES

CHAPTER 1
INTRODUCTION TO THE FORTRAN II LANGUAGE

1.1 INTRODUCTION

FORTRAN II, like all compilers, relieves the programmer from exercising a detailed knowledge of the computer language. It is problem oriented and thus accepts input closely related to the problem and converts this input into an executable machine language program. For scientists and engineers, the PDP-9® FORTRAN II system provides, in easily understood form, the means for writing PDP-9 FORTRAN II programs. The compiler accepts input in the form of statements which resemble mathematical formulas, and compiles the sequences of instructions needed to perform the procedures specified. Using this system, the programmer is able to concentrate on the problem rather than on detailed computer codes.

The PDP-9 FORTRAN II compiler can compile and run FORTRAN II programs written for other computers, provided that reasonable restrictions (such as sufficient memory capacity, use of acceptable terms and expressions; and availability of required peripheral equipment) are met. The PDP-9 FORTRAN II system includes a compiler, assembler, operating system, and subroutine library. Each of these subsections is described in later chapters. The manual is intended as a reference manual and assumes that the reader is familiar with the general principles of FORTRAN II programming.

1.2 FORTRAN II LANGUAGE

PDP-9 FORTRAN II is composed of symbols which combine to form words or are used as punctuation, and grouped into statements. These statements may be classified as follows:

a. Arithmetic Statements resemble algebraic formulas. They specify the mathematical operations to be performed.

b. Program Control Statements direct the sequence of operations of the program.

c. Specification Statements allocate data storage, determine variable and data types, and specify input/output formats.

d. Input/Output Statements control the transfer of information into and out of the computer.

The rules of FORTRAN are somewhat stylized to permit ease in interpretation by the computer, as shown in figure 1.

Symbols which are meaningful to FORTRAN include letters, numbers, and various special characters. The complete set is listed in appendix 3. Since FORTRAN ignores spaces, they may be used freely (except for two restrictions which will be noted in the text) to make a program more readable.

Order No. DEC-9B-AFT0-D

```
      FACTORIAL PROGRAM
C     THIS PROGRAM CALCULATES IX FACTORIAL FOR GIVEN IX
5     WRITE 2, 100
10    READ 1, 200, IX
      IFACT = IY = 1
      IF (IX) 5, 32, 30
30    IF (IX-IY) 41, 32, 33
32    WRITE 2, 300, IX, IFACT
      GO TO 10
33    IFACT = IFACT * (IY = IY + 1)
      GO TO 30
41    PAUSE 7777
      GO TO 5
100   FORMAT (/30H PLEASE TYPE A POSITIVE NUMBER/)
200   FORMAT (I4)
300   FORMAT (/I4, 13H FACTORIAL IS, I7/)
      END
```

Figure 1 A FORTRAN Program

1.2.1 Preparing the FORTRAN Program

Each line of a FORTRAN program contains two fields (see figure 2): the first is an identification field; the second, the statement proper.

1.2.1.1 The Identification Field - This field extends from the left-hand margin to the first tabulation, and may be left blank. If not, it may contain in the leftmost position one of the following types of identification:

a. The first digit of a statement number, which may be any integer from 1 to 99999, inclusive, identifying the statement on that line, for reference by other parts of the program. Statement numbers are used for program control or to assist the programmer in identifying segments of his program.

b. The letter C which identifies the remainder of the line as a comment. Although a FORTRAN program, using English words and mathematical symbols, can be read and understood more easily than a symbolic language program, comments throughout the program explain the procedures being used. Such comments, identified by a C in the first position of the identification field, are not interpreted by the compiler and have no effect on the executable program. Figure 2 is a section of a program with comments.

c. The letter S, which identifies the remainder of the line as symbolic machine instructions.

```
C     CALCULATE PERCENTAGE OF CORRECT RESPONSES
C     PERCENTAGE = -1 IF THERE ARE NO ITEMS IN CATEGORY
      DO 47 I = 1, 57
      DO 48 J = 1, 6
      IF (ITEMS (I, J)) 46, 46, 49
46    PERCEN (I, J) = -1.0
         .
         .
         .
```

Figure 2 Program Section with Comments

d. The continuation character $, which identifies the statement as a continuation of the preceding statement. Frequently a statement may be too long to fit on one line (this is especially true of format statements). If the character $ begins the identification field of a line, the statement field of that line is treated as a continuation of the statement on the line above. A statement may be continued on as many lines as necessary to complete it, but the maximum number of characters in the statement may not exceed 300 (approximately 4-1/2 lines). Figure 3 is an example of the continuation character.

```
3     X=X+(ARG1+2.* ARG2+2. *ARG3+
$     ARG4)/6
```

Figure 3 Example of the Continuation Character

1.2.1.2 The Statement Field - This field begins immediately after the first tabulation and extends through the next carriage return. Thus, no more than one statement can be written on one line, but a single statement can extend over one line using the continuation character.

1.2.2 Required Statements

Figure 4 is an example of a FORTRAN program, consisting of the title, the body of the program, and the END statement.

The first line of the program is the title, which may be anything the programmer wishes to write to identify his program. The title is not incorporated into the final executable program. Note that although a title is necessary, it need not be preceded by a C. A carriage return, line feed before the title is optional.

The body of the program is a series of statements, each of which specifies a sequence of mathematical operations, controls the flow of the program, or performs other tasks related to the proper working of the program.

```
      SUMMATION OF FIRST 50 INTEGERS
C     SET ITOTAL = 0 BEFORE SUMMING
      ITOTAL = 0
      DO 3 I = 1, 50
3     ITOTAL = ITOTAL +1
      END
```

Figure 4 Example of a FORTRAN Program

The END statement is a required statement and must be the last statement of every FORTRAN program. Its function is to indicate to the compiler that nothing more connected with the preceding program is to follow. The END statement should be terminated by carriage return, line feed, carriage return, line feed, form feed.

1.2.3 FORTRAN II Words

Words fall into three categories: numbers, variables, and commands. Numbers and variables are determined by the programmer and dealt with here, and commands are discussed in succeeding chapters.

1.2.3.1 Number Representation - In mathematics, there are many ways to categorize numbers. They may be positive or negative, rational or imaginary, whole numbers or fractions. In PDP-9 FORTRAN II, the treatment of numbers is separated into integers and real numbers (single decimals or numbers in decimal exponent form), distinguished as follows:

Integers are constants which are written without a decimal point. Typical integers are: 9, 17, −8192, 131071. The number 131071, $(2^{17}-1)$, is the largest magnitude that can be expressed as a FORTRAN integer. Fractional quantities and numbers larger than ± 131071 require real numbers.

Real numbers* have two forms: either they are simple decimals such as 0.0025, .4, −57., 2.71828; or they are numbers in decimal exponent form, a number multiplied by a power of 10. Examples:

Mathematical Form	FORTRAN Form
6.023×10^{23}	6.023E23
-1.66×10^{-16}	−1.66E−16
72×10^{12}	72E12

In general, a real number in decimal exponent form is expressed as ±nE±K where n may be an integer or simple decimal, and K is an integer exponent from 0 to 99, inclusive.

*This use of the term real should not be confused with the mathematical usage; in PDP-9 FORTRAN II real applies only in the limited sense described above.

Storage Modes - Another difference between PDP-9 FORTRAN II integers and real numbers is the manner in which each is represented in core memory.

A FORTRAN integer is stored as a binary number in one 18-bit computer word. This representation, shown schematically in figure 5a is called fixed point, because the decimal point is always considered to be to the right of the rightmost digit. Negative numbers are stored as the 1's complement of their magnitude, the leftmost bit being the sign bit.

A FORTRAN real number is stored as a binary number in floating-point representation. In this form, the number consists of two parts: an exponent and a mantissa. The mantissa is a decimal fraction with the decimal point assumed to be to the left of the leftmost digit. The mantissa is always normalized; that is, it is stored with leading 0s eliminated in its binary form, so that the high order bit is always 1. The exponent as stored represents the power of 2 by which the mantissa is multiplied to obtain the value of the number for use in computation.

There are two versions of the floating-point representation: normal, or three-word, mode and two-word mode. They differ in the number of words of core storage required, and, hence, in precision of the number.

The normal mode requires three 18-bit words of memory for each number. The exponent, a signed 17-bit integer (2's complement if negative), is stored in the first word. The mantissa is a 35-bit number stored in the second and third words. The sign of the mantissa is kept in the high-order bit of the second word. A negative mantissa involves a change of sign. Figure 5b is a schematic representation of a three-word floating-point number.

The second floating-point mode requires only two words of memory and can be used where space is at a premium and precision can be sacrificed. The exponent and its sign occupy the first nine bits of the first word; the mantissa occupies the rest of that word and all of the second. The sign of the mantissa is in the high-order bit of the second word. A negative mantissa involves a change of sign. Figure 5c is a schematic representation of a two-word floating-point number.

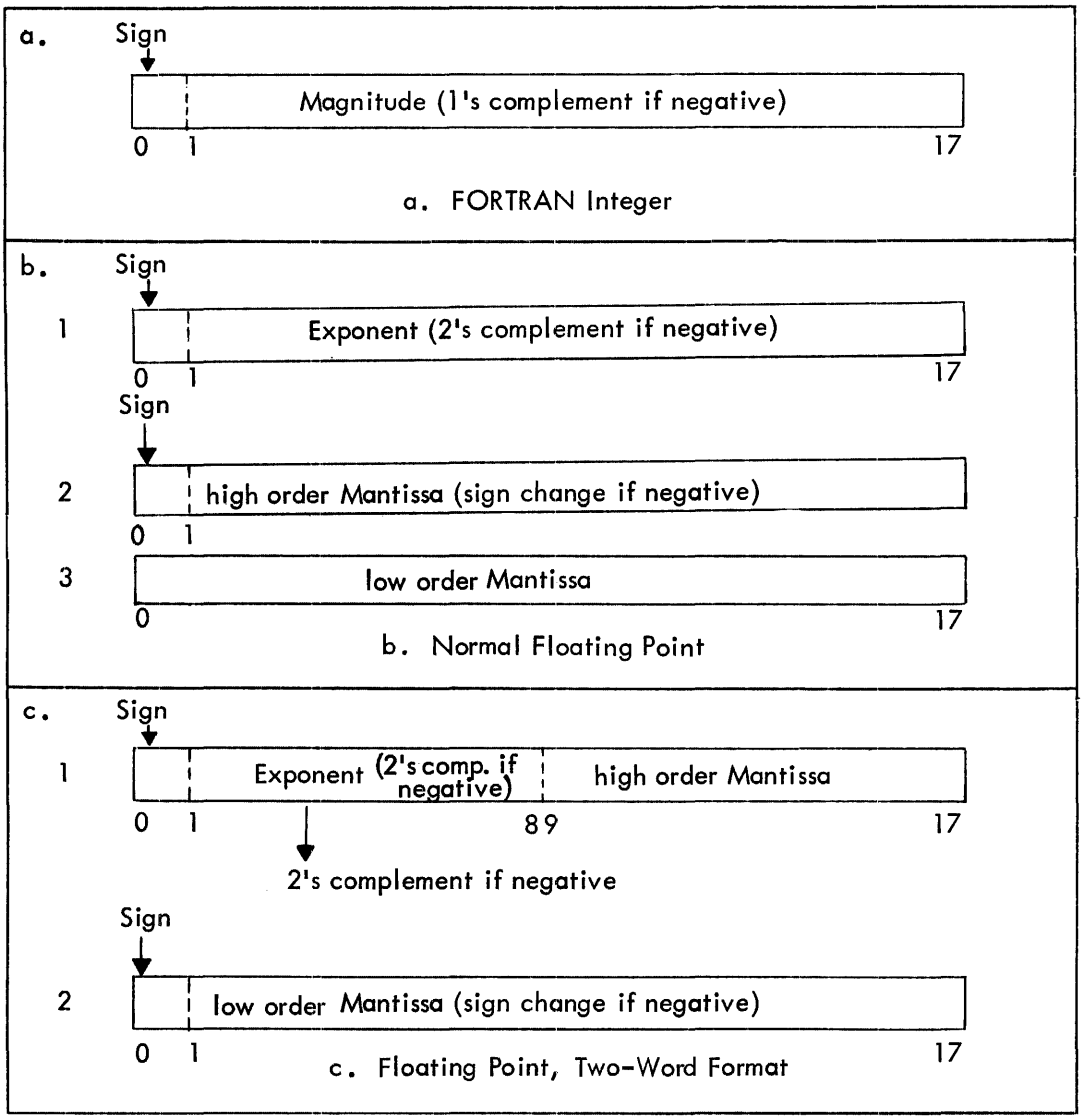

Figure 5 Number Representation, Floating Point

1.2.3.2 Variable Representation - The term "variable," as used in FORTRAN, means a quantity which may assume different values during different executions of a program or at different stages of a program's execution; hence, a variable name is a symbolic representation of this quantity. A variable name is composed of one or more characters according to three rules:

a. The only characters used in a variable name are A through Z and 0 through 9.

b. The first character must be alphabetic.

c. Only the first six characters of any variable name are meaningful; the compiler ignores all characters after the sixth.

Some examples of acceptable variables names are K, P51, ROB ROY, and EPSILON. The name ROB ROY represents one variable, not two, because blank spaces are ignored by FORTRAN. Thus, ROB ROY, ROBROY, or even ROBR OY are identical names and reference the same variable. The compiler interprets the name EPSILON as EPSILO, since only the first six characters are meaningful.

Care is necessary in selecting variable names. For example, the two names GEORGE1 and GEORGE 2 are considered identical because of the six-character restriction. Some incorrect variable names are 9SORT (first character not alphabetic) and GO#5 (illegal character included).

Since variables represent numeric quantities, the type of representation must be specified. In normal programming, variable types are specified using the standard FORTRAN conventions, as follows:

Integer variable names must begin with one of the letters I, J, K, L, M, or N.

Real variables are designated by names beginning with any other letter.

Typical integer variable names are INDEX, KDATA, M359, LIST8. Typical real variable names are XZERO, COUNT, FICA.

Subscripted Variables - An array is a grouping of data. A column of figures, the elements of a vector, a list, and a matrix are all arrays. In mathematics, an element of an array is referenced by a symbol denoting the array and subscripts identifying the position of the element. For example, the sixth element in a vector v is designated v_6. Likewise, the fourth element in the tenth column of a matrix b is identified as $b_{4,10}$. In general, an element of an n-dimensional array m is designated by $m_{i_1,i_2,i_3\ldots,i_n}$.

In PDP-9 FORTRAN II, array elements are similarly identified. The array is provided with a name, subject to the same rules as the names of variables. The name determines the mode, integer or real, of all the elements in the array. The subscripts which identify an element of the array are enclosed in parentheses and separated by commas. The two elements, v_6 and $b_{4,10}$, in FORTRAN would have the following form:

V(6) B(4,10)

Subscripts may be quite diverse in form; in fact, a subscript may be any acceptable FORTRAN arithmetic expression as long as it is integer-valued (i.e., floating-point quantities are not allowed).

Note that certain subscripts in figure 6 are themselves subscripted. Subscripting may be carried to four levels, although it is unusual to do so to more than two levels. Each subscripted subscript in figure 6, i.e., I(1), and K(2), is itself treated as a subscripted variable.

```
a. X (3,3)
b. C (I+1, J+1)
c. N (I(1), J(1), K(2))
d. Y (J/3 + (K-4))
```

Figure 6 Example of Subscripts

CHAPTER 2
ARITHMETIC AND DATA-SPECIFICATION STATEMENTS

The arithmetic statement relates a variable (V) to an arithmetic expression (E) by means of the equal sign (=), thus:

V = E

Such a statement looks like a mathematical equation, but it is treated differently. The equal sign does not merely represent a relation between left and right members, but specifies an operation to be performed; namely, replace the value of V with the value of E. A few illustrations of the arithmetic statement are given in figure 7.

```
a.  VMAX=V0+A*T0
b.  T = 2*PI*SQRTF(L/G)
c.  PI=3.14159
d.  THETA = OMEGA0*T + ALPHA*T**2/2
e.  MIN=MIN0
f.  INDEX = INDEX +2
```

Figure 7 Arithmetic Statements

2.1 ARITHMETIC EXPRESSIONS

The elements of an arithmetic expression are of four types: constants, variables, functions, and operators. An expression may consist of a single constant, a single variable, a function, or a string of constants, variables, and functions connected by operators.

Constants are explicit numerical quantities. They may be integers, decimals, or numbers in decimal exponent form.

Variables represent quantities whose values are not implicit; they may be redefined during execution of the program.

Functions are special forms of variables consisting of a name immediately followed by an argument enclosed in parentheses. The function name represents a mathematical operation to be performed on the argument such as finding the square root of a number or determining the sine or cosine of an angle. Certain basic functions are provided by the FORTRAN system and are called library functions. A detailed discussion of functions is found in chapter 5. Of interest here, however, is their treatment within an arithmetic expression:

Whenever a function is encountered, it is evaluated and the result is treated as a variable in the evaluation of the expression in which the function occurs.

Figures 8e and 8f illustrate the use of functions as variables in an arithmetic expression. Included in these examples are SINF(THETA) and COSF(THETA-1.5), corresponding to the trigonometric functions sine and cosine, and SQRTF(X), the square root operation.

Operators are symbols representing the common arithmetic operations:

Exponentiation	↑
Multiplication	*
Division	/
Addition	+
Subtraction	-
Equivalence	=

	Algebraic Expression	FORTRAN Expression
a.	$az^2 + bz + c$	A*Z↑2+B*Z+C
b.	$\frac{(a^2-b^2)}{(a+b)^2}$	(A↑2-B↑2)/(A+B)↑2
c.	$\frac{4\pi r^2}{3}$	4*PI*R↑2/3
d.	$\frac{3z^2-2(z+y)}{4.25}$	(3*Z↑2 - 2*(Z+Y))/4.25
e.	$a \sin \theta + 2a \cos (\theta-1.5)$	A*SINF(THETA)+2*A*COSF(THETA-1.5)
f.	$\frac{2\sqrt{z}}{3}$	2*SQRTF(Z)/3

Figure 8 Examples of Arithmetic Expressions

The important rule about operators in the FORTRAN arithmetic expression is that every operation must be explicitly represented by an operator symbol. In particular, the multiplication sign (*) must never be omitted. Likewise, since no superscript notation is available, a symbol for exponentiation (↑) is provided.

Figure 8 demonstrates the properties of arithmetic expressions. Each expression is shown with its corresponding algebraic form.

2.1.1 Evaluation of an Expression

Normally, a FORTRAN expression is evaluated from left to right just as an algebraic formula. As in algebra, however, there are exceptions. Certain operations are always performed before others, regardless of order. This priority of evaluation is as follows:

a.	Expressions within parentheses	()
b.	Unary Minus*	-
c.	Exponentiation	↑
d.	Multiplication	*
	Division	/
e.	Addition	+
	Subtraction	-
f.	Equivalence	=

The term "binding strength" refers to an operator's relative position in a table such as the one above. In it the operations are listed in the order of descending binding strength. Thus, exponentiation has a greater binding strength than addition, and multiplication and division have equal binding strength.

The left-to-right rule can now be stated a little more precisely:

> Operations are performed in order of decreasing binding strength. A sequence of operations of equivalent binding strength is evaluated from left to right.

2.1.2 Use of Parentheses

To change the order of evaluation, parentheses are required. Thus, the FORTRAN expression, A-B+C is algebraically evaluated as (a-b)+c, whereas A-(B+C) is evaluated as a-(b+c).

The expression	is evaluated as
A/B*C	$\frac{A}{B} \cdot C$
A/B/C	$\frac{\frac{A}{B}}{C}$
A↑B↑C	$(A^B)^C$

*The unary minus is the operator which precedes a quantity whose value is to be negated. A unary minus is recognized by the fact that it is preceded by another operator, not by an operand. Example:

A + B ↑ - 2/C - D

The first minus (indicating a negative exponent) is unary; the second indicating a subtraction) is binary.

Figure 8d illustrates the use of parentheses for grouping subexpressions within an expression. In algebra, several devices, such as square brackets ([ ]) and rococo brackets ({ }), are available for distinguishing between levels when nesting subexpressions. In FORTRAN, only parentheses are available, so the programmer must be especially careful to make certain that parentheses are properly paired; that it, in a given expression, the number of left parentheses must be equal to the number of right parentheses.

2.1.3 The Replacement (Equal) Sign

The equal sign has the lowest binding strength of all the operators; the whole of the expression on the right is evaluated before the replacement operation is performed. In an arithmetic statement, the value of the expression to the right of the equal sign replaces the value of the variable on the left.

By this definition the statement in figure 7f would mean, "Add two to the current value of INDEX. The result is the new value of INDEX."

All variables occurring to the right of an equal sign must have been previously defined. If the variable on the left of the equal sign was previously undefined, it will be defined by the arithmetic statement.

2.1.4 Internal Arithmetic Statement

The most important result of treating the equal sign as an operator is that it may be used more than once in an arithmetic statement. Consider the following:

```
Q = A/(V=SQRTF(2*G*Y))
```

Parentheses separate the internal arithmetic statement, V=SQRTF(2*G*Y), from the rest of the statement. The complete statement in this illustration is a concise way of expressing the following type of mathematical procedure:

Let $q = a/v$

where $v = \sqrt{2gy}$

In the single FORTRAN statement, both of these equations are evaluated and values are assigned to Q and V.

Another result of treating the equal sign as an operator is that there may be a series of replacements, A=B=C=D, in a single FORTRAN statement. Note that since the operand to the left of an equal sign must be a variable, only the rightmost operand, represented by D above, may be an arithmetic expression.

The statement is interpreted as follows: "Let the value of expression D replace the value of variable C, which then replaces the value of variable B" and so on*. In other words, the value of the rightmost expression is given to each of the variables in the string to the left. A common use for this construction is in setting up initial values:

```
VZERO=SZERO=AZERO=0
T=T1=T2=T3=60
P=P0=4*ATM-K
```

2.1.5 Mode of Computation

PDP-9 FORTRAN II does not restrict the use of variable types within an arithmetic expression. Integer and real variables and constants may be freely mixed. The order in which the quantities are encountered during the left-to-right evaluation determines the mode of computation; however, the result is always stored in the mode of the left-hand variable of the arithmetic statement. In general, the following rules apply:

a. For any expression, computations are carried out in fixed point until a floating-point quantity appears; thereafter, all computations are carried out in floating point.

b. An expression in parentheses is considered separately from the main computation; thus if a subexpression contains only integers, it is evaluated in fixed point. If necessary, the result is converted into floating point.

c. The value of an expression on the right of an equal sign is converted to the mode of the left-hand variable before storage, if necessary.

The following example illustrates the method of performing calculations in an arithmetic statement.

In evaluating the statement

```
        A  = C*V*(J+2)
let     T1 = C*V floating point
        T2 = J+2 fixed
the result, T2, is converted to floating point; then
        A  = T1*T2 floating
```

*This may seem at first to violate the left-to-right rule. Whenever an equal sign is encountered in scanning a statement, it cannot be executed until all operations of higher binding strength have been performed. Thus, execution of each equal sign (replacement) is deferred until the expression on the right has been evaluated. The replacements then occur in reverse order as the evaluation works back to the leftmost variable.

2.2 DATA-SPECIFICATION STATEMENTS

Data-specification statements fall into two categories: those relating to data handling and storage; and those dealing with input/output operations. This section discusses the first category (except for the COMMON statement which is directly related to subprograms and is described in chapter 5); the I/O specification statements are discussed in chapter 4.

2.2.1 Dimension Statements

Array names must be identified as such to the FORTRAN compiler. Three items of information must be provided in any program using arrays:

a. Which are the subscripted variables?

b. How many subscripts does each have?

c. What is the maximum dimension of each subscript? (When an array is used, a certain amount of storage space must be set aside for its elements, hence, this requirement.)

All the above information is provided by the following specification statement type:

DIMENSION A(I,J,K,L), B(I,J,K,L), C(I,J,K,L),....

where A, B, and C are array names, and the integer constants I, J, K, L, are the maximum dimensions of each subscript.

The rule governing the use of array names and the DIMENSION statement is as follows: All array names must appear in a DIMENSION statement, and the DIMENSION statement must precede the first use of any of the names appearing in its list.

DIMENSION LIST2 (30), MAT3(10,20), RE GRES(2, 2, 5)

In the statement above (under normal FORTRAN variable naming conventions), the names LIST2 and MAT3 designate integer arrays; that is, each element is an integer. The third name, RE GRES, designates a real array. The first array is a list of 30 elements maximum, so that 30 words of storage are set aside for its use. The second array is a matrix of 10 rows and 20 columns, making a total of 200 elements requiring 200 words of space. The third array is three-dimensional and real. There are 2 x 2 x 5 = 20 elements, each requiring 3 words of storage for floating-point representation, so that 60 words will be set aside for the array. A maximum of 4000 words is normally set aside for storage of arrays.

If a subscript is subscripted, the name of the higher-level subscript must also appear in a DIMENSION statement. For example, a program in which the following statement appears:

A(1(1),J(2),K(10)) = B(1(10),J(2),K(1))

could contain a DIMENSION statement like the following:

```
DIMENSION A (5,5,10),B(10,5,5),I(10),J(10),K(10)
```

If an array name is to be passed as an argument from one program to another (e.g., a subroutine provides values to a variable array in the calling program), both the calling program and the subroutine must agree in floating point storage mode (three-word or two-word). This restriction does not apply when the argument is a specific array element rather than an array name.

When referencing dimensioned variables, use the correct number of subscripts. For example:

```
DIMENSION    A(10,10,10)
      :
A(3,4,6)     =  42 (correct).
A(705)       =  42. (will cause haphazard
                results at object time).
A = 3.          (will also cause undesired
                results).
```

2.2.2 Floating-Point Storage Specification

Unless otherwise indicated, all real numbers in a given program are stored in three-word form where 35 bits are reserved for the magnitude of each variable or constant. If the two-word form is desired (26 bits reserved for magnitude), the following specification statement must appear as the first statement of the program to which it applies:

```
2WORD
```

The two modes may not be mixed within any one program or subprogram, and they may only be mixed between programs when the level (depth) of call is at most one; e.g., a subprogram does not call other subprograms. (Refer to section 1.2.3.1 for a discussion of storage modes.)

CHAPTER 3
PROGRAM CONTROL

Ordinarily, FORTRAN statements are executed in the order in which they are written unless contrary instructions are given. The instructions provided by the program control statements allow the programmer to alter the sequence, repeat sections, suspend operations, or halt the program.

3.1 BRANCHES AND LOOPS

3.1.1 Unconditional GOTO Statements

There are various ways in which program flow may be directed. As shown schematically in figure 9, a program may have a straight-line sequence (a), branch to an entirely different sequence (b), return to an earlier point (c), or skip to a later point (d).

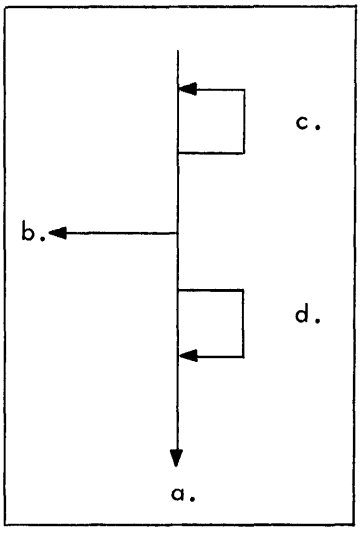

Figure 9 Schematic Representation of Program Branches

All of the branches can be performed in several ways, but the simplest is by the statement

GOTO n

where n is a statement number used in the program. This statement is described in the following example, which also illustrates the construction of a loop, the name given to program branches of the type shown in figure 9c.

```
      SUM OF FIRST N INTEGERS BY ITERATION
      KSUM=0
      INUM=1
2     KSUM=INUM+KSUM
      INUM=INUM+1
      GOTO 2
      END
```

Figure 10 Integer Summation

In figure 10, the sum of successive integers is accumulated by repeated addition. The main computation is provided by the three-instruction loop beginning with statement 2. The statements preceding this loop provide the starting conditions; this is called initialization. The partial sum (KSUM) is set to 0, and the first integer is given the value 1. The loop then proceeds to add the integer value to the partial sum, increment the integer, and repeat the operation.

3.1.1.1 The IF Statement - The program shown in figure 10 performs the required computation, but there is one flaw: the loop is endless. To get out of the loop, iteration must be stopped and the next step must be determined.

The IF statement fulfills both requirements. It has the following form:

IF (e) k,l,m

where e is any arithmetic expression, and k, l, and m are statement numbers. The IF statement is interpreted in this way:

If the value of e is less than 0, go to statement k.
If the value of e is equal to 0, go to statement l.
If the value of e is greater than 0, go to statement m.

Thus, the IF statement makes the decision of when to stop by evaluating an expression, and also provides program branch choices which can depend on the results of the evaluation. Figure 11 illustrates the use of the IF statement in the integer summation problem of figure 10.

```
      SUM OF THE FIRST 50 INTEGERS
      KSUM=0
      INUM=1
2     KSUM=INUM+KSUM
      INUM=INUM+1
      IF (INUM-50) 2,2,3
3     STOP
      END
```

Figure 11 Use of IF Statement in Integer Summation Problem

In this example, the initialization and main loop are the same as for figure 10, except that the GOTO statement of the earlier program has been replaced by an IF statement. This statement says: If the value of the variable INUM is less than or equal to 50 (which is the same as saying that the value of the expression INUM-50 is less than or equal to 0), go to statement 2 and continue the computation. If the value is greater than 50, stop.

A loop may also be used to compute a series of values. The following example is a program to generate terms in the Fibonacci series of integers, in which each succeeding member of the series is the sum of the two members preceding it: $k_n = k_{n-1} + k_{n-2}$.

```
      FIBONACCI SERIES, 100 TERMS
      DIMENSION FIB (100)
      FIB (1)=1
      FIB (2)=1
      K=3
 5    FIB(K)=FIB(K-1) + FIB(K-2)
      K=K+1
      IF (K-100) 5,5,10
10    STOP
      END
```

Figure 12 Fibonacci Series

In this program, initialization includes a DIMENSION statement to reserve space in memory for the results, and two statements which provide the starting values necessary to generate the series. Each time a term is computed, the subscript is indexed so that each succeeding term is stored in the next location in the table. As soon as the subscript becomes greater than 100, the calculation stops.

3.1.2 DO Loops

Iterative procedures such as the loop in figure 12 are so common that a more concise way of implementing them is warranted. In that example, three statements were required to initialize the subscript, increment it, and test for termination. The DO statement combines all these functions:

DO n i=k1,k2,k3

where n is a statement number, i is a simple integer variable, and k1, k2, and k3 are simple integer variables or constants used as indexing parameters to provide, respectively, the initial values of i, the final (terminating) value of i, and the indexing increment of i. The DO statement may be paraphrased as: "DO through statement n for i = k1; after statement n is completed, increment i by k3; if i is less than or equal to k2, repeat the sequence; otherwise exit from the DO loop and continue on in the

program." Upon normal exit from a DO loop, the value of the DO variable (i) will be the one generated at the final test, that is, greater than k2. If k3 is equal to 1, it may be omitted. Figure 13 shows the Fibonacci series calculation programmed using a DO loop.

```
      FIBONACCI SERIES, 100 TERMS
      DIMENSION FIB(100)
      FIB (1)=1
      FIB (2)=1
      DO 5 K=3,100
5     FIB(K)=FIB(K-1)+FIB(K-2)
      STOP
      END
```

Figure 13 Fibonacci Series Calculation Programmed as a DO Loop

The DO statement is interpreted thus: Do the sequence of statements up to and including statement 5, then index K by 1. If the new value of K is greater than 100, go to the statement following statement 5.

DO loops are commonly used in computations with multiple-subscripted variables. In these cases, it is usually necessary to perform loops within loops. Such nesting of loops is permitted in FORTRAN. A simple illustration is the initialization of array storage, shown in figure 14.

Initialize two 30 x 50 matrices and one 30-element vector, by setting the allotted storage space to 0. Note that the PDP-9 FORTRAN II, the initial condition of variable storage is never implicitly cleared.

```
C     INITIALIZATION OF STORAGE
      DIMENSION MAT1(30,50), MAT2(30,50), VEC3(30)
      DO 20 I=1, 30
      DO 10 J=1, 50
      MAT1(I, J)=0
10    MAT2(I, J)=0
20    VEC3(I)=0
      .....
      .....
```

Figure 14 Initialization of Array Storage

This sequence causes storage to be cleared in the inner loop column by column for each matrix, while the outer loop advances the column index and clears the elements of the vector.

3.1.2.1 General Rules for DO Loops - The following general rules about DO loops must be observed:

a. DO loops may be nested, but they may not overlap. Nested loops may end on the same statement, but an inner loop may not extend beyond the last statement of an outer loop. Figure 15 schematically illustrates permitted and forbidden arrangements. Those in 15a are permitted; loops 5, 6, and 7 end on the same statement. The arrangements in 15b are not permitted; loop 2 ends on a DO statement, loop 3 extends beyond outer loop 1.

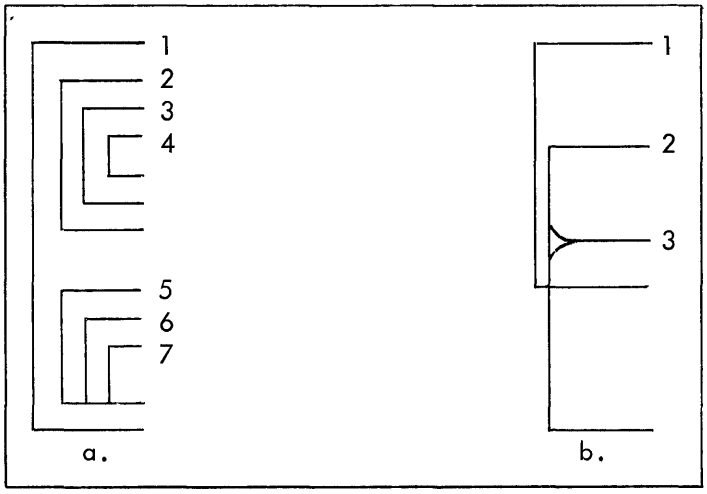

Figure 15 DO Loops

b. A program branch may not occur from a point outside a DO loop to a point inside, or from an outer DO to within an inner DO. Branches out of DO loops are permissible, however. Figure 16 illustrates this rule. Branches 2, 5, 6, and 7 are permitted; branches 1, 3, and 4 are not.

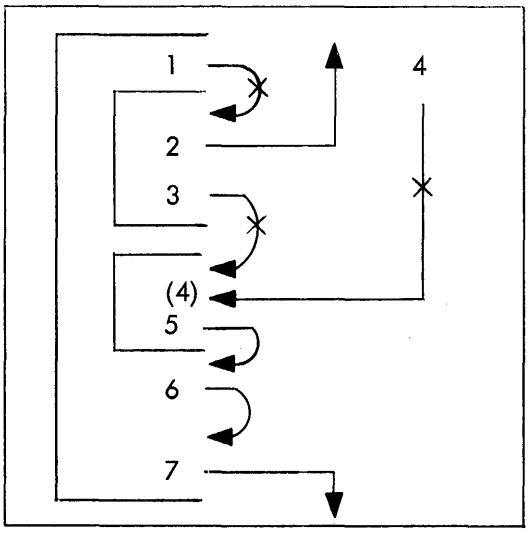

Figure 16 Program Branching in DO Loops

c. A DO loop may not end on a program branching statement (GOTO, IF) or another DO statement.

3.1.3 The CONTINUE Statement

Since a DO loop may contain alternate courses of action (such as branches to other parts of the loop, or out of the loop entirely), programmers frequently wish to make the last executable statement of the loop a test to determine which of the alternatives is to be taken. However, rule c of the above section forbids a DO loop to end on an IF or GOTO statement. To avoid this, the CONTINUE statement is provided as a dummy statement. It performs no action or computation, but provides a termination for any DO loop:

```
n  CONTINUE
```

where n is the statement number specified by the DO statement that initiated the loop.

A CONTINUE statement is not restricted to terminating DO loops; it may be used anywhere in a program; e.g., to provide points at which future program segments may be inserted.

3.1.4 Computed GOTO

The GOTO statement described previously is unconditional and provides no alternatives. The IF statement offers a maximum of three branch points. One way of providing a greater number of alternatives is by using the computed GOTO, which has the following form:

```
GOTO (k1,k2,k3,...,kn),j
```

where the ks are statement numbers, and j is a simple integer variable which may take on values of 1, 2, 3,...n according to the results of some previous computation. For example,

```
GOTO (5,7,5,7,5,7,10), IVAR
```

causes a branch to statement 5 when IVAR=1, 3, or 5, to statement 7 when IVAR=2, 4, or 6, and to statement 10 when IVAR=7. If IVAR is not one of the possible (legal) values, the GOTO is ignored and control passes to the next statement.

If an argument of a subroutine is used as the argument of a computed GOTO statement in the subroutine, the argument must be redefined (with a different name) in the subroutine; e.g.:

```
CALL SUB(JAY)

SUBROUTINE SUB(KAY)
      :
MAY = KAY
      :
GO TO (1,2), MAY
```

Otherwise, the address of JAY will be used rather than the contents of JAY.

3.1.5 Assigned GOTO

A slightly different way of providing a multibranched switch is by the assigned GOTO, which has the following form:

GOTO j, (k_1,k_2,k_3,...,k_n)

where j is a simple integer variable and the ks are statement numbers. In use, the variable j is assigned a value of one of the numbers in the list by an ASSIGN statement of the form

ASSIGN k to j

where k is one of the statement numbers in the list. The ASSIGN statement must precede the GOTO in order of execution. For example:

```
                :
    ASSIGN 30 TO IFORK
                :
25  GOTO IFORK,(20,30,40,50)
30  ASSIGN 40 TO IFORK
                :
    GOTO 25
40  A=B+C
                :
```

If the statement number assigned to j is not in the list, the computer halts. The AC contains a JMP to the address of the statement number. Hitting CONTINUE transfers control to that address. The first time statement number 25 is executed, and the program branches to statement 30 where IFORK is altered so that the next execution of statement 25 initiates a branch to statement 40. An alternate form of the ASSIGNED GOTO eliminates a list of statement numbers. It consists of the following:

GOTO j

where j is an integer variable whose value is established by an assign statement. This value must correspond to one of the statement numbers in the program.

3.2 PROGRAM TERMINATION

3.2.1 The STOP Statement

A program which is arranged so that the last written statement is the final and only stopping place needs no special termination indicator. The END statement automatically produces the final halt. Most programs, however, contain loops and branches so that the last executed statement is often somewhere in the middle of the written program. Often, there may be more than one stopping point. Such terminations are indicated by the statement:

STOP

This causes a final, complete halt; no further computation is possible. If more than one final halt is possible, each can be identified by a number as follows:

STOP n

where n is an octal integer which is displayed in the console ACCUMULATOR lights when the program stops. This feature is very useful when several stops are possible, such as stops in error routines, and it is desirable to know which one was reached.

3.2.2 The PAUSE Statement

The PAUSE statement allows suspension of operation for a time and then restarting the program by manual control. This is frequently necessary when the operator loads and unloads tapes in the middle of a program. This kind of temporary halt is provided by the following statement:

PAUSE n

The octal integer n appears in the AC lights when the pause is effected. Depressing the CONTINUE switch on the console resumes operation of the program.

CHAPTER 4
INPUT/OUTPUT STATEMENTS

In previous examples, all necessary information has been in the computer memory. Of course, a special loader reads in these programs, but the programmer must provide for the input of data and the output of results associated with his program.

For any input or output procedure, several items of information must be specified:

a. The direction of transfer (READ or WRITE).

b. The I/O device.

c. The amount, type, and location of the information to be transferred.

d. The arrangement of the data. In FORTRAN terms, the order and format of the incoming or outgoing data must be specified.

To provide all the information listed above, two statements are required for every data transfer between core memory and an external device. The first three items are supplied by the input/output statement, an example of which is shown in figure 17; the fourth is specified by the FORMAT statement.

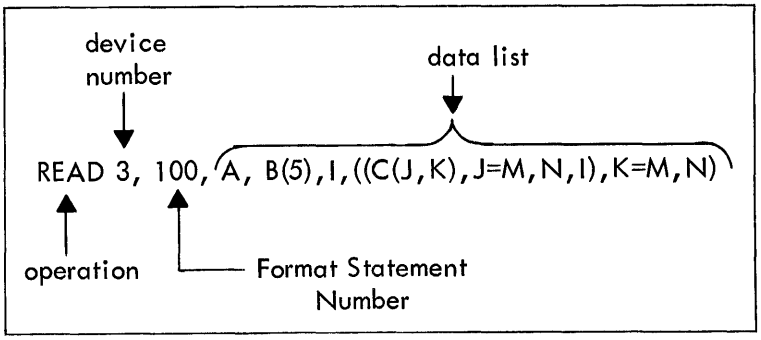

Figure 17 I/O Statement

4.1 INPUT/OUTPUT ASSIGNMENTS

The first word in the I/O statement designates the direction of the data transfer. A READ statement initiates all incoming transfers; a WRITE statement, all outgoing transfers.

The use of a device number, which follows the first word and specifies the external device involved in the transfer, eliminates the need for additional commands such as PUNCH, PRINT, etc. These device numbers are as follows:

INPUT		OUTPUT	
Device Number	Device	Device Number	Device
1	Keyboard	2	Teleprinter
3	Perforated Tape Reader	4	Perforated Tape Punch
		7	PDP-7/9 Line Printer

The format statement number is the third item in the I/O statement. It refers to a FORMAT statement which determines the arrangement of the data being transferred. Commas separate the format statement number from the device number and from succeeding items. The remaining items in the I/O statement are components of the data list.

4.2 THE I/O DATA LIST

The last part of the I/O statement is a list, the elements of which specify the locations in memory and the number of data elements being transferred. These elements, separated by commas, may be of four types:

Variables (unsubscripted)
Array elements (subscripted variables)
Array transfer expression
Constants (output list only)

The list in the statement in figure 17 contains four elements. These are, in order, a real variable, an array element, an integer variable, and an array transfer expression enclosed in parentheses.

The array transfer expression transfers whole arrays or sections thereof, under control of a single I/O statement. The expression consists of an array name with subscripts and a series of internal arithmetic statements which specify the lower and upper limits of each subscript and the increment between elements. The upper limit must not exceed the maximum vlaue for that subscript given in the DIMENSION statement in which that array name appears. If the increment is 1, it may be omitted.

The function of the array transfer expression is shown in the I/O statement in figure 17. The elements of array C are to be read into core. Every element in the K dimension between the limits M

and N, and every Ith element in the J dimension between the limits of M and N, are read into the computer. When the loops are exhausted, the reading stops. The limit parameters, I, M, and N, must be defined before they can be used in the expression.

The array transfer expression is a sequence of nested DO loops; the operations described in the preceding paragraph would have to be programmed as follows if the array transfer expression were not available:

```
      ....
      DO 15 K=M,N
      DO 15 J=M,N,I
15    READ 3, 100, C(J,K)
      ....
```

The following forms produce the indicated results:

```
READ 3, 100, (C(I,I), I=1, 10)
```

Result C(1,1) C(2,2) C(3,3) C(4,4)...C(10,10)

```
READ 3, 100, (CC(I), B(I), I=1, 10)
```

Result FORTRAN diagnostic errors

4.2.1 Ordering of Data Within an Array

In the language of matrix algebra, the data ordering specified by the array transfer expression in figure 17 is by columns. If M=1, N=5, and I=2, the elements of C must be ordered:

$$c_{11}, c_{31}, c_{51}, c_{12}, c_{32}, c_{52}, c_{13}, \ldots\ldots, c_{35}, c_{55}$$

Note that should one reverse the left-to-right arrangement of the arithmetic expressions defining K and J in the array transfer expression of figure 17, the data must be ordered by rows:

$$c_{11}, c_{12}, c_{13} \ldots \text{etc.}$$

That is, the ordering of the subscript defining expressions in an array transfer expression is independent of the subscript ordering in the array variable. The subscript encountered in the first definition expression, proceeding from left to right, varies most.

4.3 I/O SPECIFICATION STATEMENTS

4.3.1 Data Fields

The space allotted to an item of data is called the data field. The width of the data field is the number of character positions occupied by the item. The width may be greater than that required to hold

all the characters of the item of data including the sign, but no more than one item may appear in a field. For example, a five-digit integer may appear in a field seven positions wide (empty positions are denoted by the letter b, for blank):

bb34729

An item of data may be numeric (numbers), non-numeric (alphanumeric test or coded characters), or blank.

4.3.2 Data Field Formats

Information may be read in or written out in one of six data field formats: three numeric and three non-numeric. Each format is designated by a single letter contained in the format specification. The format specification indicates the type of data field (numeric or alphanumeric), the length of the data field, the form of the item it contains, and the storage mode of the item in the computer. The properties of the six format specification letters are summarized in table 1; each specification type is described in detail in this chapter.

TABLE 1 SUMMARY OF FORMAT SPECIFICATION LETTERS

Format Spec. Letter	Type of Field	Input	Storage Mode	Output
E	numeric	decimal, decimal exponent	floating point number	decimal exponent
F	numeric	decimal, decimal exponent	floating point number	decimal, decimal exponent
I	numeric	integer	fixed point number	integer
X	non-numeric	any characters and/or blanks	none	blank space
H	non-numeric	text	packed Flexowriter FIODEC	text
A	non-numeric	text	packed Line Printer FIODEC	text

4.3.3 The Format Statement

Each format specification provides information about one data field. To determine the arrangement of these fields, for both input and output, the format specifications must be combined in a FORMAT statement, as shown in figure 18.

```
a.  100 FORMAT (2E12.2,3X,F5.2) - numeric fields
b.  200 FORMAT (8HRAW DATA,5A3) - alphanumeric fields
```

Figure 18 FORMAT Statements

Following the word FORMAT is a statement list made up of format specifications, separated from each other by commas, and the whole enclosed in parentheses. Formats of all six types may be freely combined; however, the format specification, the item of data, and (H, X excepted) the variable type in the I/O statement list must all correspond.

If successive data fields have identical formats, it is not necessary to write out each specification in full; instead, the format specification may be preceded by a repetition count, equal to the number of identical data fields. The repetition count may not be larger than 15. For example, the first member of the list in figure 18a is:

2E12.2

This indicates that the next two data fields have identical E-formats. Likewise, 3X indicates three successive characters of X-format. However, groups of more than one data field may not be repeated by preceding the group with a repetition number. For example 6(F5.2,3X) is illegal.

Every FORMAT statement must be identified by a statement number so that it can be referenced by an I/O statement. When an I/O statement is executed, its data list and the list of format specifications in the corresponding FORMAT statement are scanned from left to right. Each item of data is matched with a format specification and transferred according to the format specified. This procedure continues until the I/O data list is exhausted. If the format list runs out before the I/O operation is completed, the format scan returns to the last previous left parenthesis and continues from that point.

If there are no internal parentheses in a format list, the scan would return to the beginning and repeat the format list. If the I/O list is exhausted before the format list, the remainder of the format list is simply ignored.

4.3.4 Format Specifications

The following paragraphs discuss the properties of each data field format, and the form of data permitted in each. The three numeric formats are described first, followed by the three non-numeric formats. The letter r in all cases symbolizes the optional repetition count which can precede the format specification.

4.3.4.1 Integers: I-Format - Integer fields are specified by the letter I. The general form of the specification is

rIw

where r is the number of times the specification is to be used and w denotes the field width including one position for the sign, whether or not it appears; that is, w-1 digits are allowed. The field width must be less than 7, or it will be set to 7. On input, the field must contain a FORTRAN integer not exceeding a magnitude of 131071. The integer may be signed or unsigned and may be placed anywhere within the field as long as there are no blanks embedded in the number. Input is converted to fixed point for storage. The following are examples of I-format input.

Specification	Acceptable Input	Unacceptable Input
15	32	34729 (too many characters)
	+5012	50b31 (embedded blank)
	-317	3.57b (not an integer)

On output, integers are right justified within the field. Positive integers are unsigned. If an integer is too large to fit in the field, w x's will appear in the output; w digits are allowed if the sign is positive. The storage mode for I-format output must be fixed-point.

4.3.4.2 Real Numbers: F- and E-Formats - Real number fields are specified by the letters F and E. The general forms are

rFw.d rEw.d

where w denotes the total field width including the decimal point, exponent, and sign, and d specifies the number of digits to the right of the decimal point. While w is limited to a maximum of 31 characters, d can be as large as 15. If an exponent is to be specified in input data, the character E must be present.

Both format specifications permit the same type of input. An incoming number may be in one of three forms:

a. A simple decimal: 95.34729; -7.132

b. A decimal exponent: 1.66E-16; -6.032E+23; 1.66-16; -6.032B23 (-6.032+23)

c. A string of digits with no indication of magnitude: 9534729; -6032E-23

Incoming numbers may be signed or unsigned and placed anywhere within the field. As with integers, there may be no embedded blanks, though a blank preceding an exponent is allowed; thus, 1.32E 36.

If the incoming number contains an explicit decimal point, the fraction delimiter (d) of the format specification is ignored. If there is no explicit decimal point, the number must be followed by

either an exponent or the end of the field, but not by blanks. A decimal point is inserted in the number, independently of the exponent value according to the d specification only if there are at least d digits in the number. If there are fewer than d digits, the decimal point is placed to the right of the number, regardless of d. The following illustrates the effect of the fraction delimiter.

If the specification is	And the incoming field appears as	The magnitude of the number is
E12.2	bbbbb9534729	95347.29
F10.5	bb953.4729	953.4729 (explicit decimal point overrides)
E10.2	b+1.66E-16	1.66E-16
E10.2	bbb166E-16	1.66E-16
F10.5	bbbbbbb222	222.0 (less than 5 digits input)

Both E- and F-format causes the incoming data to be converted to floating-point for storage. The two formats differ only in the form of output each produces. E-format output always appears as a decimal exponent with the exponent signed and the E omitted. The number is right justified in the field. Positive numbers are not signed. The following illustrates E-format output.

If the specification is	And the magnitude of the stored number is	The output appears as
E10.3	1.66×10^{-16}	bbb.166-15
E12.5	-1.324×10^{23}	bb-.13240 24
E12.2	-1.324×10^{23}	bbbbb-.13 24

The important points to observe in these examples are:

a. The printed number is normalized; that is, scaled so that it is less than 1.0 and no 0s appear immediately to the right of the decimal point.

b. The E representing the exponent is missing, and the exponent itself is signed whether it is positive (blank) or negative (-).

c. If the fractional part is too large to fit in the space reserved for it, the least significant digits are truncated.

F-format output is in the form of a simple decimal. The number is right justified in the field, and the decimal point is placed according to the format specification. Positive numbers are unsigned. Fractional parts are truncated if they would overflow the space reserved.

If the specification is	And the magnitude of the stored number is	The output appears as
F9.4	953.4729	b953.4729
F12.5	$-.007315 \times 10^4$	bbb-73.15000
F10.2	55.9328	bbbbb55.93

If F-format is specified, and the number to be printed is so large that the integral part would not fit in the available space, the number is automatically printed under E-format. For example,

If the format specification is	F10.5
and the stored number has a value of	57329.46
the format is interpreted as	E10.5
and the number will appear as	.57329 05

For both formats, the stored data must be in floating-point form for output. The field width w must include one position each for the sign and the decimal point; and in the case of E-format, three positions for the signed exponent.

4.3.4.3 Non-numeric Fields: X-Format - One way to separate items of data for readability is to provide wide enough fields in the format specifications so that there will be some leading blanks. Another way to separate items is by using the X-format specification, whose only function is to indicate the presence of a blank position. A string of blank spaces is indicated by a repetition count before the X:

rX

On input, an X-format specification causes the input scan to skip ahead the number of spaces specified by the repetition count and continue reading input from that point using the succeeding format specifications in the list.

On output, the X-format causes the number of spaces indicated by the repetition count to be inserted between the preceding and following items of data. For example, the specification

F5.2, 3X, F5.2

causes the output of two items of decimal data to be separated by three blank spaces.

When reading FORTRAN produced paper tape input, a 1X format specification may be used wherever a slash (/) appears in the output format statement which produced the tape. The slash causes

a carriage return to be punched following the last field. This is read as a blank item (0) unless the X specification skips the carriage return character. A more detailed description of field delimiters appears in section 4.4.1.2. The 1X specification is not required if the input format is identical to the output format.

4.3.4.4 Non-numeric Fields: H-Format (Hollerith) - The output of text such as table headings, captions, instructions to the computer operator, and descriptive information is done by including the text explicitly within the FORMAT statement, using the H-format, which specifies that the characters immediately following the H are to be taken as an item of textual data. To handle a string of text, the total number of characters, including blanks, is counted and substituted for the repetition count. An error results if the number of characters is counted incorrectly. Figure 18b shows a FORMAT statement which contains an H-format specification. Note that the field count 8 corresponds exactly to the number of characters and blanks in the field.

The programmer may mix H-format with other specifications in a format list. On output, the I/O data list scan is suspended when an H-format specification is encountered, the text in the Hollerith field is printed, and the data list scan resumes. If an output statement is to transfer H-format information only, the data lists may be omitted. Hollerith text is packed three characters to a word and stored in Flexowriter FIODEC code. Alphanumeric (A-format) information is stored 1, 2, or 3 characters per word, right justified, in line printer FIODEC code.

The information in the following format statement

```
50    FORMAT (41HSATELLITE TRACKING DATA, CAMBRIDGE, MASS./
$     23HTELOS I, SEPTEMBER 1964)
```

would be printed on the teleprinter by the statement

```
WRITE 2, 50
```

and would appear as follows:

```
SATELLITE TRACKING DATA, CAMBRIDGE, MASS
TELOS I, SEPTEMBER 1964
```

4.3.4.5 Non-numeric Fields: A-Format - Frequently, it is desirable to vary text analogous to data. For example, a billing program may process a larger number of accounts in the same manner with identical output format for each, except for the name of the person associated with the account. It would be burdensome to write a long sequence of FORMAT statements with Hollerith fields and make frequent corrections simply to store all the account names.

The A-format specification allows the user to both READ and WRITE textual information. Its general form is

rAn

where r designates the number of 18-bit words required to store the characters of text in the data field. On input, characters are read and packed 1, 2, or 3 to a word depending on the count n. On output, the stored words contain packed text, and 1, 2, or 3 characters are printed from each according to the count n. (The number of characters is r x n.) In the I/O statement, the packed text is designated by an integer variable name. If a repetition count accommodates a long string of text, the variable name is subscripted; the value of the subscript, r, is the number of words of packed text. Such an array name must appear in a DIMENSION statement. Since r $\leq$15, n $\leq$3, the format specification cannot be greater than 15A3. To read a large number of characters, FORMAT (A3) is sufficient since the specification may be indefinitely repeated.

To illustrate the use of A-format, take a hypothetical billing program. Assume that input is from paper tape and output is on the teleprinter. The first item on paper tape for each account is the name of the account in the first 36 character positions and no other information. To read this information, the following two statements would suffice (here r x n=36, the number of characters):

```
      READ 3, 300, (NAMACC(I), I=1, 12)
      ...
300   FORMAT (12A3)
```

The information in the 36 character positions is read into the 12 locations designated by the array name, NAMACC; the text is packed 3 characters to a word.

To print out A-format, the following statement is necessary:

```
WRITE 2, 300, (NAMACC(I), I=1, 12)
```

The A-format is also useful when the same program runs over a long period of time and the date of each run is recorded. A date can be provided with the input, a location reserved for it, and the information transferred using an A-format specification.

4.4 INPUT/OUTPUT DEVICES

4.4.1 Data Organization

4.4.1.1 Records - In every I/O device, data is organized into records. Because of the dissimilarity of devices, the definition of a record varies. Table 2 lists the I/O devices and the definition of a record for each.

TABLE 2 DEFINITION OF A PHYSICAL RECORD FOR I/O DEVICES

Device	Physical Record Definition
Keyboard and Teleprinter	The information typed on a single line (maximum 72 characters).
Perforated Tape Reader, Punch	The information punched between two carriage returns (practical maximum 72 characters, for compatibility with other devices).
Line Printer	120 characters (one line).

Normally, one FORMAT statement corresponds to one record and the programmer must be careful that the total number of characters in the format specifications, including repetitions, does not exceed the maximum for one record on the respective device.

4.4.1.2 Multirecord Formats - To make the arrangement of output data as flexible as possible, a single FORMAT statement can specify more than one record. The method can best be illustrated by an example. The following FORMAT statement,

```
50      FORMAT (F12.2, 5X 316, 2E15.5, 17)
```

causes the associated output to be printed on a single line. If the statement is changed to read

```
50      FORMAT (F12.2, 5X, 316/2E15.5, 17)
```

the insertion of a slash (/) in place of the comma after the third specification causes the remaining data to be written as a new record on the next line.

In general, whenever a slash appears in a FORMAT (other than an H-format) statement, it terminates the record. If two slashes appear in succession with no intervening specifications, the effect is the same as if an empty record were transferred. For example, if two slashes were inserted instead of one in the illustration:

```
50      FORMAT (F12.2, 5X, 316//2E15.5, 17)
```

and the data were written on the teleprinter, the result would be a line of data, a blank line, then another line of data. Use of multirecord formats greatly increases the flexibility with which the programmer may arrange tabular data for output. Table 3 and 4, respectively indicate the effects and limitations of multirecord formatting for input and output operations.

TABLE 3 INPUT FORMAT

	Paper Tape	Teletype
Field Delimiters[1]	Tab	Tab
Effect of Field Delimiter	Terminates current field if reached before the count in the format statement runs out.	Terminates current field if reached before the count in the format statement runs out.
Record Delimiter[2]	Carriage return	Carriage return-line feed
Effect of Record Delimiter	Terminates the current field if reached before the count in the format statement runs out.	Terminates the current field if reached before the count in the format statement runs out.
Other Delimiters	None	None

[1]Field delimiters are never required on input but will have the indicated effect if present. They are always ignored when the first character of any field (except in A-format where they should never be used; the only legal characters for A-format are the Anelex character set).

[2]Record delimiters are never required on input except where needed for the correct operation of the slash function as when using paper tape.

TABLE 3 INPUT FORMAT (continued)

	Paper Tape	Teletype
Effect of Slash in Format Statement[3]	Causes the next field requested to be taken from information after the next carriage return on the paper tape.	None

[3]Successive slashes are ignored in all cases. This can be visualized logically since the slash merely signifies that the current buffer is empty and does not cause the next record to be read. No implicit slashes are assumed at the end of an input format statement; i.e., all characters appear as one long string, irrespective of the input device, except where the slash is used explicitly.

In all cases the unit record consists of the characters requested between slashes in the output format statement or by the entire statement if no slashes are used. As noted above, the end of an output format statement is taken as an implicit slash. The maximum record which can be requested in any case is 256-(N+1) characters where N equals the number of fields desired. All requested characters above that figure are lost. The following additional limitations apply:

a. For the line printer, a request of more than 120 characters per unit record destroys the program.

b. For the paper tape punch, a request of more than 72 characters per unit record produces a tape which cannot be read off-line. If that restriction is not applicable, the maximum size record (256-(N+1) characters) can be used.

c. For the Teletype, all characters requested after the 72nd character per unit record type over the 72nd character.

TABLE 4 OUTPUT FORMAT

	Line Printer	Paper Tape	Teletype
Field Delimiters Generated	None	None	None
Cause of Generated Field Delimiter	None	None	None
Record Delimiters Generated	Space line printer 1 line	Carriage return (plus line feed in ASCII mode)	Carriage return-line feed
Effect of Slash in Format Statement	Output the present contents of the buffer, if any, and always generate the end-of-record indicator.	Output the present contents of the buffer, if any, and always generate the end-of-record indicator.	Output the present contents of the buffer, if any, and always generate the end-of-record indicator.
Effect of End of Format Statement	Implicit slash	Implicit slash	Implicit slash

4.4.2 I/O Operations with Paper Tape and Keyboard

Use of FORTRAN with paper tape and a keyboard permits a relaxation of some of the constraints on input formats for numbers. With paper tape or keyboard, an item of data being read can be delimited by a tabulation or a carriage return-line feed combination (for keyboard), or a carriage return (for paper tape), which overrides the field width allotted in the format specification. The limits on maximum field widths still apply; however, 7 for integer fields and 31 for real number fields. The number of characters in a field must be less than or equal to the width specified or the overflow will be considered another field.

```
60      FORMAT (2I4, E9.2)
```

will accept input from a keyboard thus:

```
176   -20   +16742E13 )↓   (line feed-carriage return)
```

or analogous paper tape input.

Also when using the keyboard for input, incorrect characters can be erased by striking the rubout key. The rubout key erases the last character; successive rubouts erase the next previous character. Thus to erase the last three characters, strike the rubout key three times. However, no changes in data may be made after typing a tab or carriage return since this processes the data.

CHAPTER 5
SUBPROGRAMS: FUNCTIONS AND SUBROUTINES

The programmer may employ separate subprograms to perform a sequence of operations required in the solution of a problem or to evaluate functions. For example, whenever a function occurs in an arithmetic expression, a subprogram is called into operation to evaluate the function, using as data the arguments provided. The resulting single value is then returned for use in the computation of the expression in which the function appears. A second type of subprogram, the subroutine, is used when a sequence of statements is used repeatedly or when it is necessary to generate more than one result (a matrix operation, for example).

Since a subprogram is a separate program, communication with the main program must be established; that is, an entry to the subprogram from the main program and an exit from the subprogram back to the main program must be provided. In the case of a function, entry is effected by the use of the function name in an arithmetic expression. In the case of a subroutine, entry is effected by the CALL statement. The RETURN statement provides an exit from both types of subprograms. A subprogram cannot come to a full stop (though subroutines may include pauses) but must always return control to the calling program.

5.1 FUNCTIONS

In addition to the library functions supplied with FORTRAN, the user can write his own as needed. The writing of a function subprogram follows the rules for writing any sort of program in that there must be a title, a body, and an END statement. In addition, two special statements are required: FUNCTION definition and RETURN.

5.1.1 The FUNCTION Definition Statement

The FUNCTION definition statement identifies the subprogram and has the following form:

FUNCTION f (d_1, d_2, d_3,...,d_n)

where f is the name of the function and d is a dummy name which represents a main program argument of the function. The function and dummy names follow the rules given for variables in section 1.2.3.2: they are restricted to the same character set, the first character must be alphabetic, and only the first six characters are significant. A main program argument can be a constant, a variable, or any legitimate arithmetic expression. It may itself include functions. At least one argument is required with a function subprogram.

Because a function is an element of an arithmetic expression, it must always return a value to the main computation. To do this, in the function subprogram there must be at least one arithmetic statement in which the function name appears as the left-hand variable. This also defines the mode of the return value. A function returns an integer-value if its name is defined as an integer variable, or a real-value if its name is defined as a real variable. Exceptions to this rule are: 1) a function whose name begins with the letter X is integer-valued; 2) if a function name does not begin with X but ends with F, the function returns a real-value. A function name cannot represent an array. STOP or PAUSE cannot be used in a function.

5.1.2 RETURN Statements

The statement

RETURN

terminates the subprogram (function or subroutine) and transfers control to the calling program. There may be more than one RETURN statement in a subprogram, corresponding to alternative exits. Figure 19 illustrates the writing of a function subprogram which calculates the factorial of an integer n:

$n! = 2 \cdot 3 \cdot 4 \cdot \ldots (n-2)\ (n-1)\ (n)$

```
FACTORIAL CALCULATOR
      FUNCTION NFACT (N)
      NFACT = 1
      DO 10 J=1,N
10    NFACT=NFACT * J
      RETURN
      END
```

Figure 19 Function Subprogram

In figure 19, the name NFACT specifies an integer-valued function. The dummy name N denotes an integer argument. The body of the subprogram is a loop which calculates the factorial. Statement 10 provides for the value to be returned, since the function name appears as a lefthand variable. Finally, the RETURN statement transfers control to the calling program.

5.1.3 Use of Functions

Dummy names need not agree in mode with the function name or with each other. They must agree, however, with the mode (integer, floating two- or three-word) of corresponding arguments in the calling program. For example, the factorial function above may be called this way (the dots represent parts of an arithmetic statement):

....+ NFACT(15)/....

but not this way:

....+NFACT(15.0)/....

because the argument is not an integer.

Library functions which require real arguments, such as SQRTF, COSF, are treated as floating-point variables in arithmetic expressions. No diagnostic check occurs during compilation to insure that the arguments specified by the programmer are in fact real. For example:

A=SQRTF(I*K 2.2)

compiles correctly since the real nature of the exponent forces the entire parenthetic expression to be real. However,

A=SQRTF(I*K/2)

must be written as

A=SQRTF(T=I*K/2)

or some equivalent form if correct compilation is expected.

5.1.4 Library Functions

Several common functions are provided with the FORTRAN system to save the programmer the necessity of writing them. These library functions are named according to the following conventions:

a. Every library function name ends with the letter F.

b. A function whose name begins with the letter X is integer-valued; otherwise it is real-valued.

Library functions may be used by the programmer without any special preparation; they are placed in memory from a library tape which is read prior to run time. The library functions are:

Function Name	Operation Performed
SQRTF(A)	square root: $\sqrt{a}$
SINF(A)	sin a (argument in radians)
COSF(A)	cos a (argument in radians)
ATANF(A)	arc tan a
LOGF(A)	$\log_e a$
CLOGF(A)	$\log_{10} a$
EXPF(A)	e^a
ABSF(A)	absolute value: $\|a\|$
XABSF(N)	absolute value (integer): $\|n\|$

XABSF is the only integer-valued library function.

5.2 SUBROUTINES

It is often desirable to write whole procedures, not as functions (which can return only a single quantity) but as complete subprograms which compute multiple quantities for use by the main program. Such a subprogram is called a subroutine. Its definition statement is:

SUBROUTINE s (d_1, d_2, d_3,...,d_n)

where s is the subroutine name, and d_i are dummy arguments.

Like a function subprogram, a subroutine must have at least one RETURN statement. Unlike a function, however, a subroutine does not directly return a value to the calling program; instead results are stored in locations designated by the main program, where they are available to the main program. A subroutine name differs from a function name in that it may not appear as the left-hand variable in an arithmetic statement of the subprogram. A subroutine differs further from a function in that it requires no arguments. When arguments are present, they obey the same rules as functions with regard to mode.

5.2.1 The CALL Statement

To call a subroutine into operation, the following statement is used:

CALL s (a_1, a_2, a_3,...,a_n)

where s is the subroutine name, and a_i are arguments, if any. The argument list of the CALL statement must be similar to the argument list in the subroutine.

The factorial calculator written as a function in figure 19 may be written as a subroutine as shown in figure 20. In this example the main calculation is the same as before, but to make the result available to the calling program, another argument has been provided. If a program were to use this subroutine to calculate, for instance, the factorial of an integer variable K, the following statement might be used:

CALL FACT (K,KPROD)

```
FACTORIAL CALCULATOR
      SUBROUTINE FACT (N,NPROD)
      NPROD=1
      DO 10 J=1,N
10    NPROD = NPROD * J
      RETURN
      END
```

Figure 20 Example of Factorial Calculator

The value of the resulting product, assigned the dummy name NPROD in the subroutine, is given to the variable KPROD in the calling program.

5.2.2 Common Storage

As figure 20 shows, information can be transmitted between a calling program and a subprogram via the argument list. Information can also be passed between programs through a special section of memory, set aside as common data storage. Space in this area is assigned by the following specification statement:

COMMON v1, v2, v3,...,vn

where each v represents a variable name, either simple or subscripted. Each simple variable is assigned a location (or group of locations, depending on the storage mode) beginning at the end of available memory and working backward. Arrays are assigned enough locations to store the maximum number of elements, as indicated by the DIMENSION statement. For example, if an integer array which appears in a DIMENSION statement

DIMENSION MATRIX (20, 40)

also appears in a COMMON statement

COMMON MATRIX

20 x 40 = 800 locations will be set aside in common storage.

Programs which have common data must each have a COMMON statement in which the variables are assigned in correct order and storage mode, although the names do not have to be identical from program to program. For example, two programs could share three variables if one program contained the statement

COMMON VARL, INDEX, AVAL

and the second program contained the statement

COMMON VARX, ITEST, ARGL

The first program uses the name VARL, and the second VARX, but both names refer to the same quantity. Likewise, INDEX and ITEST are corresponding names, and so on down the line of variables.

5.2.3 Array Names Used in Subroutines

Array names may be transmitted between a calling program and a subroutine either as arguments or as variables in common storage. The two methods require different treatments.

a. If an array is placed in common, it must be dimensioned in both programs and must appear in a COMMON statement in both programs. The names need not be the same but they must correspond in mode, number, and order in the COMMON statements. For example, if an array appears in the calling program as follows:

```
...
DIMENSION ARRAY (10, 10, 30)
...
COMMON VAR1, VAR2, ARRAY, IVAR
...
```

and is referred to in a subroutine by the name ARR2, this name must appear in the subroutine in statements such as these:

```
...
DIMENSION ARR2 (10, 10, 30)
...
COMMON X1, X2, ARR2
...
```

Here, the array name appears in the common list, following two real variables just as the corresponding name does in the calling program.

MATRIX MULTIPLICATION SUBROUTINE

```
      SUBROUTINE MATMUL (ID, JD, KD)
      DIMENSION DA(10,10), DB(10,10), DC(10,10)
      COMMON DC, DA, DB
      DO 10 I=1, 10
      DO 10 J=1, 10
10    DC(I,J)=0
      DO 20 J=1, JD
      DO 20 I=1, ID
      DO 20 K=1, KD
20    DC(I,J)=DC(I,J) + DA(I,K) * DB(K,J)
      RETURN
      END
```

Figure 21 Matrix Multiplication Subroutine

In figure 21, the three arrays necessary for the calculation are placed in common. A main program using this subroutine to multiply matrices of dimensions 5 x 10 and 10 x 7, respectively, must contain statements such as the following:

```
DIMENSION AMTX(10, 10), BMTX(10, 10), CMTX(10, 10)
COMMON CMTX, AMTX, BMTX
:::
CALL MATMUL (5,7,10)
...
```

b. If array names are to be transmitted as arguments, they must appear, unsubscripted, in a DIMENSION statement in the subroutine. For example, the names of the matrices required by the subroutine of figure 21 could be used as arguments in this manner:

MATRIX MULTIPLICATION SUBROUTINE

```
      SUBROUTINE MATMUL (ID, JD, KD, DA, DB, DC)
      DIMENSION DA, DB, DC
      ...
      ...
      END
```

The calling program could then have statements as follows:

```
...
DIMENSION AMTX (10, 10), BMTX (10, 10), CMTX (10, 10)
...
...
CALL MATMUL (5,7,10,AMTX,BMTX,CMTX)
```

Note that the array names appear without subscripts in the subroutine call also. Elements of an array may be transferred via the argument list when the subroutine considers the argument as an undimensioned variable, e.g.,

```
CALL SUB (C(1))
      :
      :
SUBROUTINE SUB (A)
```

5.3 MACHINE LANGUAGE CODING IN A FORTRAN CONTEXT

Since the symbolic output of the compiler is in the language of the assembler, familiarity with that language is basic to a thorough understanding of the topics discussed. Information on the assembler language can be obtained from the PDP-9 Symbolic Assembler and the PDP-9 User Handbook.

5.3.1 Handling of S Coding

As mentioned in chapter 1, whenever the letter S (for symbolic) appears in the identification field, the remainder of the line is transferred to the object program exactly as written. No error diagnosis is made for an S-coded line. Unless the programmer is absolutely certain that he can omit them in safety, all S coding should be bracketed (preceded and followed) by CONTINUE statements, where the terminating CONTINUE has a statement number attached. An LFM should precede the other S coding. Example:

```
          READ 402
          CONTINUE
S         LFM
S         LAS
S         AND (7
S         DAC J.
10        CONTINUE
```

5.3.2 Compiler Generated Coding

The FORTRAN compiler generates an object program of symbolic machine instructions and pseudo instructions from the FORTRAN statements it reads. In general, each statement is processed

individually without reference to previous or subsequent statements; as a consequence, errors such as the duplication of statement numbers are not detected by the diagnostic routines of the compiler. Coding to test the iteration variable of a DO loop is not generated until after the last statement in the DO loop.

5.3.2.1 Symbolic Conventions - To avoid conflicts with symbols which appear in the FORTRAN source program and symbols which appear in the assembler permanent symbol table (machine language instructions and assembly pseudo instructions), or symbols which reference permanent locations in the FORTRAN object time system, special conventions are established:

1. Symbols which appear in the FORTRAN source program and which are five characters or less are modified by appending the period (.) character.

2. Statement numbers are transformed into symbols by prefixing the period (.) character to the statement numbers.

3. All symbols generated internally by FORTRAN are four character symbols: the period (.) character followed by three letters.

The period (.) character is not a permissible character for symbols which appear in FORTRAN source programs. It is a permissible character for symbols in the object program as input to the assembler. A consequence is that FORTRAN source program symbols (names) and the symbols which are part of the compiled object program ordinarily differ. For example:

I=2

generates the code

```
LAC (2
DAC I.
```

hence the variables name I must be referenced as I. if it occurs in a line of S coding.

The conventions established for symbols insure that all symbols which appear in the FORTRAN object program contain a period except those which are machine language instructions, mnemonics, assembler pseudo instructions, FORTRAN object-time system references or six-character symbols from the source program. To avoid conflicts, the following six-character symbols should not appear in the FORTRAN source program:

DECIMA	VARIAB	SECPRG
ANALEX	LIBFRM	MODSET
NOSYMB	PUNDEF	MODRES
EXTERN	TELETY	EFMTEM
EXPUNG	SYMBOL	EXTADD
FIODEC	INTERN	EARITH
NOINPU	NARITH	SARITH
NINDIG	LOGCOM	SIXDIG

5.3.2.2 Floating-Point Commands - Instructions generated by the FORTRAN compiler may use fixed- or floating-point operands. Standard machine instructions, i.e., directly executable instructions, are generated when the operands are fixed point. When the operands are floating point (real), the instructions generated must be interpreted since the PDP-9 does not have such instructions in its instruction set.

The floating-point interpreting program is an integral part of the FORTRAN object-time system. It is entered by the pseudo instruction EFM (enter floating mode) which initializes the interpretive program counter. Instructions are interpreted and executed sequentially until a transfer of program control (subprogram call) or the pseudo instruction LFM (leave floating mode) is encountered. The compiler generates an EFM or LFM for each executable statement number to insure that all internal program transfers are in the proper arithmetic mode. EFM's or LFM's are also generated whenever they are needed; example:

```
I = 1
A= 1.
```

When the instruction sequence is in floating mode, the following mnemonics, which are the same as the standard PDP-9 machine instruction mnemonics, are interpreted as floating-mode commands:

LAC	load floating accumulator
ADD	floating add
DAC	deposit floating accumulator
JMP	floating jmp
JMS	floating jms

In addition, the following mnemonics are used only in the floating interpretive system:

FCS	floating clear and subtract
FSB	floating subtract
FMP	floating multiply
FDV	floating divide
CAS	floating compare accumulator to storage

Two instructions which may be generated are:

FXA	fix the floating accumulator and leave floating mode
FLO	float the fixed accumulator and enter floating mode

Notice that FXA carries an implicit LFM and that FLO carries an implicit EFM.

CAL instructions are handled by a program in the object-time system called the CAL Handler. The CAL Handler saves all relevant information in a push-down stack, and the CAL executes in fixed-point mode.

5.3.3 Subprogram Linking

5.3.3.1 Implicit Subprogram Calls - An implicit subprogram call occurs when implementation of a feature included in the source language requires an internal subsection of the object time system or use of the FORTRAN library exponential function. An example of the latter case is the generation of a JMS XPN fixed-point operand or JMS EXP floating-point operand in response to the appearance of the exponential operator (↑).

The appearance of an array name in a DIMENSION statement generates an implicit subprogram call of the following form:

```
NAME,    JMS CALSB
         LAW TWO
         LAW THREE
         LAW INTDIM
         .GS
```

The example above supposes a three-dimensional array. CALSB is the name of the subscript calculating section of the object-time system. TWO and THREE stand for the actual values of the second and third bounds. The initial bound is not needed for subscript calculation but generates storage allocation for the array. INTDIM will be 1, 2, or 3 for, respectively, fixed point, 2-word or 3-word floating point. It specifies the number of memory locations required for each array element. The symbol .GS stands for the generated symbol which actually defines the address of the array. NAME is the actual array name. (Refer to Construction of Dimensioned Variables, section 2.2.1). Four additional implicit subprograms which may be called in a FORTRAN object program are:

GOTO	computed go to
CALST	to initialize the CAL Handler
GTARG	to get arguments of a subroutine or function
SET2W	to initialize 2-word floating-point data storage

5.3.3.2 Explicit Subprogram Calls - Explicit subprogram calls are generated by the following features of the FORTRAN language:

1. A library function reference
2. A CALL statement (subroutine)
3. Reference to a subscripted variable
4. An I/O control statement

The code generated by items 1, 2, and 3 conforms to a general form; the code generated by item 4 is slightly different and is discussed under I/O statements.

A normal subprogram call generates

```
CAL A
```

where A is the name of the subprogram. If the subprogram has arguments (a function must have arguments; a subroutine may or may not have arguments; the arguments of a subscripted variable reference are its subscripts), the CAL instruction is followed by code of the following form:

```
ARz .GS
```

where ARz is ARX for fixed-point mode or ARF for floating mode. The argument name of .GS is the memory location (if floating point, the first of two or three words).

When passing an array name to a subroutine, subscripts are omitted and the array name appears as an argument:

ARz NAME.

When the CAL is processed, the CAL Handler saves the mode (fixed- or floating-point), the corresponding accumulator, and the return address. It then transfers control to the address indicated in the CAL instruction. Control is returned to the calling program by means of the return portion of the CAL Handler which restores the accumulator and mode and transfers control back to the instruction following the calling sequence.

5.3.3.3 Function Linkage - A function is always used in an arithmetic expression. It acts like a variable which is equal to the value of the function with the given arguments. When the function name occurs to the left of the equal sign (in the body of the function) the value of the arithmetic expression to the right of the equal sign is intepreted by the compiler to be the value of the function.

This value is placed in an internal location named RES, and when the return statement is encountered, the address RES returns to the return portion of the CAL Handler. This address is placed in a location called TEMAD (temporary address storage) internal to the object-time system, and the value of the function is then accessed indirectly through this location. Should another function call occur before this address is referenced by the calling program, the compiler generates a code to retrieve the previously calculated function value. Dimensioned variable references are effected through location TEMAD as though they were functions.

For example, the statement

```
100     A=SIN (B)+C
```

generates the following object code:

```
.100,   EXTERNAL SIN.
        CAL SIN.
        ARF B.
        LAC I TEMAD
        ADD C.
        DAC A.
```

A function definition statement:

```
FUNCTION FNAME (A,B,I)
```

generates the following code:

```
        INTERNAL FNAME.
FNAME., JMS GTARG
        JMP .GS
A.,     0
B.,     0
I.,     0
.GS,
```

where .GS is a symbol generated by the FORTRAN compiler. The GTARG routine, when executed, would place the addresses of the dummy arguments A., B., and I. in the locations reserved for them.

Note the use of pseudo instructions EXTERNAL and INTERNAL. When EXTERNAL is encountered by the assembler, it generates information to the loader, requiring that all references to the accompanying symbol (or symbols since more than one may occur with EXTERNAL) be saved by linking until the occurrence of a definition of that symbol; this is signaled by the occurrence of INTERNAL and one accompanying symbol (only one symbol may occur with INTERNAL).

For example, consider a factorial calculator function which returns a floating-point value.

```
FACTORIAL CALCULATOR
      FUNCTION FACT(N)
      FACT=1
      DO 10 J=1,N
10    FACT=FACT*J
      RETURN
      END
```

The compiler generates the following code:

```
FACTORIAL CALCULATOR
DECIMA*         FIODEC†
                INTERNAL FACT.
FACT.,          JMS GTARG
                JMP .AAA
N.,             0                   /GTARG PICKS UP ARZ ADDRESS
                                    /AND PLACES IT IN N.
.AAA
                LAC (1
                FLO
                DAC RES
                LFM
                LAC (1
                DAC J.
.AAB,
.10,            LFM
                LAC J.
                FLO
                FMP RES
                DAC RES             /FLOATING POINT RESULT STORED IN RES
                LFM
                LAC J.
                ADD (1
                DAC J.
                CMA
                ADD I N.
                ADD (1
                SMA
```

*DECIMA indicates that all subsequent numbers will be interpreted in decimal radix rather than in octal radix.

†FIODEC indicates that all character translations are to FIODEC code; i.e., H-format in format statements.

```
                JMP .AAB
                LAW RES          /ADDRESS OF RES IS RETURNED
                RETUR
                HLT
TEM,
TEM+0/
START
```

5.3.3.4 Subroutine Linkage - The code generated by the control word SUBROUTINE is very similar. Since a subroutine need not return a value, the user must establish storage for results either by passing the argument(s) to the subroutine in the call statement or by establishing them in a common statement.

The following is a possible version of the factorial calculator written as a subroutine:

```
FACTORIAL CALCULATOR
      SUBROUTINE FACT (N,R)
      R=1
      DO 10 J=1,N
10    R=R*J
      RETURN
      END
```

where R contains factorial N in floating point.

The compiler generates the following code:

```
FACTORIAL CALCULATOR
DECIMA      FIODEC
            INTERNAL FACT.
FACT.,      JMS GTARG
            JMP .AAA
N.,         0
R.,         0
.AAA
            LAC (1
            FLO
            DAC I R.
            LFM
            LAC (1
            DAC J.

.AAB,
.10         LFM
            LAC J.
            FLO
            FMP I R.
            DAC I R.
            LFM
            LAC J.
            ADD (1
            DAC J.
```

```
        CMA
        ADD I N.
        ADD (1
        SMA
        JMP .AAB
        RETUR
        HLT
TEM
TEM+0/
START
```

5.3.4 Construction of Dimensioned Variables

When dimensioned variable references occur in the FORTRAN source language, the code generated is very similar to that generated by a function call. A DIMENSION statement generates an internal function which has the name of the variable. When the function is called, its value is the address of the element of the array specified by the vlaues of the subscripts at the time of the call, and that address is placed in location TEMAD. For example, the statement

```
Z=A(I)
```

generates the object code

```
CAL A.
ARX I.
EFM
LAC I TEMAD
DAC Z.
```

On the other hand, the statement

```
A(I)=Z
```

generates the object code

```
CAL A.
ARX I.
EFM
LAC Z.
DAC I TEMAD
```

5.3.5 Allocation of Array Storage and the Subscript Calculator

A simple example of array storage allocation is the two-dimensional array A(2,2). A two-dimensional array is stored sequentially by rows indexed by columns (in this case):

```
A(1,1)
A(1,2)
A(2,1)
A(2,2)
```

or in general, an n-dimensional array is stored with the first dimension varying least. The standard object-time system subscript calculation program accommodates up to four dimensions. Higher dimensions may be provided for upon request.

5.3.6 I/O Statements

An I/O statement of the general form

```
RW n,m,a,b,i
```

generates the following code

```
RW
JMS .IOX
N
FOR .M
ARF A.
ARF B.
ARX I.
ENDIO
```

where RW is READ or WRITE, .IOX designates the corresponding I/O device processor. X may be 1-9, .IO1 is keyboard input, .IO7 corresponds to ASCII line printer output etc.; devices 5, 6, 8, and 9 are presently unassigned. (N is the device number (n)), and the next location designates the address (.M) of the accompanying format statement. Succeeding entries contain the addresses (A., B., I.) of the referenced variables (a, b, i). Code generated by an array transfer expression is fairly complicated but similar to the code generated by an explicit DO loop. ENDIO is the address of that section of the object-time system which terminates I/O operations. Note that Hollerith text is stored with the object program code sequence generated by the corresponding FORMAT statement.

CHAPTER 6
OPERATING PROCEDURES

This chapter details standard operating procedures for the PDP-9 FORTRAN II system. PDP-9 FORTRAN II is written for a machine having at least 8K of memory and an exclusively paper-tape configuration. In an 8K system approximately 4600_{10} locations are available for program and data.

The principal subsections of the FORTRAN system for paper tape are:

Compiler

Assembler

Operating System

I/O Library

Six Decimal Digit Arithmetic Library

Nine Decimal Digit Arithmetic Library

The compiler accepts input in the FORTRAN language and produces an object program output in computer source language acceptable to the assembler. The assembler accepts the compiler output and produces a binary relocatable version of the program and a binary version of the Linking Loader. When the user is ready to run the program, he loads the main program and any subprograms followed by any built-in functions called from the library. Once the total program is in memory, he loads the operating system and executes the program. The operating system contains an interpreter for floating-point arithmetic, an interpreter for FORMAT statements, red tape routines such as fix a floating number and vice versa, and the I/O routines. The operating system must be in memory when a FORTRAN program is executed.

6.1 PROCEDURE FOR USING FORTRAN WITH A PDP-9 PAPER TAPE SYSTEM

The Bootstrap Loader with starting address 17770_8 (for 8K machines) is called the readin mode, or RIM, Loader. Pressing the START switch on the console with 17770_8 in the ADDRESS switches is referred to as RIM start.

Step 1 Prepare programs to be compiled in accordance with the conventions described in the preceding section. Each program or subprogram on paper tape must be followed by the three-character sequence

carriage return-line feed

carriage return-line feed

form feed

Step 2 Place the paper tape labeled FORTRAN Compiler in the reader, set ADDRESS switches to 17770_8, and press START.

Step 3 Position ACCUMULATOR switches 9 and 10 as follows to indicate tape formats for, respectively, the intermediate object program (assembler source) tape and the compiler source tape: AC9 (intermediate object tape) - up for ASCII, down for FIODEC. AC10 (compiler source) - up for ASCII, down for FIODEC. Turn on the tape punch. Place the program tape to be compiled in the reader and press CONTINUE. FORTRAN punches out the intermediate object program tape.

Step 4 If other programs are to be compiled, repeat step 3. However, if more than one program is on a single tape, the tape must be pulled back before restarting, since it will have read past the END statement into the next program. If an accidental error occurs at any time, the compilation procedure may be restarted by replacing the source tape in the reader, placing 22_8 in the ADDRESS switches, and depressing START. If the punch runs out of paper tape, the machine halts with all 1s in the AC. Refill the punch and press CONTINUE to proceed with the compilation; then the two tapes can be spliced together.

Step 5 If an error occurs in the source language, the compiler types a three-letter plus two-digit code on the teleprinter followed by the current (last encountered) statement number. The compiler also prints the offending line with the errant character flagged by a line feed. See chapter 8 for the associated error conditions. As a rule, a source language error prevents proper execution of the compiled program. The error must be corrected and the program compiled again. However, compilation should be completed to uncover all errors in the same program.

Step 6 When all necessary compilations have been successfully completed, remove the output tape(s) from the punch.

Step 7 Place the paper tape labeled FORTRAN Assembler in the reader, set ADDRESS switches to 17770_8, and depress START. Set ACCUMULATOR switch 10 as follows: up if the assembler source tape is ASCII; down if it is FIODEC.

NOTE: The setting of AC10 should be identical to the setting of AC9 in step 3, above.

Step 8 Place the first program to be assembled in the reader. If several programs were compiled together, they will be separated from each other by a short length of

blank tape. The punch must be on. Depress CONTINUE. The assembler punches a partial binary output, displaying all ACCUMULATOR lights on when it is finished. Should an error occur during the assembly procedure, the assembler prints a message on the teleprinter. For a summary see chapter 8.

Step 9 Depress CONTINUE to finish punching the binary output. Undefined symbols in the source program (symbols which never appear on the left-hand side of an arithmetic statement, in an input statement or as the argument of a subroutine call, or in a COMMON statement) are printed with a relative location automatically assigned by the assembler. Any statement number which is referred to but never used as a statement label will be printed also. When finished, all ACCUMULATOR lights will again be on.

Step 10 If a printout of the relative locations of program symbols (ordered alphanumerically by symbols) is desired, put the rightmost switch of the ACCUMULATOR switches (bits 17) to the up position and press CONTINUE. With AC switch 16 up, pressing CONTINUE produces a listing ordered numerically by location counter. If the printout is not desired, leave the switch in the down position and press CONTINUE to restore the assembler for the next assembly. The ACCUMULATOR lights will all be off at this stage. If AC switch 11 is in the up position (on), listing will be done at the line printer.

Step 11 If more programs are to be assembled, place the next tape in the reader and return to step 8. If several programs were compiled together, be sure that the blank tape area separating them is under the reader light before continuing. Since the assembler uses a buffered loader, the end of one program and the beginning of the next program are likely to be read into the same buffer. It is usually necessary to withdraw a portion of tape which has already been read in order to start reading at the beginning of the second and succeeding programs on the same paper tape.

Step 12 Remove the assembled programs from the punch. Each program will have its title punched in readable format at the beginning. Since the FORTRAN Assembler is a one-pass assembler, the title will be the last item punched on the tape.

NOTE: The following steps describe the loading process. After each tape is loaded into memory the ACCUMULATOR lights display the first memory address not used.

Step 13 Load the main program through RIM start. It is important that the main program be loaded first since the Linking Loader is punched on the main program tape

only. The loader is a lengthy strip of tape immediately following the title with the eighth hole punched in every line. The RIM Loader, through use of a Bootstrap Routine, loads the Linking Loader which, in turn, loads the main program.

Step 14 Place any subprograms in the reader (readable title is always in the leader), and load through RIM start. The Linking Loader handles the problems of linking between programs. The first instruction executed by the RIM Loader is a jump to an entry in the Linking Loader.

Step 15 To obtain a printout of the absolute locations in memory of subprogram symbols and/or to determine if library subroutines are required, place 5_8 in the ADDRESS switches and depress START. If a subroutine or library function has been called but not yet loaded, its symbol will be preceded on the line by a minus sign followed by the address of the first reference to this symbol. If further subprograms are needed, they should be loaded as in step 14.

Step 16 Load the Library I/O tape; i.e., place the library tape in the reader, place 5_8 in the ADDRESS switches and depress START. If any subroutine names are preceded by " - ," load the 6DD or 9DD Library tape, i.e., set ADDRESS switches to 6_8 and depress START. When all called functions have been loaded, the loader halts, perhaps part way through the library tape.

Step 17 Load the tape labeled "FORTRAN Operating System" through RIM start. If paper tape input to the FORTRAN program is used, this should be ready in the reader. AC switches 9 and 10 must be set (up position) to indicate ASCII object-time output and input, respectively.

Step 18 Place 22_8 in the ADDRESS switches and depress START to execute the program.

NOTE: The Linking Loader does not detect when the user has loaded a program over common storage (assigned backward from the last address in memory). To guarantee that an overlay has not occurred, the first program address not used as indicated in the AC lights after loading should always be equal to or smaller than the lowest address in common storage necessary to store the arrays and common variables used in the program.

General Notes:

a. The first word of every FORTRAN program (main or subroutine) has a relative address of 1_8.

b. The initial relocation constant is 21_8.

c. After each program is loaded, the AC lights display the address of the next free location. This address is also the relocation of the next program to be loaded. (One location is unused between programs.)

CHAPTER 7
DIAGNOSTICS

The following diagnostics may be printed during compilations followed by the offending statement with a line feed after the last character processed. Each diagnostic is identified by a three-letter name, and a two-digit number. For all errors except those which indicate storage capacity exceeded, processing continues. The diagnostic error print (below) is followed by the current statement number.

As previously noted the occurrence of an error necessitates correction of the error and recompilation.

Error Name	Error Number	Reason for Error
CON		CONTROL STATEMENT
	1	Illegal control statement.
	2	Upper case character in control statement.
COM		COMMON STATEMENT
	1	Illegal entry in list.
	2	Symbol appears twice in COMMON.
ASG		ASSIGN
	1	N not a fixed-point number.
	2	Number not followed by "to."
	3	No fixed-point variable.
	4	Illegal format - variable.
SUB		SUBROUTINE AND FUNCTION
	1	Name not a variable.
	2	Dummy symbol not a variable.
	3	Dummy symbol used twice.
DIM		DIMENSION
	1	Array name not a variable.
	2	Array dimensioned twice.
	3	Dimension not a fixed-point number.

Error Name	Error Number	Reason for Error
DO		DO STATEMENT
	1	First two letters not DO.
	2	No statement number.
	3	No end test value specified.
	4	Too many characters.
ILF		ILLEGAL FORMAT
	1	Nonstatement number at left margin.
	2	Missing left parenthesis.
	3	Missing right parenthesis.
	4	Missing left parenthesis.
	5	Missing right parenthesis.
	6	Comma missing in GOTO.
	7	Variable missing in arithmetic statements.
	11	Illegal device number in input or output statement.
	12	Illegal format in accept statement.
	17	Extra right parenthesis.
	20	Extra characters in statement.
	22	Comma missing in repetitive element in I/O list.
	24	Illegal format in I/O list element.
	26	Illegal format statement number in an I/O statement.
ICH		ILLEGAL CHARACTER
	1	Illegal character.
	2	Illegal upper-case character.
	4	No more characters after an illegal one.
DIT		Miscellaneous errors. Cannot proceed.
	1	Logic error.
	2	Wrong place in table.
	3	Dispatch number too big.
	10	Too many CALS.
	11	Illegal CAL.
	12	Too many exits.

If any of the errors labeled DIT occurs, correct all other errors and recompile; if DIT errors still occur, note any pertinent data and send to DEC Programming Group.

Error Name	Error Number	Reason for Error
UFX		UNSEEN FIXED POINT
	1	Fixed-point number expected; punctuation character or no character appeared.
	2	Floating-point quantity appeared where fixed-point number expected.
	3	Fixed-point number expected; decimal number appeared.
FOR		FORMAT STATEMENT
	1	Character missing.
	2	Illegal format.
	3	Characters missing.
	4	Illegal control character.
	5	Illegal punctuation.
	6	Specification letter other than I,F,E,X,H.
	7	N too large in H format.
IFU		ILLEGAL FUNCTION USAGE
	1	Function name on left side outside function definition.
SCE		STORAGE CAPACITY EXCEEDED
		Processing may not proceed.
	1	Polish stack exhausted.
	2	Table exceeded.
	3	Table exceeded.
	4	Symbol generator exhausted.
	5	Table exceeded.
	6	Statement too long.
	7	Push down stack exceeded (too many nested DOS).

CHAPTER 8
ERROR MESSAGES

8.1 ERROR MESSAGES (FORTRAN ASSEMBLER)

The following error messages refer to the object program code generated by the compiler. Familiarity with this code is necessary for an understanding of this chapter. See the PDP-9 Symbolic Assembler program description for details.

With the exception of SCE (storage capacity exceeded) and ILP (illegal parity), assembly continues after the error message has been printed unless assembling a library tape. An error message may occur in one of three formats.

8.1.1 Format A

ERROR PREVIOUS VALUE SYMBOL NEW VALUE

Format A indicates errors in the redefinition of symbols. ERROR represents a three-letter code for the particular error. Whether the symbol was redefined depends upon the particular error.

Error	Meaning
MDT	The symbol was redefined with a comma.
RPS	A permanent symbol was redefined.
RDA	An attempt to redefine a symbol was made. The symbol was not redefined.

8.1.2 Format B

ERROR OCTAL ADDRESS SYMBOLIC ADDRESS

The general error message is printed in format B. It includes both the octal address and the symbolic address at which the error occurred.

Error	Meaning
IFP	Illegal format in parameter assignment.
IFC	Illegal format in a symbolic address tag.
IFQ	Illegal format in library list.
IFY	Illegal format in internal declaration.
IFZ	More than one symbol in internal declaration.
LIQ	Illegal term punctuation in library list.

Error	Meaning
MDT	The location counter and address disagree in an address assignment.
TUA	Too many undefined symbols in a symbolic address tag.
ILF	Illegal format in a pseudo instruction
LIT	Illegal terminator in a PUNDEF or EXTERNAL list.
IFL	Illegal format in a PUNDEF or EXTERNAL list.
IFS	Illegal format in a START.
IFI	Illegal format in an input pseudo instruction.
SCE	Storage capacity exceeded.
INS	A nonsymbol appeared in a PUNDEF list.
IFX	External symbol preceeded external declaration.

8.1.3 Format C

ERROR OCTAL ADDRESS SYMBOLIC ADDRESS CAUSE

Format C is an expanded version of Format B. CAUSE is additional information to help the programmer ascertain the cause of the error. For example, in the case of an error caused by an undefined symbol, the symbol will be printed.

Error	Cause	Meaning
ILP	character	Illegal parity (place correct character in ACS and press CONTINUE). May also be caused by reading tape in backward order.
UST	symbol	Undefined symbol in a START or PAUSE.
UAA	symbol	Undefined symbol in an absolute address assignment.
UPA	symbol	Undefined symbol in a parameter assignment.
ICH	character	Illegal character.
SYS	symbol	Previously defined symbol in internal declaration.
UPN	symbol	Undefined symbol in a punch pseudo instruction.

8.1.4 Undefined Symbol Assignment

At the end of assembly before the loader is punched, the undefined symbols and their definitions are printed. Each undefined symbol used in a storage word will be defined as the address of a register at the end of the program, and the definition printed. If the symbol was not used in a storage word, the symbol is printed and not defined. An example of the latter is a symbol which appears to the right in a parameter assignment only.

8.1.5 Error Messages from the Linking Loader

Error	Meaning
CSE	Data Block had checksum error.
SCE	Storage capacity exceeded. Program will load into the Loader's symbol table if allowed.
ASE	Assembly error. Tape is in error or failed to read properly.
LMR	Assembly error in library format. Tape is in error or failed to read properly.

8.1.6 Error Halts in the FORTRAN Object Time System

The following entries refer to the OTS itself which is always in core at run time.

PC	Symbolic	Meaning
12725	CHECK-1	Illegal device number in I/O call.
14423	EFMEND	An illegal interpretive.
15553	TOOCAL	Too many CALS. Program error, implies CAL entry but no exit.

NOTE: If the computer appears to hand up in a loop in either of the following areas:

14324 ff. , 13753 ff

an arithmetic error is the probable cause. (Division by zero, log (−0 etc.). The apparent hang-up is not real. The program will eventually resume.

CHAPTER 9
PDP-9 FORTRAN II OPERATING TEST

9.1 INTRODUCTION

This program, while not an exhaustive test of the DEC FORTRAN II System, uses most of the algebraic and control features of the language. If a user can compile, assemble, load, and then run this test successfully, it indicates that both the FORTRAN System and the PDP-9 are operating satisfactorily.

9.2 PRELIMINARY REQUIREMENTS

9.2.1 Storage

The program uses locations 22-5420 for itself and its two subroutines, plus space for the following FORTRAN requirements: SARITH, NARITH, or EARITH; IO2; all of the FORTRAN arithmetic library (SQRTF, SINF, COSF, ATANF, LOGF, CLOGF, EXPF, ABSF, and SABSF); and finally, the Object Time System.

9.2.2 Subprograms and/or Subroutines

The program consists of the Main Test, a Test Subroutine, and a Test Function, meant to be run with the FORTRAN Object Time System.

9.2.3 Equipment

8K PDP-9
Paper Tape Reader
A Type 33, or 35 Teletype

9.3 LOADING OR CALLING PROCEDURE

9.3.1 Loading

The original FORTRAN symbolic tapes should be used for the test. In the procedure that follows, the RIM Loader is assumed in core, and all FORTRAN tapes are part of the FORTRAN System.

a. Place the FORTRAN compiler in the reader, 17770 in the ADDRESS switches, and press START.

b. When the compiler has been read in, place FORTRAN symbolic tape 1 of 3 (the first subroutine) in the reader, set ACCUMULATOR switches 9 and 10 to a 1 (ASCII input), and press CONTINUE.

c. When the compiler stops with all AC lights on, it has finished punching the assembly version of the first subroutine. There is no typeout in an error-free compilation. Manually punch out one fanfold of tape leader, place FORTRAN symbolic tape 2 of 3 in the reader, and press CONTINUE to compile the second subroutine. When the computer stops, place FORTRAN symbolic tape 3 of 3 in the reader, manually punch one fanfold of tape leader, and press CONTINUE.

d. When punching stops, remove the three programs from the punch, place the FORTRAN assembler in the reader, and press START.

e. When the assembler has read in, insert the tape just punched in the reader, and press CONTINUE.

f. When the assembler stops with all AC lights on, press CONTINUE to finish assembly of the first subroutine. The only typeout should consist of the title of the tape and a few carriage returns. The same applies to the additional assemblies to be done below. When the assembler stops a second time with all AC lights on, press CONTINUE to reinitialize the assembler.

g. When the assembler stops with all AC lights off, it has finished punching the binary version of the first subroutine. Since the paper tape may be positioned past the beginning of the second subroutine (the assembler reads until a buffer is filled or a stop code is reached without reference to the end of each subprogram), move it back to the leader preceding the second subroutine and press CONTINUE.

h. When the assembler halts with all AC lights on, press CONTINUE to finish assembly of the second subroutine. When the assembler stops a second time with all AC lights on, press CONTINUE to reinitialize the assembler.

i. When the assembler halts with all AC lights off, it has finished punching the binary version of the second subroutine. Position the paper tape at the beginning of the main program, and press CONTINUE to assemble the main program.

j. When the assembler stops with all AC lights on, press CONTINUE to finish assembly of the main program. When the assembler stops a second time with all AC lights on, remove the binary tape from the paper tape punch.

k. The front of the binary tape is the last item punched. A title is punched on the tape at the end which should say TEST. Place the binary tape in the reader and press START.

l. The tape stops after the main program has loaded. Press START to read in the first subroutine and start again for the second subroutine.

m. The tape stops after the second subroutine has loaded. Place the FORTRAN I/O Library in the reader, 00006 in the ADDRESS switches, and press START.

n. When the tape runs out of the reader, press STOP and then EXAMINE to clear a read select; place the FORTRAN arithmetic library (6- or 9-digit accuracy) in the reader, and press START.

o. When the tape stops, place 00005 in the ADDRESS switches, and press START to get a memory map on the Teletype. If no names are preceded by a minus sign, loading was successful.

p. Place the FORTRAN Object Time System (OTS) in the reader, 17770 in the ADDRESS switches, and press START.

q. When the OTS has loaded, the program is ready to begin. Place 00022 in the ADDRESS switches, clear the ACCUMULATOR switches, and press START.

9.3.2 Switch Settings

During run time, if the program detects an error, it types out a message and re-executes the bad code. Setting AC switch bit 0 to a 1 suppresses re-execution of bad code and causes the program to proceed. All switches should be set to 0 when starting.

9.4 USING THE PROGRAM

The program is started or restarted by placing 00022 in the ADDRESS switches and pressing START. The program types out a set of tautologies (truth statements), calculates, and types out END OF TEST. Any departure from this procedure is an error.

9.4.1 Errors in Usage

Although the user cannot initiate any errors himself, the program attempts to detect any irregularities in its own execution. If it finds any, it writes an error message and (unless AC switch 0 is set) re-executes the faulty code. An error is indicated if one of the tautologies is false or if one of the following messages is typed.

BAD GOTO, DRAGONFLY = nnnn

BAD ARRAY, INDX5 = nnnn

BAD DO LOOP, NUM = nnnn

BAD FUNCTION VALUE = nnnn

BAD SUBROUTINE RETURN, VALUE = nnnn

where:

The first message means that in a series of different kinds of IF statements and GOTO statements, one failed, and the variable DRAGONFLY was the wrong value in the wrong place.

The second message means that a number placed in the INDX5-th position in an array was not there when it was requested later.

The third message means that a series of nested DO LOOPS was not executed in the predetermined number of times.

The last two messages indicate that the test function and/or the test subroutine returned unexpected values.

9.4.2 Recovery from Such Errors

Any error indicates that either the computer is not functioning correctly or that an incorrect or obsolete version of the compiler or assembler has been used.

9.5 DETAILS OF OPERATION AND STORAGE

The usefulness of the test lies in the fact that the program has been compiled, assembled, run previously, and, therefore, should run again without errors. It uses the five basic operations in fixed and floating point, the nine library functions, the IF statement, nested and non-nested DO loops, four different formats of GOTO statements, the ASSIGN statement, comments, CONTINUE, CALL, RETURN, SUBROUTINE, FUNCTION, and STOP.

9.5.1 Examples and/or Applications

The test should be used when there is doubt of the reliability of the hardware or software systems. If the test compiles, assembles, loads, and runs without an error, the computer and compiler can be presumed to be in correct working condition. The output from a successful run should appear as follows:

```
      SQRTF     ( 4.00000) =   2.00000
      SINF      ( 1.57079) =    .99999        (1.00000 if using 9-digit
                                               subroutines
      COSF      ( 3.14159) =   -.99999        (-1.00000 if using 9-digit
                                               subroutines
4.0*ATANF       ( 1.00000) =   3.14159
      LOGF      ( 1.00000) =    .16138-09
     CLOGF      ( 1.00000) =    .70089-10
       EXPF     (  .00000) =   1.00000
       ABSF     (-1.00000) =   1.00000
4.00000          +2.00000  =   6.00000          .
4.00000          -2.00000  =   2.00000
4.00000          *2.00000  =   8.00000
4.00000          /2.00000  =   2.00000
4.00000          ↑2.00000  =  16.00000        (15.99999 if using 9-digit
                                               subroutines)
      XABSF     (      -1) =   1
      4+              2   =   6
      4-              2   =   2
      4*              2   =   8
      4/              2   =   2
      4↑              2   =  16

END OF TEST.
```

APPENDIX 1
CHARACTER CODE EQUIVALENCES

The two test handling modes (A, H) use character code sets for which the octal equivalents differ in some respects for the two formats. Under A format, characters are stored in a 6-bit, 64-character code called "line printer FIODEC." Under H format, characters are stored in a code called flexowriter FIODEC, which includes a case shift, indicated by u in the table. The upper-case escape code is 74; the return code, 72.

When providing a dummy string of text for a READ Hollerith operation, the user must remember that the code is compiled directly into the format control list. This requires that the number of core locations occupied by the dummy text correspond exactly to the test read in at run time. Since this may vary in content, the following rules are necessary:

1. The dummy string and the input string must be the same length.

2. Corresponding characters in each string must agree in case; i.e., both the characters in the dummy string and the character in the input string must be from the set which is flagged by a u or both be from the set which is not flagged.

NOTE: If a character equivalence is indicated by NA in the table, the character is not available for use in the indicated format. On the model 33, a shift with the letters M, L, K produces the characters], \, [respectively.

TABLE 5 CORE REPRESENTATIONS OF THE ASCII CHARACTERS, A AND H FORMATS†

Character	ASCII	'A' Format	'H' Format
Space	240	0	0
!	241	15	u 05
"	242	32	u 01
#	243	56	56
$	244	60	u 40
%	245	NA	NA

†This table applies when typing a FORTRAN program either off-line or using the Symbolic Tape Editor.

TABLE 5 CORE REPRESENTATIONS OF
THE ASCII CHARACTERS, A AND H FORMATS† (continued)

Character	ASCII	'A' Format	'H' Format
&	246	NA	NA
'	247	12	u 02
(	250	57	57
)	251	55	55
*	252	72	u 73
+	253	74	u 54
,	254	33	33
-	255	54	54
.	256	73	73
/	257	21	21
0	260	20	20
1	261	01	01
2	262	02	02
3	263	03	03
4	264	04	04
5	265	05	05
6	266	06	06
7	267	07	07
8	270	10	10
9	271	11	11
:	272	NA	NA
;	273	NA	NA
<	274	17	u 07
=	275	53	u 33
>	276	34	u 10
?	277	37	NA
@	300	NA	NA
a	301	61	61
b	302	62	62
c	303	63	63
d	304	64	64

†This table applies when typing a FORTRAN program either off-line or using the Symbolic Tape Editor.

TABLE 5 CORE REPRESENTATIONS OF
THE ASCII CHARACTERS, A AND H FORMATS† (continued)

Character	ASCII	'A' Format	'H' Format
e	305	65	65
f	306	66	66
g	307	67	67
h	310	70	70
i	311	71	71
j	312	41	41
k	313	42	42
l	314	43	43
m	315	44	44
n	316	45	45
o	317	46	46
p	320	47	47
q	321	50	50
r	322	51	51
s	323	22	22
t	324	23	23
u	325	24	24
v	326	25	25
w	327	26	26
x	330	27	27
y	331	30	30
z	332	31	31
[	333	77	57
\	334	76	NA
]	335	75	55
↑	336	35	11
←	337	NA	NA
tab	211	NA	36
carriage return	215	NA	NA

†This table applies when typing a FORTRAN program either off-line or using the Symbolic Tape Editor.

Page Missing From Original
Document

Page Missing From Original
Document

Page Missing From Original
Document

Page Missing From Original
Document

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

Printed in U.S.A.