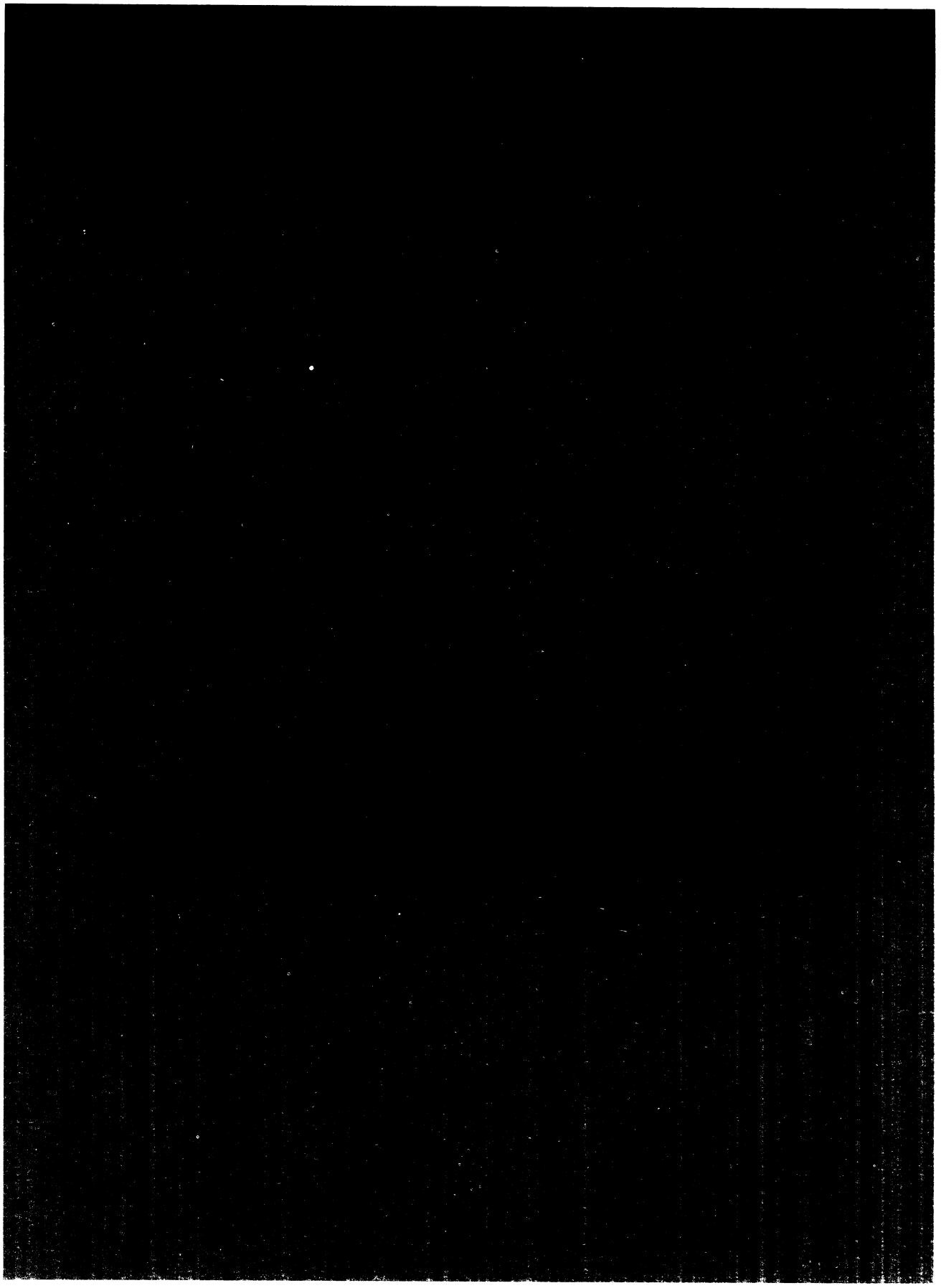


DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

DEC-08-AFAC-D

# PDP-8 FORTRAN
# PROGRAMMING MANUAL

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

Reprinted

October 1966
September 1967

Revised

May 1967
October 1967

CONTENTS

CONTENTS (continued)

CONTENTS (cont'd)



ILLUSTRATIONS

ILLUSTRATIONS (continued)

# PREFACE

The program discussed in this manual, though written for use on the Programmed Data Processor-8 computer, can also be used without change on Digital's Programmed Data Processor-5. This compatibility between the libraries of the two computers results in three major advantages:

1. The PDP-8 comes to the user complete with an extensive selection of system programs and routines making the full data processing capability of the new computer immediately available to each user, eliminating many of the common initial programming delays.

2. The PDP-8 programming system takes advantage of the many man-years of field testing by PDP-5 users.

3. Each computer can take immediate advantage of the continuing program developments for the other.

# CHAPTER 1

# INTRODUCTION

Using a digital computer to solve a problem generally involves the following series of steps:

a. Determining the correct procedures to be used, including mathematical formulas, the handling of data, the presentation of results, etc.

b. Arranging the procedures in the proper order.

c. Determining the sequence of computer instructions that will perform the operations specified.

d. Converting the sequence of instructions into binary notations in a physical medium capable of being entered into the computer for execution.

Much of the progress and development in programming has been made in discovering ways to make the computer perform more of the steps listed, above, and leaving the programmer free to concentrate more on the problem itself. At first, programmers entered instructions manually from the computer console or prepared the binary program for direct input. Later, a symbolic notation was developed for a computer instruction set, and programs called assemblers were written that could interpret a tape or card deck punched with this notation. These assemblers translate each symbolic instruction into a machine operation and assemble an executable program. Thus, an assembler can accept input from step c above.

An assembler requires that the programmer be familiar with the particular instruction set of the computer being used. To solve the same on another computer would usually require complete reprogramming.

To free the programmer from the need of learning a given computer's language before using the machine to solve problems, compilers were developed which accept input more closely related to the problem and convert the input into an executable program. FORTRAN is such a compiler. It accepts input in the form of statements which resemble mathematical formulas

(hence its name, which stands for FORmula TRANslation) and compiles sequences of instructions necessary to perform the procedures specified. Non-mathematical operations are specified by English words. In terms of the steps given above, a FORTRAN source program is a product of step b; the computer performs steps c and d.

A FORTRAN compiler can thus be written for any digital computer to convert a source program into an executable program. Extensive reprogramming is made unnecessary, since the same source program can be compiled on different machines with only minor changes.

The PDP-8 FORTRAN System consists of two subsystems: the compiler and the object time system.

The FORTRAN Compiler contains the instructions the computer requires to perform the clerical work of translating the FORTRAN version of the problem statement into an object program in the language of the object system.

When the compiler detects errors in statement format or usage, it prints out diagnostic messages (see Appendix A). The programmer or operator may then take the appropriate corrective measures.

After compilation, the object time system is used to execute the program. This system contains the interpreter, the arithmetic function subroutines, and the input-output packages. When program execution is required, the object time system, object program, and the data it will work with are loaded into the computer for solution of the problem.

This is a one-pass compiler, which means the source language tape must be read only once. The compiler generates one tape which contains coding in a form that is executable under control of the object time system.

To use the system, it is only necessary to load the compiler. The compiler then processes the source language tape and generates the object program tape. This object program tape can be run at any time simply by loading the object system, which, in turn, loads and executes the object program.

# CHAPTER 2

# THE FORTRAN LANGUAGE

## STATEMENTS

Figure 1 is an example of a FORTRAN program, consisting of a 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 writes to identify the program. It is not incorporated into the final executable program.

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.

The end statement must be physically 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.

```
    C;  THIS PROGRAM CALCULATES FACTORIALS
    5;  TYPE 200
   10;  ACCEPT 300,X
        FACT=Y=1.
        IF (X) 5,32,30
   30;  IF (X-Y) 41,32,33
   32;  TYPE 400,X,FACT
        GO TO 10
   33;  FACT=FACT*(Y=Y+1.)
        GO TO 30
   41;  PAUSE
        GO TO 5
  200;  FORMAT (/, "PLEASE TYPE A POSITIVE NUMBER", /)
  300;  FORMAT (E)
  400;  FORMAT (/,E, "FACTORIAL IS",E)
        END
```

Figure 1 A Fortran Program

## PROGRAM FORMAT

Each line contains two fields: the first, which begins at the margin, is an identification field; the second contains the statement proper (see Figure 1).

The identification field extends from the left-hand margin up to a semicolon character. This field may be left blank, or it may contain one of the following types of identification:

1. The first digit of a statement number. This number, which may be any positive integer from 1 to 2047 inclusive, identifies 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. Up to 40 statements can have statement numbers.

2. The letter C. This identifies the remainder of the line as a comment (see Section Comments).

The semicolon (;) is necessary only if the statement is numbered or is a comment, (i.e., if the identification field is blank, the semicolon may be omitted).

The statement field begins immediately after the semicolon and extends through the next carriage return. Although the continuation character (') allows a single statement to extend over two or more lines, no more than one statement can be written on one line.

## TYPES OF STATEMENTS

FORTRAN statements are of several types with differing functions distinguished as follows:

1. Arithmetic statements resemble algebraic formulas. They specify the mathematical operations to be performed.

2. Program control statements direct the flow of the program.

3. Specification statements allocate data storage, determine variable and data types, and specify input-output formats.

4. Input-output statements control the transfer of information into and out of the computer.

## COMMENTS

Although a FORTRAN program using English words and mathematical symbols can be read and understood more easily than a symbolic language program, it is helpful to provide comments freely throughout the program to explain the procedures being used. Such comments, identified by a C in the first position of the line, are not interpreted by the compiler and have no effect on the executable program.

```
      .
      .
      .
C;    CALCULATE PERCENTAGE OF CORRECT RESPONSES
C;    PERCENTAGE = -1  IF THERE ARE NO ITEMS
C;                     IN CATEGORY
      DO 47 I=1, 57
      DO 48 J=1, 6
      IF (ITMS (J)) 46, 46, 51
46;   PRCN (I) = -1.0
      .
      .
      .
```

Figure 2 Example of Comments

## CONTINUATION

Frequently, a statement may be too long to fit on one line. If the character single quote (') appears as the last character of a line before the carriage return, the next line is treated as a continuation of the statement on the line above (see Figure 3). 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 128.

```
1;  FORMAT
 ;  IF (NR (1) -1)) 2, 2, 3
2;  AP=-14.73
 ;  GO TO 6
3;  IF (NR (1) -2) 4, 4, 5
4;  AP=-44.19
 ;  GO TO 6
5;  AP=SH (2)*3.0-AG (4)/'
    AG (1)+SQTF (AG (14))
                .
                .
                .
```

Figure 3 The Continued Statement

## THE CHARACTER SET

The characters which are meaningful in FORTRAN belong to the ASCII set listed in Appendix D. Of these, the acceptable characters are: all letters and numbers — A through Z, 0 through 9; control characters — semicolon (;), carriage return (CR), line feed (LF), single quote ('), double quote ("), left parenthesis (, right parenthesis ), period (.), comma (,); and the operators — plus (+), minus (-), slash (/), asterisk (*), equal sign (=). All other characters are ignored by the compiler except in Hollerith information of FORMAT statements where all Teletype characters are legal. The character space has no grammatical function except in FORMAT statements, but may be used freely to make a program easily readable.

# CHAPTER 3

# FORTRAN ARITHMETIC

## ARITHMETIC EXPRESSIONS

An algebraic formula such as the following

$$[5a + 4b\ (x^2 - x_0)]\ /2a \sin \theta$$

represents a relationship between literal symbols (a, b, x, $x_0$, $\theta$) and constants (5, 4, 2) indicated by mathematical functions and arithmetic signs (+, -, /, multiplication, exponentiation, sine). This same formula can be written as a FORTRAN arithmetic expression with very little change in appearance:

```
(5.*A + 4.*B*(X**2 - XZRO) )/ (2.*A*SINF(THTA))
```

The construction of both expressions is the same; the differences are notational.

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

The following examples demonstrate the properties of arithmetic expressions. Each expression is shown with its corresponding algebraic form.

| Algebraic Expression | FORTRAN Expression |
|---|---|
| $ax^2 + bx + c$ | `A*X**2 + B*X + C` |
| $\frac{(a^2 - b^2)}{(a + b)^2}$ | `(A**2 - B**2) / (A+B) **2` |
| $\frac{4 \pi r^2}{3}$ | `4.*PI*R**2/3.` |

| | |
|---|---|
| $\frac{3x\pi - 2(x+y)}{4.25}$ | (3.*X*PI-2.* (X+Y)) /4.25 |
| $a \cdot \sin\theta + 2a \cdot \cos(\theta - 45)$ | A*SINF(THTA)+2.*A*COSF(THTA-0.7853982) |
| $\frac{2\sqrt{x}}{3}$ | 2.*SQTF(X)/3. |

## Constants

Constants are explicit numerical quantites. They may be integers, decimals, or numbers in decimal exponent form. Some examples follow:

integers

| | | |
|---|---|---|
| 5 | -70 | 2047 |

decimals

| | | |
|---|---|---|
| 18.75 | 3.14159 | -0.00025 |

decimal exponent

1.66E-16 (meaning $1.66 \times 10^{-16}$)

These different forms of numerical representation are described in detail in Chapter 4.

## Variables

A variable is a literal symbol whose value is not implicit; its value may be changed during the execution of the program. A variable name is composed of one or more characters according to these three rules:

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

2. The first character must be alphabetic (i.e., A through Z).

3. Only the first four characters of any variable name are meaningful. All characters after the fourth are ignored by the compiler.

Some examples of acceptable variable names are:

| | | | |
|---|---|---|---|
| A | XZRO | DC8B | EPSL |
| K | LST8 | | |
| THTA | P51 | XSUM | |

The name EX IT represents one variable, not two. (Remember that blank spaces have no function in FORTRAN.) Thus, EX IT, EXIT, or even EXI T, are identical names as far as the compiler is concerned because they all reference the same variable.

The name EPSILON would be interpreted by the compiler as EPSI, since only the first four characters are meaningful. For example, the two names XSUM1 and XSUM2 would be considered identical.

Some incorrect variable names are:

| | |
|---|---|
| 9SRT | (first character not alphabetic) |
| GO(5 | (illegal character included) |
| CSH$ | (illegal character included) |

## Operators

The operators are symbols representing the common arithmetic operations. The important rule about operators in the FORTRAN arithmetic expressions is this: Every operation must be explicitly represented by an operator. In particular, the multiplication sign must never be left out. A symbol for exponentiation is also provided since superscript notation is not available. To illustrate the rule, here are the FORTRAN and algebraic forms given in the section on arithmetic expressions:

```
(5.*A + 4.*B*(X**2-XZRO) ) / (2.*A*SINF (THTA))
```

$$[5a + 4b\ (X^2 - X_0)]\ /2a\ \sin\theta$$

Normally, a FORTRAN expression is evaluated from left to right just as an algebraic formula is. There are exceptions to this rule. Certain operations are always pertormed before others regardless of order. This priority of evaluation is as follows:

| | | |
|---|---|---|
| 1st. | Expressions within parentheses | ( ) |
| 2nd. | Unary minus | — |
| 3rd. | Exponentiation | ** |
| 4th. | Multiplication | * |
| | Division | / |
| 5th. | Addition | + |
| | Subtraction | - |

The term binding strength is sometimes used to refer to the relative position of an operator in a table such as the one above, which is 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 unary minus is simply the operator which indicates a quantity whose value is less than zero, such as -53, -GAMME, -K. It refers only to the operand which it precedes as opposed to a binary operator, which refers to operands on either side of itself, as in the expression a-b. A unary minus is recognized by the fact that it is preceded by another operator, not by an operand. For example:

A + B**-2/C-D

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

The left-to-right rule can now be stated more precisely as follows:

A sequence of operations of equal binding strength is evaluated from left to right.

To change the order of evaluation, parentheses are required. Thus, the expression A-B*C is evaluated as A-(B*C), not (A-B)*C. The example below gives a few more illustrations of the left-to-right rule.

| The expression | is evaluated as |
|---|---|
| A/B*C | (A/B)*C |
| A/B/C | (A/B)/C |
| A**B**C | (A**B)**C |

An easy way to check the proper pairing of parentheses is by counting out, illustrated in the following example:

```
(Z+AM*(ZM+1.)) / ((X**2+C**2)*P)
1      2      10  12          1 0
```

The procedure is this: Reading the expression from left to right, assign the number 1 to the first left parentheses (if you encounter a right parenthesis first, the expression is already wrong!) Increase the count by one each time a left parenthesis is read, and decrease the count by one when a right parenthesis is used. When the expression has been completely scanned, the count should be zero. If it becomes less than zero during the scanning, there are too many right parentheses. If it is greater than zero at the end of an expression, then the pairing is incorrect.

Use of Parentheses

Note the use of parentheses in the following example below. They are used to enclose the subscript of the dimensioned variable D, to specify the order of operations of the expression involving A, B, C, and to enclose the argument of the function.

```
D(I+J) = (A+B)**C+SINF (X)
```

In algebra there are several devices, such as square brackets ( [ ] ), rococo brackets ({ } ), etc., for distinguishing between levels when subexpressions are nested. In FORTRAN, only the curved parentheses are available, so the programmer must be especially careful to make certain that parentheses are properly paired. In a given expression, the number of left parentheses must be equal to the number of right parentheses.

## Functions

Functions are used in FORTRAN just as they are in ordinary mathematics -- as variables in an arithmetic expression.

The function name represents a special subprogram which performs the calculation necessary to evaluate the function; the result is used in the computation of the expression in which the function occurs.

PDP-5/8 FORTRAN provides several mathematical functions: square root, sine, cosine, arc tangent, exponentiation, and natural logarithm.

The argument of a function can be a simple or subscripted variable or an expression. The argument must be in floating point. FORTRAN recognizes a term as a function when the term is a predefined symbol ending in F followed by an argument enclosed in parentheses (if the F is missing from the term, the symbol is treated as a subscripted variable). The argument of a function can consist of another function or group of functions. For example, the expression:

LOGF(SINF(X/2)/COSF(X/2)) is equivalent to log*tan $\frac{X}{2}$.

The PDP-5/8 FORTRAN library currently consists of the following functions:

| Function Name | Meaning |
|---|---|
| SQTF (X) | square root of X |
| SINF (X) | sine of X, where X is expressed in radians |
| COSF (X) | cosine of X, where X is expressed in radians |
| ATNF (X) | arc tangent X, where the angle is given in radians |
| EXPF (X) | exponential of X |
| LOGF (X) | logarithm of X |

## THE ARITHMETIC STATEMENT

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 is interpreted in a special sense; it does not represent a relation between left and right members, but it specifies an operation to be performed.

> NOTE: 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.

This means that the value of the left-hand variable will be different after the execution of an arithmetic statement. A few illustrations of the arithmetic statement are given below.

a. VMAX = VO + AXT

b. T = 2.*PI*SQTF(1./G)

c. PI = 3.14159

d. THTA = OMGA + ALPH*T**2/2.

e. MIN = MIN0

f. INDX = INDX +2

With the interpretation of the equal sign defined previously, example f becomes meaningful as an arithmetic statement. If, for example, the value of INDX is 40 before the statement is executed, its value will be 42 after execution.

Perhaps another way of looking at the equal sign illustrates its use and interpretation more fully. In arithmetic expressions, a binary operator requires an operand on its left and right. The equal sign of an arithmetic statement is considered to be a binary operator also. This interpretation is demonstrated in the following revised table of operators:

| Operator | Use | Interpretation |
|---|---|---|
| — (Unary) | -A | negate A |
| ** | A**B | raise A to the Bth power |
| * | A*B | multiply A by B |
| / | A/B | divide A by B |
| + | A+B | add B to A |
| - (Binary) | A-B | subtract B from A |
| = | A=B | replace A with B |

Treated this way, the equal sign is considered to have the lowest binding strength of all the operators. This means that the expression on the right is evaluated before the operation indicated by = is performed.

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

CPRM = (CKL - CKG) / (CPG = P*(Q + 1.))

The internal arithmetic statement, CPG = P*(Q + 1.), is set off from the rest of the statement by parentheses. The complete statement is a concise way of expressing the following common type of mathematical procedure:

$$\text{Let} \quad c' = \frac{c_{kl} - c_{kg}}{c_{pg}}$$

$$\text{where} \quad c_{pg} = p*(q+1)$$

The stating of a relation followed by the conditions for evaluating any of the variables can be expressed in a single arithmetic statement in FORTRAN.

Another important result of treating the equal sign as an operator is that the operations may be performed in sequence. Just as there may be a series of additions, A+B+C, so may there be a series of replacements, A=B=C=D. Note that since the operand to the left of an equals sign must be a variable, only the rightmost operand, represented by D in the example, may be an arithmetic expression. The statement is interpreted as follows: "Let the value of the expression D replace the value of the variable C, which then replaces the value of the 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:

```
XZRO=SZRO=AZRO=0
T=T1=T2=T3=60
P=FP=4.*ATM-AK
```

Only single level replacements will compile correctly in this manner.
For example: Statements of the type A(1) = A(2) = R(1) = 0.123 are not allowed and will not compile properly.

Another useful result in treating the equal sign as an operator is that 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.

Example:

```
A = M
K = B
```

# CHAPTER 4

# NUMBER REPRESENTATION AND VARIABLE TYPES

## INTEGERS AND FLOATING POINT NUMBERS

In mathematics there are many ways to categorize numbers. They may be positive or negative, rational or irrational, whole numbers or fractions. In FORTRAN, the treatment of numbers is separated into integers and decimals, distinguished as follows:

1. Integers are positive or negative numbers written without a decimal point. These numbers are integers: 9, 17, 147, 1024, 2047. The last number, 2047, is the largest quantity that can be expressed as a FORTRAN integer. For fractional quantities and for numbers larger than $\pm 2047$ (which is $2^{11}-1$), the second type of number is required.

When using integer arithmetic, any fractional results are truncated. For example, the expression M=N/3 with N=8 would result in M=2. This applies only to division because multiplication, addition and subtraction yield integral results.

2. Floating point numbers have two forms. They are simple decimals, such as 0.0025, .4, 57., 2.71828; or numbers in decimal exponent form. Numbers in decimal exponent form are simple decimals multiplied by a power of 10.

Examples:

| Mathematical Form | FORTRAN Form |
|---|---|
| $6.023 \times 10^{23}$ | 6.023E23 |
| $1.66 \times 10^{-16}$ | 1.66E-16 |
| $72. \times 10^{12}$ | 72E12 |

In general, a floating point number in decimal exponent form is NE±K, where N may be an integer or simple decimal, and K is an integer from 0 to 99, inclusive. The construction NE±K is used to represent the number $Nx10^K$. The following are floating point representations of the number 19:

19.0
.19E2
1.9E+1
1900E-2
190.0E-1

## FIXED-AND FLOATING-POINT REPRESENTATION

The difference between integers and real numbers in FORTRAN is the way in which each is represented in core memory.

A FORTRAN integer is stored in one 12-bit computer word. The sign of the number is kept in the high-order bit and the magnitude in the remaining 11 bits. This representation, shown schematically in Figure 4 is called fixed point, because the decimal point is always considered to be to the right of the rightmost digit. A FORTRAN integer may not exceed the range of -2047 through +2047. All integers greater than ±2047 are taken modulo 2048 (that is 2049 is taken as 0001, 4099 is taken as 3).

The floating point format consists of two parts: an exponent (or characteristic) 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 zeros eliminated so that the leftmost bit is always significant. The exponent represents the power of two by which the mantissa is multiplied to obtain the true value of the number for use in computation. The exponent and mantissa each are stored in 2's complement form.

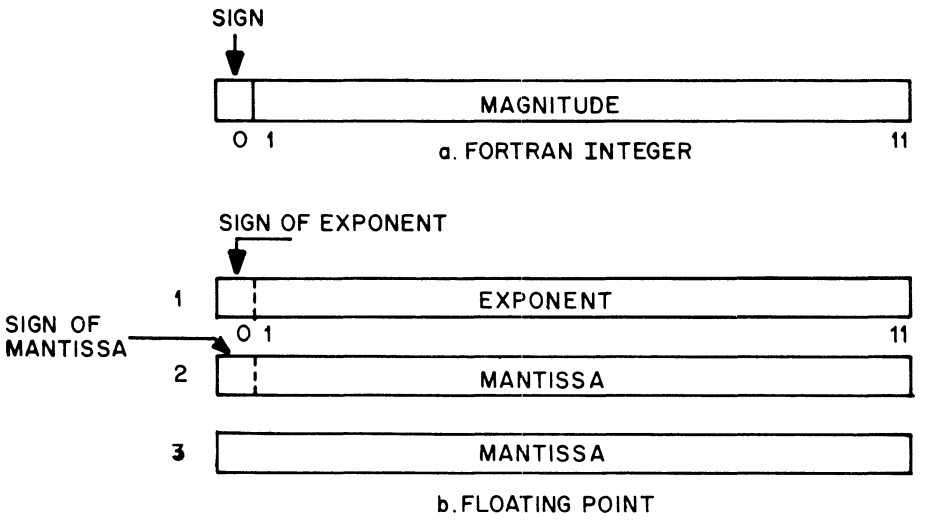



Figure 4 Number Representation

## TYPES OF VARIABLES

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

1. Integer variable names must begin with one of the letters I, J, K, L, M, or N.
2. Floating point variables are designated by names beginning with any other letter.

These are integer variable names: INDX, KDTA, M359. These are floating-point variable names: ZXRO, CONT, FICA.

Integers cannot appear in floating point expressions except as exponents or subscripts. Some examples of illegal and legal expressions are as follows:

| Expression | Legal | Mode |
|---|---|---|
| A(I)*B(J)**2 | Yes | Floating |
| I(M)*K(N) | Yes | Fixed |
| 4.*J | No | - |
| I+D | No | - |
| 16.*B | Yes | Floating |
| (K+16)*3 | Yes | Fixed |
| A**(I+2)/B | Yes | Floating |
| 8*A | No | - |

# CHAPTER 5

# SUBSCRIPTED VARIABLES

## ARRAYS

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 means of a symbol denoting the array and subscripts identifying the position of the element. For example, the sixth element in a vector v is designated $v_6$.

In FORTRAN, array elements are similarly identified. The array is given a name subject to the same rules as the names of variables, described in Chapters 3 and 4. The subscript which identifies an element of the array is enclosed in parentheses. The element referred to in the preceding paragraph would have the following form in FORTRAN:

```
V(6)
```

Such a name designates a subscripted variable, which may be used in computation just like a simple variable. The array name determines the mode, integer, or floating point of all the elements in the array.

The example below gives a few illustrations of the use of subscripted variables.

```
a. X(I+L)=X(I)+ALPH(I)*P(I)
b. X(I+3)=X(I+2)+X(I+1)/2.
c. C(J)=A(I*J+3)
d. A=B(6)
```

## Subscripts

As the example above illustrates, 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. This means that there may not be any floating-point quantities in a subscript expression.

## The Dimension Statement

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

1. Which are the subscripted variables?

2. What is the maximum dimension of the 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:

```
DIMENSION A(20), B(15)
```

where A and B are array names, and the integer constants 20 and 15 are the maximum dimensions of each subscript.

The rules governing the use of array names and the dimension statement are as follows:

All array names must appear in a dimension statement. DIMENSION may be used as many times as desired and may appear anywhere in the FORTRAN program, provided that the DIMENSION of an array appears before any statement which references the array.

```
DIMENSION LIST(30), MAT(100), REGR(20)
```

In the statement in the example above, the names LIST and MAT designate integer arrays; that is, each element is an integer. The third name, REGR, designates a floating-point array. The first array is a list of 30 elements maximum, so that 30 words of storage are set aside for its use. The third array is floating-point and there are 20 elements in it. Since this array is floating, each element requires 3 words of storage so that 60 words are set aside for the array.

```
DIMENSION B(30), I(15)
```

This version of the PDP-5/8 FORTRAN does not have the facility for double subscripted variables. To accomplish double subscripting, the programmer has to include indexing statements in the source program as illustrated in Figure 5.

```
C;    MATRIX MULTIPLY
      DIMENSION A(36), B(36), C(36)
      ACCEPT 1, I
 1;   FORMAT (I)
      DO 10 M=1, I
      DO 10 N=1, I
      INDX=M+I*(N-1)
      ACCEPT 2, A(INDX)
 2;   FORMAT (E)
10;   CONTINUE
      TYPE 15
15;   FORMAT (/,/,/)
      DO 20 M=1, I
      DO 20 N=1, I
      INDX=M+I*(N-1)
      ACCEPT 2, B(INDX)
      C(INDX)=0
20;   CONTINUE
      DO 30 M=1, I
      DO 30 N=1, I
      DO 30 K=1, I
      IC=N+I*(M-1)
      IA=K+I*(M-1)
      IB=N+I*(K-1)
      C(IC)=C(IC)+A(IA)*B(IB)
30;   CONTINUE
      TYPE 15
      DO 40 M=1, I
      TYPE 21
      DO 40 N=1, I
      INDX=N+I*(M-1)
      TYPE 2, C(INDX)
40;   CONTINUE
21;   FORMAT (/)
      TYPE 15
      END
```

Figure 5 Indexing Statements

In this example the matrices are stored column wise in memory, that is, sequential locations in memory are used as follows:

| Element | Relative Position in Memory (INDX) |
|---|---|
| a11 | 1 |
| a21 | 2 |
| a31 | 3 |
| a41 | 4 |
| a51 | 5 |
| a61 | 6 |
| ------ | ----- |
| a12 | 7 |
| a22 | 8 |
| . | . |
| . | . |
| . | . |
| a56 | 35 |
| a66 | 36 |

If referencing element $a_{56}$ in the array, M=5, N=6 (I would be =6 for a 6 by 6 array.), and INDX=M+I*(N-1)=5+6*5=35. If referencing element $a_{22}$, INDX=2+6*1=8.

# CHAPTER 6

# PROGRAM CONTROL

In this chapter, the FORTRAN statements which have been described as isolated elements are discussed in their proper context -- in program sequences. It is obvious that FORTRAN statements are executed in the order in which they are written unless instructions are given to the contrary. Such instructions are provided by the program control statements, which allow the programmer to alter the sequence, repeat sections, suspend operations, or bring the program to a complete halt.

## PROGRAM TERMINATION

A program arranged so that the last written statement is the final and only stopping place needs no special terminating indication. The end statement automatically determines 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. Frequently, 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.

When a STOP is encountered during program execution at object time, the system signifies that a stop has occurred by typing an exclamation mark (!) on the tape teleprinter.

The stop statement prevents further computation after it has been executed. There is a way, however, to suspend operation for a time and then restart the program manually. This procedure is frequently necessary when the operator must do such tasks as loading and unloading tapes or card decks in the middle of a program. This kind of temporary halt is provided by the following statement:

PAUSE

This brings the program to a halt, but the operator may restart it at any time by pressing the CONTINUE key on the computer console.

## END Statement

END occurs alone on a line and indicates the physical end of the program to the FORTRAN compiler. It must be followed by carriage return and line feed. Every program must contain an END statement.

## BRANCHES AND LOOPS

### The GO TO Statement

There are various ways in which program flow may be directed. As shown schematically in Figure 6, a program may be a straight-line sequence (1), or it may branch to an entirely different sequence (2), return to an earlier point (3), or skip to a later point (4).

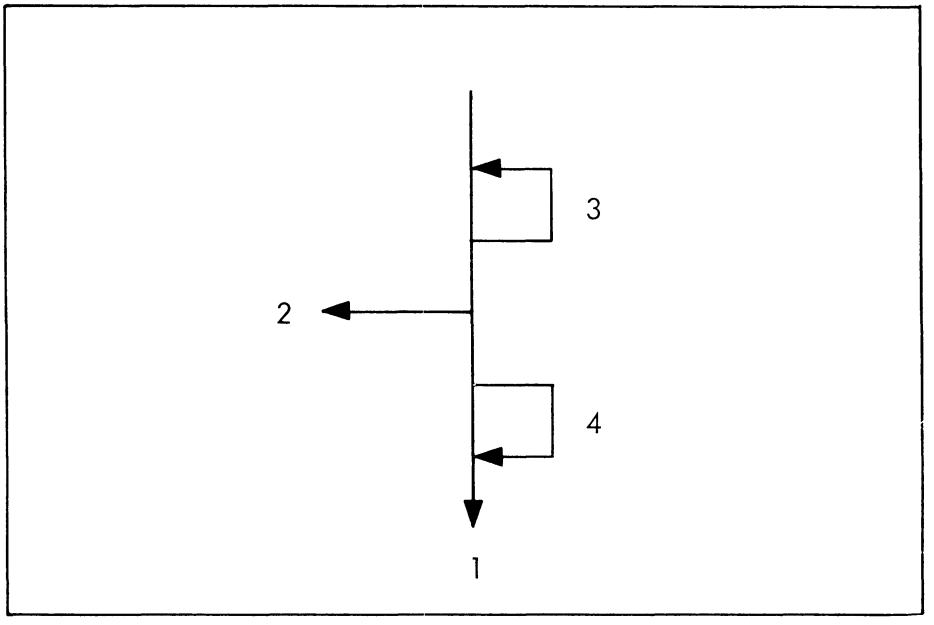



Figure 6 Schematic Representation of Program Branching

All of these branches can be performed in several ways, the simplest of which is by using the statement:

GO TO N

where N is a statement number used in the program. The use of 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 6, No. 3.

## Integer Summation

In the example below, 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 the initialization. The partial sum is set to zero, and the first integer is given the value of one. The loop then proceeds to add the integer value to the partial sum, increment the integer, and repeat the operation.

```
C;    SUM OF FIRST N INTEGERS BY ITERATION
      KSUM=0
      INUM=1
2;    KSUM=INUM+KSUM
      INUM=INUM+1
      GO TO 2
```

Figure 7 Integer Summation

## Limits and Decisions - The IF Statement

The program shown in the preceding example performs the required computation, but there is one flaw: the loop is endless. To get out of the loop, the user must know when to stop the iteration and what to do afterwards.

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

IF (E)K,L,M

where E is any variable name, arithmetic expression, or arithmetic statement, and K, L, and M are statement numbers. The statement is interpreted in this way:

if the value of E is less than 0, GO TO statement K
value of E is equal to 0, GO TO statement L
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 the program branch choices which depend on the results of the evaluation.

```
C;    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
```

Figure 8 Use of IF Statement in Integer Summation Problem

In this example, the initialization and main loop are the same as for the preceding example except that the GO TO 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 if the value of the expression INUM-50 is less than or equal to zero), go to statement 2 and continue the computation. If the value is greater than 50, stop.

A further improvement on the example above can be made if the feature of substatements within an expression is incorporated (refer to pages 13 and 14).

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

Figure 9 IF Statement with Substatement Feature

In this example, the sum is formed by counting down, but the same results are achieved. The initialization is changed so that INUM starts with the value of 50 instead of 0, and the statement INUM=INUM+1 is no longer required.

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:

$$K_N = K_{N-1} + K_{N-2}$$

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

Figure 10 Fibonacci Series

In this program, the initialization includes a dimension statement which reserves space in storage, 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 reaches 100, the calculation stops.

## DO Loops

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

DO n I=K1, K2, K3

Here, n is a statement number, I is a simple integer variable, and K1, K2, and K3 are indexing parameters which provide, in order, the initial value of I, the final (terminating) value of I, and the indexing increment of I. If K3 is omitted from the statement, it is assumed equal to one. Statement n must be a CONTINUE statement.

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

Figure 11 Fibonacci Series Calculation Programmed As a DO Loop

In words, the DO statement says: Do all statements through statement 5 for K=3, when statement 5 is encountered. Perform the following test: If K+1 is less than or equal to 100, set K=K+1 and continue on in the program by executing the first statement after the DO. If the K+1 is greater than 100, the next sequential statement following statement 5 is executed. In this example this is a STOP.

DO loops are commonly used in computations with subscripted variables. In these cases, it is usually necessary to perform loops within loops. Such nesting of loops is permitted in FORTRAN

```
 ;      DO 10 I=1,20
 ;      X(I)=0
 ;      DO 5 J=2,40,2
 ;      X(I)=X(I)+(B(J)-Z(J))**2
5;      CONTINUE
 ;      A(I)=X(I)**2+C(I)
10;     CONTINUE
```

Figure 12 Nested DO Loops

In the previous example, sequential elements in the X array are formed by summing the square of the difference of every second element in the B and Z arrays. Then the A array is formed by summing every element in a C array and the square of every element in the X array. The algebraic expression for the loop is as follows:

$$A_i = X_i^2 - C_i \quad \text{for } i = 1, 2, 3, \ldots 20$$

where

$$X_i = \sum_{j=2}^{40} (b_j - z_j)^2 \quad \text{for } j = 2, 4, 6, \ldots 40$$

The following general rules about DO loops must be observed.

1. 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 13 schematically illustrates permitted and forbidden arrangements.

2. If the user transfers into the range of a DO, the variable I is not initialized as specified in the DO statement. Transferring into the range of a DO is allowed as long as:

   a. Incrementing and testing start with the present value of I.

   b. Control was originally transferred out of the DO other than by completing it.

3. A DO loop must end on a CONTINUE statement.

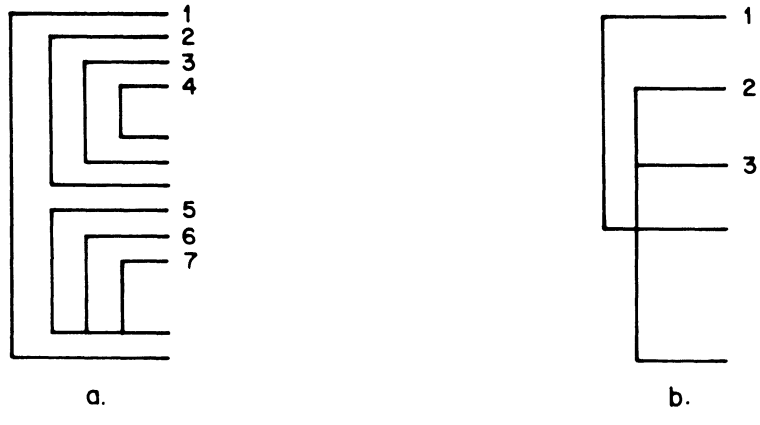



Figure 13 DO Loops

Those in a are permitted; loops 5, 6, and 7 end on the same statement. The arrangements in b are not permitted; loop 3 ends on a statement outside the range of loop 1.

Illegal DO Nesting

```
      ;     DO 10 I=1, 20
      ;     DO 20 J=1,100,2
      ;     SUM=(X(I)-Y(I))**2
     10;    CONTINUE
        1   Z(J)=SUM+A(J)
     20;    CONTINUE
```

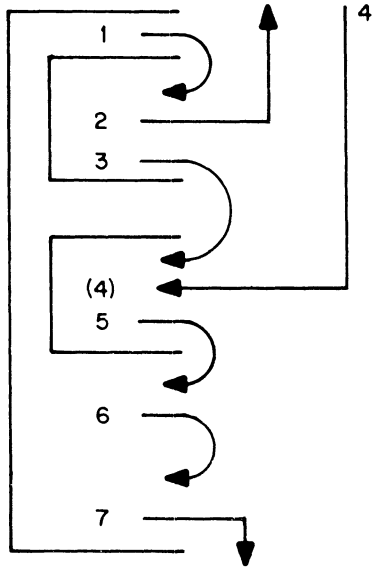



Figure 14 Program Branching in DO Loops

Branches 2, 5, 6, and 7 are permitted; branches 1, 3, and 4 are not.

## The CONTINUE Statement

Since the DO loop may contain alternate courses of action, programmers frequently wish to make the last executable statement of a loop, a test to determine which of the alternatives should be taken next. However, Rule 3 above forbids a DO loop to end on an IF or GO TO; so a special statement is provided which is not an executable statement itself, but provides a termination for such a DO loop. The statement is:

CONTINUE

DO loops must be terminated on a CONTINUE statement.

## Computed GO TO

The GO TO statement described in the section on branches and loops 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 GO TO, which has the following form:

```
GO TO (K1,K2,K3,...Kn),J
```

where the K's are statement numbers, and J is a simple integer variable, which takes on values of 1, 2, 3, ...n according to the results of some previous computation. For example,

```
IVAR = 14*J/2+K
GO TO (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.

When IVAR is less than one or greater than seven, the next sequential statement after the GO TO is executed.

# CHAPTER 7

# INPUT AND OUTPUT

## AVAILABLE DEVICES

So far, we have assumed that all information (programs, data, and subprograms) was in memory, without regard to how it got there. Programs, of course, are read in by a special loader, but the programmer is responsible for the input of data and the otuput of results by including these operations in his program.

For any input-output procedure, several items must be specified:

1. In which direction is the data going? In FORTRAN terms, the data coming in is being read into memory; information going out is being written on whatever medium is specified.

2. Which device is being used? Information can be transferred between core and either of two different input-output devices; each I/O operation must specify which device is involved.

3. Where in core memory is the data coming from or going to? The amount of data and its location in the computer storage must be specified.

4. In what mode is the data represented? In addition to floating- and fixed-point modes for numeric data, there is the Hollerith mode for transferring alphanumeric or text information.

5. What is the arrangement of the data? In FORTRAN terms, the format of incoming or outgoing data is specified.

For every data transfer between core memory and an external device, two statements are required to provide all of the information listed above. The first three items are specified by the input-output statement and the last two items are determined by the FORMAT statement.

PDP-5/8 FORTRAN provides for communication of data to and from a program in the following ways.

1. ASCII Coded Data - (Appendix E)

The Teletype can be used to transfer data to the program either via the keyboard on which the user types the data, or from previously punched paper tape read via the teletype reader.

Data can be output from a program to the Teletype, producing a printed copy with or without the corresponding punched paper tape (depending on whether or not the punch is turned on).

The high-speed reader and punch can also be used for data transfer via punched paper tape. No printed copy is made when output is to the high-speed punch.

2. BINARY Coded Data

DECtape can also be used for data transfer in which case the data is stored as a core image on tape in 128 word blocks of 12-bit binary words.

## INPUT-OUTPUT STATEMENTS

The input-output statements control this transfer of information. As illustrated in Figure 15, I/O statements consist of three basic items of information: the device being accessed and the direction of transfer; the number of the FORMAT statement that controls the arrangement of data; and the list of names of the variables whose values are to be output or changed by new inputs.

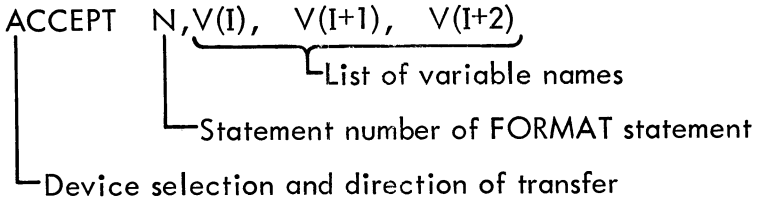



Figure 15 Input-Output Statement

## Device Selection and Direction of Transfer

ACCEPT and TYPE transfer information between the Teletype and the PDP-5/8.

ACCEPT causes information to be accepted into core memory from either the Teletype paper-tape reader, the keyboard, or the photo-electric reader, depending on a switch option selected at run time.

TYPE causes information to be transferred from core memory to the Teletype printer, or the printer and paper tape punch depending on whether the punch is activated or not, or to the high-speed punch depending on a switch option selected at run time.

READ causes information to be read into core memory from DECtape. (See Chapter 8 for details.)

WRITE causes information to be written on DECtape from core memory.

```
;READ  UNIT, BLOCK, FORMAT, LIST
;WRITE UNIT, BLOCK, FORMAT, LIST
```

where UNIT and BLOCK specify the DECtape unit to be used, and the position of information on tape respectively (UNIT and BLOCK can be either constants or variables). The balance of these statements is exactly analogous to the corresponding information in a TYPE statement. FORMAT specifies the format statement, and LIST specifies the variables to be written from, or read into core.

Bit 0 of the SWITCH REGISTER must be set to 1 (up) when compiling or running a program containing READ and WRITE statements and 0 (down) otherwise. Failure to properly set the switch will cause error diagnostics. (See Appendix G.)

## Statement Number of Format Statement

Following the instruction that selects the device and direction of transfer is the statement number of the FORMAT statement that controls the arrangement of the information being transferred.

Example:

```
      ACCEPT 10, A
10;   FORMAT (E)
```

Every I/O statement must have a reference to a FORMAT statement.

## List

The next item of specification in the I/O statement is the names of the variables or array elements that are involved in the transfer. These names are arranged in a sequential list in the order that their values are transferred. There is no restriction on the modes of the variables in the list or the number of names that can appear in a list as long as they are compatible with the corresponding FORMAT statement.

```
      ;   TYPE 10,A,I,B,C(I+K),N(J+L)
   10;    FORMAT (E,I,E,/)
                  .
                  .
                  .
      ;   A=A+(C(J)**2-C(N)**2)
                  .
                  .
                  .
      ;   TYPE 10,A,J,B,C(N)
                  .
                  .
                  .
```

Figure 16 A List Example

If the list contains more names than there are elements in the FORMAT statement, when the elements are exhausted the FORMAT statement is reinitialized, and the first element in the FORMAT statement corresponds to the next name in the list.

For instance in the preceding example when the value of the variable B is typed in the E format, the control character slash (/) causes a carriage-return line-feed to occur. Then the FORMAT statement is reinitialized, and the array element C(I+K) is typed in the E format and the array element N(J+L) in the I format.

Correspondingly, the list does not have to exhaust the elements of a FORMAT statement. If there are fewer names in the list than there are elements in the FORMAT statement, the program completes the I/O operation and proceeds to the next sequential FORTRAN statement. If this next statement is another I/O statement that references a previously unexhausted FORMAT

statement, that FORMAT statement is reinitialized. In other words, FORMAT statements are reinitialized when they are first referenced or when all of their elements are exhausted.

## FORMAT SPECIFICATIONS STATEMENT

As already mentioned in the previous description of input-output statements, the FORMAT statement controls the arrangement and mode of the information being transferred. The values of the names appearing in the list of the I/O statement are transferred in the mode specified by the corresponding element in the FORMAT statement. These controlling elements consist of the characters E, I, slash (/) and quote ("). The set of elements must be enclosed in parentheses and separated by commas.

Example: FORMAT (E,I,/, "HOLLERITH")

### Control Elements E and I

The control elements E and I are used for defining the mode of the data being transferred. When a variable is transferred in the E format, it is stored or output in floating point. If the variable is transferred in the I format, it is stored or output in fixed point. Mode conversion on input or output can be accomplished because the elements in the FORMAT statement define the mode of the data and the mode of the variable is overriden.

Example:

```
    ;  TYPE 10, A
10;  FORMAT (I)
```

The variable A is typed as an integer and the fractional part of A is truncated. For instance, if A has a value of 14.96, only the integer part, 14, would be typed. If A has an absolute value of less than one, zero would be typed.

### Input

Input data words consist of a sign, the decimal value, an exponent value if the data is floating point, and a field terminating character such as space. Any character that is not a number, decimal point, sign, or E can be used to terminate a field except the character rub out. When typing data, any number of spaces or other non-numeric characters can be typed before the sign

or decimal value is typed to make the data sheet more readable. If a mistake is made when typing data words, the last word or partial word can be erased from core memory by typing the character rub out.

These input words can be transferred into core memory from either the Teletype paper-tape reader, the keyboard, the photo-electric reader or DECtape. They can be entered in either fixed- or floating-point modes for integers or decimal fractions. The mode in which they will be stored is controlled by the corresponding element in the FORMAT statement.

Integer Values - Fixed Point - FORMAT (I)

An integer data field consists of a sign (minus or space) and up to four decimal characters. Some examples of integer values are as follows:

| Typed Numbers | Values Accepted |
|---|---|
| −2001 | −2001 |
| −40 | −0040 |
| −0040 | −0040 |
| 16 | 0016 |
| −2047 | −2047 |

Decimal Fraction Values - Floating Point - FORMAT (E)

A floating-point input word consists of a sign*, the data value of up to seven decimal characters, an E if an exponent is to be included, the sign of an exponent, and the exponent which is the power of ten that the data word is multiplied by.

Example:

dddd.dddEnn

The d's represent characters in the data word and n represents the power of ten of the exponent. Either the sign, the decimal point, or the entire exponent part can be omitted. If the sign is left out, the number is assumed to be positive; if the decimal point is left out, it is assumed to appear after the rightmost decimal character. If the exponent is omitted, the power of ten is taken as zero.

*Plus sign can be represented by a plus or space character. Minus is represented by a minus character. If a sign character is absent from the data word, the data is stored as positive.

Examples of floating-point values are as follows:

| Typed Numbers | Values Accepted |
|---|---|
| 16 | $0.16 \times 10^2$ |
| .16E02 | $0.16 \times 10^2$ |
| 1600.E-02 | $0.16 \times 10^2$ |

## Correcting Typing Errors

If a mistake is made when typing data words into a FORTRAN program, the mistake can be corrected by canceling or erasing the data word before typing the terminating character and then retyping the data word that is in error.

To cancel or erase a word, type a rub out character.

When this character is detected during the acceptance of a data word and before the termination character has been transmitted, the data word appearing before the character rub out is erased from memory. Operations on the names in the list do not advance to the next sequential name until a complete data word and the terminating character have been received.

## Output

### Data Word Output - Floating and Fixed Modes - FORMAT (E) and FORMAT (I)

Integer values are always printed as the sign and a maximum of four characters with spaces replacing leading zeros. Floating-point values are printed in a floating-point format which consists of sign, leading zero, decimal point, seven decimal characters, the character E, the sign of the exponent (minus or plus), and an exponent value of two characters.

Examples:

| Integer Values | Output Format |
|---|---|
| -1043 | -1043 |
| -0016 | - 16 |
| +0016 | + 16 |

Floating-point values are printed as per example

SO.ddddddsxx

where

S is the sign, minus sign, or space
d is the seven decimal digits of the data word
s is the sign of the exponent value
xx is the exponent value

| Output Format | Decimal Value |
|---|---|
| -0.8388608E+07 | -8,388,608.0 |
| 0.1192092E-06 | +.0000001192092 |

## OTHER FORMAT CONTROL ELEMENTS

In most cases when data is to be presented it must be labeled and arranged properly on a data sheet. In order that this can be accomplished with FORTRAN, a provision has been made so that text information and spacing can be typed out along with the data words. These features are provided by the special FORMAT control elements quote (") and slash (/).

### Quote (") (Hollerith Output)

When text information is contained as part of a FORMAT and this information is enclosed in quotes, it is output to the specified device as it appears in the statement. This output occurs when a TYPE or WRITE statement references a FORMAT statement containing text and all other elements of that FORMAT statement previous to the text have been used.

```
                .
                .
                .
                TYPE 10
10;             FORMAT (/, "THIS IS HOLLERITH",/)
                .
                .
                .
                TYPE 100, AMIN, AMAX
100;            FORMAT (/, "MINIMUM=",E,/,"MAXIMUM=",E,/)
                .
                .
                .
                TYPE 210
210;            FORMAT (/,/,"                CUMULATIVE DISTRIBUTION",/,/'
                "         INCREMENTS            FREQUENCY",/)
                DO 220 K=1,100
                TYPE 250, K,VALU(L), VALU(K+1), COUNT(K)
220;            CONTINUE
250;            FORMAT (I,"   ",E,"   ",E,"   ",E,/)
                .
                .
                .
```

Figure 17 Examples of Quote and Slash

All legal Teletype characters can be contained within quotes and are output as text (Appendix D).

If a statement continues on another line the Hollerith field must be ended before typing the continuation character (') ; it may be re-opened on the next line. Before text is output, the elements of the FORMAT statement that appear in front of the Hollerith information must have been used.

Example:

```
    1   TYPE 10, VAR,SD
   10;  FORMAT (E,E,E, "VARIANCE AND STANDARD DEVIATION",/)
```

In this example, the text is not typed because one of the E elements was not used.

## Slash (/)

The slash character is used for typing a carriage return and line feed for advancing the paper of the tape teleprinter. A carriage-return line-feed will by typed for every slash that appears in the statement.

Example:

```
;     TYPE 10, A, B
10;   FORMAT(/,/,/,E,/,/,E,/,/)
```

Three carriage-return line-feeds will be typed before the value of A; then two carriage-return line-feeds will be typed before and after the value of B is typed.

The input subroutine of the object time system ignores all non-numeric characters except as data word delimiters so that input data can be labeled and spaced in intermixing the appropriate text and carriage-return line-feeds with the data.

# CHAPTER 8

# FORTRAN WITH DECTAPE OPTION

PDP-8 FORTRAN includes provisions for storage of data on DECtape. In this way FORTRAN programs requiring large amounts of accessible data may be readily written to run on a 4K PDP-5 or PDP-8.

The standard library version of the FORTRAN Compiler is written such that the type of DECtape hardware to be used is irrelevant. The standard library version of the FORTRAN Operating System is written to handle the TC01 DECtape Control with TU55 Tape Transports since these are more common. There is, however, an overlay tape to convert the OP SYS to do the same operations using the 552 Control with 555 DECtape Transports. If this overlay is required the user should so specify when making requests to the library.

## FORTRAN Compiler with DECtape I/O Option

When the FORTRAN Compiler is read into core, it is equipped with a switch option governing the compilation of DECtape I/O statements (READ and WRITE). If the user wishes to compile a program containing DECtape I/O statements, he must set Switch Register bit 0 to 1 (up) before starting any compilation.

The Compiler is designed so that the space occupied by the processing routines for this option becomes part of the input statement buffer if SR bit 0 is set to 0. This means that the DECtape I/O processing routines are destroyed if any compilation is done with bit 0 set to 0 and the Compiler must be reloaded into core to regain the option.

Any program containing DECtape I/O statements must limit the length of the source statements to 100 characters per statement.

## Use of Symbolprint with FORTRAN

Symbolprint destroys a portion of the DECtape Compiler in core. The compiler must be reloaded if it is to be used to accept a symbolic program containing DECtape I/O statements.

## FORTRAN Operating System with DECtape I/O Option

When the FORTRAN Operating System is read into core, it is equipped with a switch option governing the execution of DECtape I/O statements. If the user wishes to run a program containing such statements he must set Switch Register bit 0 to 1 before running his program.

There is a further condition which must be observed, since DECtape I/O requires a considerable amount of additional processing routines. Like the Compiler, the OP SYS destroys its DECtape handling routines if it is used with SR bit 0 set to 0, thus gaining extra space. This requires, however, that the OP SYS must be reloaded into core to regain the option.

## DECtape FORTRAN Statements and Operation

When using DECtape, the FORTRAN Operating System contains a buffer area which is defined as a page of memory reserved to handle transfers to and from a block of DECtape ($128_{10}$ data words).

The DECtape routines transfer one full block from tape to one page of core (and vice versa), therefore, even if the block contains only one data word, the whole block will be read into the OP SYS buffer, overlaying whatever had been there.

To store variables or arrays of data on DECtape requires two steps:

1. From the locations assigned by the OP SYS to the variables or allotted to the arrays by a DIMENSION statement, the programmer must collect the data and put it in the OP SYS buffer. This is done with pseudo WRITE statements (in a DO loop in the case of arrays). (See below.)

2. He must write the buffer onto a block of DECtape. This is done by a physical WRITE statement. (See below) The programmer must be aware of how much data he has in the buffer and write it out on DECtape, before he overflows the buffer. Overflow will cause an error diagnostic.

To retrieve data from DECtape is also a multiple operation.

1. The programmer is responsible for remembering which block contains the data he wishes to retrieve.

2. He must read this block into the OP SYS buffer using a physical READ. (See below.)

3. He must remember in which order he stored the variables or arrays and reference them here in the same order.

4. He must disperse the data from the buffer to the locations assigned by the OP SYS or allotted by a DIMENSION statement. This is done by pseudo READ (in a DO loop for arrays). (See below.)

NOTE: Data which has been brought from tape into the buffer is not yet available for use within the program. It must be dispersed first.

Pseudo WRITE and pseudo READ statements operate between the user program and the buffer only. They are used to collect into the buffer data from within the user program and to disperse into the user program data in the buffer. They have no effect on the physical DECtape.

The user specifies pseudo READ or WRITE by specifying UNIT 0 and BLOCK 0 in the READ or WRITE statement. Specifying any unit other than 0 will indicate that the user wishes to read from DECtape into the buffer or write the buffer out on tape. Pseudo READ and WRITE are of the form;

```
READ 0, 0, FORMAT, LIST
WRITE 0, 0, FORMAT, LIST
```

Physical READ and physical WRITE statements operate between the buffer of the OP SYS and the DECtape. They cause the actual reading or writing of tape. The user specifies physical READ or WRITE by specifying a UNIT number from 1-7 and the number of the actual block on which the data has been stored or is to be stored. They are of the form;

```
READ  UNIT, BLOCK, FORMAT, LIST
WRITE UNIT, BLOCK, FORMAT, LIST
```

It is not necessary to specify a list on a physical READ or WRITE but it is advisable, since it does no harm and is an aid to remembering which variables in which order are on which block.

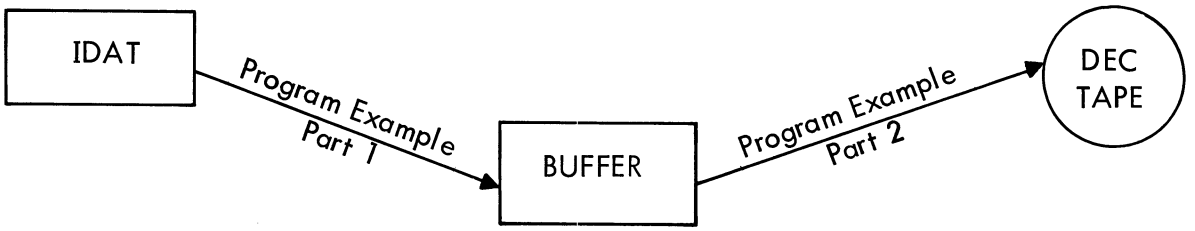
Two examples follow which demonstrate the storage and retrieval of data.

```
                DIMENSION IDAT (128)
                110; FORMAT (I)
Part 1          DO 100 I=1,J
                WRITE 0, 0, 110, IDAT (I)
                100; CONTINUE
                      .
                      .
                      .
Part 2          WRITE MU, MBLK, 110
```

J has been previously defined $\leq$ 128

This DO loop will WRITE J number of elements of IDAT into the buffer

This statement will then store data from buffer to tape




```
C;              : : : : : : : : : : : : :
C;              MUNT=TAPE UNIT TO BE SELECTED
C;              MBLK=BLOCK TO BE WRITTEN ON
C;              IBLK=INITIAL BLOCK TO BE SEARCHED FOR (TO REWIND TAPE)
C;              VALUES FOR J AND K AS DETERMINED BY USER MUST BE LESS THAN 200
C;              : : : : : : : : : : : : :

DIMENSION IBF1 (200), IBF2 (200)
90;FORMAT (I)
199;ACCEPT 90, J,K, MUNT, MBLK, IBLK
DO 10 I=1, J
ACCEPT 90, IBFI (I)
10;CONTINUE

C;              : : : : : : : : : : : : :
C;              DATA HAS BEEN ACCEPTED NOW WE READ BLOCK '0'
C;              THIS INITIALIZES AND REWINDS THE TAPE
C;              A GOOD IDEA TO DO THIS BUT NOT ESSENTIAL
C;              : : : : : : : : : : : : :

READ MUNT, IBLK, 90
```

```
C;          : : : : : : : : : : : : :
C;          TIME TO TAKE ACCEPTED DATA AND WRITE IT INTO
C;          THE BUFFER, NOT ONTO THE ACTUAL TAPE.  SEE PART ONE OF DIAGRAM.
C;          BUFFER IS WRITTEN BY SELECTING UNIT  0
C;          DO LOOP NEEDED SINCE DATA IS IN AN ARRAY
C;          : : : : : : : : : : : : :

DO 31 I=1,J
WRITE0,90,IBF1 (I)
31;CONTINUE

C;          : : : : : : : : : : : : :
C;          TAPE IS NOW PHYSICALLY WRITTEN BY SELECTING
C;          A LOGICAL UNIT, OTHER THAN ZERO.  SEE PART TWO OF DIAGRAM.
C;          : : : : : : : : : : : : :

WRITE MUNT, MNLK,90

C;          : : : : : : : : : : : : :
C;          NOW READ BLOCK BACK INTO BUFFER
C;          : : : : : : : : : : : : :

READ MUNT, MBLK, 90

C;          : : : : : : : : : : : : :
C;          READ BUFFER AND STORE IT IN ANOTHER ARRAY
C;          DO LOOP NEEDED BECAUSE OF ARRAY
C;          : : : : : : : : : : : : :

DO 32 I=1,K
READ 0, 0, 90, IBF2(I)
32;CONTINUE
C;          * * * * * * * * * * * * *
C;
C;          ALL I/O DONE HALT.
C;
C;          * * * * * * * * * * * * *

DO 13 I=1,K
TYPE 91,IBF2 (I)
91;FORMAT (/,I)
13;CONTINUE

PAUSE

C;          : : : : : : : : : : : : :
C;          ALL DECTAPE I/O DONE,TYPE OUT CONTENTS OF ARRAY READ FROM DECTAPE
C;          : : : : : : : : : : : : :

GO TO 199

END
```

# CHAPTER 9

# PDP-5/8 FORTRAN SYMBOLPRINT

FORTRAN Symbolprint is a useful aid in finding where a FORTRAN program is stored in interpretive memory, the exact memory locations assigned to each FORTRAN variable, and the amount and location of interpretive core memory that is not used by a FORTRAN program.

Symbolprint loads over the FORTRAN Compiler and starts at address 600. The following is a typical example of the typeout.

| List of Variable Names | Assigned Location |
|---|---|
| . | . |
| . | . |
| . | . |
| HW | 7546 |
| TB | 7543 |
| G | 7540 |
| TF | 7535 |
| MC | 7534 |
| DSR | 7531 |
| C1 | 7526 |
| C2 | 7515 |
| C3 | 7504 |
| C4 | 7470 |
| . | . |
| . | . |
| . | . |
| 6312 | 7241 |

Note that a single word only has been assigned for the fixed point variable MC.

The last two octal constants typed indicate respectively the highest address used by the program in interpretive memory and the lowest address used for data. Therefore the area of core between these two addresses is available. In the example there are

$$7241 - 6312 - 1 = 726$$

octal locations free.

A machine language subroutine may occupy this available space. Use the FORTRAN PAUSE statement to link the FORTRAN program to the subroutine.

If PAUSE is followed by a number (decimal), FORTRAN compiles (in effect) JMS to that address. For example:

```
;PAUSE 3328
```

effects JMS 6400. Location 6400 should contain coding such as the following:

```
SUBR, 0
      :
      :
      JMP I SUBR
```

constituting the desired machine language program.

# APPENDIX A

# OPERATING PROCEDURES
# FOR RIM AND BIN PAPER TAPE LOADERS

## READ-IN-MODE LOADER (RIM)

1. The RIM Loader is a minimum-length, basic paper tape loader for the PDP-8. It is initially stored in memory by way of the CONTROL console switches. Once stored, it is considered to be a permenent occupant of locations 7756 through 7777 (absolute octal addresses) and care should be taken to keep it from being destroyed.

2. A paper tape to be read in by the RIM Loader must be in RIM format:

| Tape Channel | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 7 | 6 | 5 | 4 | S | 3 | 2 | 1 | |
| 1 | 0 | 0 | 0 | 0 | . | 0 | 0 | 0 | Leader/Trailer code |
| 0 | 1 | | | A1 | . | | | A2 | Absolute address to |
| 0 | 0 | | | A3 | . | | | A4 | contain next 4 digits |
| 0 | 0 | | | X1 | . | | | X2 | Contents of previous |
| 0 | 0 | | | X3 | . | | | X4 | 4 digit address |
| 0 | 1 | | | A3 | . | | | A4 | |
| 0 | 0 | | | A3 | . | | | A4 | Address |
| 0 | 0 | | | X1 | . | | | X2 | |
| 0 | 0 | | | X3 | . | | | X4 | Contents |
| | | | | (ETC.) | | | | | (ETC.) |
| 1 | 0 | 0 | 0 | 0 | . | 0 | 0 | | Leader/Trailer code |

3. The complete PDP-8 RIM Loader for the ASR-33 (SA=7756) is as follows:

| Abs. Addr. | Octal Contents | Tag | Instruction i z | Comments |
|---|---|---|---|---|
| 7756, | 6032 | BEG, | KCC | /clear AC and flag |
| 7757, | 6031 | | KSF | /skip if flag=1 |
| 7760 | 5357 | | JMP .-1 | /looking for char |
| 7761 | 6036 | | KRB | /read buffer |
| 7762, | 7106 | | CLL RTL | |
| 7763, | 7006 | | RTL | /ch 8 in AC0 |
| 7764, | 7510 | | SPA | /checking for leader |
| 7765, | 5357 | | JMP BEG +1 | /found leader |
| 7766, | 7006 | | RTL | /OK, ch7 in link |
| 7767, | 6031 | | KSF | |
| 7770, | 5367 | | JMP .-1 | |
| 7771, | 6034 | | KRS | /read, do not clear |
| 7772, | 7420 | | SNL | /checking for address |
| 7773, | 3776 | | DCA I TEMP | /store contents |
| 7774, | 3376 | | DCA TEMP | /store address |
| 7775, | 5356 | | JMP BEG | /next word |
| 7776, | 0 | TEMP, | 0 | /temp storage |
| 7777, | 5301 | | 0 | /jump to start of bin loader |

4. Placing the RIM Loader in memory by way of the CONTROL console switches is accomplished as follows:

a. Set 7756 in the switch register (SR)

b. Press LOAD ADDRESS

c. Set the first instruction in the SR (6032)

d. Press DEPOSIT

e. Set the next instruction in the SR

f. Press DEPOSIT

g. Repeat steps e and f until all 16 instructions have been deposited.

5. To load a tape in RIM format, place the tape in the reader, set the SR to 7756, press LOAD ADDRESS, press START, and start reader.

6. The complete PDP-8 RIM Loader for the high-speed reader 750 (SA=7755) is as follows:

| Abs. Addr. | Octal Contents | | Symbolic | |
|---|---|---|---|---|
| 7756 | 6014 | BEG, | RFC | /clear flag and fetch char. into buffer |
| 7757 | 6011 | | RSF | /skip when flag=1 |
| 7760 | 5357 | | JMP .-1 | |
| 7761 | 6016 | | RRB RFC | /read buffer into AC, get next char. into buffer |
| 7762 | 7106 | | CLL RTL | /rotate channel 8 into |
| 7763 | 7006 | | RTL | /AC bit 0 |
| 7764 | 7510 | | SPA | /is it leader |
| 7765 | 5374 | | JMP TEMP-2 | /yes clear AC |
| 7766 | 7006 | | RTL | /NO rotate channel 7 to LINK |
| 7767 | 6011 | | RSF | |
| 7770 | 5367 | | JMP .-1 | |
| 7771 | 6016 | | RRB RFC | |
| 7772 | 7420 | | SNL | /link set=origin |
| 7773 | 3776 | | DCA I TEMP | /store data |
| 7774 | 3376 | | DCA TEMP | /store address |
| 7775 | 5357 | | JMP BEG +1 | /next word |
| 7776 | 0000 | TEMP, | 0 | /temporary storage |
| 7777 | 5301 | | 0 | /JMP to start of BIN loader |

## BINARY LOADER (BIN)

1. The BIN Loader is used to read in the machine language tapes. A binary-formatted tape is about one half the length of a comparable RIM formatted tape. It can, therefore, be read in about twice as fast as a RIM tape and is, for this reason, the more desirable format to use with the 10 cps ASR-33 Reader.

2. To load a tape in BIN format, place the tape in the reader, set the SR to 7777; press LOAD ADDRESS, press START, and start reader.

3. After a BIN has been read in, one of the two following conditions exist:

   a. No checksum error: halt with AC=0

b. Checksum error: halt with AC=(computer checksum) - (tape checksum). If a checksum error exists, a character was misread from the binary tape or is mispunched on the tape. The operator should reload the binary tape; and if the same checksum error appears in the AC indicator after readin, the binary tape was mispunched and a new copy should be obtained. If a different checksum error appears after readin, the appropriate maintenance procedure should be followed.

# APPENDIX B

## PREPARATION OF SYMBOLIC (SOURCE) TAPE

1. Symbolic tape preparation using Symbolic Tape Editor. (It is to the user's benefit to use the Editor to put his source program on tape since using the Editor minimizes the chance of extraneous characters getting on the tape and also facilitates deletion and correction of statements.)

   a. Load Symbolic Tape Editor using Binary Loader.

   b. Start Editor at 176 (Load Address, Start).

   c. Type A↓ and type the symbolic source program.

   d. Hold the CTRL key and press the FORM key; the bell will sound.

   e. Type P↓, followed by F↓ when punching stops.

   NOTE: For complete explanation of Editor, see Symbolic Tape Editor Manual.

2. Symbolic tape preparation off-line using Teletype only.

   a. Turn power on in computer (key on left in PDP-8, switch on right in PDP-5).

   b. Turn Teletype LINE-OFF-LOCAL knob to LOCAL to disconnect Teletype from computer.

   c. Press PUNCH ON button on the Teletype.

   d. Generate leader.*

   e. Type the source program.

   f. Generate trailer.*

*To generate leader/trailer (200 code), hold the CTRL and SHIFT keys with the left hand, depress the REPT key and then the P with the right hand. Release in reverse order or a P will be punched on the tape.

3. Manual symbolic tape editing using the ASR-33.

a. An incorrect character might be typed while preparing the symbolic tape. Use the following procedures to correct the tape: (the error is detected N characters after typing the incorrect character) press the PUNCH B.SP. button N+1 times, press rubout N+1 times, and continue.

b. Characters, words, or statements can be inserted or deleted after the entire symbolic tape has been prepared. Use the following procedures to accomplish such changes.

(1) Insertions - Duplicate the tape up to the point at which it is desired to make an insertion (by turning the punch on, placing the tape in the reader, starting the reader, and stopping the reader with the READER switch using the printout as a guide). Next, type the insertion. Continue by pressing the READER switch to start and duplicate the remainder of the tape.

(2) Deletions - Duplicate the tape up to the point at which it is desired to make a deletion (see Insertions). Next, turn the punch off; start the reader; and using the printout of the information to be deleted as a guide, stop the reader. Continue by turning the punch on and starting the reader to duplicate the remainder of the tape.

# APPENDIX C

# FORTRAN OPERATING PROCEDURES

## COMPILER

1. Load the Compiler with the Binary Loader (see Appendix A).

2. Put the starting address of the Compiler (0200 octal) into the switch register and press LOAD ADDRESS.

3. Set I/O switches. (Conditional) See I/O Control.

4. Place the source language tape in the selected reader and turn on the reader and punch.

5. Press START.

6. At the end of compilation, the computer will halt with the run light off.

7. To compile additional programs, place the source language tape in the appropriate reader, turn the reader and punch on, and press CONTINUE. I/O selections cannot be changed without reloading compiler.

## SYMBOLPRINT

Symbolprint is run immediately after compiling a program and before compiling another or loading the Operating System. (It cannot be run if the Operating System has been loaded into core.)

Use of Symbolprint destroys the portion of the Compiler which processes DECtape READ and WRITE statements. The Compiler must therefore be reloaded if it is to compile a source program containing such statements.

a. Load Symbolprint with the Binary Loader.

b. Set 0600 in the switch register.

c. Press LOAD ADDRESS and START, see Chapter 8.

## OPERATING SYSTEM
## (OBJECT TIME SYSTEM)

1. To load a compiled program:

   a. Load the FORTRAN Operating System using the Binary Loader.

   b. Place the Compiler output (interpretive code object tape) in the Teletype or photo-electric reader. Turn on the reader, making sure ASR-33 is ON LINE.

   c. Load the SWITCH REGISTER with 0200, and press LOAD ADDRESS.

   d. Set switch register bit 1 to read in compiled tape from photo-electric reader or Teletype (as shown below).

   e. Press START. The Operating System reads the compiler output tape.

   f. The Operating System halts at the end of loading. The loading is correct if the checksum difference which appears in the AC equals 0.

   g. Turn off the reader and remove the compiler output tape from it.

2. To execute a program after loading

   a. Set SWITCH REGISTER bits 0, 1, and 2 (as shown below).

   b. If input is to be from paper tape, put the data tape in the appropriate reader and turn the reader on. If output is to be punched, turn punch on.

   c. Press CONTINUE.

   NOTE: Once loaded, a program can be executed any number of times.

   NOTE: The Operating System need not be reloaded to run more than one program in succession. To do so start at step b of section 1.

3. To Re-execute a Loaded Program

   a. Set the SWITCH REGISTER to 0201; press the LOAD ADDRESS key.

   b. Set the SWITCH REGISTER for the I/O (as shown below).

   c. Press START.

The FORTRAN Operating System checks each of its internal stacks after the execution of each interpretive instruction to insure that there is neither stack overflow nor stack underflow. If a FORTRAN program has been debugged and is known to operate correctly, this test may be NOPed by changing C(0404) to 7000 (NOP). This will speed up the execution of the program by a factor of about 2.

## I/O CONTROL

The selection of I/O devices for both compiler and OP SYS is controlled by setting the switches as shown below:

| Bit Number | Switch Position | Meaning |
|---|---|---|
| 0 | 0 | The program contains only paper tape I/O statements. |
| | 1 | The program contains DECtape I/O statements. |
| 1 | 0 | Compiler: Use the Teletype reader for input of source tape.<br>OP SYS: Use the Teletype reader for loading the object program and the keyboard for ACCEPT statements. |
| | 1 | Use the high speed reader. |
| 2 | 0 | Compiler: Use the Teletype printer/punch for compiler output (interpretive code) tape and error diagnostics.<br>OP SYS: Use the Teletype printer/punch for TYPE statements. |
| | 1 | Use the high speed punch (error diagnostics still come on Teletype). |

# APPENDIX D

# FORMAT OF COMPILER OUTPUT

## INTERPRETIVE CODE

1. 200 codes (leader, ignored by loader)

2. Data blocks, each as follows:

    a. origin (2 frames, first has bit 7 punched)

    b. data words (2 frames/word)

3. Forward referencing table - first frame has bits 7 and 8 (only) punched

4. Checksum

    a. first has bits 6, 7, and 8 (only) punched

    b. next two frames are checksum

5. 200 codes (trailer, ignored by loader which stops after checksum)

6. Error comments, if any, in ASCII

# APPENDIX E

## ASR-33 8-BIT CHARACTER SET

| Character | 8-Bit Code (in Octal) | Character | 8-Bit Code (in Octal) |
|---|---|---|---|
| A | 301 | ! | 241 |
| B | 302 | " | 242 |
| C | 303 | # | 243 |
| D | 304 | $ | 244 |
| E | 305 | % | 245 |
| F | 306 | & | 246 |
| G | 307 | ' | 247 |
| H | 310 | ( | 250 |
| I | 311 | ) | 251 |
| J | 312 | * | 252 |
| K | 313 | + | 253 |
| L | 314 | , | 254 |
| M | 315 | - | 255 |
| N | 316 | . | 256 |
| O | 317 | / | 257 |
| P | 320 | : | 272 |
| Q | 321 | ; | 273 |
| R | 322 | < | 274 |
| S | 323 | = | 275 |
| T | 324 | > | 276 |
| U | 325 | ? | 277 |
| V | 326 | @ | 300 |
| W | 327 | [ | 333 |
| X | 330 | / | 334 |
| Y | 331 | ] | 335 |
| Z | 332 | ↑ | 336 |
| 0 | 260 | → | 337 |
| 1 | 261 | Leader/Trailer | 200* |
| 2 | 262 | Line-Feed | 212* |
| 3 | 263 | Carriage-Return | 215 |
| 4 | 264 | Space | 240 |
| 5 | 265 | Rub-out | 377* |
| 6 | 266 | Blank | 000* |
| 7 | 267 | | |
| 8 | 270 | *Ignored by the operating system | |
| 9 | 271 | | |

# APPENDIX F

## PDP-8 FORTRAN SOURCE PROGRAM RESTRICTIONS

The following limits are imposed upon all FORTRAN source programs for the PDP-8:

a. Not more than 1000 data cells. This includes all dimensioned variables, user-defined variables, constants, and all constants generated by the usage of a DO loop.

b. Not more than 20 undefined forward references to unique statement numbers per program. An undefined forward reference is a reference to any statement label that has not previously occurred in the program. Multiple references to the same undefined statement numbers are considered as one reference.

c. Not more than 64 different variable names per program.

d. Not more than 128 characters per input statement. (When using the DECtape Compiler, the input statement size is reduced to 100 characters.)

e. Not more than 40 numbered statements per program.

### PDP-8 COMPILER AND OPERATING SYSTEM CORE MAP

The Compiler occupies the following core locations:

| | | |
|---|---|---|
| 3 | - 7600 | Compiler itself plus tables |
| 7200 | - 7600 | Compiler tables (undefined forward reference table, etc.) |

The Operating System occupies locations:

| | |
|---|---|
| 0 - 5200 | Operating System for paper tape I/O |
| 0 - 6000 | Operating System for DECtape I/O |

Locations 5200 - 7576 are available for the user's program when using paper tape input/output or locations 6000 - 7576 when using DECtape.

NOTE: The 1000 data word restriction applies.

# APPENDIX G

# DIAGNOSTICS

Diagnostic procedures are provided in the compiler to assist the programmer in program compilation. When the compiler detects errors in a FORTRAN source program, it prints out error messages on the on-line tape-teleprinter. These messages indicate the source of the error and direct the programmer's efforts to correct the error.

To speed up the compiler process, the compiler prints out only an error code. The programmer then looks up the error message corresponding to the code in table A-1 and takes the appropriate corrective measures.

## DYNAMIC ERROR CORRECTION

A user may choose to compile in either of two modes: the normal mode or the dynamic correction mode. The latter allows the user to correct a statement, which the compiler has determined contains a source-language error, by reentering the offending line via the tape teleprinter without having to physically correct the symbolic tape and recompile. This feature is not implemented in the high-speed reader version of the compiler since the higher speed of the device makes recompilation easy.

To choose the dynamic correction mode:

1. Load the starting address of the compiler (0200) in the console switches and press LOAD ADDRESS.

2. Set SR bit 11 to 1, press START (can only be used with low speed paper tape I/O).

If an error is detected, the diagnostic prints out in the normal fashion and the computer halts.

To correct the statement:

1. Turn READER switch to FREE.

2. With the READER switch still in the FREE position, press CONTINUE.

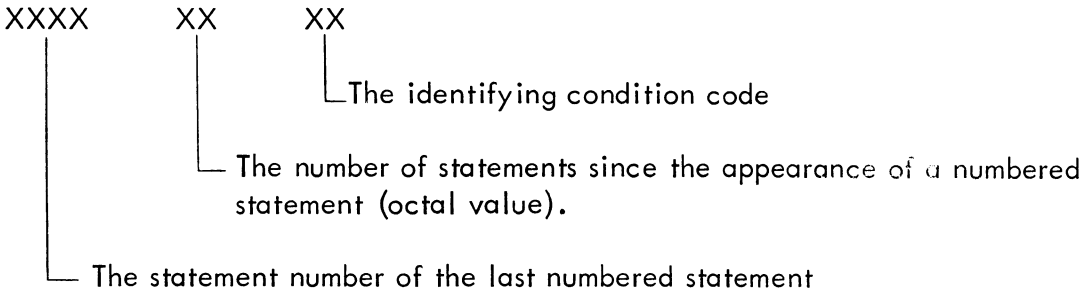
3. Type the new line in its entirety,* obeying all rules for the source language and terminating the statement with a carriage-return line-feed.

4. Turn reader on and compilation will continue.

To leave the dynamic correction mode, restart the compiler in the normal fashion.

## COMPILE TIME DIAGNOSTICS

### Format of Diagnostics




Example:

```
        .
        .
        .
10;     A=I (J+1)
        B=A*(B+SINF(THTA)
        .
        .
        .
```

During compilation of the above statements the following error code would be printed,

10 11 11

*If the statement was numbered, do not reenter the statement number unless it was in error.

indicating that a statement which occurs eleven statements octal (eight decimal) after the appearance of statement 10 is in error. The message corresponding to code 11 shows that the number of left and right parentheses is the statement is not equal. The statement is examined and corrected; then compilation is resumed.

TABLE A-1

| Diagnostic Code | Conditions |
|---|---|
| 00 | Fixed- and floating-point modes have been mixed in an expression. |
| 01 | Two operators appear adjacent to each other (i.e., a variable has been left out of an expression) e.g., A=C + * D. |
| 02 | Compiler error - Reload Compiler and repeat compilation process. Contact Software Quality Control, PDP-8 Division if this reoccurs. |
| 03 | A comma has been used illegally in an arithmetic statement. |
| 04 | Too many operators appear in a single statement, |
| 05 | A function argument is in fixed mode, e.g., SINF(INC). |
| 06 | A variable subscript is in floating point mode. This could also indicate that an operator is missing, e.g., A+B(C+1.) for A=B*(C+1.). |
| 07 | More than 64 (decimal) different variable names have been used in the program. |
| 10 | Program too large - program and data requirements have overlapped. |
| 11 | There is an unequal number of right and left parentheses in a statement. |
| 12 | An illegal character was detected and ignored. |
| 13 | The compiler is unable to recognize or process this statement due to some error in its format. |
| 14 | Program too large; program and data requirements have overlapped. |
| 15 | A subscripted variable is defined before the appearance of a dimension statement, or a subscripted variable does not appear in a dimension statement. It might also indicate that an operator is missing in a fixed-mode expression, e.g., A=I(J-K) for A=I*(J-K). |

TABLE A-1 (continued)

| Diagnostic Code | Conditions |
|---|---|
| 16 | Statement too long; more than 128 characters have been counted not including spaces except in format statements where all legal TTY characters are counted. |
| 17 | A floating-point operand should have been fixed-point, e.g., DO 10 I=1, 7.3. |
| 20 | A statement number that has been referenced does not appear in the program. See the paragraph on the next page. |
| 21 | There are more than 40 numbered statements in the source program. |
| 22 | A statement cannot be compiled because it has too many incompleted operations, e.g.,C=A+(C+(D+(E+.... |
| 23 | Too many statements have been referenced before they are defined. |
| 24 | Attempt to compile a READ or WRITE progam statement after starting program without switch 0 set. |

If a statement number is referenced but does not appear in the source program, the diagnostic code will be printed as follows:

xxxx 77 20

where the number usually reserved for the last numbered statement (xxxx) is replaced by the missing statement number.

e.g., GO TO 100

The diagnostic would appear as follows where statement 100 is never defined.

100 77 20

## OPERATING SYSTEM DIAGNOSTICS

Not all errors are detected by the compiler. Some errors can only be detected by the object time system. Also, there are some conditions which indicate errors on the part of the compiler and/or object system. When such an error occurs during running of a program, the computer types out an error message containing an error number. The computer then halts. If the CONTINUE toggle is pressed, the computer takes the action listed in the following table.

TABLE A-2

| Error Number | Possible Cause | Action Taken |
|---|---|---|
| 11 | Attempt to divide by zero | Quotient set to plus or minus largest number representable in computer; then continue executing instructions. |
| 12 | Floating point exponent on input greater than plus or minus 2047 | System executes next instruction. |
| 13 | Illegal operation code (either compiler error, or data stored over program, or transfer to data section) | System executes next instruction. |
| 14 | Transfer to core location zero or one | No recovery possible. |
| 15 | Non-format statement used for a format | System executes next instruction. |
| 16 | Illegal format statement constituent | System examines next constituent. |
| 17 | Attempt to fix large floating point number | |
| 20 | Attempt to take square root of a negative number | System takes square root of absolute value. |
| 21 | Attempt to raise a negative number to a power | System raises absolute value to the power specified. |
| 22 | Attempt to find the logarithm of zero or a negative number | System attempts to find logarithm of absolute value. Note that log (absolute value (0)) still gives an error halt. |
| 31 | Select error. | The operating system halts with the called unit in bits 0-2 of the AC (0-3 if using 552/555). Recovery is possible by correcting the logical Unit and pressing continue. |
| 32 | Physical Tape Error. | The program halts with the error status in the AC. (The configuration of bits is dependent upon the tape control being used.) |
| 33 | DECtape buffer exceeded | |
| 34 | DECtape control switch set incorrectly | |

TABLE A-2 (continued)

| Error Number | Possible Cause | Action Taken |
|---|---|---|
| 76 | One of the stacks used by the system has underflowed. (i.e., more data has been requested than was placed on the stack) | No recovery possible. Since this may be a system error, communicate program and circumstances to DEC. |
| 77 | One of the stacks has overflowed (i.e., more data placed on it than there is storage in the machine.) | Same as Error 76 |


EQUIPMENT
CORPORATION
MAYNARD, MASSACHUSETTS

PRINTED IN U.S.A.