DEC-6-0-TP-FII-LM-FP-ACTOO

# FORTRAN II LANGUAGE PROGRAMMING MANUAL

# PDP-6

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

DEC-6-O-TP-FII-LM-FP-ACTOO

# PDP-6 FORTRAN II LANGUAGE PROGRAMMING MANUAL

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

# FOREWORD

This is a reference manual describing the specific statements and capabilities of PDP-6 FORTRAN II. Familiarity with the basic concepts of FORTRAN programming on the part of the reader is assumed.

# CONTENTS



## APPENDIXES

# CONTENTS (continued)

## APPENDIXES (continued)



## ILLUSTRATIONS



## TABLES

# CHAPTER 1

# INTRODUCTION

## BASIC ELEMENTS

### The Character Set

The symbols or characters which are meaningful in PDP-6 FORTRAN are:

| | | | |
|---|---|---|---|
| Blank | " | H | W |
| . | # | I | X |
| ) | % | J | Y |
| ] | = | K | Z |
| < | @ | L | 0 |
| ! | : | M | 1 |
| + | ' | N | 2 |
| $ | \ | O | 3 |
| * | ; | P | 4 |
| [ | A | Q | 5 |
| > | B | R | 6 |
| ↑ | C | S | 7 |
| - | D | T | 8 |
| / | E | U | 9 |
| , | F | V | ← |
| ( | G | | |

Blanks, with two exceptions, are ignored and may be used as desired to make the program neater in appearance and more readable. For example, READTAPE and READ TAPE are equivalent. The exceptions are FORMAT statements and ASCII constants.

### FORTRAN Words

FORTRAN words fall into seven categories: constants, variables, subscripted variables, function names, operation symbols, statement labels, and commands. Of these, the first four are most similar to the "words" used in writing ordinary formulas. For example, as in ordinary mathematics:

$$y = 2x + 3\cos(z)$$

where x, y, and z are variables, 2 and 3 are constants, = and + are operators, and cos is a function name. Statement labels are integers used to "name" statements within a program. Commands are special FORTRAN words, e.g., DIMENSION, GO TO, ASSIGN, IF, etc. Each command word is an integral part of its associated type of statement. Therefore, commands will be discussed along with their statement types.

### Number Representation

There are two types of numbers represented in FORTRAN: integers and real*. Both types of numbers may assume positive, negative, and zero values. Integers may, of course, assume only integral values and always are represented exactly in the machine provided their magnitude is not greater than the largest integer containable in the machine registers. In the PDP-6, this largest integer is $2^{35}-1$. A real number x may take on values in the range

$$10^{-38} \leq |x| \leq 10^{+38} \text{ and } x = 0$$

---

*This use of the term real should not be confused with the mathematical usage. In FORTRAN, real applies only in the limited sense described above (formerly referred to as floating point).

However, though a number is within the range, its representation in the machine may only be approximate. The conversion to machine representation is accurate to approximately eight significant digits (27 binary bits).
Except for zero, two numbers of different types never have the same representation in the machine. For example, if 3 is the integer three, and 3.0 is the type real three, the representation of 3 is not at all similar to the representation of 3.0.

## Constants

### Numerical

In FORTRAN II, a number may be explicitly named by writing a constant. However, there are two forms for naming constants: integer and real.

Integer Constants - Any number representable by a string of digits, from the set 0, 1, 2, .....9, written without a decimal point or exponents

$$\pm X_1 X_2 \ldots\ldots X_n$$

Where $X_i$ is any digit. The plus sign is optional; if no sign appears, plus is assumed.

The range of an integer constant I acceptable to PDP-6 FORTRAN is:

$$-2^{35}+1 \leq I \leq +2^{35}-1$$

Note: $2^{35} = 34,359,738,368$.

Some examples of integers are:

| | | | |
|---|---|---|---|
| 9 | -17 | 8192 | +131701 |

Real Constants - Any number representable by a string of digits, from the set 0, 1, 2,.....9, written with either a decimal point, exponent or both. The general form is:

$$\pm X_1 X_2 \ldots X_n . Y_1 Y_2 \ldots Y_n \; E \pm Z_1 Z_2$$

where $X_i$, $Y_i$, and $Z_i$ are digits. The plus signs are optional; if no signs are given, plus is assumed. Either the string $X_1 \ldots\ldots X_n$ or the string $Y_1 \ldots\ldots Y_n$ may be omitted if their value is zero. If $Z_1$ and $Z_2$ are both zero, $E \pm Z_1 Z_2$ may be omitted. (However, the number must contain either a decimal point or an E.)

The range of real constants acceptable to PDP-6 FORTRAN II is:

$$10^{-38} \text{ to } 10^{+38} \text{ and zero.}$$

Some examples of real numbers are:

| | |
|---|---|
| 6.023E23 | -6.023E23 |
| 1.66E - 16 | +72E12 |
| -.0056 | 4.2 |

### Boolean Constants

Any number representable by a string of digits from the set 0, 1, 2,...7.

$$\pm X_1 X_2 \ldots X_n$$

where $X_i$ is any octal digit. The plus sign is optional; if no sign appears, plus is assumed. A minus sign implies the 1's complement of the number. The range of the Boolean constant B is

$$0 \leq B \leq 2^{36}-1$$

Examples:

| | |
|---|---|
| 377777 | 40404040 |
| -0 | +15 |

### ASCII Constants

ASCII constants are used to convey textual information. They have the form:

$$i\,H\,c_1\,c_2 \ldots\ldots\ldots c_i$$

where $0 < i \leq 5$. The c's are any of the characters from the character set. Note that there must be exactly i characters including blanks.

Examples:

```
A = 5HPDP-6
S = 4H YES
```

In the example above the variable on the left must be of a real type.

## Variables

### Variable Names

A variable name is composed of from one to six characters according to the rules:

1. The only characters which may be used in a variable name are A through Z and 0 through 9. (Blanks are ignored.)
2. The first character must be alphabetic.

Some examples of acceptable variable names are K, P51, and EPSILO.

Some incorrect variable names are 6MERGE (first character not alphabetic), G1.5 (illegal character included), and EPSILON (too many characters).

### Type Rules for Variables

There are two types of entities designated by variable names: integers and reals. Corresponding to the two types of entities there are two types of variable names identified as follows:

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

Some integer variable names are INDEX, KDATA, K359, and MSIX. These are real variable names: XZERO, COUNT, and 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 means of a symbol denoting the array and subscript 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-dimentional array m is designated by

$$m_{i_1, i_2, i_3, \ldots\ldots\ldots\ldots i_n}.$$

In FORTRAN II, array elements are similarly identified. The array is provided with a name, subject to the same formation and type rules as the names of variables. All the elements in the array have the same type. The subscripts which identify an element of the array are enclosed in parentheses and separated by commas. The two elements, $v_6$ and $b_{4,10}$ would have the following notation: V(6) and B(4,10). Subscripts may be quite diverse in form; in fact, a subscript may be any acceptable FORTRAN arithmetic expression

(see Chapter 2) as long as it is integer-valued and less than $2^{18}$ (real quantities are not allowed).

```
a. X(3,3)
b. C(I+1, J+1)
c. N(I(I), J(1), K(2))
d. Y(J/3,(K-4) +12)
```

Figure 1 Example of Subscripts

Note that the subscripts in Figure 1, example c, are themselves subscripted. Subscripting may be carried to any level. Each subscripted subscript, i.e., I(1), J(1), and K(2) in Figure 1, is itself treated as a subscripted variable.

### Function Names

Function names 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 operations such as these are provided by the FORTRAN system and are called library functions. A detailed discussion of functions will be found in Chapter 5.

### Operation Symbols

The operation symbols may be any one of the following: ** or ↑ , / , + and -. These denote arithmetic or Boolean operations depending on the type of statement in which they occur. The rules for the use and binding strength of these operators are given in Chapter 2. An important rule about operators in FORTRAN II expressions is that every operation must be explicitly represented by an operator. In particular, the multiplication sign must never be omitted. Likewise, since superscript notation is not available, a symbol for exponentiation is provided.

### Statement Labels

It is sometimes necessary for statements in a program to refer to other statements, e.g., in transferring control, referring to format statements, etc. To provide for such references a statement may be preceded by a label. A statement label may be any string of one to five decimal digits, i.e., any integer from 1 to 99999. Leading zeros are ignored; thus 99, 099, 0099, and 00099 are all considered to be the same label.

## GENERAL FORMATION RULES

Certain general rules hold for the formation of all FORTRAN programs and all FORTRAN statements. These rules will be dealt with here.

Physically a FORTRAN program is divided into a sequence of lines. One line may be either one punched card or all the characters punched on a paper tape by one line of typing on a Teletype. Logically a FORTRAN program is divided into statements. Therefore, some correspondence must be established between lines and statements. The rules are:

1. There can be no more than one statement per line.
2. If a statement is too long to fit on one line, it may be continued onto additional

lines. In this case, the first line is known as the initial line and each succeeding line is known as a continuation line.

Each continuation line must be identified as such by a special character. Which character is used, and where it appears on the line, depends on the input medium. (See below under Line Formats for each medium). A PDP-6 FORTRAN II statement may have a maximum length of 660 characters.

## Line Formats

The format for a line depends on the input medium. Since punched cards and punched paper tape may both be used on the PDP-6, both formats are discussed below:

### Card Input

Each card contains exactly one line. The line is further subdivided into four fields, i.e., four sets of contiguous columns.

The four fields are columns 1 to 5, column 6, columns 7 to 72, and columns 73 to 80. This is known as a fixed field format where all the information within a certain field is to receive a specified interpretation. (The interpretation for each field is often printed at the top of the card to assist the programmers, see Figure 2).

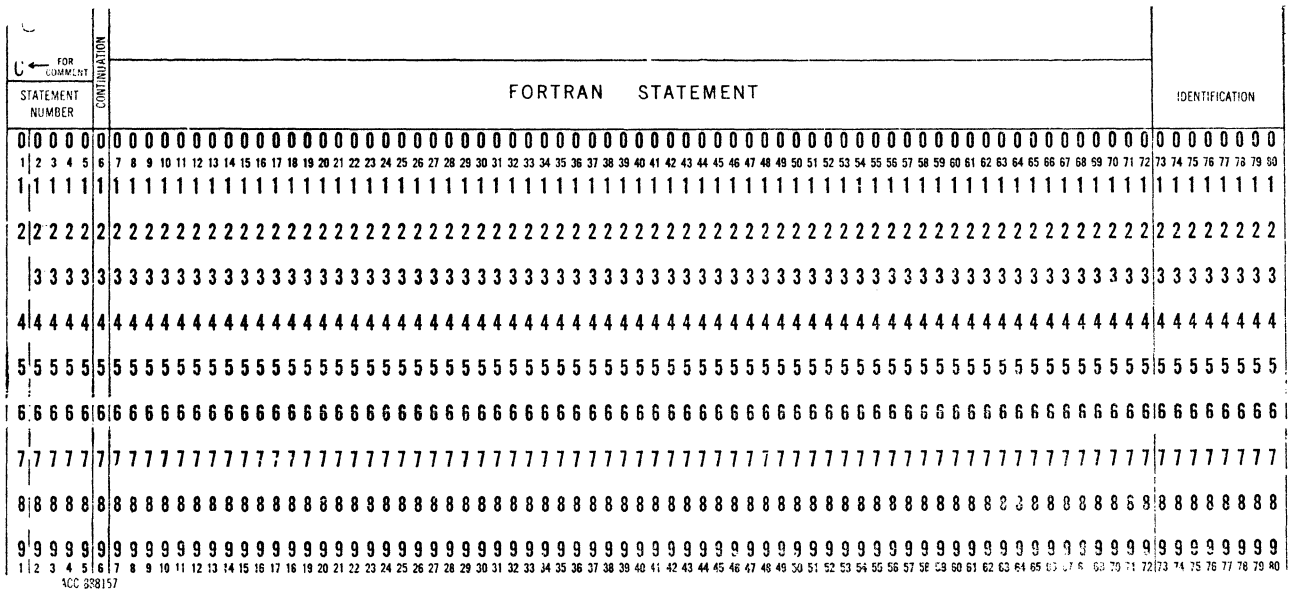



Figure 2 FORTRAN Statement Card

The interpretation for each field is:

Field 1 (Columns 1-5). This field is further subdivided into two overlapping subfields, i.e., column 1 and columns 1-5. Column 1 may contain one of several special control characters (see below). Statement labels appear in columns 2-5 and may extend to column 1 if there is no control character.

Field 2 (Column 6). This one column field contains the continuation character; if it is blank, or punched with a zero, the line is an initial line. Any other character, punched in column 6, denotes a continuation line.

Since nine continuation lines are allowed, the

programmer may use the digits 1 to 9 to indicate their proper order.

Field 3 (Columns 7-72). This field contains the statement. It cannot contain more than one statement; however, it may contain as much of a single statement as will fit in 66 or fewer characters.

Field 4 (Columns 73-80). This field is ignored by the compiler and may be used to place any kind of identifying information on the card, e.g., sequence numbers.

## Punched Tape Format

On punched tape, lines must be delineated by a carriage return/line feed. There are two formats for punched paper tape:

1. Card Simulated Format - Paper tape may be punched in a card format using the tab to space to column 7. Each line ends with a carriage return/line feed. Continuation characters are punched in column 6. Lines with no statement numbers or control characters begin with a tab or six spaces. Statement numbers in columns 1-5 or control characters in column 1 may be followed by a tab or enough spaces to reach column 7.

2. Column Free Format - Lines are delineated by carriage return/line feed. A continuation line is identified by a back slash as the last character before the terminator for the previous line. Statement labels and control characters must precede the statement and must be followed by a colon.

Figure 3 gives an example of the use of the back slash as a continuation character.

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

Figure 3 Use of Continuation Characters

## Control Characters

Occasionally it is necessary to instruct the compiler to interpret a statement in some special way. Two control characters are provided for this which must appear in the first character position of the statement's initial line. They are:

C Designates a comment statement. Such a statement is ignored by the compiler except for being retained for printing along with the compilation. A comment statement may not have continuation lines. However, this is not a serious restriction since each line of the comment is simply preceded by a C.

B Indicates that Boolean operations are to be compiled. (See Chapter 2.)

## END STATEMENT

The END statement 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.

# CHAPTER 2

# ASSIGNMENT STATEMENTS

Assignment statements are statements of the form v = e, where v is a variable name and e is some arithmetic expression. The execution of an assignment statement replaces the value of v with the value of e. There are two types of assignment statements: arithmetic statements and Boolean statements.

## ARITHMETIC EXPRESSIONS

The elements of an arithmetic expression are constants, variables, subscripted 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.

### Arithmetic Operators

Arithmetic operators are symbols representing the common arithmetic operations as follows:

| | |
|---|---|
| Exponentiation | ** or ↑ |
| Multiplication | * |
| Division | / |
| Addition | + |
| Subtraction or unary minus | - |

The symbols are compiled as arithmetic operations unless the statement has the control character 'B' prefixed.

A unary minus is the operator which precedes a quantity whose value is to be negated.

### Formation Rules

An arithmetic expression may be:

1. A constant, a variable name, subscripted variable name, or function name.
2. Any entity conforming to rule 1, with a prefixed (unary) minus sign.
3. Two entities conforming to rules 1 or 2 with an infixed operation symbol.
4. Two entities conforming to rules 1, 2, or 3 with an infixed operation symbol.
5. Two entities conforming to rules 1, 2, 3, or 4 and enclosed in parentheses.
6. An entity conforming to rule 5 with a prefixed unary minus.
7. Two entities conforming to any of the above with an infixed operation symbol.

All variables of an arithmetic expression must be of the same type, except for exponents which may be of type integers in either integer or real expressions.

Figure 4 demonstrates the properties of arithmetic expressions. Each expression is shown with its corresponding algebraic form.

### 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:

| | | |
|---|---|---|
| 1. | Expressions within parentheses | ( ) |
| 2. | Unary minus | - |
| 3. | Exponentiation | ** or ↑ |
| 4. | Multiplication | * |
| | Division | / |
| 5. | Addition | + |
| | Subtraction | - |

The term binding strength is used to refer to an operator's relative position in a table such as the one above in which 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 strength.

The left-to-right rule can now be stated a little more precisely as follows: Operations are performed in order of decreasing binding strength. A sequence of operations of equivalent binding strength is evaluated from left to right.

Examples g and h in Figure 4 illustrate the use of functions as variables in an arithmetic expression. Included in these examples are SINF (THETA), COSF (THETA -1.5), and SQRTF (Z) corresponding to the trigonometric functions sine, cosine, and $\sqrt{(x)}$.

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

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

Figure 4 Properties of Arithmetic Expressions

## Use of Parentheses

As with ordinary algebra, parentheses may be used to change the normal order of evaluation. For example:

```
A + B * C
```

would result in A being added to the product of B and C. However, if we want, instead, the product of A + B times C we may write:

```
(A + B) * C
```

Also, parentheses may be embedded in parentheses. For example:

```
(A * (B + C) )** D
```

The expression is evaluated from the innermost expression outward. That is, B + C is formed, then the product with A, and finally the exponentiation. The expression within parentheses simply becomes a numerical argument for the rest of the expression.

Expressions with many nested subexpressions can become very difficult to read making it difficult to be sure that each left parenthesis is properly paired with a right parenthesis. If they are not properly paired, an error diagnostic is printed. Fortunately, the test used by the computer turns out to be a very simple way of checking by hand. Consider the following example:

```
Z*(P*(SINF(THETA)+S) / (Z**2-(B**2+C**2)) )
  1 2    3      2 1   2     3          2 1 0
```

The procedure is this: starting with a count of zero, scan the expression from left to right. Increase the count by one for each right parenthesis. Decrease the count by one when a right parenthesis is read. When the expression has been completely scanned, the count should be zero.

## The Arithmetic Statement

The arithmetic statement relates v, a variable name or array element name of type integer or real, 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 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 5.

```
a. VMAX=VO+Z*TO
b. T = 2.*PI*SQRTF(1.0/G)
c. PI=3.14159
d. THETA=OMEGAO*T+ALPHA*T**2/2.
e. MIN=M + N + 5
f. INDEX = INDEX +2
```

Figure 5 Arithmetic Statements

The equal sign is considered to have a lower binding strength than all of the operators. This means that the whole of the expression on the right is evaluated before the operation indicated by = is performed. By this definition the statement in example f of Figure 5 would mean, "add two to the current value of INDEX. The result is the new value of INDEX."

It should be noted that all variables occurring to the right of an equal sign must have been defined and calculated when the expression is evaluated. If the variable on the left of the equal sign was previously undefined, it will be defined by the arithmetic statement.

## Assignment Rules

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, according to the following rules:

| v is type | e is type | Rule |
|---|---|---|
| integer | integer | v = e |
| integer | real | v = FIX(e) |
| real | integer | v = FLOAT(e) |
| real | real | v = e |

The name FIX* means truncate any fractional part and convert to integer representation. The name FLOAT means convert to real representation.

## Internal Arithmetic Statement

An 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) )
```

The internal arithmetic statement, V=SQRTF (2.*G*Y), must be set off from the rest of the statement by parentheses. 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 these equations are evaluated starting with the innermost statement and values are assigned to V and Q.

Another result of treating the equal sign as an operator is that just as there may be a series of additions, a+b+c, there may be a series of replacements, a=b=c=d. 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 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:

```
VZERO=SZERO=AZERO=0.
P=PO=4*ITM-K
```

Each replacement conforms to the above rules.

---

*Truncate in 2's complement means find the greatest integer in X. Hence XFIXF(0.5)= 0 and XFIXF(-0.5)=-1. The function XINTE(X) performs truncation to the greatest integer in the absolute value. Hence XINTF(0.5)=0 and XINTF(-0.5)=0.

**This may seem at first to violate the left-to-right rule. However, 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.

## BOOLEAN STATEMENTS

The arithmetic statement may be used to perform Boolean operations if a B is placed in the first character position of the statement line. In this case the symbols for the arithmetic operations take on the following meanings:

| | |
|---|---|
| + | inclusive OR |
| - (binary) | exclusive OR |
| - (unary) | 1's complement |
| * | AND |
| / | equivalence |
| ** or ↑ | shifting |

For all but the shift operation, the mode of the data is in a sense irrelevant. The Boolean operations are performed on the full 36 bits of the data word, complementing etc., on a bit by bit basis without regard for the significance of the bits.

The effect of these operations may be defined as follows: Let $r_i$ be the $i^{th}$ bit of the 36-bit result. Let $arg1_i$ be the $i^{th}$ bit of the first argument; let $arg2_i$ be the $i^{th}$ bit of the second argument. Then for each operation we have:

| Boolean Algebra | | Result of Operation ($r_i$) | | | |
|---|---|---|---|---|---|
| $arg1_i$ | $arg2_i$ | + | - | * | / |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 |

For the unary minus:

$$r_i = 0 \text{ if } arg_i = 1$$

$$r_i = 1 \text{ if } arg_i = 0$$

For the shifting operation the second argument must be an integer, i.e., the form must be:

arg ** ivar

where arg is an arbitrary type and ivar is an integer. arg is shifted right if ivar is negative, left if ivar is positive, by $ivar_{mod36}$ positions. That is:

$$r_i = arg_{i \pm ivar_{mod36}}$$

Examples:

```
B  A = B+213776347777
B  C = C*A-D
B  F = C**-3
B  H = E/G
```

# CHAPTER 3

# SPECIFICATION STATEMENTS

There are three types of specification statements: dimension statements, common statements, and equivalence statements. These are called specification statements since they specify structural properties of the program which must be known to the compiler if it is to set aside enough storage for arrays, properly relate different variables, etc.

> NOTE: All specification statements must appear before any executable statement. (Executable statements are assignment statements, control statements, and the executable input-output statements.)

## Array Declarators

In an array of dimension n, each subscript is allowed to assume all integral values between 1 and some maximum, where $d_1, d_2, \ldots\ldots d_n$ represent the maximum values for each subscript. The array will require:

$$d_1 \times d_2 \times \ldots\ldots d_n$$

words of storage. An array declarator is used to inform the compiler of this requirement. It has the form:

$$\text{aname } (d_1, d_2, \ldots\ldots d_n)$$

where aname is the array name. It is subject to the same formation rules and type conventions as ordinary variable names. All of the elements of the array are of the type declared by the name. The d's, the declarator subscripts, must be integer constants. Their presence is sufficient to inform the compiler that an array is being declared. Their number indicates the dimensionality of the array; their magnitude, the maximum value which each subscript may assume.

## Array Successor Function

Arrays are stored by columns in the array storage area. That means the following function may be used to establish the location of a given element relative to the beginning of the storage area:

$$s = i_1 + d_1 (i_2 - 1) + d_1 d_2 (i_3 - 1)$$
$$+ \ldots\ldots + d_1 d_2 \ldots d_n (i_n - 1) - 1$$

where $i_1, i_2, \ldots i_n$ are the values of the n subscripts. This function is important when an array element must be identified by a single subscript.

> NOTE: An array declarator must appear for each array. It may appear in either a dimension statement, or a common statement.

## DIMENSION STATEMENTS

Dimension statements are provided explicitly to declare arrays. They have the form:

$$\text{DIMENSION } a_1, a_2, \ldots\ldots\ldots a_n$$

where the a's are array declarators. No two array declarators may have the same array name.

## COMMON STATEMENTS

The common statement allows the programmer to assign the same meaning to variable names in different program units. Names appearing in common statements are assigned storage locations in the order in which they appear, beginning at location $140_8$.

Common statements have the form:

$$\text{COMMON } v_1, v_2, \ldots v_m$$

where each v may be either a variable name, an array name, or an array declarator. Each name must be distinct; e.g., a listed variable name must not be repeated as an array name.

Since the storage location of a variable is determined by its order in the common list, variables in different program units need not have the same name to have the same meaning. For example:

```
COMMON X, I, Z
COMMON A, J, C
```

may be two common statements appearing in two different program units. The effect of the two statements is to assign the following locations to the listed variables:

| | |
|---|---|
| LOC (X) = $140_8$ | LOC (A) = $140_8$ |
| LOC (I) = $141_8$ | LOC (J) = $141_8$ |
| LOC (Z) = $142_8$ | LOC (C) = $142_8$ |

If an array name appears in a common statement, enough sequential common locations must be set aside to contain the array. Consequently, the array must be declared in a dimension statement. The alternative is to list an array declarator in the common statement rather than simply the array name.

Considerable care must be taken to assure that variables which are to be identical occupy the same position in common. For example, assume in one program we have the following statement:

```
COMMON A, B, C
```

where B is an array name. Then, assume in a different program unit we have:

```
COMMON X, Y, Z
```

where X, Y, and Z are all single variables. Then LOC (X) = LOC (A); however, C will not be in the same location as Z. Rather, Y will equal the first element of B, and Z will equal the second element of B. To have LOC (C) = LOC (Z) and still write the list in the above order, we must declare Y to be an array of the same size as B. That is, if B is a 3 x 3 array, we could declare Y to be a 3 x 3 array or a 1 x 9 array.

## EQUIVALENCE STATEMENTS

Equivalence statements are used to assign two or more variables in a program unit to the same storage locations. Such an assignment may be made for several reasons, perhaps the most common being to save storage. If a program contains several variables which are never used simultaneously or, more accurately, the value of one need not be retained while another is being used; storage may be saved by having them all use the same location.

The general form of the equivalence statement is:

EQUIVALENCE $(e_1)$, $(e_2)$, ... $(e_n)$

where each e is a list of the form $v_1, v_2, \ldots v_n$ and $n \geq 2$. In turn, each v may be a variable name or an array element name. All of the entities named within a single set of parentheses are assigned to the same location in storage.

The array element is identified by a single subscript, which is equal in value to the array successor function. If an array name is listed without the linear subscript, a subscript of 1 is assumed.

If an element in an equivalence group also appears in a common statement, all the elements of the group are considered to be in common. The common block is reordered so that the items appearing in equivalence statements appear first and in the same order as in the equivalence statements. For example, suppose a program had the common statement:

COMMON A, B, C, D

This would make the following assignment:

LOC (A) = $140_8$ LOC (C) = $142_8$
LOC (B) = $141_8$ LOC (D) = $143_8$

If there were also an equivalence statement:

EQUIVALENCE (D, G), (F,B)

The result would be:

LOC (G) = LOC (D) = $140_8$
LOC (F) = LOC (B) = $141_8$
LOC (A) = $142_8$
LOC (C) = $143_8$

A question arises as to what happens if an array element appears as equivalent to a common variable. The answer is:

> The common variable is placed in common according to its position in an equivalence statement. The array is then assigned to common locations so that the specified element is equivalent to the specified common variable.

For example, consider:

COMMON A, B, C, D
EQUIVALENCE (E(1),C)

The result would be:

LOC (E(1)) = LOC (C) = 140
LOC (E(2)) = LOC (A) = 141
LOC (E(3)) = LOC (B) = 142
LOC (E(4)) = LOC (D) = 143
LOC (E(n+1)) = 140 + n

However, the pair of statements:

COMMON A, B, C, D
EQUIVALENCE (E(5), C)

would have the following effect:

LOC (E(1)) = 140
LOC (E(2)) = 141
LOC (E(3)) = 142
LOC (E(4)) = 143

LOC (E(5)) = LOC (C) = 144

LOC (E(6)) = LOC (A) = 145

LOC (E(7)) = LOC (B) = 146

LOC (E(8)) = LOC (D) = 147

Restrictions

A common variable may appear in only one equivalence statement unless its storage requirements are equal to or exceed all other variables in the succeeding equivalence groups in which it appears.

# CHAPTER 4

# CONTROL STATEMENTS

Normally, the executable statements of a FORTRAN program are executed in the order of their appearance. However, the order of execution may be altered by any of the following control statements: GO TO, IF, CALL, RETURN, CONTINUE, STOP, PAUSE, and DO.

The CALL and RETURN statements are used with subroutines and will be discussed in Chapter 5.

The control statements (except for STOP, CONTINUE, and RETURN) contain labels to identify those statements which are to be executed following execution of the control statement itself. All labels in a control statement must refer to executable statements.

## GO TO STATEMENTS

Unconditional GO TO - This statement has the form:

GO TO n

where n is a statement label. Execution of this statement results in the statement labeled n being executed next.

Example:

```
GO TO 10
```

Computed GO TO - The computed GO TO is an n-way branch, where the branch selected depends on the value of an integer variable. The form is:

GO TO (label1, label2,...labeln), iname

where iname is an integer variable name. The next statement executed, following the computed GO TO, will be the statement corresponding to the nth label in the list; n being the current value of iname.

Example:

```
GO TO (21, 4, 36, 18, 100), INT
```

If INT = 3, the next statement to be executed is statement 36.

Assigned GO TO - There are two forms of this statement, one being similar to the unconditional GO TO and one similar to the computed GO TO. In both cases, the effect of the statement depends on the current value of an integer variable. The two forms are:

GO TO iname

and

GO TO iname, (label1, label2,...labeln)

The first form simply transfers the execution sequence to the statement whose label is equal to the value of iname. The second form matches the current value of iname against the labels in the list. If a successful match is found, the effect of the statement is the same as the first form. If not, control passes to the next executable statement.

The statement:

ASSIGN n TO iname

assign the label n to the integer variable iname.

If more than one ASSIGN statement refers to the same integer variable name, the value assigned by the last executed statement is the current value.

Examples:

```
ASSIGN 21 TO INT
      .
      .
      .
GO TO INT
```

```
ASSIGN 1000 TO INT
      .
      .
      .
GO TO INT, (2,21,1000,310)
```

## IF STATEMENTS

Arithmetic IF - The arithmetic IF is a three way branch. It has the form:

IF (exp) label1, label2, label3

where exp is any arithmetic expression of type real or integer; and label1, label2, and label3 are three statement labels. The next statement to be executed is:

label1 if exp < 0
label2 if exp = 0
label3 if exp > 0

The three labels need not all be distinct; less than or equal, greater than or equal, and not equal conditions may be tested by repeating label.

Examples:

```
1. IF (K) 23,64,100
2. IF (A**I*B+C) 23,64,23
3. IF (K-100) 5,5,10
```

Boolean IF - The Boolean IF is a two way branch. It has the form:

IF (exp) label1, label2

where exp is a Boolean expression. The next statement executed after the IF is:

label1 if exp = 0
label2 if exp $\neq$ 0

Examples:

```
IF (A*672) 101,3
IF (-(A+B)) 4,76
IF (-(A/B)) 36,21
```

The Boolean IF must be identified by a B in column 1.

NOTE: The character * may replace any statement label in an arithmetic or Boolean IF statement; this causes control for the branch corresponding to the * to pass to the next executable statement.

### Special IF Statements

There are several forms of IF used to detect various conditions in the state of the machine.

Sense Lights

IF (SENSE LIGHT i) label1, label2

where i is an integer constant and $1 \leq i \leq 36$. If the designated light is on, it is turned off and the statement corresponding to label1 is executed next; otherwise, control transfers to the statement corresponding to label2.

NOTE: Although the PDP-6 has no sense lights, the FORTRAN II Compiler sets aside a register for sense light tests.

The statement:

SENSE LIGHT i

where i is an integer constant and $1 \leq i \leq 36$ causes sense light i to be turned on. The statement SENSE LIGHT 0 causes all sense lights to be turned off.

Sense Switch

Sense switch settings may be tested by the statement:

IF (SENSE SWITCH i) label1, label2

Control goes to label1 if sense switch i is up and label2 if it is down; i is an integer constant in the range $0 \leq i \leq 35$.

Overflow - There are two overflow conditions which may be tested. The statements provided are:

IF ACCUMULATOR OVERFLOW label1, label2
IF QUOTIENT OVERFLOW label1, label2

If an overflow condition is indicated, the indicator is cleared and the statement corresponding to label1 is executed next. Otherwise, the statement corresponding to label2 is executed next.

Examples:

```
IF ACCUMULATOR OVERFLOW 14,2
IF QUOTIENT OVERFLOW 13,51
```

NOTE: Both ACCUMULATOR and QUOTIENT OVERFLOW tests examine the overflow flag in the PDP-6.

## DO STATEMENTS

DO statements provide a convenient means for causing the repeated execution of a series of statements along with incrementing an index for each repetition.

The general form of a DO statement is:

DO label j = k, l, m

where label is the statement label of an executable statement in the same program unit as the DO and sequentially following the DO. This statement is known as the terminal statement. The terminal statement must not be any form of GO TO, IF, STOP, PAUSE, or DO statement.

### Range

Each DO statement has an associated range which includes the first executable statement following the DO and extends to and includes the terminal statement.

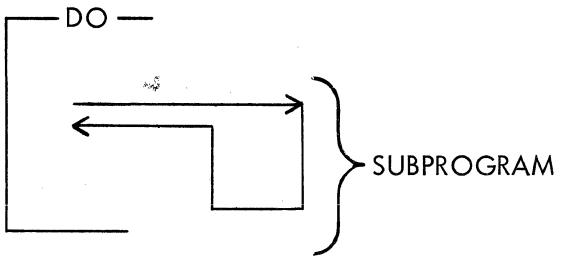
j is an integer variable name known as the control variable; it is the index to be incremented.

k, l, and m are integer expressions. j is initially set to the value of k. After execution of the terminal statement, the value of j is incremented by the value of m. If the value of j is less than or equal to the value of l, control passes to the first executable statement following the DO. Otherwise control passes out of the range of the DO.

m may be omitted, in which case its value is assumed to be 1.

No statement within the range of the DO should change the value of j, k, l, or m.

Not all statements in the range of the DO need appear sequentially between the DO and the terminal statement. For example, a statement within a DO may transfer control to a procedure, which, when terminated, returns control to a statement within the range of the DO. This may be diagrammed as follows:

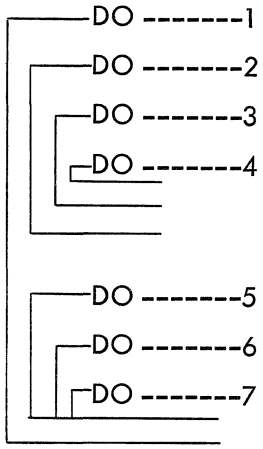



In this case, the subprogram is considered to be within the range of the DO.

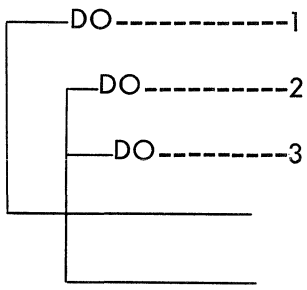
It is often desirable to have DO's within a DO. This might occur, for example, when a calculation is being performed with multiply subscripted parameters. An innermost DO will completely run through all the values of one subscript for each step of an outer DO.

When a DO is contained within a DO, i.e., nested, the range of the contained DO must be a subset of the range of the containing DO. That is, all the statements of the inner DO, from the DO itself to and including its terminal statement, must be statements in the range of the outer DO.

The following diagram illustrates proper and improper nesting of DO's.



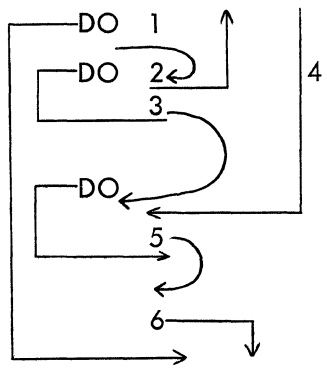

Allowable Nesting




Improper

A program branch must not occur from a statement outside the range of a DO to a statement within the range of a DO. However, branches are permitted out of the DO, in which case the control variable is defined and is available.

Branches 2, 5, and 6 in the following diagram are permitted; branches 1, 3, and 4 are not.




Finally, if two or more nested DO's share the same terminal statement, no statement outside the range of the innermost DO may cause a branch to the terminal statement.

Examples:

```
DO 100 I1 = N1*2,M,K*(N2*3-1)
DO  40 I2 = 1,10,4
DO  10 I3 = 7,J
```

## CONTINUE STATEMENT

The general form of a CONTINUE statement is:

CONTINUE

Its execution has no effect on the program beyond causing control to go to the next statement in sequence. Nevertheless, this can be a useful statement. For example, suppose it is necessary to have a conditional GO TO or IF statement in the range of a DO so that, if the stated conditions are met, the control variable is incremented and another repetition of the entire range begins. This can be done by making the terminal statement of the DO a CONTINUE statement. The GO TO or IF statements then simply transfer control to the terminal sequence of the DO.

## PAUSE STATEMENT

PAUSE statements may have the form:

PAUSE n or PAUSE 'MESSAGE'

where n is an octal constant and may be omitted. Execution of a PAUSE statement causes PAUSE, PAUSE n or PAUSE 'MESSAGE' to be typed on the user Teletype. The user may type G to return control to the next statement or X to terminate the program. A carriage return must follow the G or X.

## STOP STATEMENT

The general form of the STOP statement is:

STOP n

where n is an octal constant and may be omitted. The STOP statement causes termination of a program.

# CHAPTER 5

# SUBPROGRAMS

There are two broad classes of subprograms, defined by the way the subprogram is called and whether or not it may return one or many values to the main program. Subroutine subprograms must be explicitly called by a CALL statement. They may return one, or several, values. Function subprograms are implicitly called by using their names in an arithmetic expression. They may return only a single explicit value. Implicit values may be returned through arguments or COMMON storage.

## DUMMY ARGUMENTS

All subprograms use dummy arguments in their argument lists which accompany their definitions. The calling argument list must match the dummy argument list in the number, order, and type of argument. Beyond that, there need be no correspondence between the names of the arguments in the two lists.

## FUNCTIONS

The two types of functions are intrinsic and external. Intrinsic functions are predefined and are part of the FORTRAN language. External functions are defined by subprograms external to the program unit in which they are called.

### External Functions

External functions, along with subroutine subprograms, correspond most closely to what is normally thought of as a separate and closed subroutine. That is, an external function is a single and separate program. It exists only once in memory and is separate from the main program. Whenever it is called, control leaves the main program temporarily and goes to the external function.

The following rules must be observed in defining an external function:

1. The first statement must be a FUNCTION statement. This is a statement of the form

   FUNCTION name $(a_1, a_2, \ldots\ldots, a_n)$

   where name is the symbolic name of the function. Name may not have a terminal F unless it consists of three characters or less. The a's are dummy arguments which may be either variables or array names.

   Symbolic function names are constructed by the same rules as for variables. The type of the function name, integer or real, determines the type of the result.

2. The symbolic name of the function must appear as a variable name within the program. Upon execution of the RETURN statement, the value of the variable is considered the value of the function.

3. The symbolic name of the function must not appear in any non-executable statement within this program unit except for the FUNCTION statement.

4. The symbolic names of the dummy arguments must not appear in any COMMON or EQUIVALENCE statements within this program unit.

5. The subprogram may not contain a SUBROUTINE statement, another FUNCTION statement, or a reference to itself.

6. The subprogram must be logically terminated by a RETURN or CALL statement.

7. The subprogram must be physically terminated by an END statement.

External functions are called by using the symbolic name of the function, followed by a list of actual arguments. The arguments must agree in number, order, and type with the dummy arguments. Further, if a dummy argument is an array name, the corresponding actual argument must be an array name. The actual arguments may be variable names, array elements, array names, or expressions. The expressions may, in turn, contain calls to other external functions.

An external function to calculate the factorial of an integer n:

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

Library Functions

Certain external functions are used so commonly that they are considered basic, and are supplied in the library. Table 1 shows these basic functions. If desired, additional functions may be added to the library. When library functions are called, a terminal F is appended to the function name.

Example:

```
ROOT1 = (-B+SQRTF(B**2-4.*A*C))/2.*A
```

TABLE 1 EXTERNAL FUNCTIONS

| Basic External Functions | Definition | Number of Arguments | Symbolic Name | Type of Argument | Type of Function | Restrictions |
|---|---|---|---|---|---|---|
| exponential | $e^x$ | 1 | EXP | real | real | |
| natural logarithm | $\log_e$ | 1 | LOG | real | real | |
| sine | sine (arg. in radians) | 1 | SIN | real | real | |
| | sine (arg. in degrees) | | SIND | real | real | |
| cosine | cos (arg. in radians) | 1 | COS | real | real | |
| | cos (arg. in degrees) | | COSD | real | real | |
| hyperbolic tangent | tanh | 1 | TANH | real | real | |
| square root | $x^{1/2}$ | 1 | SQRT | real | real | |
| arctangent | $\tan^{-1}$ | 1 | ATAN | real | real | |
| arctangent | $\tan^{-1}$ (y/x) | 2 | ATAN2 | real | real | |

TABLE 1 EXTERNAL FUNCTIONS (continued)

| Basic External Functions | Definition | Number of Arguments | Symbolic Name | Type of Argument | Type of Function | Restrictions |
|---|---|---|---|---|---|---|
| arcsin | $\sin^{-1}$ | 1 | ASIN | real | real | |
| arccos | $\cos^{-1}$ | 1 | ACOS | real | real | |
| hyperbolic sine | sinh | 1 | SINH | real | real | |
| hyperbolic cosine | cosh | 1 | COSH | real | real | |
| common logarithm | $\log_{10}$ | 1 | LOG10 | real | real | |
| truncation | sign of a times largest int. $\leq$ \|a\| | 1 | XINT<br>INT | real<br>real | integer<br>real | $\|a\| < 2^{34}$<br>a normalized |
| remaindering (*note) | $a_1 \pmod{a_2}$ | | MOD | real | real | $\|a_1/a_2\| < 2^{27}$ |
| choose largest value | max $(a_1, a_2 .. a_n)$ | $n \geq 2$ | XMAX0<br>XMAX1<br>MAX0<br>MAX1 | integer<br>real<br>integer<br>real | integer<br>integer<br>real<br>real | |
| choose smallest value | min $(a_1, a_2, .. a_n)$ | $n \geq 2$ | XMIN0<br>XMIN1<br>MIN0<br>MIN1 | integer<br>real<br>integer<br>real | integer<br>integer<br>real<br>real | |
| transfer sign | sign of $a_2$ times $\|a_1\|$ | 2 | XSIGN<br>SIGN | int/either<br>real/either | integer<br>real | |
| positive difference | max $(a_1 - a_2, 0)$ | 2 | XDIM<br>DIM | integer<br>real | integer<br>real | |

*Note: The functions MODF($a_1$, $a_2$) and XMODF($a_1$,$a_2$) are defined as $a_1 - [a_1/a_2]a_2$, where [ x ] is the largest integer which does not exceed the absolute value of x, and whose sign is the same as x.

## Intrinsic Functions

Table 2 lists the intrinsic functions for PDP-6 FORTRAN. The intrinsic functions are sometimes referred to as open subroutines. This means each time the function name is encountered, the sequence of program steps to evaluate the function becomes part of the calling program.

The function name must agree exactly with the name shown in Table 2, and it must be followed by a parenthesized list of arguments. Each ar-

gument may be any arithmetic expression which agrees in order, type, and number with the specifications of Table 2.

An initial X is always present in the names of functions which produce results of type integer.

Some examples of arithmetic expressions with intrinsic functions:

```
A * Z + ABSF (C) * D
I * J + XABSF (K) * L
```

TABLE 2 INTRINSIC FUNCTIONS

| Intrinsic Function | Definition | No. of Args. | Function Name | Mode of Args. | Mode of Function | Restriction |
|---|---|---|---|---|---|---|
| absolute value | $\|a\|$ | 1 | XABSF<br>ABSF | integer<br>real | integer<br>real | none<br>none |
| remaindering (*note) | $a_1 \pmod{a_2}$ | 2 | XMODF | integer | integer | none |
| float | convert integer to real | 1 | FLOATF | integer | real | none |
| fix | convert real to integer the result is the largest integer $\leq a$ | 1 | XFIXF | real | integer | none |

*Note: The functions MODF($a_1$, $a_2$) and XMODF($a_1$,$a_2$) are defined as $a_1 - [a_1/a_2]\, a_2$, where [x] is the largest integer which does not exceed the absolute value of x, and whose sign is the same as x.

## SUBROUTINES

### Defining Subroutines

Subroutines are defined by writing a program much like any other program except for the following restrictions:

1. The first statement must have the form

   SUBROUTINE name

   or

   SUBROUTINE name ($a_1$,$a_2$,...$a_n$)

   where the name is the symbolic name of the subroutines and the a's, the dummy arguments, are either variable or array names.

2. The subroutine name must not appear in any other statement other than the first.

3. No dummy argument name can appear in a COMMON or EQUIVALENCE statement. However, they must appear in a DIMENSION statement if they are array names.

4. The subprogram may not contain a FUNCTION statement or another SUBROUTINE statement.

5. The subroutine must be logically terminated by a RETURN, CALL EXIT, or CALL DUMP statement.

6. The subroutine must be physically terminated by an END statement.

## Calling Subroutines

A subroutine is called by a statement of the form

CALL name

or

CALL name ($a_1$,$a_2$,....,$a_n$)

where name is the symbolic name of the subroutine and the a's are the actual arguments. An argument may be a variable name, an array element, an array name, or any other expression. However, the actual arguments must agree in order, number, and type with the dummy argument list in the subroutine definition.

None of the arguments should have the same name as the subroutine.

A subroutine may return one or more calculated values by redefining one or more of the arguments in the argument list. If this is done, the redefined argument should not appear in this list as an expression.

Example:

```
CALL FOFX (A(3), 5.43,SQRTF(Y),5H ERROR)
```

## RETURN

The RETURN statement causes control to be transferred to the next executable statement in the calling program. RETURN statements may appear only in subprograms.

The following example contains a subroutine, for multiplying two 3 x 3 matrices to form a third:

```
      SUBROUTINE MATMPY (A, B, C)
      DIMENSION A(3, 3), B(3, 3), C(3, 3)
      DO 100 I=1, 3
      DO 100 J=1, 3
100   C(I, J)=A(I, 1)*B(1, J)+A(I, 2)*B(2, J)
     X +A(I, 3)*B(3, J)
      RETURN
      END
```

This subroutine could be called by the statement

```
CALL MATMPY (AMAT1,AMAT2,PMAT)
```

The routine would then form the product of AMAT1 times AMAT2 and return all the values in an array named PMAT.

An alternative method for calling and transmitting information is:

```
      SUBROUTINE MATMPY
      COMMON A(3, 3), B(3, 3), C(3, 3)
      DO 100 I=1, 3
      DO 100 J=1, 3
100   C(I, J)=A(I, 1)*B(1, J)+A(I, 2)*B(2, J)
     X +A(I, 3)*B(3, J)
      RETURN
```

called by:

```
      COMMON AMAT1(3,3), AMAT2(3,3),
     X PMAT(3,3)
          .
          .
          .
      CALL MATMPY
```

# CHAPTER 6

# INPUT-OUTPUT

## INPUT-OUTPUT LISTS

Data transfer statements contain input-output lists. These are ordered lists of variable names, array element names, or array names (also expressions in the case of output) each separated by commas. The order of the list, from left to right, specifies the order in which the data will appear.

Example:

```
A, B, C(3), F(7), NUM, CONST
```

If data was being read in, the first number read would be placed in a memory location referenced by A, the second by B, the third by C(3), etc.

The appearance of an array name without subscripts causes input or output of all the elements of the array, ordered by the array successor function.

A DO-implied list is a list which is enclosed in parentheses and ends in an indexing specification, i.e., a list of the form:

(name1, name2,....namen, i=m1, m2, m3)

where i, m1, m2, and m3 all have the same meanings as in DO statements, and name1, name2, and namen are subscripted variables.

Example:

```
(VAR(K), MAT(K), K=1, 10)
```

This list is equivalent to VAR(1), MAT(1), VAR (2),....VAR(10), MAT(10).

Implied-DO lists may be nested. In this case, the range of the inner DO is exhausted for each increment of the outer DO.

For example, suppose it is desired to read in a matrix by row rather than column order. Then a nested pair of implied DO's may be used as follows:

```
((MAT (I,J), J=1, m), I=1, n)
```

This is equivalent to the list (MAT(1,1), MAT(1, 2),....MAT(1,m), MAT(2,1),....MAT(m,n).

## INPUT-OUTPUT STATEMENTS

Input-output statements are of two types: executable statements cause transmission of information from or to input-output devices. FORMAT statements, referred to by executable input-output statements, specify the arrangement and conversion of information denoted by the input-output lists.

### Executable Statements

Input statements have the form:

```
READ label, list
REREAD label, list
```

ACCEPT label, list
READ INPUT TAPE n, label, list
or RIT n, label, list

REREAD INPUT TAPE n, label, list

READ TAPE n, list

where label is a FORMAT statement label, list is an input list, and n is an integer expression denoting a device number.

For the READ, TYPEIN, READ INPUT TAPE, and RIT statements, information is read in beginning with the next item in position to be read, until the input list is satisfied. Each item of data is converted as specified in the FORMAT statement.

For the REREAD statements, the previous record is rescanned with a new format.

The READ TAPE statement causes input of binary information from device n.

Examples:

```
READ 101,(A(I),I=M,N),J
REREAD 102,(A(I),I=M,N),J
RIT N+1,5,TERM1,TERM2
READ TAPE 4, (((BINDAT(I,J,K)
I=1,I1),J=1,J1),K=1,K1)
```

Output statements have the form:

PRINT label, list

PUNCH label, list

TYPE label, list

WRITE OUTPUT TAPE n, label, list
or WOT n, label, list

WRITE TAPE n, list

where label is a FORMAT statement label, list is an output list, and n is an integer expression denoting a device number.

For the PRINT, PUNCH, TYPEOUT, WRITE OUTPUT TAPE, and WOT statements, information is written until the output list is satisfied. Each item of data is converted as specified in the FORMAT statement.

The WRITE TAPE statement causes output of binary information onto device n.

Special Tape Statements - In each of the following statements n is an integer expression:

REWIND n

This command causes tape n to be positioned at its load point.

UNLOAD n

This command causes tape n to be rewound and unloaded.

BACKSPACE n

This command causes tape n to backspace one record.

SKIP RECORD n

This command causes tape n to space at the beginning of the next record.

END FILE n

This command causes an end-of-file mark to be written on tape n.

## FORMAT Statements

The general form of a FORMAT statement is:

$$\underset{\text{label}}{\phantom{123---m}} \quad \underset{}{123\text{---}m} \qquad\qquad \underset{}{123\text{--}p}$$
$$\text{label FORMAT}(///\text{---}/f_1 s_1 f_2 s_2 \text{----} f_n s_n ///\text{--}/)$$

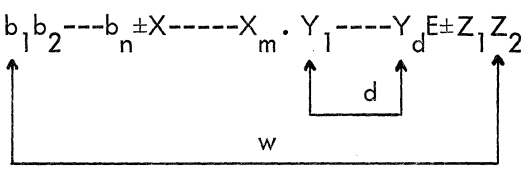
where m, n, or p, but not all three, may be zero, the f's are field descriptors or a group of field descriptors enclosed in parentheses, and the s's are either commas or slashes.

## Basic Field Descriptors

The basic field descriptors have the following forms: Fw.d, Ew.d, Gw.d, Iw, Ow, Lw, Aw, wHh, $h_2$---$h_3$, and wX. The letters F, E, G, O, I, L, A, H, and X indicate the method of conversion. 'w' is a non-zero integer constant denoting the width of the field in the external character string. 'd' is an integer constant specifying the number of digits in the fractional part. F, E, G, I, and O are all considered to be numerical fields. The remaining four are alphanumeric fields.

The fields designated by F, E, and G descriptors all contain real data which is to be converted internally to a real datum. For all three, the input fields are identical. However, the output fields may be quite different.

F, E, G Input Fields - The general form of the input field is:




To allow ease of preparing input data several variations are allowed on this basic form:

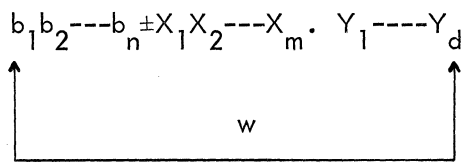
1. The decimal point may be omitted. If so, and if the field descriptor is Fw.d, Ew.d, or Gw.d, d digits are assumed to follow the implicit point. If the decimal point is present, it overrides the 'd' specification.
2. The initial plus sign may be omitted.
3. The exponent may have the following forms:
   a. A signed integer.
   b. E followed by an integer (if positive).
   c. E followed by a signed integer.
4. The exponent may be omitted.

Thus, suppose the field descriptor was E10.4, F10.4, or G10.4. The following numbers would all be equal:

| | | |
|---|---|---|
| bbbb427.67 | bbb4276700 | (decimal implied by descriptor) |
| 42.7670E+1 | b+42.767E1 | |
| b+42.767+1 | +427670E01 | (decimal implied by descriptor) |

Note that E+1, E1, E01, E+01, +1 and +01 are all equivalent.

F Conversion Output - For F conversion, the form of the output field is:

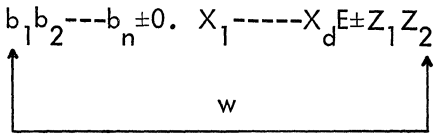



The field consists of as many blanks as necessary to fill out its width, followed by a decimal number without exponent. Plus signs are omitted.

The following table illustrates the correspondence between internal and external numbers for an F10.4 field descriptor:

| Internal | External |
|---|---|
| +4270. | b4370.0000 |
| -437. | b-437.0000 |
| +4.37 | bbbb4.3700 |
| +.437 | bbbb0.4370 |
| -.00437 | bbb-0.0044 |
| +.0000437 | bbbbb.0000 |

E Conversion Output - The external field for conversion output is of the form:




For this conversion, all numbers are normalized so that the decimal point appears to the left of the first significant digit. The number is rounded to d significant digits.

The following table illustrates the correspondence between internal and external numbers for an E12.4 field descriptor:

| Internal | External |
|---|---|
| +4370 | bb0.4370E+04 |
| -4.37 | b-0.4370E+01 |
| +.00437 | bb0.4370E-02 |
| +.0000437 | bb0.4370E-04 |
| -437.19 | b-0.4372E+03 |
| +437.23 | bb0.4372E+03 |

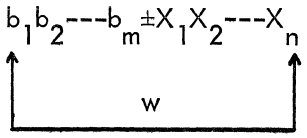
G Output Conversion - G conversion depends on the magnitude of the internal number. For the field descriptor Gw.d, the following table shows the correspondence between the magnitude of the internal number and the conversion that takes place.

| Magnitude | External |
|---|---|
| | ←W→ |
| $0.1 \leq N < 1$ | F(W-4). d  lbbbbl |
| $1 \leq N < 10$ | F(W-4).(d-1)  lbbbbl |
| . | . |
| . | . |
| . | . |
| $10^{d-2} \leq N < 10^{d-1}$ | F(W-4). 1  lbbbbl |
| $10^{d-1} \leq N < 10^{d}$ | F(W-4). 0  lbbbbl |
| Otherwise | Ew.d |

The following table illustrates G output conversion for the field descriptor G11.4.

| Internal Number | Conversion | External Field |
|---|---|---|
| 43700.0 | E11.4 | b0.4370E+05 |
| -4370.0 | F7.0bbbb | b-4370.bbbb |
| 437.0 | F7.1bbbb | bb437.0bbbb |
| 43.7 | F7.2bbbb | bb43.70bbbb |
| -4.37 | F7.3bbbb | b-4.370bbbb |
| .437 | F7.4bbbb | b0.4370bbbb |
| .0437 | E11.4 | b0.4370E-01 |

I Conversion - Fields specified by the field descriptor Iw contain decimal integers. The form of the external field is the same for both input and output:

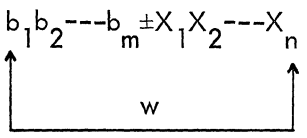



On output the plus sign is omitted.

Examples of external fields corresponding to the descriptor I6:

| | | |
|---|---|---|
| bbbbb2 | bbbb-2 | b+2763 (if output bb2763) |
| -57296 | 647198 | |

O Conversion - Fields specified by the field descriptor Ow contain octal integers. The form of the external field is:




For input the sign is optional. All O-type output conversions are unsigned.

Examples of conversion for O4 specification:

| External (input) | Internal | External (output) |
|---|---|---|
| bb27 | 000000000027 | 0027 |
| bb-1 | 777777777777 | 7777 |

L Conversion - L conversions are used to read or write logical data. The form of the field descriptor is Lw.

For input, the external field contains a string of up to w nonblank characters beginning with either T or F. For example,

| Descriptor | Input Field |
|---|---|
| L1 | T |
| L3 | bbT or bTb or Fbb |
| L15 | bbTRUEbHONESTbb |

For output, a T or F is written in the last character position of the field. The rest of the field is filled in with blanks.

Scale Factors - A scale factor may be used for E, F, and G conversions. A scale factor has the form:

nP

where n, the scale factor, is an integer constant or a minus sign followed by an integer constant. The scale factor precedes the three basic field descriptors. When execution of a format statement is begun, a scale factor of zero is assumed. Once a scale factor is encountered in a FORMAT statement, it holds for all remaining E, F, and G fields, in that FORMAT statement or until a new scale factor is encountered.

The effects of the scale factor are:

1. For F, E, and G input the scale factor has no effect if there is an exponent in the external field. Otherwise; for a scale of n:
external number = internal number x $10^n$

2. For F output with a scale factor of n
external number = internal number x $10^n$

3. For E output, the scale factor controls the decimal normalization between the number and the exponent so that:

   a. If $n \leq 0$, there will be |n| leading zeros followed by d significant digits to the right of the point.

   b. If $n \geq 0$, there will be exactly n significant digits to the left of the decimal point and d-n+1 to the right of the decimal point.

4. For G output, the scale factor is suspended unless E conversion is used.

The following chart provides some examples of scale factor effects.

| Internal | Format | External-No Scale | External with Scale |
|---|---|---|---|
| +4370 | 2PF12.4 | 4370.0 | b438000.0000 |
| 4370 | 2PE12.4 | bb0.4370E+04 | bb43.700E+02 |
| 4370 | -2PE12.4 | bb0.4370E+04 | b.004370E+06 |

### The Blank Field Descriptor

A blank field descriptor has the form:

wX

On input, w characters of the external record are skipped. On output, w blanks are written in the external record.

### The ASCII Field Descriptors

There are three forms of ASCII field descriptors, wH,'....',and Aw. The wH descriptor has the following effects:

1. On input, the next w characters are read as ASCII text into the w character positions following the H in the FORMAT statement.
2. On output, the w characters following the H in the FORMAT statement are written into the record.

For example, if we should wish to insert text in a record, the following field descriptor might appear in a FORMAT statement.

32HbTHISbISbAbSAMPLEb0FbOUTPUTbTEXT

Note that w, the character count, must include all characters, including blanks. The following example illustrates the means for altering textual information in a FORMAT statement.

```
      READ   100
       .
       .
       .
100   FORMAT (5Hbbbbb)
```

The field read by the READ statement would have to include exactly five characters of text including blanks. This text would replace the blanks in the format statement.

The '...' descriptor has the same effect as wH except that the text is embedded between the single quotes. When used as part of the text, the quote character appears twice in succession: DON'T is represented as 'DON''T'.

The Aw field descriptor causes ASCII characters to be read into, or written from, a specified list element. Since up to five ASCII characters can be stored in one memory word, the following rules apply to the Aw specification.

1. If on input w>5, only the rightmost five characters, including blanks, are read from the external field.
2. If on input w≤5, all w characters will be read and stored in memory left justified with 5-w trailing blanks to fill out the memory words.
3. If on output w>5, the five characters from the internal representation will be written in the last five characters positions of the external field. The leading character positions will be filled in by w-5 blanks.
4. If on output w≤5, the leftmost w characters of the internal representation will be written in the external field.

### Variable Field Input

The PDP-6 FORTRAN allows certain relaxation of the input formats for use in preparing input data. Namely, the E, F, G, I, and O field descriptors can appear without w and d. If so, numbers must be separated by an explicit delimiter. Such a delimiter may be any character which is illegal as the next character in the number representation.

Examples:

| FORMAT | External Field |
|---|---|
| (I) | -5+4/3,6-7/ |
| (I,E,F,O) | 9/.5E12/6.501/776/ |

## Variable Field Output

If the E, F, G, I, and O descriptors appear without w and d in a FORMAT specification for output, the field width w is set to 15. For the E, F, and G descriptors, d is set to 7. Thus the output FORMAT specification

```
10   FORMAT (E,F,G,I,O)
```

would automatically become

```
10   FORMAT(E15.7, F15.7,
     X G15.7, I15, O15)
```

## Repeat Count

The repeat count is an integer constant that specifies how many times a given field descriptor, or group of field descriptors enclosed in parentheses, is to be repeated. For example, consider the following equivalent FORMAT specifications:

(I2,I2,E10.4,E10.4,E10.4) ≡ (2I2,3E10.4)

(I2,I2,1HQ,F5.2,I2,1 HQ,F5.2,I2,1HQ,F5.2) ≡
(I2,3(I2,1HQ,F5.2))

When a repeat count and scale factor are both used, the general form is

n P r f

where f is the basic field descriptor. All basic field descriptors, except for wH and wX, may have a prefixed repeat count.

## Carriage Control

The first character of each printed line is interpreted as a control character for the line printer. The following table contains the special control characters which may be used as the first character in an Aw or wH field. Up to 120 characters may be printed on a line.

| Character | Effect |
|---|---|
| space | skip to next line |
| 0 | skip a line |
| 1 | form feed - go to top of next page |
| + | suppress skipping - will repeat line |
| - | skip 2 lines |
| 2 | skip to next 1/2 page |
| 3 | skip to next 1/3 of page |
| / | skip to next 1/6 of page |
| * | skip to next 1/10 of page |
| . | skip to next 1/20 of page |
| , | skip to next 1/30 of page |

## Slash

Besides being a field separator, the character / closes a record and starts a new one.

On output, a series of n+1 slashes will produce n blank records.

Example: Output according to the FORMAT statement.

```
FORMAT (9HbMATRIXbA//5H+SINE)
will produce
MATRIXbA
(blank line)
SINE
```

## Terminating FORMAT Statements

When the entire format has been used and the final right parenthesis is reached, the current

record is closed and the input or output list is examined for further entries. If no further entries are found, the data transfer is complete. If items remain, a new record is started and the FORMAT statement is repeated according to the following conditions:

1. If the FORMAT specification contains one or more repeat groups, begin with that group with the rightmost closing parentheses.

2. If the FORMAT specification has no repeat groups, return to the beginning of the specification.

# APPENDIX 1

# DIAGNOSTICS

The compilation process proceeds in two parts. First, the compiler translates the source language into an intermediate assembler language. Second, an assembler translates the intermediate language into an object language, the binary machine language. Therefore, there are two sets of diagnostics: those printed out by the compiler and those printed out by the assembler.

## COMPILER DIAGNOSTICS

| Error Message | Meaning | Steps for Correction |
|---|---|---|
| Parse Table Overflow | Statement too long | Break up the statement into one or more smaller statements. |
| Pushdown Depth Excessive | Same as above | Same as above |
| Rule Storage Overflow | The program can not be compiled since it generated too many rules. | Try reducing the number of variables in DIMENSION statements and/or don't nest the DOs so deeply. |

If the compiler cannot completely parse a statement, it will parse as much of the statement as it can and generate the corresponding code. It will indicate its failure to parse the statement completely by printing an up-arrow ↑ under the last character correctly parsed.

If the compiler cannot parse any of a given statement, it will print three up-arrows ↑↑↑ under the offending statement.

## ASSEMBLY DIAGNOSTICS

Assembly error flags consist of single characters printed in the left margin of the assembly listing of the compiler output.

| Error Flag | Explanation |
|---|---|
| C | Common Error - A variable has appeared more than once in a set of COMMON statements or is also a subroutine argument. |
| D | Dimension Error - A variable has appeared more than once in a set of DIMENSION statements. |

| Error Flag | Explanation |
|---|---|
| E | Equivalence Error - A subroutine argument has appeared in an EQUIVALENCE statement. |
| I | Equivalence Inconsistency - More than one variable in an equivalence group has appeared in another equivalence group. |
| S | Storage Assignment Error in EQUIVALENCE Statement - A variable in an equivalence group which has appeared in a previous equivalence group does not account for all the storage in the current group. |
| M | Multiple Symbols - A statement label has been used more than once. |
| U | Undefined Symbol - A statement label which has been referenced is missing, or an op-code which is incorrect. |
| O | Table Overflow - Symbol table; COMMON, DIMENSION, EQUIVALENCE table; or literals table has overflowed. |
| X | External Symbol Definition - A variable or array name is the same as a library function name. |
| R | Relocation Error - Illegal arithmetic involving relocatable symbols.* |
| = | Assignment Error - illegal use of =*. |
| N | Null Symbol - Use of illegal symbol structure such as label in assembly which does not begin with alphabetic character or % or.* |
| # | A constant is too large to fit in a PDP-6 machine word. |
| L | A statement is too long to be assembled correctly. |
| T | The statement label for the terminating statement of the current DO loop has already been processed. |

*Errors likely to occur in use of ASSEMBLE - COMPILE feature.

# APPENDIX 2

# SPECIAL PDP-6 FORTRAN II STATEMENTS

## DECTAPE INPUT-OUTPUT

The statements INFILE N, FILE and OUTFILE N, FILE provide a means for referencing DECtape in FORTRAN input-output statements. On output, files are created by issuing the OUTFILE statement. For example, the statement

```
OUTFILE 10, FILE1
```

causes the file name (in 7-bit ASCII) in FILE1 to be entered in the file directory on the DECtape referenced by device assignment 10. Subsequent to the OUTFILE statement, all output statements which reference device 10 will cause data to be written in file FILE1. The statement

```
END FILE 10
```

is necessary for closing the output, i.e., emptying buffers and completing the entry in the directory for file FILE1.

For input, the INFILE statement causes subsequent input statements to reference data in a particular file. The special tape statements, except for END FILE, are ignored in DECtape operations.

If INFILE or OUTFILE statements are not used, the file name FORTR. DAT will be assumed. Thus one file can be created automatically on any DECtape.

```
      :
      :
C     WRITE FILE1
        ANAME=5HFILE1
        OUTFILE 3, ANAME
        WRITE TAPE 3, (A(I),I=1,100)
        END FILE 3
      :
      :
C     WRITE FILE2
        BNAME=5HFILE2
        OUTFILE 3, BNAME
        WRITE TAPE 3, (B(I),I=100)
        END FILE 3
      :
      :
C     READ FILE1
        INFILE 3, ANAME
        READ TAPE 3, (A(I),I=1,100)
      :
      :
```

## TITLE

A program name may consist of up to six characters and is declared in a TITLE statement which has the form

```
TITLE NAME
```

The use of the TITLE statement is optional. If not used, the title .MAIN. is generated for main programs and subprogram names for subprograms. The title is essential for initiating use of a program's symbols in DDT. When used, the TITLE statement must be the first statement in the program.

## ASSEMBLE-COMPILE

The PDP-6 FORTRAN II compiler will accept MACRO-6 code directly when inserted between the statements ASSEMBLE and COMPILE.

The following components of MACRO-6 code are permissible:

1. All basic PDP-6 operation code mnemonics.
2. The pseudo-operations ASCII, ASCIZ, EXP, XWD, and BLOCK.
3. Literals containing
   a) digits 0-9
   b) the characters ., +, -, E
   c) the symbol ↑O for changing the radix to 8
   d) the pseudo-operations ASCII and SIXBIT.
4. The use of . to represent the value of the current location counter.
5. Storage may be allocated by the use of the character #.

NOTE: The radix for all instructions is 10 unless changed temporarily by ↑O.

The following restrictions apply to the above:

1. Terms in expressions may be combined only with +, -, and *.
2. Only non-relocatable terms may be combined by *.
3. Relocatable symbols may not be used in the left half of the XWD pseudo-operation.
4. Statement labels must begin with an alphabetic character or "%", "$" or ".". If a label starts with ".", the second character must be alphabetic. FORTRAN statement labels of the form XXX may be referred to by %XXX.

## ASSEMBLE-COMPILE Examples

```
1.  C     IF N IS NEGATIVE, GET RANDOM
          NUMBER IN M
          IF (N) 10, 20, 20
          ASSEMBLE
    %10   MOVE 3, [↑O1425364750 76]
          ADD 3, RAN
          ROT 3, -1
          EQVB 3, RAN
          MOVEM 3, M#
          JRST %11
    RAN   ↑O123456707654
          COMPILE
    11    DO 12 I=1,100
            :

2.  C     REPLACE THE STATEMENTS:
    C     DO 10 I=1,M
    C     DO 10 J=1,N
    C     10 A(I,J) = B(I,J)*A(I,J)
    C     AND EXECUTE IN ACCUMULATORS
          DIMENSION A(10,20), B(10,20),
          C(10,20)
          ASSEMBLE
          MOVEM 2, TWOSAV#; SAVE AC 2
          HRLI 2, BEGN; Set up block transfer
                        ; in AC 2
          HRRI 2, 6
          BLT 2, 17     ; Move code to AC
                        ; 6-AC 17
          JRST 6
                        ; The following block is
                        ; executed in accumu-
                        ; lators 6-17
    BEGN  MOVE 2,M      ; OUTER LOOP
    MLUP  MOVE 3,N      ; INNER LOOP
          MOVE 4,J
          IMULI 4,10
          ADD 4,I
          MOVE 5, B-11 (4)  ;B(I,J)
          FMPRM 5, A-11 (4) ;A(I,J)=B(I,J)*A(I,J)
          SOJG 4, NLUP
          SOJG 3, MLUP
          JRST %20      ; RETURN FROM AC 17
    %20   MOVE 2, TWOSAV
          COMPILE
            :
```

## ASSEMBLE-COMPILE Format

### Cards

On cards, statement labels for assembly code are limited to four characters and must be punched in columns 1-5. The instruction and comments follow in columns 7-72 with the format

OPCODE AC, ADDR (IR) ;COMMENT

The fields for the operation, accumulator, address, index register, and comments are not fixed and may fall anywhere in columns 7-72.

The ASSEMBLE and COMPILE statements must appear without labels.

### Punched Tape

1. Card simulated format

   Tabs may be used to skip to column 7 and also to delimit fields within an instruction.

2. Column free format

   Labels must be followed by a colon.

# APPENDIX 4

# SUMMARY OF STATEMENTS

## CONTROL STATEMENTS

ASSIGN n TO iname
CONTINUE
DO label i = k, l, m
GO TO n
GO TO iname
GO TO iname, (label1, label2,...labeln)
GO TO (label1, label2,...labeln), iname
IF (exp) label1, label2, label3
B IF (exp) label1, label2
IF ACCUMULATOR OVERFLOW label1, label2
IF QUOTIENT OVERFLOW label1, label2
IF (SENSE LIGHT i) label1, label2
IF (SENSE SWITCH i) label1, label2
PAUSE i
PAUSE "MESSAGE"
SENSE LIGHT i
STOP

## INPUT-OUTPUT STATEMENTS

### Input

ACCEPT label, list
READ label, list
REREAD label, list
READ INPUT TAPE n, label, list
REREAD INPUT TAPE n, label, list
READ TAPE n, list
RIT n, label, list

### Output

PRINT label, list
PUNCH label, list
TYPE label, list
WRITE OUTPUT TAPE n, label, list
WOT n, label, list
WRITE TAPE n, list

### Tape Commands

BACKSPACE n
END FILE n
REWIND n
SKIP RECORD n
UNLOAD n

## SPECIFICATION STATEMENTS

COMMON $v_1, v_2, \ldots v_m$
DIMENSION $a_1, a_2, \ldots a_n$
EQUIVALENCE (list1), (list2),...(listn)

## SUBPROGRAM STATEMENTS

CALL name $(a_1, a_2, \ldots a_n)$
FUNCTION name $(d_1, d_2, \ldots d_n)$
RETURN
SUBROUTINE name $(d_1, d_2, \ldots d_n)$

# APPENDIX 5

## FORTRAN II OPERATING SYSTEM

1. Device Assignments

Logical device assignments for run-time I/O are made with the use of a table called DEVTB. in the FORTRAN library. Each entry in the table consists of a 6-bit ASCII device name, and the numerical position of each entry corresponds to the logical number used in FORTRAN I/O statements. The first location of DEVTB. contains the number of entries in the table. The last five entries in the table are special and correspond to the FORTRAN statements READ, ACCEPT, PRINT, PUNCH, and TYPE. Any entry in DEVTB. may be changed by reassembling the table.

Example:

```
DEVTB.:  ↑D13
         SIXBIT   .DTA0.
         SIXBIT   .TTY.
         SIXBIT   .CDR.
         SIXBIT   .LPT.
         SIXBIT   .MTA0.
         SIXBIT   .MTA1.
         SIXBIT   .DTA1.
         SIXBIT   .DTA2.
         SIXBIT   .CDR.    ;READ
         SIXBIT   .TTY.    ;ACCEPT
         SIXBIT   .LPT.    ;PRINT
         SIXBIT   .PTP.    ;PUNCH
         SIXBIT   .TTY.    ;TYPE
         END
```

With this table, the statement WRITE OUTPUT TAPE 6 would refer to magnetic tape unit 1. The PRINT statement would refer directly to the line printer.

2. Special Library Programs

a. EXIT

A call EXIT statement causes a run to be terminated. All I/O devices are released from the job.

b. PDUMP, DUMP

CALL PDUMP ($A_1$, $B_1$, $F_1$, $A_2$, $B_2$, $F_2$, ...) and CALL DUMP ($A_1$, $B_1$, $F_1$, $A_2$, $B_2$, $F_2$, ...) are statements which cause portions of core to be dumped on the device corresponding to the PRINT statement. If no arguments are present, the entire user core area is dumped in octal. The argument list ($A_1$, $B_1$, $F_1$, ...) displays the arguments for the dump and the mode in which the dump is to take place. Core is dumped between the limits $A_i$ and $B_i$ in the mode $F_i$. Either $A_i$ or $B_i$ may be upper or lower limits.

The modes are

0. Octal
1. Floating Point
2. Integer
3. ASCII

If the final mode is missing, the core area between $A_n$ and $B_n$ is dumped in octal.

If the last two arguments $B_n$ and $F_n$ are missing, an octal dump is made from $A_n$ to the end of the user's area.

DUMP calls EXIT while PDUMP returns control to the calling program when the dump has been completed.

c. CHANG

The statement A = CHANGF(X) causes X to be changed from a sign-magnitude negative number to a 2's complement negative number (or vice versa). If X is positive, CHANG has no effect on X.

3. Error Messages

All errors which are detected by the Operating System result in terminating the run with a CALL EXIT. The errors detected are:

a. Illegal character in FORMAT statement

b. End of file on input

c. Illegal character in input string

d. Device not available

e. Illegal FORTRAN device number

f. Too many devices referenced (15 allowed)

g. File name not found in a DECtape directory

h. DECtape directory full

i. Tape record too short for list specification

j. Device error or tape parity error

k. End of file while reading binary file

l. End of tape

For more detailed information about FORTRAN I/O, see DEC-06-0-OS-FII-GM-FP-ACT00-FORTRAN II Format and I/O Processor.

# APPENDIX 6

## PDP-6 FORTRAN II COMPILER OPERATING INSTRUCTIONS

The PDP-6 FORTRAN II Compiler contains two basic sections: a compiler which generates assembly code from the FORTRAN source statements and an assembler which generates relocatable binary programs. The 22K compiler contains both parts in one program. The 9K compiler, however, prepares an intermediate file for input to the assembler which is a separate program (FOLA).

### COMMAND STRING FOR 22K COMPILER

The command string is used to specify the input and output file designations for the compiler. The 22K compiler expects up to two output files and one input file. The general form of the command string is

FILE1, FILE2←— FILE3

FILE1 will contain the relocatable binary output; FILE2 will contain the listing of the compiled output (source, assembly, binary, errors), and FILE3 is the source or input file. Each file may have one of the following forms:

DEVICE:
DEVICE: FILENAME
DEVICE: FILENAME.EXTENSION

where DEVICE may be any device mnemonic acceptable to the PDP-6 executive system, FILENAME may be up to six letters and/or digits, and EXTENSION may be up to three letters and/or digits.

The file name extensions REL and LST are assumed for FILE1 and FILE2 unless specified otherwise.

Example:

PTP: ,DTA3:LIST ←—DTA1:SORC.TXT

If FILE1 is not desired, the command string should be of the form:

,FILE2 ←— FILE3

If FILE2 is not desired, either of the following command strings is valid:

FILE1 ←—FILE3
FILE1, ←— FILE3

If neither output file is desired, the valid command strings are:

←— FILE3
, ←—FILE3

### SWITCHES FOR THE 22K COMPILER

Switches are letters specifying optional and extra functions to be performed by the compiler. These letters may appear within parentheses or after a forward slash. Only a single letter may follow a slash, while more than one letter may appear within parentheses. The switches are as follows:

- K Skip one file on the device (magnetic tape only).
- M Do not print storage map.
- N Do not list errors on Teletype console if there is a listing file.

R The source contains rules for the compiler (for building a new compiler).

S List only source and errors (no assembly code).

T The source is in column-free punched tape format.

W Rewind the device (magnetic tape only).

Z Clear the directory on the device before inserting the new file.

Any switches not recognized are ignored.

The switches M, N, R, S, T may appear anywhere in the command string. K, W and Z must appear before the termination character of the applicable file.

For example, the command string

(W)MTA0:, (Z)DTA2: LIST ←— CDR: /M

calls for binary output on magnetic tape 0 (rewind first), listing on DECtape 2 (clear directory) and source from the card reader. The storage map is to be deleted from the listing.

Command String for the 9K Compiler: The command string for the 9K Compiler is similar to that for the 22K Compiler except that there is no binary file. However, two identical output files are permitted (for example a listing of input to the assembler FOLA). The general forms of the command string are

FILE1, FILE2 ←— FILE3
FILE1 ←— FILE3

The file name extension for the output files are assumed to be FOL unless otherwise specified.

Switches for the 9K Compiler: The switches K, R, T, W, and Z (as described for the 22K compiler) are recognized by the 9K Compiler.

Example:

```
DTA1:FOLAIN, LPT: ←— PTR: /T
```

This command string calls for identical output files (input to FOLA) to be written on DECtape 1 and the line printer. The input (punched in column-free format) is coming from the paper tape reader.

Command String for FOLA: The form of the FOLA command string is identical to that of the 22K compiler with the exception that more than one input file may be specified:

FILE1, FILE2 ←— FILE3, FILE4, ...

Switches for FOLA: The switches K, M, N, S, and W are recognized by FOLA. M, N, S, and T must appear before the termination character for the applicable file and must appear for each file for which they are intended.

Example:

```
PTP:,(K)MTA1: ←— CDR:, (M)TTY:
```

In this example, the binary is to be punched on paper tape, the listing is to go on magnetic tape 1 (after skipping one file), and the input is to come from the card reader and the teletype. The storage map is not to appear on the listing for the input file from the teletype.

# APPENDIX 7

## LIMITATIONS ON 9K FORTRAN II COMPILER

1. Boolean statements are not allowed.
2. Use of * in IF statements is not allowed.

digital
EQUIPMENT
CORPORATION
MAYNARD, MASSACHUSETTS

Cambridge, Mass. • Washington, D. C. • Parsippany, N.J. • Rochester, N.Y. • Los Angeles
Palo Alto • Chicago • Ann Arbor • Pittsburgh
Denver • Huntsville • Orlando • Carleton Place
and Toronto, Ont. • Reading, England • Paris,
France • Munich, Germany • Sydney, Australia