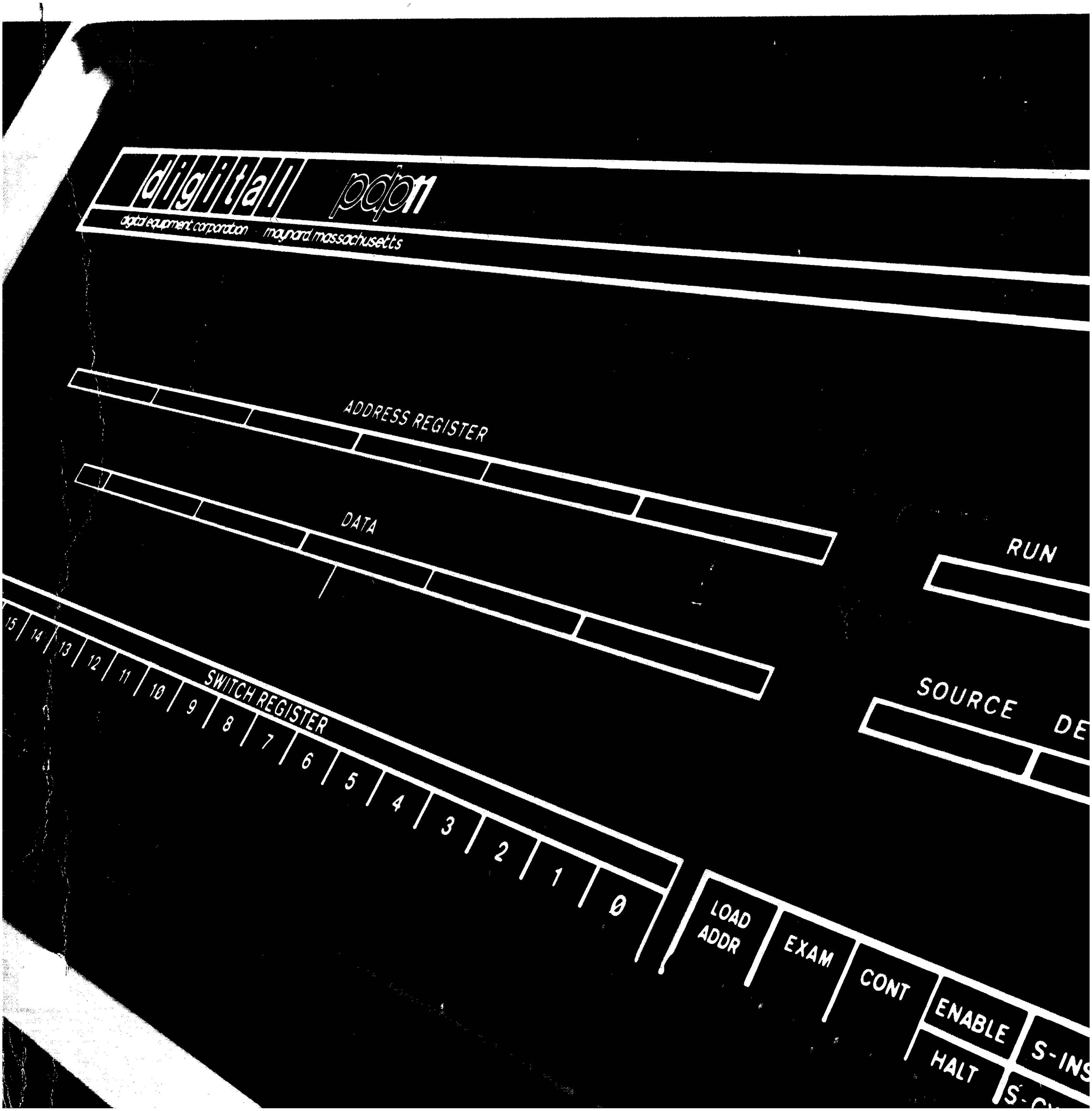
digital

pdp11

# Fortran IV
## programmer's manual



DEC-11-KFDA-D

# PDP-11
# FORTRAN IV

## COMPILER and OBJECT TIME SYSTEM

## Programmer's Manual

For additional copies, order No. DEC-11-KFDA-D from Digital Equipment Corporation, Direct Mail, Bldg. 1-1, Maynard, Massachusetts 01754

First Printing, May 1971

Your attention is invited to the last two pages of this document. The "How To Obtain Software Information" page tells you how to keep up-to-date with DEC's software. The "Reader's Comments" page, when filled in and mailed, is beneficial to both you and DEC; all comments received are considered when documenting subsequent manuals.



Associated documents:

Disk Operating System Monitor, Programmer's Handbook, DEC-11-MWDA-D
PAL-11R Assembler, Programmer's Manual, DEC-11-ASDB-D
Link-11 Linker and Libr-11 Librarian, Programmer's Manual, DEC-11-ZLDA-D



Acknowledgment

This manual was prepared by the Documentation Department of Cambridge Computer Associates, Inc. for Digital Equipment Corporation.

# PREFACE

PDP-11 FORTRAN IV is part of the PDP-11 Disk Operating System. For the convenience of the FORTRAN programmer and the operator actually concerned with compiling the FORTRAN program, the manual is separated into two distinct parts:

Part I - The PDP-11 FORTRAN IV Language
Part II - The FORTRAN Operating Environment

The Index is also separated into two parts.

Any configuration that supports the DOS will support FORTRAN. The reader of this manual is expected to have some familiarity with FORTRAN programming.

MASTER CONTENTS

MASTER CONTENTS (Cont)

Page

# Part I

# THE PDP-11 FORTRAN IV LANGUAGE

PART I
TABLE OF CONTENTS

## ILLUSTRATIONS



## TABLES

# CHAPTER 1
# INTRODUCTION

The following chapters describe the FORTRAN IV (FORmula TRANslation) language, a problem-oriented language designed to permit scientists and engineers to express a computation in notation with which they are familiar. A FORTRAN source program is composed of statements in an easy to read form. Commands are descriptive of the functions they perform, and computational elements are expressed in a notation similar to that of standard mathematics. The source program is compiled by the FORTRAN IV compiler into code which is subsequently assembled by the PAL-11 Assembler Program. The resultant program runs in conjunction with the FORTRAN Object Time System described in Part II of this manual. Note that there is not a one-to-one correspondence between a FORTRAN statement and a machine-language instruction. Many statements will result in several machine instructions while others will yield none. The latter type, non-executable statements, provide information to the compiler on how to interpret other elements of the source program.

## 1.1 LANGUAGE COMPONENTS

The basic unit of expression in FORTRAN is the statement. A statement consists of a command portion which characterizes the statement's function and, as required, arguments upon which the command operates. A statement may be numbered for reference by other statements. The argument of a command may be data values upon which the program is to operate. These may be expressed explicitly (constants) or symbolically (variables). Using these primary units together with FORTRAN operators, the programmer may construct expressions to derive new values by combining known values.

The character set from which FORTRAN statements may be constructed is given below.

| | | | |
|---|---|---|---|
| The letters A-Z | | * | Asterisk |
| The digits 0-9 | | / | Slash |
| | Blank | ( | Left parenthesis |
| = | Equals | ) | Right parenthesis |
| + | Plus | , | Comma |
| - | Minus | . | Decimal point |
| | | $ | Currency symbol |

Other characters may appear only within a Hollerith constant (see Section 2.1.7) text string.

FORTRAN statements fall into five categories according to their functions. Arithmetic statements are used to assign values to variables. Control statements are used to govern the sequence in which program statements are executed. Data transmission statements govern the transfer of information between the computer and peripheral devices. Specification statements provide the compiler with information about data the compiled program will process. Subprogram statements are used to define subprograms.

## 1.2 PROGRAM STRUCTURE

A FORTRAN program is a sequence of statements. The end of the program is signified by the characters END. Control originates at the first executable statement and continues in sequence unless explicitly transferred by the occurrence of a control statement.

Non-executable statements must appear before the executable portion of the program. The one exception to this rule is the FORMAT statement (described in Section 5.1.1).

A statement is composed in lines; that is, a series of characters terminated by a line feed. Although most source programs for the PDP-11 FORTRAN compiler will be prepared using the EDIT-11 program, a line generally conforms to the format described below for punched card input.

A line is divided into three fields - the statement number field (columns 1-5), the line continuation field (column 6), and the statement field (columns 7-72). For non-card input, the appropriate number of spaces may be typed, or the character TAB which will automatically advance to the appropriate field. Columns 73-80, which are ignored by the FORTRAN compiler, may be used for any purpose, for example, for sequence or identification numbers.

The statement number is optional. If supplied, it must be a number greater than zero, composed of 1 to 5 digits of any value, placed anywhere within the field. Leading zeros are ignored. Statement numbers may be assigned in any order since the sequence of operations is dependent on the order of the statements rather than the value of their numbers. They must, however, be unique.

The line continuation field is used only when a statement requires more than one line. Additional lines (up to a maximum of five) are indicated by the appearance of any character other than blank or zero in column 6. If a TAB is used rather than spacing, continuation lines are assumed when a numeric character follows the TAB. The end of a line is indicated by a line feed.

The statement field contains a FORTRAN statement (or portion thereof). Blanks which appear within a statement will be ignored with the exception of alphanumeric data appearing in a FORMAT statement, in a DATA statement, or in a Hollerith constant.

A comment line, denoted by a C in column 1 (first character), may appear anywhere in the source program. Comment text may then appear anywhere in columns 2-72.

# CHAPTER 2
# EXPRESSING DATA VALUES

Data values in a FORTRAN program may be represented by the primary units - constants and variables - or by expressions. Expressions are composed of primary units and operators which indicate operations to be performed on their values.

## 2.1 CONSTANTS

A constant is a value used by the object program which does not change from one execution of the program to another. Six types of constants are permitted in a FORTRAN IV source program: integer or fixed point, real or single-precision floating point, double-precision floating point, complex, logical, and Hollerith.

### 2.1.1 Integer Constants

An integer constant is a string of from one to five decimal digits written without a decimal point. A negative integer may be indicated by a preceding minus sign. A positive integer may be preceded by an optional plus sign.

Examples:

```
3
+10
-528
8085
```

An integer constant must fall within the range $-2^{15}$ to $2^{15}-1$.

### 2.1.2 Real Constants

A real constant is a string of decimal digits which includes a decimal point. A real constant may consist of any number of digits but only the leftmost eight digits not including leading zeros are used by the compiler.

A real constant may be followed by a decimal exponent, represented by the letter E followed by a signed integer constant. The field following the letter E must not be blank, but may be zero.

Examples:

```
 15.
  0.0
   .579
-10.794
  5.0E3  (i.e., 5000.)
  5.0E+3 (i.e., 5000.)
  5.0E-3 (i.e., 0.005)
  5.0E0 (i.e., 5.0)
```

A real constant has precision to 24 bits or about seven decimal digits. The magnitude must lie approximately within the range $0.14 \times 10^{-38}$ to $1.7 \times 10^{38}$. Real constants occupy two words of PDP-11 storage.

### 2.1.3 Double-Precision Constants

A double-precision constant may consist of any number of decimal digits, but only the leftmost fifteen digits, not including leading zeros, are used by the compiler. It is specified by a string of decimal digits, including a decimal point, which is followed by the letter D and a signed integer constant. The field following the letter D must not be blank, but may be zero.

Examples:

```
24.671325982134D0
 3.6D2 (i.e., 360.)
 3.6D-2 (i.e., .036)
 3.0D0
```

The magnitude of a double-precision constant must lie approximately between $0.14 \times 10^{-38}$ and $1.7 \times 10^{38}$. Double-precision constants occupy four words of PDP-11 storage.

### 2.1.4 Octal Constants

An octal constant is a string of from one to six octal digits (only the digits 0-7 may be used) preceded by the letter O.

Examples:

```
O120
O0
O177777
```

An octal constant is valid only in the context of three statements - DATA, PAUSE, and STOP. The maximum value which may be expressed as an octal constant is 177777.

### 2.1.5 Complex Constants

FORTRAN IV permits direct operations on complex numbers. A complex constant is written as an ordered pair of real constants separated by a comma and enclosed in parentheses.

Examples:

```
(.70712,-.70712)
(8.763E3,2.297)
```

The first constant of the pair represents the real part of the complex number, and the second constant represents the imaginary part; each may be signed. The enclosing parentheses are part of the constant and always appear, regardless of context. The two parts are each internally represented by one single-precision floating point value occupying consecutive locations of PDP-11 storage.

### 2.1.6 Logical Constants

The two logical constants, represented in the source language as .TRUE. and .FALSE., have the internal integer values -1 and 0,* respectively. These values may be entered, via DATA or input statements, as TRUE and FALSE. Logical quantities may be operated upon both by arithmetic and logical operators.

### 2.1.7 Hollerith Constants

A Hollerith constant is a string of characters. There are two forms by which a Hollerith constant may be represented.

| | |
|---|---|
| Form 1: | nH character string |
| Where: | n is the number of characters |
| Examples: | 5HWORDS<br>3H123 |
| Form 2: | 'character string' |
| Examples: | 'WORDS'<br>'123' |

The single quote character which delimits a Hollerith constant in Form 2 may be included in the character string if immediately preceded by a single quote character. Thus, 'DON''T' will be stored as DON'T.

*The value -1 is equivalent to the octal number 177777.

A Hollerith value may be entered in a DATA statement or input statement as a string of one or two ASCII characters per integer variable, one to four per real variable, and one to eight per complex or double-precision variable.

Hollerith constants are stored in memory as byte strings. The constants will always fill up to word boundaries. If a Hollerith constant is specified with an odd number of characters, a blank will be appended to the right-hand end of the constant.

Example: the constant 5HABCDE , stored at location $20000_8$ in memory, would look like this:

| 20001 | 20000 |
|---|---|
| B | A |

| 20003 | 20002 |
|---|---|
| D | C |

| 20005 | 20004 |
|---|---|
| (blank) | E |

Hollerith constants are used in different ways depending on context. See Paragraph 2.3.1 and especially Table 2-1 for detailed information on the effect of Hollerith constants in arithmetic expressions.

## 2.2 VARIABLES

A variable is a quantity which is represented by a symbolic name. The value of a variable may change during the execution of a program. A variable name is a string of from one to six characters, the first of which must be alphabetic. Variable names longer than six characters are rejected by the compiler.

Examples:

| Valid Names | Invalid Names |
|---|---|
| ALPHA | 2A |
| MAX | MAXIMUM |
| A34 | |

A variable has a principal attribute-type. The variable's type indicates the type of value it may be assigned (integer, real, logical, double-precision, or complex). Type is assigned to a variable via an explicit type declaration statement (6.3), implicitly via an IMPLICIT statement (6.4), or, if neither of these methods is used, by virtue of the initial letter of its name. I, J, K, L, M, or N indicate type integer (fixed point). All other letters indicate type real (floating point).

The extent of a variable refers to the extent of the values which may be referred to by a single name. A scalar variable represents a single quantity.

An array variable represents an element of an array, an ordered set of data of one, two, or three dimensions. An entire array is identified by its name; an element of the array is identified by the subscripted array name.

Up to three levels of subscripting may be given for an array variable.

Examples:

| Variable | Refers to |
|---|---|
| ARRAY (1) | An element of one-dimensional array ARRAY. |
| MAT (1,2,3) | An element of the three-dimensional array MAT. |

The subscripts of an array variable may be integer or floating point constants or expressions. Floating point subscripts will be converted to integers before use.

An array variable's extent is determined by the dimensions it is assigned. This may be done by a DIMENSION or COMMON statement or as part of a type-declaration statement. Array dimensioning is discussed in Chapter 6.

## 2.3 EXPRESSIONS

An expression is a combination of primary units (constants and variables) with operators which specify a computation to be performed to obtain a new value. An expression may, itself, function as a primary unit in another expression if it is enclosed in parentheses.

### 2.3.1 Arithmetic Expressions

An arithmetic expression is a combination of constants, variables, and expressions separated by the arithmetic operators given below.

| Operator | Operation |
|---|---|
| - | unary minus |
| ** | exponentiation |
| * | multiplication |
| / | division |
| + | addition |
| - | subtraction |

Additional computations (such as sine, cosine, square root) may be specified via a function reference (see Chapter 7 for a description of function definition). A function reference acts as a basic element in an expression since all functions return a single value. The reference SQRT(4.) (assuming the existence of a function named SQRT which returns the square root of its argument) represents the value 2. in an expression.

An arithmetic expression need not have operators at all but may simply be a basic element. Thus,

```
2.718
Z(N)
MAX
```

are all legal expressions.

Any numeric expression may be enclosed in parentheses and considered to be a basic element.

```
(X+Y)/2
(ZETA)
(COS(SIN(PI*M)+X))
```

Numeric expressions which are preceded by a + or - sign are also numeric expressions:

```
+X
-(ALPHA*BETA)
-SQRT(-GAMMA)
```

If the precedence of numeric operations is not given explicitly by parentheses, it is understood to be the following (in order of decreasing precedence):

| Operator | Explanation |
|---|---|
| ** | numeric exponentiation |
| * and / | numeric multiplication and division |
| + and - | numeric addition and subtraction |

In the case of operations of equal hierarchy, the calculation is performed from left to right.

No two numeric operators may appear in sequence. For instance:

```
X*-Y
```

is improper. Use of parentheses yields the correct form:

```
X*(-Y)
```

A typical numeric expression using numeric operators and a function reference, the expression for one of the roots of the general quadratic equation

$$\frac{-b+\sqrt{b^2-4ac}}{2a},$$

would be coded as:

```
(-B+SQRT(B**2-4.*A*C))/(2.*A)
```

Any type of quantity (logical, integer, real, double-precision, complex) may be combined with any other in an arithmetic expression. The type of resultant expression when any two types are combined may be found in Table 2-1 on the following page.

Logical, octal and Hollerith (literal) constants are treated as integer constants when they are combined with other elements in arithmetic expressions. Data in a Hollerith constant beyond its first 16 bits (2 characters) is ignored.

Example:

```
I = 1
J = I +'A B C D'
K = I*.TRUE.
```

J will contain the result of adding 1 to the word whose low order byte is a 101(A) and whose high order byte is a 102(B). The result is an octal 041102 or the ASCII 'BB'. K will contain a -1, since the value of .TRUE. taken as an integer is -1.

In mixed-mode expressions the logical, octal, or Hollerith entity will be converted as an integer to the appropriate mode and then combined.

### 2.3.2 Logical Expressions

A logical expression combines logical constants, logical variables, logical function references, and arithmetic expressions, using the logical or relational operators given below.

| Logical Operator | Meaning |
|---|---|
| .NOT. expression | Has the value .TRUE. only if expression is .FALSE., and has the value .FALSE. only if expression is .TRUE. |

(Continued on next page)

| Logical Operator | Meaning |
|---|---|
| expr1.AND.expr2 | Has the value .TRUE. only if expr1 and expr2 are both .TRUE., and has the value .FALSE. if either expr1 or expr2 is .FALSE. |
| expr1.OR.expr2 | (Inclusive OR) Has the value .TRUE. if either expr1 or expr2 is .TRUE., and has the value .FALSE. only if both expr1 and expr2 are .FALSE. |

| Relational Operator | Relation |
|---|---|
| .GT. | greater than |
| .GE. | greater than or equal to |
| .LT. | less than |
| .LE. | less than or equal to |
| .EQ. | equal to |
| .NE. | not equal to |

Table 2-1
Types of Resultant Subexpressions

| +,-,*,/ | | Type of Quantity | | | | |
|---|---|---|---|---|---|---|
| | | Real | Integer | Complex | Double Precision | Logical |
| Type of Quantity | Real | Real | Real | Complex | Double Precision | Real |
| | Integer | Real | Integer | Complex | Double Precision | Integer |
| | Complex | Complex | Complex | Complex | Complex | Complex |
| | Double Precision | Double Precision | Double Precision | Complex | Double Precision | Double Precision |
| | Logical | Real | Integer | Complex | Double Precision | Logical |

NOTE: the following special rules apply for determining the type resulting from expressions of the form A**B:

if B is type INTEGER, the expression is of the same type as A
if A and B are both REAL, the expression is REAL
if A or B, or both A and B, are double-precision, the expression is double-precision.

These are the only cases allowed.

Logical operators can combine only basic elements whose type is LOGICAL (see Chapter 6). Relational operators compare units of type integer, real, or double-precision. Real and double-precision units may be combined. The value of such an expression will be of type LOGICAL (that is, .TRUE. or .FALSE.). The relational operators .EQ. and .NE. may also be used with complex expressions. (Complex quantities are equal if the corresponding parts are equal.)

A logical expression, like an arithmetic expression, may consist of basic elements or a combination of elements, as in

```
.TRUE.
X.GE.3.14159
```

and

```
TVAL.AND.INDEX
BOOL(M).OR.K.EQ.LIMIT
```

A logical expression may also be enclosed in parentheses and function as a basic element. Thus, the expressions

```
A.AND.(B.OR.C)
```

and

```
(A.AND.B).OR.C
```

are evaluated differently.

No two logical operators may appear in sequence, except in the case where .NOT. appears as the second of two logical operators. Any logical expression may be preceded by the unary operator .NOT. as in:

```
.NOT.T
.NOT.X+7.GT.Y+Z
BOOL(K).AND..NOT.(TVAL.OR.R)
```

Logical and relational operations (unless overridden by parentheses) are carried out in the following order:

.GT.,.GE.,.LT., .LE.,.EQ.,.NE.

.NOT.

.AND.

.OR.

For example, the logical expression

```
.NOT.ZETA**2+Y*MASS.GT.K-2.OR.PARITY.AND.X.EQ.Y
```

is interpreted as

```
(.NOT.(((ZETA**2)+(Y*MASS)).GT.(K-2))).OR.(PARITY.AND.(X.EQ.Y))
```

# CHAPTER 3
# ASSIGNMENT STATEMENTS

A variable may be assigned a value at any point in the source program. During program execution, the most recent assignment determines the variable's value in subsequent statements. There are two statements which may be used to assign a value to a variable - the Arithmetic statement which assigns a numeric or logical value and the ASSIGN statement which assigns a statement number.

## 3.1 THE ARITHMETIC STATEMENT

| | |
|---|---|
| Form | A = B |
| Where | A is a variable name<br>B is an expression<br>= is the replacement operator |
| Effect | The variable named A is assigned the value of expression B. |

The Arithmetic statement associates a variable name with a value. The name may then be used in subsequent expressions to represent this value. Thus, if the Arithmetic statement A = 2 is executed first, the statement B = A + 1 is equivalent to the statement B = 3.

Since the equal sign in an Arithmetic statement does not indicate equality but, rather, a replacement, statements of the form

```
I = I + 1
```

are perfectly legal. The Arithmetic statement is, in fact, the only means in FORTRAN by which the results of computations represented by expressions may be stored.

In the following examples, the expression to the right of the equal sign is evaluated and converted when necessary to conform to the type of the variable to the left before assignment. That is, if a real expression is assigned to an integer variable, the value of the expression will be converted to an integer before assignment.

Examples:

```
ANS = Y*(X**2+Z)
I = I*N
X(J) = A(J)-B(J)
P = .TRUE.
```

The expression to be assigned must be capable of yielding a value which conforms to the type attribute of the variable which is being assigned. The compiler will perform conversions in accordance with Table 3-1 below.

Table 3-1
Conversion Rules for Assignment Statements

| Variable Type | Expression Type | | | | | |
|---|---|---|---|---|---|---|
| | Real | Integer | Complex | Double Precision | Logical, or Octal Constant | Literal Constant |
| Real | D | C | R,D | H,D | C | D,4 |
| Integer | C | D | R,C | H,C | D | D,2 |
| Complex | D,R,I | C,R,I | D | H,D,R,I | D,R,I | D,8 |
| Double Precision | D,H,L | C,H,L | R,D,H,L | D | D,H,L | D,8 |
| Logical | C | C | R,C | H,C | D | D,2 |

D - Direct replacement
C - Conversion between integer and floating point
R - Real only (imaginary part set to 0)
I - Set imaginary part to 0
H - High order portion of expression assigned
L - Set low order part to 0
2 - Use the first character in the literal and one character following
4 - Use the first character in the literal and three characters following
8 - Use the first character in the literal and seven characters following

## 3.2 THE ASSIGN STATEMENT

| | |
|---|---|
| Form | ASSIGN _n_ TO _var_ |
| Where | _n_ is a statement number<br>_var_ is a variable of type INTEGER |
| Effect | The variable represents the assigned statement number and may be used in an assigned GO TO statement (Chapter 4). |

The ASSIGN statement is used in conjunction with an assigned GO TO statement (4.1.3) to permit symbolic referencing of statements. The statement number assigned must be that of an executable statement. An integer variable which has obtained its value via an ASSIGN statement must be redefined via an Arithmetic statement before it can be used in any context other than the GO TO statement. For example, the statement:

```
ASSIGN 10 TO COUNT
```

associates the variable name COUNT with statement number 10 and the statement:

```
COUNT = COUNT+1
```

is invalid. The statement becomes valid, however, if preceded by the statement:

```
COUNT = 10
```

which assigns count the integer value of 10.

# CHAPTER 4
# CONTROL STATEMENTS

Statements are normally executed in the sequence in which they appear in the source program. This sequence may be altered by the occurrence of any of the FORTRAN control statements described in this chapter. These are: GO TO, IF, DO, CONTINUE, PAUSE, STOP, CALL and RETURN. The CALL and RETURN statements, which transfer control to and from subroutines, are described in Chapter 7.

## 4.1 THE GO TO STATEMENT

The GO TO statement transfers control directly to a specified statement. There are three forms of the GO TO statement - unconditional, computed, and assigned. A GO TO statement may appear anywhere in the executable portion of the source program except as the terminal statement in a DO loop (4.3).

### 4.1.1 Unconditional GO TO Statements

| | |
|---|---|
| Form | GO TO n |
| Where | n is the statement number of an executable statement |
| Effect | Control is transferred to statement n. |

When control is transferred by a statement of the form GO TO n, the usual sequential processing continues at the statement whose number is n.

### 4.1.2 Computed GO TO Statements

| | |
|---|---|
| Form | GO TO $(n_1,n_2, \ldots, n_k)$ i<br><br>NOTE: An optional comma may follow the right parenthesis. |
| Where | $n_1, n_2, \ldots, n_k$ are statement numbers<br>i is an integer variable or constant |
| Effect | Control is transferred to the statement whose number is ith in the list. |

The integer expression in a computed GO TO statement acts as a switch, as in the example given below.

```
GO TO (20,10,5),K
```

If K = 1, control will be transferred to statement 20; if K = 2, to statement 10; or if K = 3, to statement 5. If K has a value less than 1 or greater than 3 in this example, an error will be reported when the program is executed.

### 4.1.3 Assigned GO TO Statements

| | |
|---|---|
| Form | GO TO K<br>or<br>GO TO K $(n_1,n_2, \ldots, n_k)$<br><br>NOTE: An optional comma may follow K. |
| Where | K is an integer variable<br>$n_1,n_2, \ldots, n_k$ are statement numbers |
| Effect | Control is transferred to the statement whose number is currently associated with the variable K via an ASSIGN statement. |

An ASSIGN statement, as discussed in Chapter 3, defines an integer variable as a statement number. Thus, when the statement

```
ASSIGN 10 TO LOOP
```

has been executed, the programmer may subsequently transfer control to statement 10 by saying:

```
GO TO LOOP
```

He may also say:

```
GO TO LOOP, (10, 20, 100)
```

which will transfer control to whichever statement number is currently associated with LOOP. If the name LOOP is not defined as one of the listed statement numbers, the GO TO statement will not be executed and an error message will be printed.

## 4.2 THE IF STATEMENT

An IF statement causes control to be transferred on the basis of the values of specified expressions. There are two forms of the IF statement - arithmetic and logical.

### 4.2.1 Arithmetic IF Statements

| | |
|---|---|
| Form | IF (arithmetic expression) $n_1$, $n_2$, $n_3$ |
| Where | $n_1$, $n_2$, $n_3$ are statement numbers |
| Effect | Control is transferred to:<br>$n_1$ if expression $<0$<br>$n_2$ if expression $= 0$<br>$n_3$ if expression $>0$ |

An IF statement transfers control to one of three statements, as shown in the model, according to the value of the expression given. For example, the statements:

```
ALPHA = 3
.
.
.
IF (ALPHA) 10, 20, 30
```

will transfer control to statement number 30. Complex expressions may not be used in an IF statement.

### 4.2.2 Logical IF Statements

| | |
|---|---|
| Form | IF (logical expression) statement |
| Where | statement may be any executable statement except a logical IF or a DO |
| Effect | The statement given is executed if the expression has the value .TRUE.; otherwise, the next statement in sequence is executed. |

Examples:

```
IF (T.OR.S) X = Y + 1
IF (Z.GT.X(K)) CALL SWITCH (S,Y)
IF (K.EQ.INDEX) GO TO 15
```

## 4.3 THE DO STATEMENT

| | |
|---|---|
| Form | DO n i = $m_1$, $m_2$, $m_3$ |
| Where | n is a statement number<br>i is an integer variable<br>$m_1$, $m_2$, $m_3$ are positive integer variables or constants |
| Effect | Statements following the DO up to and including statement n are executed repeatedly for values of i starting with $m_1$, and incremented by m3 until i is greater than or equal to m2. |

The statements which are executed as a result of a DO statement are called the range. The variable i is called the index. The values $m_1$, $m_2$, and $m_3$ are, respectively, the initial, limit, and increment values of the index. When the DO statement occurs, its range is first executed for $i = m_1$. Subsequent iterations are for $i = i + m_3$. If $m_3$ is not supplied by the programmer, an increment of 1 is assumed. The final iteration is for $i \geq m_2$. A zero or negative $m_3$ value is not permitted. The range of a DO is always executed at least once, regardless of the values of the limit and increment. After each execution of the range, the increment value is added to the value of the index and the result is compared with the limit value. If the value of the index is not greater than the limit value, the range is executed again using the new value of the index.

Examples:

```
DO 20 I = 5, 100, 2
(final iteration for I = 99)
DO 100 I = 0, 100, 2
(final iteration for I = 100)
```

After the last execution of the range, control passes to the statement immediately following it. This exit from the range is called the normal exit. Exit may also be accomplished by the execution of a control statement within the range.

The values of the limit and increment variables and the index of the DO loop may not be altered within the range of the DO statement. When a statement transfers control outside the range of a DO loop, e.g., by a GO TO or IF, the index retains its current value and is available for use as a variable. The value of the index variable becomes undefined when the DO loop it controls is exited normally. A transfer from outside the range into a DO loop is not legal.

The terminal statement of a DO range may not be a GO TO, DO, RETURN, STOP, PAUSE, or an arithmetic IF statement. A logical IF statement is allowed as the last statement of the range, provided that it does not contain any of the statements mentioned above.

As an example, consider the sequence:

```
  DO 5 K = 1, 4
5 IF (X (K) .GT. Y (L)) Y (K) = X (K)
6 ...
```

In this case, the range is considered ended when, and if, control would normally pass to the statement following the entire logical IF statement. Statement 5 is executed four times whether the statement Y(K) = X(K) is executed or not. Statement 6 is not executed until statement 5 has been executed four times. Note that if statement 5 were:

```
5 IF (X (K) .GT. Y (L)) GO TO 10
```

it would be an error.

The range of a DO statement may also include other DO statements. This is referred to as nesting. The range of each nested DO statement must fall entirely within the range of the outer DO statement; that is, the ranges of two DO statements must intersect completely or not at all. Figure 4-1 illustrates the order in which nested DOs are executed.

# CHAPTER 5
# DATA TRANSMISSION STATEMENTS

Data transmission statements govern the transfer of data between internal storage and peripheral devices. These include three distinct types of statement - data description statements (FORMAT and DEFINE FILE); input-output statements (READ and WRITE); and device control statements (FIND, BACKSPACE, REWIND, and END FILE).

## 5.1 DATA DESCRIPTION STATEMENTS

The data description statements - FORMAT and DEFINE FILE - describe the form and arrangement of data on the selected peripheral device; FORMAT describes a record, DEFINE FILE a disk file.

### 5.1.1 The FORMAT Statement

| | |
|---|---|
| Form | n FORMAT (field description$_1$ .../...) |
| Where | n is a statement number |
| Effect | Specified either type of conversion to be performed between the internal and external representation of data or format of fixed data. |

A FORMAT statement may describe one or more records. The character / (slash) indicates that a new record is being described. For example, the statement:

```
FORMAT (3O8/I5,2F8.4)
```

is equivalent to:

```
FORMAT (3O8)
```

for the first record and:

```
FORMAT (I5,2F8.4)
```

for the second record. Each record description may consist of one or more field specifications, a field being a consecutive series of characters within the record. Field specifications are separated by commas as shown above. The separating comma may be omitted when a slash is used. When n slashes appear at the end or beginning of a format, n blank records may be written on output or records skipped on input. When n slashes appear in the middle of a format, n-1 blank records are written or n-1 records skipped.

Both the slash and the closing parenthesis at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated starting with that group repeat specification terminated by the last right parenthesis of level one (or level zero if no level one group exists).

Thus, the statement

```
          FORMAT (F7.2,(2(E15.5,E15.4),I7))
                 ↑       ↑ ↑             ↑ ↑ ↑
level 0 ─────────┘       │ └level 2 ─────┘ │ └── level 0
              level 1 ───┘      level 1 ───┘
```

causes the format

F7.2,2(E15.5,E15.4),I7

to be used on the first record, and the format

2(E15.5,E15.4),I7

to be used on succeeding records.

As a further example, consider the statement:

FORMAT (F7.2/(2(E15.5,E15.4),I7))

The first record has the format

F7.2

and successive records have the format

2(E15.5,E15.4),I7

The ASCII character string comprising a format specification may be stored as an array. Input/output statements may then refer to the format by giving the array name, rather than the statement number of a FORMAT statement. The stored format has the same form as a FORMAT statement excluding the word "FORMAT." The enclosing parentheses are required.

Repetition of a field specification may be indicated by preceding a field descriptor by an unsigned integer giving the number of repetitions desired.

A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number.

FORMAT statements may be placed anywhere within the executable portion of the source program. Unless the FORMAT statement contains only alphanumeric data for direct input/output transmission, it will be used in conjunction with the list of a data transmission statement.

The form of a field specification depends on the type of field being described. There are three basic types - numeric, logical, and Hollerith. In addition, a blank field description may be given to skip portions of an input record or to imbed blanks within an output record.

5.1.1.1 Numeric Fields - Numeric fields are specified by one-letter codes which designate the type of conversion to be performed. Two parameters may appear in a numeric field description, depending on the field type. These are: an integer (w) specifying the field width (which may be greater than required to provide for blank columns between numbers) and an integer (d) specifying the number of decimal places to the right of the decimal point or, for G conversion, the number of significant digits. (For D, E, F, and G input, the position of the decimal point if present in the external field, takes precedence over the value of d in the format.) Conversion codes and the corresponding internal and external forms of the numbers are listed in Table 5-1 below.

Table 5-1
Numeric Field Codes

| Conversion Code | Internal Form | External Input Form | External Output Form |
|---|---|---|---|
| D | Double precision | Decimal number with or without a . or exponent field | Decimal number with a D exponent field and a decimal point |
| E | Real | Decimal number with or without a . or exponent field | Decimal number with a decimal point and an E exponent field |
| F | Real | Decimal number with or without a . or exponent field | Decimal number with a decimal point |
| G | Real | Decimal number with or without a . or exponent field | Decimal number with a decimal point and with or without an E exponent field (see Table 5-2) |
| I | Integer | Decimal number without a . or exponent | Decimal number without a decimal point or exponent |
| O | Integer | Octal number | Octal number |

The allowable numeric field description forms are:

(1) Dw.d
(2) Ew.d
(3) Fw.d
(4) Iw
(5) Ow
(6) Gw.d

For example,

```
FORMAT (I5,F10.2,D18.10)
```

could be used to output the line

```
bbb32bbbb-17.60bbb.5962547681D+03
```

on the output listing. (The letter b represents a blank or a space.)

The G format is the general format code that is used to transmit data. The rules for input are the same as E format. The form of the output conversion is a function of the magnitude of the data being converted. Table 5-2 shows the magnitude of the external data, M, and the resulting method of conversion.

Table 5-2
Magnitude of Internal Data

| Magnitude of Data | Resulting Conversion |
|---|---|
| $0.1 \leq M < 1$ | F(w-4).d, 4x |
| $1 \leq M < 10$ | F(w-d).(d-1), 4x |
| ⋮ | ⋮ |
| $10^{d-2} \leq M < 10^{d-1}$ | F(w-4).1, 4x |
| $10^{d-1} \leq M < 10^{d}$ | F(w-4).0, 4x |
| All others | Ew.d |

The field width (w) should always be large enough to include spaces for the decimal point, sign, and exponent. In all numeric field conversions, if w is not large enough to accommodate the converted number, asterisks will be printed for the field. If the number is less than w spaces in length, the number is right-adjusted in the field.

Scale factors may be specified for D, E, F, and G conversions. A scale factor is written:

nP

where P is the identifying character and n is a signed or unsigned integer that specifies the scale factor.

For F type conversions, the scale factor specifies a power of ten so that

$$\text{external number} = (\text{internal number})* 10^{(\text{scale factor})}$$

For D and E conversions, the scale factor multiplies the fraction by a power of ten, but the exponent is changed accordingly leaving the number unchanged except in form. For example, if the statement:

```
FORMAT (F8.3,E16.5)
```

corresponds to the line

```
bb26.451bbbb-0.41321E-01
```

then the statement

```
FORMAT (-1PF8.3,2PE16.5)
```

would correspond to the line

```
bbb2.645bbb-41.3215E-03
```

For G type output conversion, the scale factor is not used unless the magnitude of the number is such that E format is used.

In input operations, the scale factor is not used if there is an exponent in the external field.

When no scale factor is specified, a scale factor of zero is assumed. Once a scale factor has been specified, however, it holds for all subsequent D, E, F, and G type conversions within the same format unless another scale factor is encountered. A zero scale factor may be resumed via an explicit specification. Scale factors have no effect on I and O type conversions.

Complex quantities are transmitted as two independent real quantities. The format specification consists of two successive real specifications or one repeated real specification. For instance, the statement

```
FORMAT (2E15.4,2(F8.3,F8.5))
```

could be used in the transmission of three complex quantities.

5.1.1.2 Logical Fields - Logical data can be described in a manner similar to numeric data. A logical field description has the form:

Lw

where L is the conversion code character and w is an integer specifying the field width. The data is transmitted as the value of a logical variable in the input/output list. On input, the first nonblank character in the data field must be T or F; the value of the logical variable will be stored as true or false, respectively. If the data field is blank or empty, a value of false will be stored. On output, w minus 1 blanks followed by the letter T or F, according to the variable's value, will be transmitted. For example, if the specification were L10, the output for the value .TRUE. would be:

bbbbbbbbbT

5.1.1.3 Hollerith Fields - Hollerith data can be described in a manner similar to numeric data, as in:

Aw

where A is the conversion code character and w, the number of characters in the field. The alphanumeric characters are transmitted as the value of a variable in an input/output list. The variable may be of any type. The sequence:

```
  READ(2,5)V
5 FORMAT (A4)
```

causes four characters to be read and placed in memory as the value of the variable V.

The value of w is limited to the maximum number of characters which can be stored in the space allotted for a single variable.

If w exceeds this amount, the leftmost characters are lost on input, and on output the w characters will appear right-justified in the external output field, with blanks filled in on the left.

If w is less than the number of characters which can be stored in the space allotted to the variable, on input the characters are left-justified and blank-filled on the right of each list item. On output the leftmost w characters in the variable are transmitted to the output field.

Hollerith data may also be transmitted directly into or from the FORMAT statement. The Hollerith string may be specified in two forms. One, called H-conversion, is:

nH

where H is the control character and n is the number of characters in the string (including blanks). For example, the format in the statement below can be used to print PROGRAM COMPLETE on the output listing.

```
FORMAT (17H PROGRAM COMPLETE)
```

Referring to this format in a READ statement would cause the 17 characters to be replaced with a new string of characters from the input file.

In the second form, the Hollerith data is simply enclosed in single quotes. The result is the same as in H-conversion; on input, the characters between the quotes are replaced by input characters, and, on output, the characters between the quotes (including blanks) are written as part of the output data. A quote character within the data is represented by two successive single quotes as with Hollerith constants.

A Hollerith format field may be placed among other fields of the format. For example, the statement:

```
FORMAT (I5,7H FORCE=F10.5)
```

can be used to output the line:

```
bbb22bFORCE=bb17.68901
```

Note that the separating comma may be omitted after a Hollerith format field.

5.1.1.4 Carriage Control - The first character of each ASCII record controls the spacing of the line printer or teleprinter. This character may be established by beginning a FORMAT statement for an ASCII record with 1Ha, where a is the desired control character. The line spacing actions, listed below, occur before printing.

| Character | Effect |
|---|---|
| blank | advance carriage to next line |
| 0 zero | skip a line (double space) |
| 1 one | form feed - go to top of next page |
| + plus | suppress skipping - will overprint line |

If any other character appears first, it will be treated as a blank.

5.1.1.5 Record Layout Specification – Input and output can be made to begin at any position within a FORTRAN record by use of a field description of the form:

```
Tw
```

where T is the spacing control character and w is an unsigned integer constant specifying the character position in a FORTRAN record where the transfer of data is to begin. For printed output, w corresponds to the (w-1)th print position, since the first character of the output buffer is a carriage control character and is not printed. (A blank carriage control indicator is assumed.)

For example,

```
2 FORMAT (T50, 'BLACK'T30, 'WHITE')
```

would cause the following line to be printed:

```
Print Position 29         Print Position 49
↓                          ↓
    WHITE                      BLACK
```

For input, the statement

```
1 FORMAT (T35, 'MONTH')
  READ (3,1)
```

causes the first 34 characters of the input data to be skipped, and the next five characters would replace the characters M, O, N, T, and H in storage. If an input record containing

```
ABCbbbXYZ
```

is read with the format specification

```
10 FORMAT (T7,A3,T1,A3)
```

then the characters XYZ and ABC are read, in that order.

Blanks may be introduced into an output record or characters skipped on an input record by use of the specification:

```
nX
```

where the spacing control character is X and n is the number of blanks or characters skipped and must be greater than zero. For example, the statement

```
FORMAT (5H STEPI5,10X2HY=F7.3)
```

may be used to output the line

```
bSTEPbbb28bbbbbbbbbbY=b-3.872
```

The preceding blank would not be printed on teleprinter or line printer.

### 5.1.2 The DEFINE FILE Statement

| | |
|---|---|
| Form | DEFINE FILE $a_1$ ($m_1$, $1_1$, U, $v_1$),<br>$a_2$ ($m_2$, $1_2$, U, $v_2$), ... |
| Where | a is an integer constant or variable name that is the symbolic designation for this file (see WRITE statement, Section 5.2.3, for more information on this field). m is an integer constant or variable name that defines the number of records in the file.<br>1 is an integer constant or variable name that defines the length (in words) of each file record.<br>U is a fixed argument designating that the file is unformatted.<br>v is an integer variable name, called the associated variable, which is set at the conclusion of an input-output operation on the file to point to the next record. |
| Effect | Describes a disk file for use with input-output statements. |

The DEFINE FILE statement is applicable to disk files only, and is required so that they may be referenced as direct access files by input-output statements.

The associated variable (v) in a DEFINE FILE statement is used to maintain an index of records processed. It is set automatically after an input-output statement is executed.

The statement:

```
DEFINE FILE 1(1000,100,U,V1)
```

specifies a 1000-record file, each record of which is 100 words long. The variable V1 will maintain an index of records processed, providing a pointer to the next record to be processed.

## 5.2 INPUT-OUTPUT STATEMENTS

The input-output statements, READ and WRITE, govern transfer of data records between internal storage and peripheral devices. Each statement may contain an input-output list naming the variables and array elements to be given values on input or whose values are to be transmitted on output.

Both formatted and unformatted records may be transmitted. A formatted record, a string of characters, requires the use of a format specification.

### 5.2.1 Input-Output Lists

An input-output list contains variable names and array elements whose values will be assigned on input or written on output. During input, the new values of listed variables may be used in subscript or control expressions for variables appearing later in the list. For example:

```
READ(13)L,A (L),B(L+1)
```

reads a new value for L and uses this value in the subscripts of A and B.

The transmission of array variables may be controlled by indexing similar to that used in the DO statement by including as a list element a parenthesized list of control variables followed by the index control. For example,

```
READ(7) (X(K),K=1,4),A
```

is equivalent to:

```
READ(7)X(1),X(2),X(3),X(4),A
```

The indexing may be compounded by nesting as in the following:

```
READ(11) ((MASS(K,L),K=1,4),L=1,5)
```

The above statement reads in the elements of array MASS in the following order:

```
MASS(1,1),MASS(2,1),...,MASS(4,1),MASS(1,2),...,MASS(4,5)
```

If an entire array is to be transmitted, the indexing may be omitted and only the array name written. The array is transmitted in order of increasing subscripts with the first subscript varying most rapidly. Thus, the example above could have been written:

```
READ(11)MASS
```

assuming that the array MASS is dimensioned MASS(4,5).

5.2.2 Input-Output Records

All data is transmitted by an input-output statement in terms of records. The maximum amount of information in one record and the manner of separation between records depends upon the medium. For punched cards, each card constitutes one record; on a teletypewriter, a record is one line; for ASCII records, the amount of information is specified by the FORMAT reference and the I/O list; for magnetic tape binary records, the amount of information is specified by the I/O list; for disk records, DEFINE FILE is used.

Each execution of an input or output statement initiates the transmission of a new data record. If an input-output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another input-output statement without repositioning the record. Repositioning is not, however, possible on all devices (see Section 5.3). If an input-output list requires more than one ASCII record of information, successive records are read.

5.2.3 The WRITE Statement

| | |
|---|---|
| Form | WRITE (u,f) list } ------formatted WRITE<br>WRITE (u,f)<br>WRITE (u) list -------- unformatted WRITE<br>WRITE (a'r) list -------direct access disk WRITE<br>WRITE (u,f,END=n) list<br>WRITE (u,f,ERR=n) list } - WRITE and transfer control if errors<br>WRITE (u,f,END=n,ERR=n) list |
| Where | u is a unit designation<br>f is a format reference<br>list is an I/O list<br>a is a symbolic disk file number<br>r is an associated variable (record pointer) |
| Effect | Output is performed as specified by the arguments of the WRITE statement. |

A formatted WRITE statement may appear with or without an I/O list. If a list is provided, the values of the variables in the list are read from memory and written on the unit designated in ASCII form. The data is converted to external form as specified by the designated FORMAT statement. If no list is supplied, information is read directly from the specified format and written on the unit designated in ASCII form.

An unformatted WRITE statement must have an I/O list. The values of the variables in the list are read from memory and written on the unit designated in binary form.

A direct access WRITE statement outputs a fixed-length record directly into a disk file. The file must be defined previously via the execution of an appropriate DEFINE FILE statement.

Notes On Unit Designation (u) And Symbolic Disk File Number (a)

The unit designation (u) and symbolic disk file number (a) referred to in READ (see next section), WRITE, and DEFINE FILE statements may be integers in the range 1 to 8. Of the eight numbers, 6 is the keyboard, 5 is the line printer and 4 is the high-speed paper-tape reader. The remaining numbers are assumed to refer to files on a disk.

Thus, READ (4,10) refers to input from the high-speed paper-tape reader, WRITE (5,1) refers to output on the line printer, and READ (7,11) refers to input from the disk. Note that u and a are both drawn from the same set of numbers, and they cannot conflict.

The user may override these device number assumptions by two methods. He may either use the SETFIL subroutines to override the unit number assumptions, or he may employ the ASSIGN command of the DOS Monitor. The ASSIGN command (see Disk Operating System Monitor Manual) allows the override to occur at run-time, just before the program is executed. The SETFIL subroutine allows the override to be specified in the program, thereby requiring no intervention at run-time.

### 5.2.4 The READ Statement

| | | |
|---|---|---|
| Form | READ (u,f) list<br>READ (u,f) | formatted READ |
| | READ (u) list<br>READ (u) | unformatted READ |
| | READ (a'r) list | direct access disk READ |
| | READ (u,f,END=n) list<br>READ (u,f,ERR=n) list<br>READ (u,f,END=n,ERR=n) list | READ and transfer control |
| Where | f is a format reference<br>u is a unit designation<br>r is an associated variable record pointer<br>a is a symbolic disk file number<br>n is a statement number | |
| Effect | Input is performed according to the arguments of the READ statement | |

A formatted READ statement causes information to be read from the specified unit and put in memory. The data are converted from external to internal form as specified by the referenced FORMAT statement. If an I/O list is provided, the data are stored as the values of listed variables. The second form of the READ statement is used if the data are transmitted directly into the specified format.

An unformatted READ statement causes binary information to be read from the unit designated and stored in memory as values of the variables in the I/O list, if any.

A direct access READ statement provides random access to fixed-length records in a disk file. The file whose records are to be read must be defined by the DEFINE FILE statement.

READ and transfer control statements cause control to be transferred to the statement specified if an end-of-file or error condition is encountered during input. The arguments END=n and ERR=n may appear separately or together. If an end-of-file is encountered during a READ, control transfers to the statement specified by END=n. If an END parameter is not specified, I/O on that device terminates and the program halts with an error message. If an error on input is encountered, control transfers to the statement specified by ERR=n. If an ERR=n parameter is not specified, the program halts with an error message.

Example:

```
    READ (7,7,END=888,ERR=999)A
    .
    .
    .
888 (control transfers here if an end-of-file is encountered)
    .
    .
    .
999 (control transfers here if an error on input is encountered)
```

## 5.3 DEVICE CONTROL STATEMENTS

There are four device control statements - FIND (which applies to a moving head disk only) and BACKSPACE, END FILE, and REWIND which apply to any device which may be automatically repositioned (magnetic tape, DECtape, and disk). Their forms and effects are listed below in Table 5-3.

Table 5-3
Device Control Statements

| Statement | Effect |
| --- | --- |
| FIND (a'b) | The disk read/write mechanism is positioned to record b of file a. (a is assigned via a DEFINE FILE statement. The record number b is an integer constant or variable.) |
| BACKSPACE u | Repositions the designated unit to the beginning of the file and spaces forward to n-2 records (n is the number of the record processed before the BACKSPACE). |

(Continued on next page)

Table 5-3 (Cont)
Device Control Statements

| Statement | Effect |
|---|---|
| END FILE u | Activates the Monitor's CLOSE facility for the designated unit, thereby writing an END-OF-FILE. |
| REWIND u | Repositions the designated unit to the beginning of the file. |

# CHAPTER 6
# SPECIFICATION STATEMENTS

Specification statements may be divided into three categories. First, there are storage specification statements - DIMENSION, COMMON, and EQUIVALENCE - which give the compiler storage allocation instructions. Second, there are data specification statements - DATA and BLOCK DATA - which are used to enter values. Third, there are type declaration statements - INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BYTE, and IMPLICIT - which specify the type attribute of a variable. These are all nonexecutable statements which must precede the executable portion of the program. DATA statements must follow all other specification statements.

## 6.1 STORAGE SPECIFICATION

### 6.1.1 The DIMENSION Statement

| | |
|---|---|
| Form | DIMENSION array name ($V_1$, $V_2$, $V_3$)... |
| Where | $V_1$, $V_2$, and $V_3$ are the maximum value the subscript they represent may assume |
| Effect | The array name is assigned the type array. Storage is allocated according to the dimensions given. |

Each array specification gives the array name and the maximum values which each of its subscripts may assume. Each value must be an unsigned positive integer constant or variable. Arrays may also be declared in the COMMON or TYPE declaration statements in the same way:

```
COMMON X(10,4),Y,Z
INTEGER A(7,32),B
```

No array for which dimension information is not supplied may be referenced as an array variable.

A subprogram may establish adjustable arrays via reference to an array which has been allocated storage by the calling program. In this case, both the array name and the subscript values are expressed as dummy arguments in the subroutine, as in:

DIMENSION A(X,Y,Z)

In order to do this, the programmer must establish A, X, Y, and Z as required arguments. The dummy array must not exceed the dimensions of the main-program array but may be smaller if the call provides lower subscript values than those of the main program dimensioning or if the initial array element referenced is not the beginning of the main-program array.

### 6.1.2 The COMMON Statement

| | |
|---|---|
| Form | COMMON/BLOCK1/A,B,C/BLOCK2/D,E,F/... |
| Where | BLOCK1,BLOCK2,...are the block names<br>A,B,C...are the variables to be assigned to each block |
| Effect | Specified variables or arrays are stored in an area available to other programs. |

By means of COMMON statements, the data of a main program and/or the data of its subprograms may share a common storage area. The common area may be divided into separate blocks which are identified by block names. A block is specified as follows:

/block name/var1, var2, ...

The variables which follow the block name indicate scalar or array variables assigned to the block. They are placed in the block in the order in which they appear in the block specification. For example, the statement

COMMON/R/X,Y,T/C/U,V,W,Z

indicates that the elements X, Y, and T are to be placed in block R in that order, and that U, V, W, and Z are to be placed in block C. A common block may have the same name as a variable in the same program.

Block entries are linked sequentially throughout the program, beginning with the first COMMON statement. For example, the statements

COMMON/D/ALPHA/R/A,B/C/S
COMMON/C/X,Y/R/U,V,W

have the same effect as the statement

```
COMMON/D/ALPHA/R/A,B,U,V,W/C/S,X,Y
```

One block of common storage, referred to as blank common, may be left unlabeled. Blank common is indicated by two consecutive slashes. For example,

```
COMMON/R/X,Y//B,C,D
```

indicates that B, C, and D are placed in blank common. The slashes may be omitted when blank common is the first block of the statement, as in:

```
COMMON B,C,D
```

Storage allocation for blocks of the same name begins at the same location for all programs executed together. For example, if a program contains

```
COMMON A,B/R/X,Y,Z
```

as its first COMMON statement, and a subprogram has

```
COMMON/R/U,V,W//D,E,F
```

as its first COMMON statement, the quantities represented by X and U are stored in the same location. A similar correspondence holds for A and D in blank common.

Common blocks may be of any length. No program must, however, attempt to enlarge a common block declared by a previously linked* program. Array names appearing in COMMON statements may have dimension information appended if the arrays have not been declared via a DIMENSION statement or a type declaration. For example,

```
COMMON ALPHA,T(15,10,5),GAMMA
```

specifies the dimensions of the array T while entering T in blank common. Each array name appearing in a COMMON statement must be dimensioned somewhere in the program containing the COMMON statement.

### 6.1.3 The EQUIVALENCE Statement

| | |
|---|---|
| Form | EQUIVALENCE $(V_1,V_2,\ldots),(V_k,V_{k+1},\ldots),\ldots$ |
| Where | V's are variable names |
| Effect | The set of parenthesized variables identify the same storage location. |

*Programs are linked via the LINK-11 program as described in the LINK-11 manual.

For example,

```
EQUIVALENCE(RED,BLUE)
```

specifies that the values of the variables RED and BLUE are stored in the same location.

The relation of equivalence is transitive; thus, the two statements

```
EQUIVALENCE(A,B),(B,C)
EQUIVALENCE(A,B,C)
```

have the same effect.

The subscripts of array variables in an EQUIVALENCE statement must be integer constants.

Example:

```
EQUIVALENCE(X,A(3),Y(2,1,4)),(BETA(2,2),ALPHA)
```

### 6.1.4 EQUIVALENCE AND COMMON

Variables may appear in both COMMON and EQUIVALENCE statements, but no two quantities in COMMON may be set equivalent to one another.

Quantities placed in a common block by means of EQUIVALENCE statements may cause the end of the common block to be extended. For example, the statements

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(A,Y)
```

causes the common block R to extend from X to A(4), arranged as follows:

```
X
Y       A(1)
Z       A(2)
        A(3)
        A(4)
```

EQUIVALENCE statements which would require extension of the start of a common block are not allowed. For example, the sequence

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(X,A(3))
```

is not permitted, since it would require A(1) and A(2) to extend the starting location of block R.

## 6.2 THE DATA STATEMENT

| | |
|---|---|
| Form | DATA (var $list_1$)/values $list_1$/(var $list_2$)/values $list_2$/,... |
| Where | (var list) contains a string of variables separated by commas<br>/values list/ contains a string of data items separated by commas |
| Effect | A value from values list is assigned to the corresponding variable in var list. |

The DATA statement is used to supply initial or constant values for variables. The specified values are compiled into the object program, and become the values assumed by the variables when program execution begins. Such values may also be provided via a BLOCK DATA subprogram (see Chapter 7). Variables in a labeled common block can only be specified in a BLOCK DATA subprogram. Variables in blank common may not be initialized.

Variables in the variable list may be either single subscripted or unsubscripted arrays, or the name of an entire array.

When an entire array is given, data values must be specified for each and every element of the array. Data elements are stored in the array in the same order used for the data transmission and storage arrays, i.e., in order of increasing subscripts with the first subscript varying most rapidly.

Allocation to memory locations in the array stops when:

a. the data item list is exhausted; or
b. data items have been allocated to the entire array. If so, additional data items will be allocated to additional items in the varaible list.

When Hollerith or literal constants are encountered in the values list, they are assigned to the associated variables in the same manner that such constants are handled in assignment statements. Specifically, let the site of the variable in bytes be v, and the size of the literal by l.

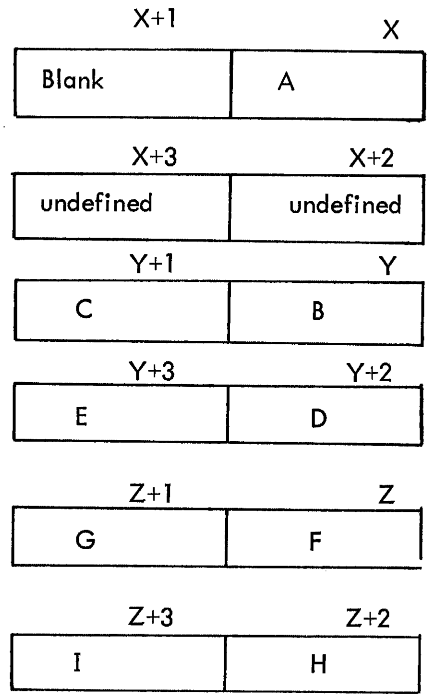
a. if $v \leq l$, the first (leftmost) v digits of the literal will be stored in the variable; the remaining digits will be ignored.

b. if $l < v$, the Hollerith literal will occupy the l low order bytes of the variable. The remaining bytes will be undefined. Note, however, that since Hollerith literals are always blank-filled to word boundaries, the first byte following any Hollerith constant with an odd number of bytes will be a blank.

Example:

```
DATA    X,Y,Z/'A','BCDE','FGHIJKL'/
```

produces in memory:




The data items following each list of variables must have a one-to-one correspondence with the variables of the list, and must agree in type, since each item of the data specifies the value given to its corresponding variable.

Data items assigned may be numeric, Hollerith, octal, hexadecimal, or logical constants. For example,

```
DATA ALPHA, BETA/5,16.E-2/
```

specifies the value 5 for ALPHA and the value .16 for BETA. Any item of data may be preceded by an integer constant followed by an asterisk. This notation indicates that the item is to be repeated. For example,

```
DATA(A(1),A(2),A(3))/3*0./
```

specifies the value zero for array elements A(1) - A(3).

As another example:

```
DIMENSION A(2,2),B(3)
DATA A,B/2*1.0,3*2.0,3.0,4./
```

will initialize

A(1,1), and A(2,1) to 1
A(1,2),A(2,2) and B(1) to 2
B(2) to 3, and B(3) to 4.

## 6.3 TYPE DECLARATION STATEMENTS

| | |
|---|---|
| Form | type $V_1, V_2, V_3, \ldots$ |
| Where* | type may be: INTEGER (INTEGER*2), REAL (REAL*4), DOUBLE PRECISION (REAL*8), COMPLEX, LOGICAL, BYTE (LOGICAL*1) |
| | $V_1$, $V_2$, $V_3$ are variables |
| Effect | All variables in the list are assigned the given type. |

A variable may appear in only one type statement. Type statements may be used to give dimension specifications for arrays. Adjustable arrays in subprograms may also be defined via type statements.

## 6.4 THE IMPLICIT STATEMENT

| | |
|---|---|
| Form | IMPLICIT type ($a_1, a_2, \ldots$) |
| Where | type is INTEGER, REAL, LOGICAL, COMPLEX or DOUBLE PRECISION<br>$a_1, a_2, \ldots$ represent single alphabetic characters, each separated by commas, or a range of characters (in alphabetic sequence) denoted by the first and last characters of the range separated by a minus sign (e.g., (A-D)) |
| Effect | Any program variable which is not mentioned in a type statement, and whose first character is one of those listed in the IMPLICIT statement, is classified according to the type appearing before the list in which the character appears. |

As an example, the statement

IMPLICIT REAL(A-D,L,N-P)

causes all variables starting with the letters A through D, L, and N through P to be typed as real, unless they are explicitly declared otherwise.

*Parenthesized items are synonyms.

The initial state of the compiler is set as if the statements

```
IMPLICIT REAL(A-H,O-Z)
IMPLICIT INTEGER(I-N)
```

were at the beginning of the program. This state is in effect unless an IMPLICIT statement changes the above interpretation, i.e., identifiers, whose types are not explicitly declared, are typed as follows.

- Identifiers beginning with I, J, K, L, M, or N are assigned integer type.
- Identifiers not assigned integer type are assigned real type.

# CHAPTER 7
# SUBPROGRAM STATEMENTS

There are two categories of subprograms in FORTRAN - functions and subroutines. Both consist of one or more FORTRAN statements which may be invoked by name and, as appropriate, with values upon which they are to operate. A function differs from a subroutine in that it always returns a single numeric value; by convention, the function reference represents this value in an expression. A subroutine, on the other hand, may return several or no values.

The transmission of arguments between a subprogram reference and the subprogram itself is accomplished by the use of dummy variables within the subprogram definition. Those variables in the subprogram which are dummy variables are listed in the subprogram definition statement. References to the subprogram may then supply values for these arguments in the same order and be substituted for them whenever they appear in the subprogram.

## 7.1 FUNCTION DEFINITIONS

Functions may be internal or external. An internal function is defined via a form of the Arithmetic statement and may be referenced only by the program in which it is defined. An external function, which may be referenced by other programs, is defined via the FUNCTION statement. All functions must have at least one argument.

A function name must be a legal symbol. A function reference may only appear within an expression and must, like other elements of expressions, have a specified type. Type may be specified in the definition itself or via any other FORTRAN type-specification facility.

### 7.1.1 The Arithmetic Statement Function Definition

| | |
|---|---|
| Form | t name (arg1, ...) = expression |
| Where | t is an optional type specification<br>name is the function name<br>arg1, ... are dummy variables<br>expression is the function definition |
| Effect | Defines an internal function. |

An Arithmetic statement function definition is a single statement. The expression which defines the function may include dummy arguments, ordinary variables, external functions and previously defined internal functions.

In the following definition:

```
ACOSH(X) = (EXP(X/A) + EXP(-X/A))/2.
```

X is a dummy argument and A an ordinary variable. When the function is referenced, the current value of A and the supplied value of X will be used to evaluate it. All function definitions of this type must precede the first executable statement of the program in which they appear, and follow the last specification statement appearing in the program.

### 7.1.2 The FUNCTION Statement

| | |
|---|---|
| Form | t FUNCTION name (arg1, ...) |
| Where | t is an optional type specification<br>name is the function name<br>arg1, ... are dummy arguments |
| Effect | Defines an external function. |

The function name must be a legal symbol and must be assigned a value within the definition. This value is the function's value. Arguments must agree in number, order, and type with actual arguments given by the calling program.

Dummy arguments may represent the following elements in the function definition: expressions, alphanumeric strings, array names or elements and subprogram names. Dummy arguments which represent array names must appear within the subprogram either in a DIMENSION statement, or in one of the type statements that provide dimension information. Dimensions given as constants must not exceed the dimensions of the corresponding arrays in the calling program. Dimensions given as dummy variables may be used to specify adjustable dimensions for array name arguments. For example, in the statement sequence:

```
FUNCTION TABLE (A,M,N,B,X,Y)
  .
  .
  .
DIMENSION A(M,N), B(10), C(50)
```

the dimensions of array A are specified by the dummy arguments M and N, while the dimension of array B is given as a constant. The various values given for M and N by the calling program must be within the limits of the actual arrays which the dummy array A represents. Various arrays may be substituted for A. These arrays may each be of different size. Dummy dimensions may only be given for dummy arrays. Note in the example above that the array C, which is not a dummy argument, must be given absolute dimensions. A dummy argument may not appear in an EQUIVALENCE statement in the FUNCTION subprogram.

A function must not modify any arguments which appear in the FORTRAN arithmetic expression calling the function. The only FORTRAN statements not allowed in a FUNCTION subprogram are SUBROUTINE, BLOCK DATA, and another FUNCTION statement.

## 7.2 SUBROUTINE SUBPROGRAMS

A SUBROUTINE subprogram is defined external to the program which references it. Subroutine definition is initiated by a SUBROUTINE statement. A subroutine is referenced by a CALL statement and returns control to the calling program by means of one or more RETURN statements.

### 7.2.1 The SUBROUTINE Statement

| | |
|---|---|
| Form | SUBROUTINE name<br>SUBROUTINE name (arg1, ...) |
| Where | name and arg are as for functions |
| Effect | The program which follows is declared a SUBROUTINE subprogram. |

The arguments in the parenthesized list are dummy arguments representing the arguments of the subprogram. The dummy arguments must agree in number, order, and type with the actual arguments used by the calling program. A SUBROUTINE subprogram need not have any arguments at all. When supplied, they may be expressions, alphanumeric strings, array names, array elements, scalar variables, and subprogram names.

Dummy variables which represent array names must be dimensioned within the subprogram by a DIMENSION or type declaration statement. As in the case of a FUNCTION subprogram, either constants or dummy identifiers may be used to specify dimensions in a DIMENSION statement. The dummy arguments must appear in an EQUIVALENCE or COMMON statement in the SUBROUTINE program.

A SUBROUTINE subprogram may use one or more of its dummy arguments to represent results. For example,

```
SUBROUTINE COMPUTE (A,B,ANS)
```

requires the user to supply numeric values for A and B to be computed, and a variable for ANS in which to store the results. The only FORTRAN statements not allowed in a SUBROUTINE subprogram are FUNCTION, BLOCK DATA, and another SUBROUTINE statement.

### 7.2.2 The CALL Statement

| | |
|---|---|
| Form | CALL name<br>CALL name (arg1, ...) |
| Where | name identifies a subprogram<br>arg1, ... are actual arguments |
| Effect | Control is transferred to the SUBROUTINE subprogram. |

The arguments of a CALL statement may be expressions, array names, array elements, scalar variables, alphanumeric strings or subprogram names; arguments may be of any type, but must agree in number, order, type, and array size (except for adjustable arrays, as discussed under the DIMENSION statement) with the corresponding arguments in the SUBROUTINE statement of the called subroutine. Unlike a function, a subroutine may produce more than one value and cannot be referred to as a basic element in an expression.

### 7.2.3 The RETURN Statement

The RETURN statement consists of the text:

```
RETURN
```

This statement returns control from a subprogram to the calling program. Normally, the last statement executed in a subprogram is a RETURN statement. Any number of RETURN statements may appear in a subprogram.

## 7.3 THE BLOCK DATA STATEMENT

The BLOCK DATA statement is used to establish a BLOCK DATA subprogram, a data specification subprogram which is used to enter initial values for variables in labeled common blocks. No executable statements may appear in a BLOCK DATA subprogram. A BLOCK DATA subprogram is established by a BLOCK DATA statement consisting of the text:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram and it must be the first statement of the subprogram.

The subprogram contains only type-statements, EQUIVALENCE, DATA, DIMENSION, and COMMON statements. A complete set of specifications must be given for an entire COMMON block. A single BLOCK DATA subprogram may initialize any number of named COMMON blocks.

## 7.4 THE EXTERNAL STATEMENT

| | |
|---|---|
| Form | EXTERNAL identifier, identifier, ... identifier |
| Where | identifier is the name of a subprogram |
| Effect | The identifier is declared a subprogram name and may be used as the argument of other subprograms |

FUNCTION and SUBROUTINE subprogram names may be used as the actual arguments of subprograms. When they are, their names must be distinguished from ordinary variables by their appearance in an EXTERNAL statement.

Any subprogram name given as an argument to another subprogram must have previously appeared in an external declaration in the calling program (i.e., as an identifier in an EXTERNAL).

Example:

```
EXTERNAL SIN, COS
.
.
.
CALL TRIGF(SIN,1.5,ANSWER)
.
.
.
CALL TRIGF(COS,187,ANSWER)
.
.
.
END
SUBROUTINE TRIGF(FUNC,ARG,ANSWER)
.
.
.
ANSWER = FUNC(ARG)
.
.
.
RETURN
END
```

# Part II

# THE FORTRAN OPERATING ENVIRONMENT

PART II
TABLE OF CONTENTS



TABLES

# CHAPTER 1
# GENERAL PROCEDURES

There are two steps involved in obtaining an executable computer program from a FORTRAN source program. The first step, preparing an object module, requires use of the FORTRAN compiler and PAL assembler to obtain both compilation and assembly. The second step, preparing a load module, requires the use of the LINK-11 program to obtain those portions of the FORTRAN Object Time System required to run the user program.

## 1.1 PREPARING AN OBJECT MODULE

The FORTRAN compiler produces code which must be assembled by the PAL assembler.

To request compilation, the user first types:

.RUN FORTRN

When the compiler is ready to accept input, the character # is printed. On the same line, the user issues a command string* of the form:

device: obj-file, device: list file <device: source file

where device specifies the location of the file using one of the mnemonics given in Table 3-1 of Chapter 3.

Either or both output file specifications may be omitted. However, if an output file is specified without an extension, the compiler creates one as follows:

Object file - PAL
Source list file - LST

If no extension is specified for the input file name, the compiler looks for, and expects the extension FTN.

---

*The command string adheres to the requirements of the Disk Operating System (DOS) Command String Interpreter (CSI).

As an example, the command string:

BESSEL, OUTPUT <BESSEL

will cause the compiler to compile the program BESSEL.FTN. The source program listing will be written on a file called OUTPUT.LST, and the compiler will create an object file called BESSEL.PAL.

If a syntactical error is detected in the command string, the compiler will output the command up to and including the error, advance the carriage and print the character #. The user must retype the entire string. Compilation may be aborted and the compiler restarted by typing CTRL/C[1] and the Monitor command REstart.

## 1.2 PREPARING A LOAD MODULE

A user program produced by the FORTRAN compiler is executed in conjunction with the FORTRAN Object Time System (OTS), a library of programs which support a variety of source-language facilities. The OTS is divided into four parts - input-output processing routines, mathematical subroutines and function generators, miscellaneous service routines, and input-output device tables and buffers and run switches.

The input-output portion of the OTS includes routines to build input and output records and to manipulate files via the system monitor. This section also includes a format processor, which associates items in a FORTRAN FORMAT statement with items in an I/O list and I/O record and performs required conversions, and a set of monitor interface routines which act as device drivers.

The mathematical subroutines perform arithmetic operations not supported by the PDP-11 hardware, such as floating point and double-precision arithmetic. The function generator routines include the standard mathematical functions supported by FORTRAN such as SIN and ATAN. (See Appendix C for a list of standard functions.)

Miscellaneous service routines perform a variety of functions such as array-index arithmetic and error processing.

The final portion of OTS maintains information required for input-output operations (link blocks, file blocks, device status switches and buffers). It also contains any global values or switches required for program execution.

---

[1] Holding down the CTRL key and typing C.

A load module consists of the user's object module and those programs in OTS required for its execution. A load module is prepared using the LINK-11 program. Information on linking object modules and performing library searches may be obtained in the LINK-11 manual.

## 1.3 ERROR PROCESSING

The Object Time System detects run-time errors and prints error messages on the assigned message logging device. Errors are divided into classes on the basis of functional similarity such as FUNCTION errors, recoverable I/O errors, and so on. Each class of error will have a maximum allowed occurrence level before which it will not terminate execution. This number may be reset by the user via the subroutine SETERR (Appendix C). Error messages are given in Appendix F.

# CHAPTER 2
# SUBPROGRAMS

All subprograms which are explicitly invoked by the user (as described in Part I, Chapter 7) are called via the convention described in Section 2.1 below. Those Object Time System Subprograms which are automatically invoked by FORTRAN statements to perform operations not supported by the PDP-11 hardware are called using the convention described in Section 2.2.

## 2.1 STANDARD SUBROUTINE CALLS

All user-defined or system subprograms which are invoked by a call or a function reference in the source program obey the calling conventions described below.

Argument addresses are placed in a list following the subprogram call. The standard sequence will be:

```
        .GLOBL SUBR
        .
        .
        .
        JSR     R5,SUBR
        BR      XX
        Arg1
        Arg2
        .
        .
        .
        ARGn
     XX:
```

Note that the even byte of the branch instruction following the JSR contains the number of arguments* and is pointed to by R5 when SUBR is entered.

Functions store the result in registers R0-R3 depending on the function type and return control via RTS R5. Thus, an integer function result is returned in R0 and a real function result in R0 and R1; double-precision and complex in R0, R1, R2 and R3.

---

*See PAL-11R Assembler Manual (especially Section 7.12) for more information on the machine format of the Branch instruction.

## 2.2 THREADED CODE

Most FORTRAN statements generate calls to internal subprograms. These calls are based on the simple Polish method for evaluating expressions. This method assumes that a typical expression consists of a large number of very simple operations done in a linear sequence. These operations use the stack for evaluating all expressions.

For example, the FORTRAN program

```
A=1.
B=1.
```

would generate the following code for each expression:

```
;A=1.
$P0001
.GLOBL $POP3
$POP3,A

;B=1.
$P0001
$POP3,B
```

Most routines referred to by the calls generated above are found at the end of the assembly listing. Other routines are linked in from the FORTRAN library.

The routine $P0001 would be

```
$P0001:    MOV     #$R0000+4,R0      ;GET VALUE
           BR      $F0001

$R0000:    040200                    ;FLOATING POINT CONSTANT 1
           000000

$F0001:    MOV - (R0),-(SP)          ;PUSH 2 WORD VALUE ONTO
           MOV - (R0),-(SP)          ;STACK
           JMP @ (R4)+               ;GO TO NEXT ROUTINE
```

The routine $POP3 is in the library. $POP3 pops a value off the stack into the memory location whose address follows the call to $POP3 in the threaded code. $POP3, A pops the value on top of the stack into 2 memory words reserved for A. Similarly, $POP3, B saves the two word value found on top of the stack, in B.

The expression C=A+B would result in

```
$P0002
$P0003
.GLOBL  $ADR
$ADR
$POP3,C
```

where $P0002 and $P0003 would push the values of A and B onto the stack, $ADR would add the two real values on top of the stack, and $POP3 would save the result as variable C.

In order to call one of these internal subprograms from an assembly language program, an entry to this Polish mode of execution must be made via JSR R4, $POLSH, which invokes the routine:

```
$POLSH:  TST (SP) +     ;DELETE USELESS OLD
                        ;VALUE OF R4
         JMP @ (R4)+    ;PUSHED ON ENTRY
                        ;BY JSR
```

The next word following the call to $POLSH will be the first word of Polish code to be executed.

Internal subprograms are listed in Appendix E.

To exit from Polish mode, simply direct the last Polish routine entered to jump to the next location in sequence. This is accomplished by placing the address of that word following the last Polish call. Example:

```
        $POP3,A     ;POP3 WILL JUMP TO @(R4),
        .+2         ;WHICH IS NEXT.
NEXT:   ~~~~~~
```

Note that this mode of execution is exited for execution of subroutine and function calls via the standard PDP-11 calling convention.

In the last example above, if C = A+B were followed by a CALL SUB (ARG), then the code following $P0003, C would be

```
.+2
JSR R5,SUB
BR .+6
ARG
etc.
```

# CHAPTER 3
# FORTRAN INPUT-OUTPUT

Input-output functions of a FORTRAN-compiled user program are performed by the Object Time System. All input-output is accomplished through the Monitor and is device-independent. The user may, therefore, do logical assignments at run-time using Monitor ASSIGN commands or by calling the SETFIL subroutine (Appendix C).

## 3.1 FILE STRUCTURES

OTS input-output facilities are provided by one of three packages of OTS routines - formatted, unformatted, and random access. The formatted input-output routines will read or write formatted ASCII records whose maximum length is 133 characters. On input, longer records will simply be truncated. For shorter records, the last character (line feed, form feed, or vertical tab) will be deleted and the record will be padded with blanks. The next-to-last character is also deleted if it is a carriage return. For output, if the device is a line printer or teleprinter, a carriage return and vertical tab are appended to the end of each record. The first character of each record is interpreted as a line spacing command. If the device is not a printer, a carriage return and line feed are appended.

Unformatted input-output routines read or write formatted binary records of any size with parity checking. Records will be transmitted in segments up to 63 words long. The first word of each segment is a control word with one of the following meanings.

| Value | Meaning |
|---|---|
| 0 | Not first or last segment |
| 1 | First segment |
| 2 | Last segment |
| 3 | First and last segment |

The random access routines read or write binary records. The maximum allowable record length is 32767 bytes. These routines determine the block number and the displacement to the proper record from the user program's DEFINE FILE statement and the record number given in the input-output request.

## 3.2 DEVICE ASSIGNMENT

If the user does not supply run-time assignment information, FORTRAN logical device number 6 is assigned to the teleprinter and all others are assigned to disk and given the name FOR0nn.DAT (where nn is the device number). Device number 6 is also assumed to be the message logging device and must be available for formatted ASCII output when required. Table 3-1 gives the available devices.

Table 3-1
PDP-11 FORTRAN IV Standard Peripheral Devices

| Name | Mnemonic | Input/Output | | Operation |
|---|---|---|---|---|
| | | Formatted | Unformatted | |
| Disk (includes disk packs and drums) | DC DF DK | Yes | Yes | READ/WRITE |
| DECtapes | DT | Yes | Yes | READ/WRITE |
| Line Printer | LP | Yes | No | WRITE |
| Magtape | MT | Yes | Yes | READ/WRITE |
| Paper Tape Punch (High-speed) | PP | Yes | Yes | WRITE |
| Paper Tape Reader (High-speed) | PR | Yes | Yes | READ |
| Low-Speed Punch and Reader | PT | Yes | Yes | READ/WRITE |
| Teletype - User | KB | Yes | No | READ/WRITE |

Logical device assignment is governed by the Device Table which contains entries for eight devices but may be expanded to handle more. Each entry, as shown in Table 3-2 below, is 16 words long and preceded by an 11-word header.

Table 3-2
Device Table Entry

| | | |
|---|---|---|
| HEADER | Word 1 | Address of entry for error routine message file |
| | Word 2 | Number of entries in device vector table |
| | Word 3 | Device number of message logging file |
| | Words 4-11 | Addresses of device table entries for each of the devices one through eight |
| | Word 1 | Link Pointer (from Link Block, after INIT) |
| | Word 2 | Physical Device Name (RAD50 /XXX/; XXX = DF (is KB for Log Dev) |
| | Word 3 | Unit Num (Default 0) /How Open (File Block - 2) |

Table 3-2 (Cont)
Device Table Entry

| | | |
|---|---|---|
| ENTRY | Words 4 & 5 | File Name (RAD50 /FOR/, /NNN/; NNN = Entry Num) |
| | Word 6 | File Extension (RAD50 /DAT/) |
| | Word 7 | Switches* and Protect Code (Default = 233) |
| | Word 8 | Status/Mode (from Line Buff Header) |
| | Word 9 | Count of I/O Operations for this Device |
| | Words 10-14 | Unused for formatted and unformatted I/O |
| | Word 15 | User ID code (UIC) - default = 0 |
| | Word 16 | Addr of Error Value VAR (from CALL SETFIL) |
| For Random I/O Words 8=14 are: | | |
| | Word 8 | Function Word |
| | Word 9 | Block Number |
| | Word 10 | Buffer Addr |
| | Word 11 | Buffer Length |
| | Word 12 | Associated VAR addr (from DEFINE FILE) |
| | Word 13 | Max Num of Records (from DEFINE FILE) |
| | Word 14 | Record Length (from DEFINE FILE) |

*Switches are as follows:

| Bit | Setting | Meaning |
|---|---|---|
| 0-1 | 0 | Closed |
| | 1 | Open formatted |
| | 2 | Open unformatted |
| | 3 | Open random |

By changing the number in word 2, the user may modify the number of entries to be considered. If fewer are desired, he may change one of the eight device words to zero. If more are desired, he may expand this 8-word sequence.

## 3.3 INPUT-OUTPUT BUFFERS

Both input and output use a single buffer. A wait will be issued after each READ or WRITE request; that is, I/O will be synchronous. The buffer is preceded by a link block, file block, and buffer header.

# APPENDIX A
# STATEMENT SUMMARY

| Statement | Form | Effect | See Section |
|---|---|---|---|
| Arithmetic | a = b | the value of expression b is assigned to the variable a | 3.2 |
| Arithmetic function definition | $f(a_1 \ldots) = x$ | the value of expression x is assigned to $f(a_1 \ldots)$ after parameter substitution | 7.1 |
| ASSIGN | ASSIGN n TO v | statement number n is assigned as the value of integer variable v for use in an assigned GO TO statement | 3.2 |
| BACKSPACE | BACKSPACE u | peripheral device u is back-spaced one record | 5.3 |
| BLOCK DATA | BLOCK DATA | identifies a block data sub-program | 7.3 |
| CALL | CALL prog<br>CALL prog $(a_1 \ldots)$ | invokes subroutine named prog, supplying arguments when required | 7.2.2 |
| COMMON | COMMON/block1/a,b,c,/... | variables (A,B,C) are assigned to a common block | 6.1.2 and 6.1.4 |
| CONTINUE | CONTINUE | no processing, target for transfers | 4.4 |
| DATA | DATA var $\text{list}_1$/val $\text{list}_1$/... | assigns initial or constant values to variables | 6.2 |
| DEFINE FILE | DEFINE FILE $a_1(m_1,1_1,U,v_1)\ldots$ | describes a disk file for sequential I/O | 5.1.2 |
| DIMENSION | DIMENSION array $(v_1,v_2,v_3)\ldots$ | storage allocated according to dimensions specified for the array | 6.1.1 |
| DO | DO n i = $m_1,m_2,m_3$ | statements following the DO up to statement n are iterated for values of integer variable i, starting at i = $m_1$, incrementing by $m_3$, terminating when i $\geq m_2$ | 4.3 |

| Statement | Form | Effect | See Section |
|---|---|---|---|
| END FILE | END FILE u | invokes the monitor CLOSE facility for device u | 5.3 |
| EXTERNAL | EXTERNAL subprog, . . . | declares a subprogram for use by other subprograms | 7.4 |
| FIND | FIND(a'b) | disk read/write mechanism positioned to record b of file a | 5.3 |
| FORMAT | n FORMAT (field description$_1$. . ./. . .) | specifies conversions between internal and external representation of data | 5.1.1 |
| FUNCTION | t FUNCTION f($a_1$. . .) | indicates an external function definition (t is an optional type specification) | 7.1.2 |
| GO TO | | transfers control to: | |
| | (1) GO TO n | (1) statement n | 4.1.1 |
| | (2) GO TO($n_1$, . . .$n_k$),i<br>GO TO($n_1$, . . .$n_k$)i | (2) to statement $n_1$ if i = 1,<br>to statement $n_k$ if i = k | 4.1.2 |
| | (3) GO TO var<br>GO TO var($n_1$, . . .$n_k$)<br>GO TO var,($n_1$, . . .$n_k$) | (3) transfers control to statement number assigned to var optionally checking that var is assigned one of the labels $n_1$. . .$n_k$ | 4.1.3 |
| IMPLICIT | IMPLICIT type$_1$($a_1$. . .). . . | the given type is assigned to any variable (not mentioned in an explicit type specification) which begins with one of the letters given as an argument | 6.4 |
| PAUSE | PAUSE<br>PAUSE number | program execution interrupted and number printed, if given | 6.5 |
| READ | READ(u,f) list<br>READ(u,f)<br>READ(u) list<br>READ(a'r) list<br>READ(u,f,END=n) list<br>READ(u,f,ERR=n) list<br>READ(u,f,END=n,ERR=n) list | reads a record from a peripheral device according to specifications given in the arguments of the statement | 5.2.4 |
| RETURN | RETURN | returns control from a subprogram to the calling program | 7.2.3 |
| REWIND | REWIND u | repositions designated unit to the beginning of the file | 5.3 |
| STOP | STOP<br>STOP number | terminates program execution and prints number specified | 4.6 |

| Statement | Form | Effect | See Section |
|---|---|---|---|
| SUBROUTINE | SUBROUTINE prog($a_1$,...) | declares prog to be a subroutine subprogram and $a_1$..., if supplied, as dummy arguments | 7.2.1 |
| WRITE | WRITE(u,f)<br>WRITE(u,f) list<br>WRITE(u) list<br>WRITE(a'r) list<br>WRITE(u,f,END=n) list<br>WRITE(u,f,ERR=n) list<br>WRITE(u,f,END=n,ERR=n) list | writes a record to a peripheral device according to specifications given in the arguments of the statement | 5.2.3 |

# APPENDIX B
# ASCII CHARACTER SET

| EVEN PARITY BIT | 7-BIT OCTAL CODE | CHARACTER | REMARKS |
|---|---|---|---|
| 0 | 000 | NUL | NULL, TAPE FEED, CONTROL SHIFT P. |
| 1 | 001 | SOH | START OF HEADING; ALSO SOM, START OF MESSAGE, CONTROL A. |
| 1 | 002 | STX | START OF TEXT; ALSO EOA, END OF ADDRESS, CONTROL B. |
| 0 | 003 | ETX | END OF TEXT; ALSO EOM, END OF MESSAGE, CONTROL C. |
| 1 | 004 | EOT | END OF TRANSMISSION (END); SHUTS OFF TWX MACHINES, CONTROL D. |
| 0 | 005 | ENQ | ENQUIRY (ENQRY); ALSO WRU, CONTROL E. |
| 0 | 006 | ACK | ACKNOWLEDGE; ALSO RU, CONTROL F. |
| 1 | 007 | BEL | RINGS THE BELL. CONTROL G. |
| 1 | 010 | BS | BACKSPACE; ALSO FEO, FORMAT EFFECTOR. BACK-SPACES SOME MACHINES, CONTROL H. |
| 0 | 011 | HT | HORIZONTAL TAB. CONTROL I. |
| 0 | 012 | LF | LINE FEED OR LINE SPACE (NEW LINE); ADVANCES PAPER TO NEXT LINE, DUPLICATED BY CONTROL J. |
| 1 | 013 | VT | VERTICAL TAB (VTAB). CONTROL K. |
| 0 | 014 | FF | FORM FEED TO TOP OF NEXT PAGE (PAGE). CONTROL L. |
| 1 | 015 | CR | CARRIAGE RETURN TO BEGINNING OF LINE. DUPLICATED BY CONTROL M. |
| 1 | 016 | SO | SHIFT OUT; CHANGES RIBBON COLOR TO RED. CONTROL N. |
| 0 | 017 | SI | SHIFT IN; CHANGES RIBBON COLOR TO BLACK. CON-TROL O. |
| 1 | 020 | DLE | DATA LINK ESCAPE. CONTROL P (DC0). |
| 0 | 021 | DC1 | DEVICE CONTROL 1, TURNS TRANSMITTER (READER) ON, CONTROL Q (X ON). |
| 0 | 022 | DC2 | DEVICE CONTROL 2, TURNS PUNCH OR AUXILIARY ON. CONTROL R (TAPE, AUX ON). |
| 1 | 023 | DC3 | DEVICE CONTROL 3, TURNS TRANSMITTER (READER) OFF, CONTROL S (X OFF). |
| 0 | 024 | DC4 | DEVICE CONTROL 4, TURNS PUNCH OR AUXILIARY OFF. CONTROL T (~~TAPE,~~ AUX OFF). |
| 1 | 025 | NAK | NEGATIVE ACKNOWLEDGE; ALSO ERR, ERROR. CONTROL U. |

| EVEN PARITY BIT | 7-BIT OCTAL CODE | CHARACTER | REMARKS |
|---|---|---|---|
| 1 | 026 | SYN | SYNCHRONOUS IDLE (SYNC). CONTROL V. |
| 0 | 027 | ETB | END OF TRANSMISSION BLOCK; ALSO LEM, LOGICAL END OF MEDIUM. CONTROL W. |
| 0 | 030 | CAN | CANCEL (CANCL). CONTROL X. |
| 1 | 031 | EM | END OF MEDIUM. CONTROL Y. |
| 1 | 032 | SUB | SUBSTITUTE. CONTROL Z. |
| 0 | 033 | ESC | ESCAPE. PREFIX. CONTROL SHIFT K. |
| 1 | 034 | FS | FILE SEPARATOR. CONTROL SHIFT L. |
| 0 | 035 | GS | GROUP SEPARATOR. CONTROL SHIFT M. |
| 0 | 036 | RS | RECORD SEPARATOR. CONTROL SHIFT N. |
| 1 | 037 | US | UNIT SEPARATOR. CONTROL SHIFT O. |
| 1 | 040 | SP | SPACE. |
| 0 | 041 | ! | |
| 0 | 042 | " | |
| 1 | 043 | # | |
| 0 | 044 | $ | |
| 1 | 045 | % | |
| 1 | 046 | & | |
| 0 | 047 | ' | ACCENT ACUTE OR APOSTROPHE. |
| 0 | 050 | ( | |
| 1 | 051 | ) | |
| 1 | 052 | * | |
| 0 | 053 | + | |
| 1 | 054 | , | |
| 0 | 055 | - | |
| 0 | 056 | . | |
| 1 | 057 | / | |
| 0 | 060 | 0 | |
| 1 | 061 | 1 | |
| 1 | 062 | 2 | |
| 0 | 063 | 3 | |
| 1 | 064 | 4 | |
| 0 | 065 | 5 | |
| 0 | 066 | 6 | |
| 1 | 067 | 7 | |
| 1 | 070 | 8 | |
| 0 | 071 | 9 | |
| 0 | 072 | : | |
| 1 | 073 | ; | |
| 0 | 074 | < | |
| 1 | 075 | = | |
| 1 | 076 | > | |
| 0 | 077 | ? | |
| 1 | 100 | @ | |
| 0 | 101 | A | |
| 0 | 102 | B | |
| 1 | 103 | C | |
| 0 | 104 | D | |
| 1 | 105 | E | |
| 1 | 106 | F | |

| EVEN PARITY BIT | 7-BIT OCTAL CODE | CHARACTER | REMARKS |
|---|---|---|---|
| 0 | 107 | G | |
| 0 | 110 | H | |
| 1 | 111 | I | |
| 1 | 112 | J | |
| 0 | 113 | K | |
| 1 | 114 | L | |
| 0 | 115 | M | |
| 0 | 116 | N | |
| 1 | 117 | O | |
| 0 | 120 | P | |
| 1 | 121 | Q | |
| 1 | 122 | R | |
| 0 | 123 | S | |
| 1 | 124 | T | |
| 0 | 125 | U | |
| 0 | 126 | V | |
| 1 | 127 | W | |
| 1 | 130 | X | |
| 0 | 131 | Y | |
| 0 | 132 | Z | |
| 1 | 133 | [ | SHIFT K. |
| 0 | 134 | \ | SHIFT L. |
| 1 | 135 | ] | SHIFT M. |
| 1 | 136 | ↑ | |
| 0 | 137 | ← | |
| 0 | 140 | ` | ACCENT GRAVE. |
| 0 | 175 | } | THIS CODE GENERATED BY ALT MODE. |
| 0 | 176 | ~ | THIS CODE GENERATED BY ESC KEY (IF PRESENT). |
| 1 | 177 | DEL | DELETE, RUB OUT. |
| | | | LOWER CASE ALPHABET FOLLOWS (TELETYPE MODEL 37 ONLY). |
| 1 | 141 | a | |
| 1 | 142 | b | |
| 0 | 143 | c | |
| 1 | 144 | d | |
| 0 | 145 | e | |
| 0 | 146 | f | |
| 1 | 147 | g | |
| 1 | 150 | h | |
| 0 | 151 | i | |
| 0 | 152 | j | |
| 1 | 153 | k | |
| 0 | 154 | l | |
| 1 | 155 | m | |
| 1 | 156 | n | |
| 0 | 157 | o | |
| 1 | 160 | p | |

| EVEN PARITY BIT | 7-BIT OCTAL CODE | CHARACTER | REMARKS |
|---|---|---|---|
| 0 | 161 | q | |
| 0 | 162 | r | |
| 1 | 163 | s | |
| 0 | 164 | t | |
| 1 | 165 | u | |
| 1 | 166 | v | |
| 0 | 167 | w | |
| 0 | 170 | x | |
| 1 | 171 | y | |
| 1 | 172 | z | |
| 0 | 173 | { | |
| 1 | 174 | \| | |

## C.1 FUNCTIONS

| Function | Function Name | Definition | Number of Arguments | Type of | |
|---|---|---|---|---|---|
| | | | | Argument | Function |
| Absolute value: | | | | | |
| Real | ABS | \|arg\| | 1 | Real | Real |
| Integer | IABS | \|arg\| | 1 | Integer | Integer |
| Double-precision | DABS | \|arg\| | 1 | Double | Double |
| Complex to real | CABS | $c=(x^2+y^2)^{1/2}$ | 1 | Complex | Real |
| Conversion: | | | | | |
| Integer to real | FLOAT | | 1 | Integer | Real |
| Real to integer | IFIX | Result is largest integer $\leq a$ | 1 | Real | Integer |
| Double to real | SNGL | | 1 | Double | Real |
| Real to double | DBLE | | 1 | Real | Double |
| Complex to real (obtain real part) | REAL | | 1 | Complex | Real |
| Complex to real (obtain imaginary part) | AIMAG | | 1 | Complex | Real |
| Real to complex | CMPLX | $c=Arg_1+i*Arg_2$ | 2 | Real | Complex |
| Truncation: | | | | | |
| Real to real | AINT | Sign of arg * largest integer $\leq$\|arg\| | 1 | Real | Real |
| Real to integer | INT | | 1 | Real | Integer |
| Double to integer | IDINT | | 1 | Double | Integer |

# APPENDIX C
# FORTRAN-IV LIBRARY SUBPROGRAMS

| Function | Function Name | Definition | Number of Arguments | Type of | |
|---|---|---|---|---|---|
| | | | | Argument | Function |
| Remaindering: | | | | | |
| Real | AMOD | The remainder when Arg 1 is divided by Arg 2 | 2 | Real | Real |
| Integer | MOD | | 2 | Integer | Integer |
| Double-precision | DMOD | | 2 | Double | Double |
| Maximum value: | | | | | |
| | AMAX0 | $Max(Arg_1, Arg_2, \ldots)$ | $\geq 2$ | Integer | Real |
| | AMAX1 | | | Real | Real |
| | MAX0 | | | Integer | Integer |
| | MAX1 | | | Real | Integer |
| | DMAX1 | | | Double | Double |
| Minimum value: | | | | | |
| | AMIN0 | $Min(Arg_1, Arg_2, \ldots)$ | $\geq 2$ | Integer | Real |
| | AMIN1 | | | Real | Real |
| | MIN0 | | | Integer | Integer |
| | MIN1 | | | Real | Integer |
| | DMIN1 | | | Double | Double |
| Transfer of sign: | | | | | |
| Real | SIGN | $Sgn(Arg_2)*\|Arg_1\|$ | 2 | Real | Real |
| Integer | ISIGN | | 2 | Integer | Integer |
| Double-precision | DSIGN | | 2 | Double | Double |
| Positive difference: | | | | | |
| Real | DIM | $Arg_1 - Min(Arg_1, Arg_2)$ | 2 | Real | Real |
| Integer | IDIM | | 2 | Integer | Integer |
| Exponential: | | | | | |
| Real | EXP | $e^{Arg}$ | 1 | Real | Real |
| Double | DEXP | | 1 | Double | Double |
| Complex | CEXP | | 1 | Complex | Complex |

| Function | Function Name | Definition | Number of Arguments | Type of | |
|---|---|---|---|---|---|
| | | | | Argument | Function |
| Logarithm: | | | | | |
| Real | ALOG | $\log_e$ (Arg) | 1 | Real | Real |
| | ALOG10 | $\log_{10}$ (Arg) | 1 | Real | Real |
| Double | DLOG | $\log_e$ (Arg) | 1 | Double | Double |
| | DLOG10 | $\log_{10}$ (Arg) | 1 | Double | Double |
| Complex | CLOG | $\log_e$ (Arg) | 1 | Complex | Complex |
| Square root: | | | | | |
| Real | SQRT | $(\text{Arg})^{1/2}$ | 1 | Real | Real |
| Double | DSQRT | $(\text{Arg})^{1/2}$ | 1 | Double | Double |
| Complex | CSQRT | $c=(x+i\ y)^{1/2}$ | 1 | Complex | Complex |
| Sine: | | | | | |
| Real (radians) | SIN | sin (Arg) | 1 | Real | Real |
| Double (radians) | DSIN | sin (Arg) | 1 | Double | Double |
| Complex | CSIN | sin (Arg) | 1 | Complex | Complex |
| Cosine: | | | | | |
| Real (radians) | COS | cos (Arg) | 1 | Real | Real |
| Double (radians) | DCOS | cos (Arg) | 1 | Double | Double |
| Complex | CCOS | cos (Arg) | 1 | Complex | Complex |
| Hyperbolic: | | | | | |
| Tangent | TANH | tanh (Arg) | 1 | Real | Real |
| Arc - sine | ASIN | asin (Arg) | 1 | Real | Real |
| Arc tangent | | | 1 | Real | Real |
| Real | ATAN | atan (Arg) | 1 | Real | Real |
| Double | DATAN | atan (Arg) | 1 | Double | Double |
| quotient of | ATAN2 | atan $(\text{Arg}_1/\text{Arg}_2)$ | 2 | Real | Real |
| two arguments | DATAN2 | atan $(\text{Arg}_1/\text{Arg}_2)$ | 2 | Double | Double |
| Complex conjugate | CONJG | Arg=X+i*Y, C=X-i*Y | 1 | Complex | Complex |
| Random number | RAN | result is a random number between zero and one (uniform distribution) | 1 | Integer, Real, Double, or Complex | Real |

## C.2 SUBROUTINES

| Subroutine Name | Call Format (Optional Arguments are underlined) | Effect |
|---|---|---|
| DATE | CALL DATE (array) | Places today's date into the three-word array specified in the call. The date consists of left-justified ASCII characters in the form:<br><br>mmddyy<br><br>where mm is a 2-digit month, dd a 2-digit day, and yy is a 2-digit year. |
| PDUMP | CALL PDUMP $\underline{(L_1,U_1,F_1,\ldots,L_n,U_n,F_n)}$ | Causes specified portions of core to be dumped.<br><br>$L_1$ and $U_1$ are variables giving the limits of the dump (either may be upper or lower limits).<br><br>$F_1$ is an integer indicating the format in which the dump is to be performed:<br><br>0 = octal, 1 = real, 2 = integer, and 3 = ASCII.<br><br>If no limits are given, the entire job area is dumped. If one limit is given, core is dumped from that point to the end of the job area. If F is not given, octal is assumed.<br><br>Control is returned to the calling program when the dump is completed. |
| SETERR | CALL SETERR (CLASS, MAX) | Resets maximum occurrence count for specified class of errors. The argument CLASS is an integer indicating the error class affected. MAX is an integer with the following meanings:<br><br>>0 = log until MAX<br>0 = log and ignore<br>-1 = no log and ignore<br>-2 = no log and exit<br>-3 = immediate abort |

| Subroutine Name | Call Format (Optional Arguments are underlined) | Effect |
|---|---|---|
| SETFIL | CALL SETFIL (n, FILE, ERR, DEV, Un, ID, PC, CS, RECL, NREC)<br><br>NOTE: Optional arguments can only be provided in sequence as above; that is, any trailing set may be omitted. n and FILE are always required. | Overrides default values for a FORTRAN device assignment. Arguments are as follows:<br><br>n = logical device number<br>FILE = file name and extension<br>ERR = a variable into which both error returns from this routine and from I/O with the ERR option will be placed<br>DEV = a device mnemonic (e.g., DT or LP)<br>Un = unit number (e.g., 1 if device DT1)<br>ID = user ID code<br>PC = protect code<br>CS = 1 for non-random or<br>2 for random<br>RECL = record length for CS=1<br>NREC = number of records for CS=1 |

# APPENDIX D
# FORTRAN WORD FORMATS

## D.1 INTEGER FORMAT

| Sign<br>0 +<br>1 - | Binary number |
| --- | --- |
| 15 | 14 … 0 |

In two-word format, an integer is assigned two words. Only the high-order word is significant.

## D.2 REAL FORMAT

| | Sign<br>0 +<br>1 - | Binary excess 128 exponent | high-order mantissa |
| --- | --- | --- | --- |
| word n | 15 | 14 … 7 | 6 … 0 |

| | Low order-mantissa |
| --- | --- |
| word n+2 | 15 … 0 |

This format is limited to normalized numbers. Since the high-order bit of the mantissa is always 1, it is discarded, giving an effective precision of 24 bits.

## D.3 DOUBLE-PRECISION FORMAT

| | Sign<br>0 +<br>1 - | Binary excess 128 exponent | high-order mantissa |
| --- | --- | --- | --- |
| word n | 15 | 14 … 7 | 6 … 0 |

| word n+2 | Low order mantissa | |
|---|---|---|
| | 15 | 0 |

| word n+4 | Lower order mantissa | |
|---|---|---|
| | 15 | 0 |

| word n+6 | Lowest order mantissa | |
|---|---|---|
| | 15 | 0 |

The effective precision is 56 bits.

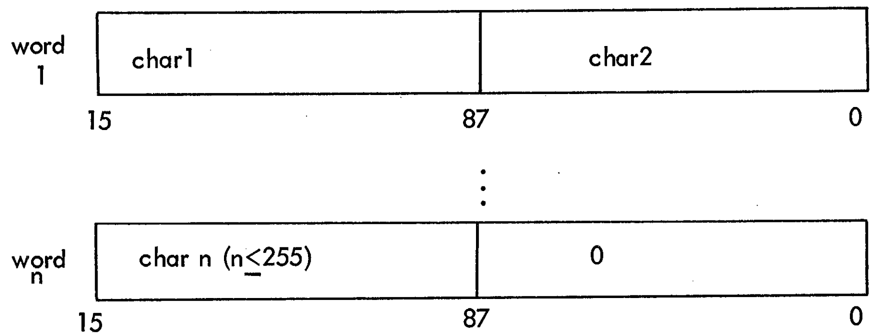
## D.4 COMPLEX FORMAT

| | | | | |
|---|---|---|---|---|
| word n | Sign<br>0 +<br>1 - | Binary excess 128 exponent | high-order mantissa | Real Part |
| | 15 | 14 … 7 | 6 … 0 | |
| word n+2 | Low order mantissa | | | |
| | 15 … 0 | | | |
| word n+4 | Sign<br>0 +<br>1 - | Binary excess 128 exponent | high-order mantissa | Imaginary Part |
| | 15 | 14 … 7 | 6 … 0 | |
| word n+6 | Low order mantissa | | | |
| | 15 … 0 | | | |

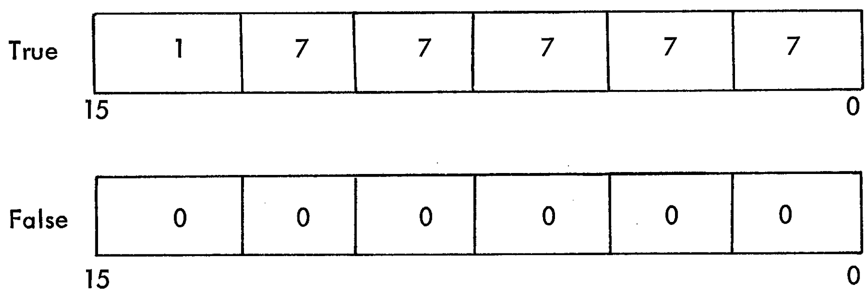
## D.5 BYTE FORMAT

| unspecified | data $item_1$ |
|---|---|
| 15 … 8 | 7 … 0 |

## D.6 HOLLERITH FORMAT




## D.7 LOGICAL FORMAT



# APPENDIX E
# INTERNAL SUBPROGRAMS

| Subprogram Name | Function |
| --- | --- |
| $IR | FLOAT THE INTEGER ON THE TOP OF THE STACK. (65 words) |
| $ID | ENTRY INTO $IR WHICH FIRST MOVES THE ARGUMENT DOWN TWO WORDS (FILLING IN WITH ZEROS) BEFORE EXECUTING THE $IR CODE. |
| $DR | PUT THE HIGH ORDER WORDS OF THE DOUBLE PRECISION QUANTITY ON THE TOP OF THE STACK TRUNCATING TO REAL FORMAT. (5 words) |
| $RD | APPEND A DOUBLE WORD OF ZEROS TO THE REAL QUANTITY ON THE TOP OF THE STACK. (10 words) |
| $RI | TRUNCATE AND FIX THE REAL NUMBER OF THE TOP OF THE STACK. (40 words) |
| $DI | ENTRY INTO $RI WHICH MOVES THE ARGUMENT UP THE STACK TWO WORDS (DISCARDING THE LOW ORDER PART) BEFORE EXECUTING THE $RI CODE. |
| $ADR | REPLACE THE TWO REAL NUMBERS ON THE TOP OF THE STACK WITH THEIR SUM. NO CODES WILL BE SET. (135 words) |
| $SBR | ENTRY IN $ADR WHICH NEGATES THE NUMBER ON TOP OF THE STACK BEFORE DOING THE ADD. |
| $ADD | REPLACE THE TWO DOUBLE PRECISION NUMBERS ON THE TOP OF THE STACK WITH THEIR SUM. NO CODES WILL BE SET. (210 words) |
| $SBD | ENTRY IN $ADD WHICH NEGATES THE NUMBER ON THE TOP OF THE STACK BEFORE DOING THE ADD. |
| $CMR | COMPARE CORRESPONDING WORDS OF THE TWO ITEMS ON THE STACK UNTIL A MISMATCH IS FOUND (IF ONE EXISTS). CLEAR THE STACK AND RETURN THE Z AND N CODES DEFINED IN 130-309-001 SECTION 3.1.2.2. (25 words) |
| $CMD | THIS IS THE SAME AS $CMR EXCEPT THAT THE ITEMS ARE DOUBLE PRECISION. (30 words) |

(Continued on next page)

| Subprogram Name | Function |
|---|---|
| $ISR | TEST AND FLUSH THE REAL NUMBER ON TOP OF THE STACK AND RETURN TO @(R4) IF IT IS NEGATIVE, @(R4+2) IF ZERO, AND @(R4+4) IF POSITIVE. (20 words) |
| $ISD | ENTRY IN $ISR FOR DOUBLE PRECISION. |
| $MLI | REPLACE THE TWO INTEGERS ON THE TOP OF THE STACK WITH THEIR PRODUCT. (50 words) |
| $MLR | REPLACE THE TWO REAL NUMBERS ON THE TOP OF THE STACK WITH THEIR PRODUCT. (100 words) |
| $MLD | REPLACE THE TWO DOUBLE PRECISION NUMBERS ON THE TOP OF THE STACK WITH THEIR PRODUCT. (170 words) |
| $DVI | REPLACE THE TWO INTEGERS ON THE TOP OF THE STACK WITH THE INTEGER PART OF THE QUOTIENT OF THE TOP STACK ITEM DIVIDED INTO THE SECOND ITEM. A ZERO DIVISOR RESULTS IN A CALL TO ERROR. (125 words) |
| $DVD | REPLACE THE TWO DOUBLE PRECISION NUMBERS ON THE TOP OF THE STACK WITH THEIR QUOTIENT. A ZERO DIVISOR CALLS ERROR. (210 words) |
| $EXP | EXPAND EXP (X) IN A TAYLOR SERIES AND RETURN REAL RESULT IN R0, R1. |

# APPENDIX F
# ERROR MESSAGES

Format of the message is:

FORTcccnnn - message text

where ccc is the class number in octal ASCII and nnn is the message number in octal ASCII.

If no message file exists, the format reduces to FORTcccnnn. A subroutine trace back will follow each message.

Class 0

| | |
|---|---|
| 0 | INVALID ERROR CALL |
| 1 | NO SPACE TO DO I/O |
| 2 | SUBROUTINE DIRECTLY OR INDIRECTLY REFERENCES ITSELF |

Class 1

| | |
|---|---|
| 0 | VALUE OUT OF BOUNDS (COMPUTED OR ASSIGNED GO TO) |
| 1 | DEVICE PARITY |
| 2 | CHECKSUM/PARITY ERR OR END OF DATA ERROR (RANDOM) |
| 3 | I/O ERROR |
| 4 | EOF/EOM |
| 5 | UNABLE TO ALLOCATE CONTIGUOUS FILE |
| 6 | DEFINE FILE NOT DONE (RANDOM) |
| 7 | DEFINE FILE DONE (NOT RANDOM) |
| 10 | INVALID PROTECT CODE |
| 11 | FILE DOES NOT EXIST/OR ALREADY OPEN |
| 12 | UNABLE TO OPEN |
| 13 | COMPATIBILITY ERROR |
| 14 | INVALID DEVICE NUMBER |
| 15 | INVALID RECORD NUMBER (RANDOM) |

Class 2

| | |
|---|---|
| 0 | FORMAT HAS ITEMS AND NO CONVERSION SPECS |
| 1 | PARENTHESES NESTING TOO DEEP IN FORMAT |
| 2 | CONVERSION ERROR |
| 3 | FORMAT SYNTAX ERROR |
| 4 | REFERENCE OUTSIDE OF RECORD BOUNDARIES |

INDEX

PART I FORTRAN IV COMPILER

INDEX

PART II FORTRAN Operating Environment

PDP-11 FORTRAN IV Compiler and
Object Time System
Programmer's Manual
DEC-11-KFDA-D
May 1971

# READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback -- your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability and readability.

______________________________

Did you find errors in this manual? If so, specify by page.

______________________________

How can this manual be improved?

______________________________

Other comments?

______________________________

Please state your position. ______________ Date: ________

Name: ______________ Organization: ______________

Street: ______________ Department: ______________

City: ____________ State: ____________ Zip or Country ________

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Digital Equipment Corporation
Software Information Services
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

digital equipment corporation