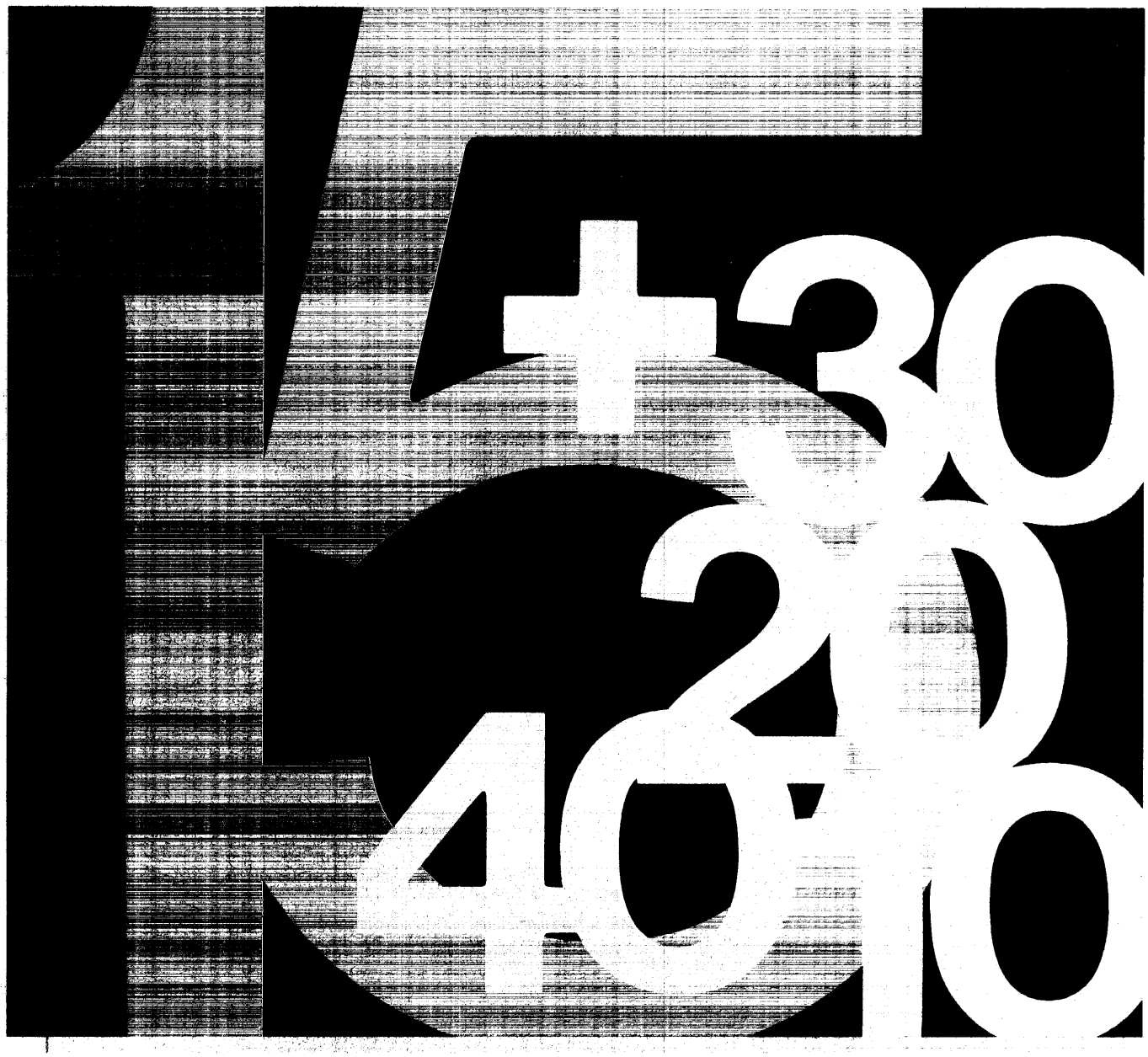
Digital Equipment Corporation
Maynard, Massachusetts

digital

Programmers' Reference Manual

PDP-15 FORTRAN IV



PDP-15
FORTRAN IV PROGRAMMERS' REFERENCE MANUAL

Order No. DEC-15-KFZB-D from Program Library, Maynard, Mass. Price: $2.50

Direct comments concerning this manual to Software Information Service, Maynard.

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Printing July 1969
2nd Printing (Rev) June 1970

Your attention is invited to the last two pages of this manual. The Reader's Comments page, when filled in and returned, is beneficial to both you and DEC. All comments received are considered when documenting subsequent manuals, and when assistance is required, a knowledgeable DEC representative will contact you. The Software Information page offers you a means of keeping up-to-date with DEC's software.

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

APPENDICES (Cont)

ILLUSTRATIONS

TABLES

PREFACE

This manual describes the FORTRAN IV language and compiler system for either the PDP-15 or PDP-9 Computer; it provides the user with the information needed to write, compile and execute FORTRAN programs on either of these computers.

The manual consists of three major parts:

Part 1, Basic FORTRAN IV Language

Part 1 is intended to familiarize the user with the FORTRAN IV coding procedures in the PDP-15 and -9 environment.

Part 2, FORTRAN IV Object Time System

Part 2 describes the group of subprograms which process compiled FORTRAN statements, particularly I/O statements, at the time of execution.

Part 3, FORTRAN Science Library

Part 3 provides detailed descriptions of the intrinsic functions, external functions, subfunctions, and arithmetic routines contained in the system Science Library.

FORTRAN IV (as described in this manual) is essentially the language specified by the United States of America Standards Institute (X3.9 - 1966) with the exceptions noted in Appendix A of this manual (located at the end of Chapter 9).

PART 1
LANGUAGE

CHAPTER 1
INTRODUCTION

1.1 FORTRAN

FORTRAN makes it unnecessary for the scientist or engineer to learn the machine language for specific computers. With FORTRAN, the user can write programs in a simple language that adapts easily to scientific usage. The FORTRAN language is composed of statements constructed in mathematical form in accordance with precisely formulated rules. A FORTRAN program consists of meaningful sequences of FORTRAN statements that direct the computer to perform specific operations and calculations; such a program is called a source program. The source program must be translated by the FORTRAN compiler program before execution; the translated version of the program is referred to as an object program. The object program is in binary code that the machine can understand.

1.2 SOURCE PROGRAM FORMAT

The FORTRAN character set consists of the 26 letters (A through Z); 10 digits (0 through 10); and 11 special characters:

Blank	
Equals	=
Plus	+
Minus	-
Asterisk	*
Slash	/
Left Parenthesis	(
Right Parenthesis	)
Comma	,
Decimal Point	.
Dollar Sign	$

1.2.1 Card Format (IBM Model 029 Keypunch Codes)

The FORTRAN source program is written on a standard FORTRAN coding sheet (see Figure 1-1), which consists of the following fields:

a. statement number field
b. line continuation field
c. statement field
d. identification field.

The FORTRAN statement is written in columns 7 through 72. If the statement is too long for one line, it can be continued in the statement field of as many lines as necessary if column 6 of each continuation line contains any numeric character other than blank or zero. There are two exceptions to this rule:

a. the DO statement must be on one line
b. the equal sign (=) of an assignment statement must appear on the first line.

For one statement to be referenced by another, a statement number must be placed in columns 1 through 5 of the first line of the referenced statement. This number is made up of digits only, and can contain from one to five digits. Leading zeros and all blanks in this field are ignored. Because statement numbers are used only for identification, they can be assigned in any order.

The FORTRAN compiler ignores the last eight columns (columns 73 through 80), which can be used for program identification, sequencing, or any other purpose desired by the user. Comments can be included in the program by putting the letter C in column 1 of each line containing a comment (or continuation of a comment). The compiler ignores these comments except for printing them.

Blanks can be used to aid readability of a FORTRAN statement, except where otherwise indicated in this manual.

1.2.2 Paper Tape Format

When FORTRAN source program statements are prepared on paper tape, the sequence of characters is exactly the same as for card input, and each line is terminated with a carriage return-line feed sequence.

A statement number (all digits) can be written as the first five characters, or the letter C can appear as the first character to indicate a comment line or a continuation of a comment line. For statement continuation lines, any numeric character other than blank or zero is written as the sixth character or as the first character after a TAB. The seventh character, which begins the statement, must be alphabetic. Each line is terminated with a carriage return-line feed.

The TAB key can increase the speed of writing FORTRAN statements on paper tape. A TAB followed by an alphabetic character begins the statement in column 7. A TAB followed by a digit causes the digit to be placed in column 6, indicating a statement continuation line. A statement number of less than five digits, followed by a TAB, causes the next character to be placed in column 6 if it is a digit, or in column 7 if it is a letter.

If a title is desired at the beginning of the tape for a listing, it must be entered as a comment line.

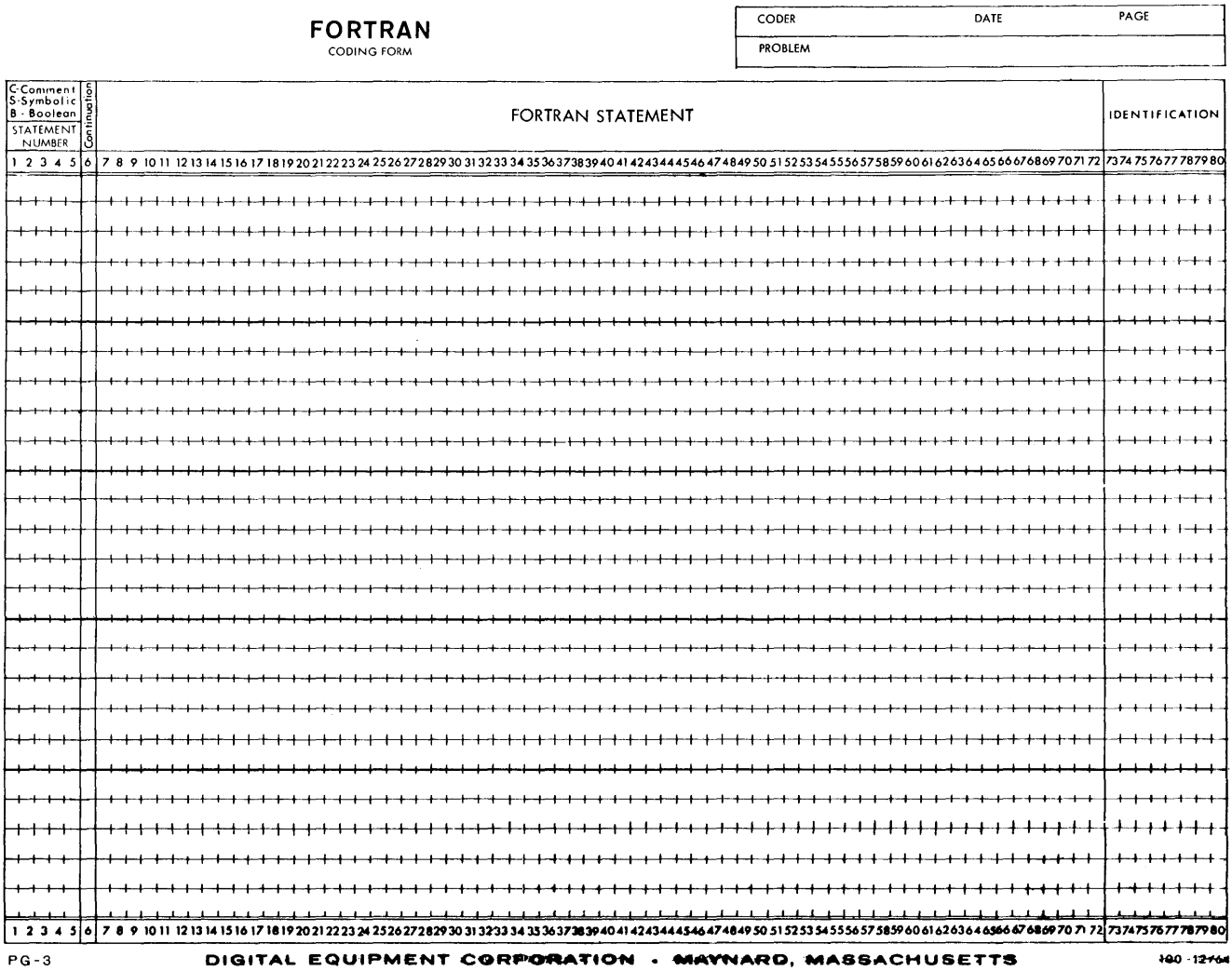


Figure 1-1 FORTRAN Coding Form

Part 1
LANGUAGE

CHAPTER 2
ELEMENTS OF THE FORTRAN LANGUAGE

2.1 CONSTANTS

There are five types of constants allowed in the FORTRAN source program: integer, real, double-precision, logical, and Hollerith.

2.1.1 Integer Constants

An integer constant consists of one to six decimal digits written without a decimal point. A + or - sign preceding the number is optional. The magnitude of the constant must be less than or equal to 131071 ($2^{17}-1$).

Examples:

```
+97
0
-2176
576
```

If the magnitude$\geq 2^{17}-1$, an error message will be output. Negative numbers are represented in 2's complement notation.

2.1.2 Real Constants (Six-decimal-digit accuracy)

A real constant is an integer, fraction, or mixed format number written in the following forms:

a. A constant consisting of one to six significant decimal digits with a decimal point included within the constant. A + or - sign can precede the constant; the + sign is optional.

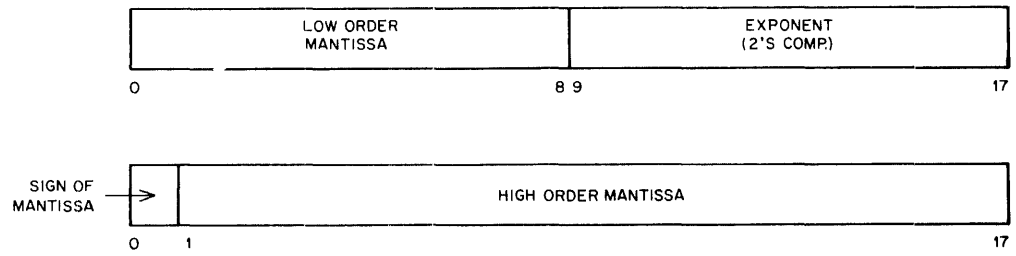
b. A constant followed by the letter E, indicating a decimal exponent, and a one or two digit exponent with magnitude less than 76* indicating the appropriate power of 10. A + or - sign can precede the exponent. The decimal point is not necessary in real constants having a decimal exponent.

Examples:

```
352.
+12.03
-.0054
5.E-3
+5E7
```

*If the adjusted magnitude exceeds 75, an error results. The constant .999999E75 is legal, but 999.999E73 is illegal.

Real constants are stored in two words in the following format:



NOTE

Negative mantissae are indicated with a change of sign.

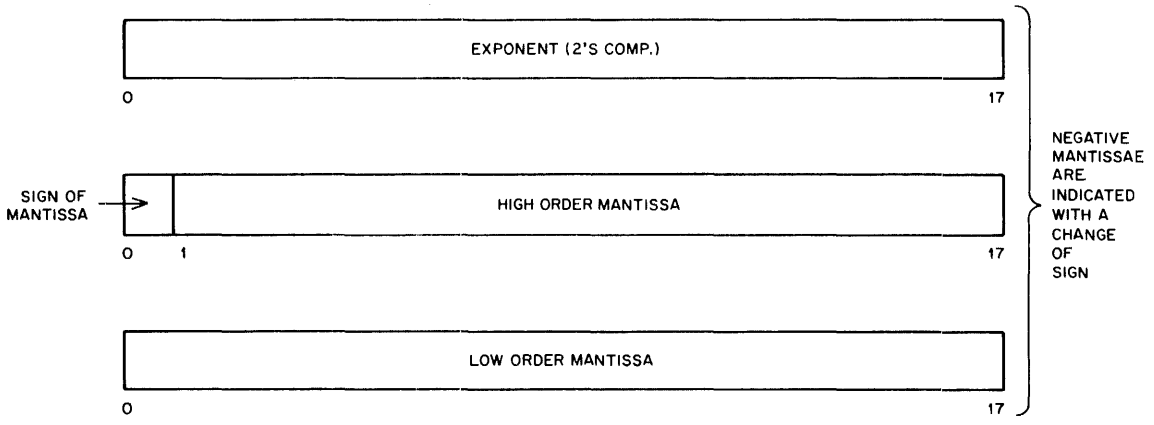
2.1.3 Double-Precision Constants (nine-decimal-digit accuracy)

A double-precision constant is written as a real number with a decimal exponent, followed by the letter D and the one- or two-digit exponent with magnitude not greater than 76. A + or - sign can precede the constant and also the exponent. A decimal point within the constant is optional. A double-precision constant is interpreted the same as a real constant, except that the degree of accuracy is greater.

Examples:

```
-3.0D0
987.6542D15
32.123D+7
```

Double-precision constants are stored in three words:



2.1.4 Logical Constants

The two logical constants are the words TRUE and FALSE, each enclosed by periods, with values as indicated below.

```
.TRUE.   777777
.FALSE.  0
```

2.1.5 Hollerith Constants

A Hollerith constant is written as an unsigned integer constant, the value of which (n) must be ≥ 1 and ≤ 5, followed by the letter H, followed by exactly n characters, which are the Hollerith data. Any FORTRAN character, including blank, is acceptable. The Hollerith constants are used only in CALL and DATA statements and must be associated with real variable names. (For examples, refer to Paragraph 8.11.1.) The Hollerith constants are packed in 7-bit ASCII, five per two words of storage with the rightmost bit always zero.

Examples:

```
1HA
4HA$CD
```

2.2 VARIABLES

A variable is a representation of a numeric quantity, the value of which can change by assignment or computation during the execution of a program. The representation, or name, consists of from one to six alphanumeric characters, the first of which must be alphabetic.

Example:

```
X = Y + 10.    Both X and Y are variables; X by computation, and Y by assignment
               in some previous statement.

TEST
GAMMA
X12345
```

NOTE

If three characters or less are used for each symbol, considerable core space can be saved during compilation.

2.2.1 Variable Types

Variables in FORTRAN can represent one of the following types of quantities: integer, real, double-precision, or logical.

2.2.2 Integer Variables

Variable names beginning with the letters I, J, K, L, M, or N are considered to be integer variables. If the first letter is not one of the above letters, it is an integer variable only if it was named in a previous integer type specification statement.

2.2.3 Real Variables

Variable names beginning with letters other than I, J, K, L, M, or N are considered real variables. If the first character is one of the foregoing letters, it is a real variable only if it was named in a previous real type specification statement.

Example:

```
      REAL ITIN
        .
        .
        .
C     ITIN WILL BE TREATED AS A REAL VARIABLE SUM=ITIN+1
```

2.2.4 Double-Precision and Logical Variables

A type specification statement is the only way to assign a variable value to one of these two types. This is done with either a double precision statement or a logical statement.

2.3 ARRAYS AND SUBSCRIPTS

An array is an ordered set of data identified by a symbolic name. Each individual quantity in this set of data is referred to in terms of its position within the array. This identifier is called a subscript. For example,

```
A (3)
```

represents the third element of a 1-dimension array named A. To generalize further, in an array A with n elements, A (I) represents the Ith element of the array A where I = 1, 2,. . .,n.

FORTRAN allows for 1-, 2-, and 3-dimension arrays; thus, there can be up to three subscripts for the array, each subscript separated from the next by a comma. For example,

```
B (1, 3)
```

represents the value located in the first row and the third column of a 2-dimension array named B. A dimension statement defining the size of the array (i.e., the maximum values each of its subscripts can attain) must precede the array in the source program. (A COMMON statement can also be used for dimensioning.)

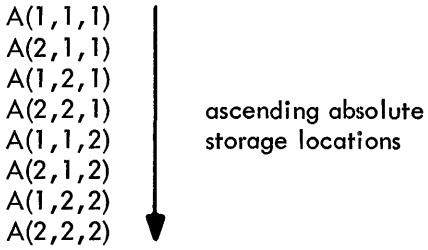
FORTRAN IV does not check constant subscripts to ascertain that they are positive and nonzero. For example, the following statements are not flagged, although they are illegal.

```
N(0)=1
N(-1)=1
```

(These statements are illegal because the array N cannot have a 0 or -1 member.)

2.3.1 Arrangement of Arrays in Storage

Arrays are stored in column order in ascending absolute storage locations. The array is stored with the first of its subscripts varying most rapidly and the last varying least rapidly. For example, a 3-dimension array A, defined in a DIMENSION statement as A (2,2,2), is stored sequentially in this order:



2.3.2 Subscript Expressions

Subscripts can be written in any of the following forms:

```
V
C
V + k
V - k
C * V
C * V + k
C * V - k
```

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

Example:

```
I
I3
IMOST + 3
ILAST - 1
5 * IFIRST
2 * J + 9
4 * M1 - 7
```

2.3.3 Subscripted Variables

A subscripted variable is a variable followed by a pair of parentheses enclosing one to three subscripts separated by commas.

Example:

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

2.4 EXPRESSIONS

An expression is a combination of elements (constants, subscripted or nonsubscripted variables, and functions), each of which is related to another by operators and parentheses. An expression represents one single value that is the result of calculations specified by the values and operators that make up the expression. The FORTRAN language provides two kinds of expressions: arithmetic and logical.

2.4.1 Arithmetic Expressions

An arithmetic expression consists of arithmetic elements joined by the arithmetic operators +, -, *, /, and **, which denote addition, subtraction, multiplication, division, and exponentiation, respectively. An expression may consist of a single element (meaning a constant, a variable, or a function name). An expression enclosed in parentheses is considered a single element. Compound expressions use arithmetic operators to combine single elements.

Examples:

2.71828	(single element: a constant)
Z(N)	(single element: a variable)
TAN(THETA)	(single element: a function name)
(X Y)/2	(single element: because it is enclosed in parentheses)
(X+Y)-(ALPHA*BETA)	(compound expression: arithmetic operators combining single elements)

2.4.1.1 Mode of an Expression - The type of quantities making up an expression determines its mode; e.g., a simple expression consisting of an integer constant or an integer variable is said to be in the integer mode. Similarly, real constants or variables produce a real mode of expression, and double-precision constants or variables produce a double-precision mode. The mode of an arithmetic expression is important because it determines the accuracy of the expression.

In general, variables or constants of one mode cannot be combined with variables or constants of another mode in the same expression. There are, however, exceptions to this rule.

a. The following examples show the modes of the valid arithmetic expressions involving the use of the arithmetic operators (+, -, *, and /). I, R, and D indicate integer, real, and double-precision variables or constants. A plus sign (+) is used to indicate any one of the four operators:

I + I	Integer result
R + R	Real result
R + D D + R D + D	Double-precision result

b. When raising a value to a power, the mode of the power can be different than that of the value being raised. The following examples show the modes of the valid arithmetic expressions using the arithmetic operator (**). As above, I, R, and D indicate integer, real, and double-precision.

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

The subscript of a subscripted variable, which is always an integer quantity, does not affect the mode of the expression.

2.4.1.2 Hierarchy of Operations - The order in which the operations of an arithmetic expression are to be computed is based on a priority rating. The operator with the highest priority takes precedence over other operators in the expression. Parentheses can be used to determine the order of computation. If no parentheses are used, the order is understood to be as follows:

1. Function reference
2. **(Exponentiation)
3. Unary minus evaluation
4. *(multiplication), /(division)
5. +(addition), -(subtraction)

Within the same priority, operations are computed from left to right.

Example:

```
FUNC + A*B/C-D(I,J) + E**F*G-H
```

interpreted as,

```
FUNC + ((A*B)/C) - D(I,J) +(E^F * G) - H
```

2.4.1.3 Construction of Arithmetic Expressions - The following rules apply to constructing arithmetic expressions:

a. Any expression can be enclosed in parentheses.

b. Expressions can be preceded by a + or - sign.

c. Simple expressions may be connected to other simple expressions to form a compound expression, provided that:

(1) No two operators appear together.

(2) No operator is assumed to be present.

d. Only valid mode combinations can be used in an expression (Refer to Section 2.4.1.1).

e. The expression must be constructed so that the priority scheme determines the order of operation desired (Refer to Section 2.4.1.2).

Arithmetic expression examples:

```
3
A(I)
B + 7.3
C*D
A + (B*C) - D**2 + E/F
```

2.4.2 Relational Expressions

A relational expression is formed with the arithmetic expressions separated by a relational operator. The result value is either TRUE or FALSE depending on whether the condition expressed by the relational operator is met or not met. The arithmetic expressions can both be integer mode expressions or a combination of real/double-precision. No other mode combinations are legal. The relational operators must be enclosed by periods. They are:

.LT.	Less than (<)
.LE.	Less than or equal to (≤)
.EQ.	Equal to (=)
.NE.	Not equal to (≠)
.GT.	Greater than (>)
.GE.	Greater than or equal to (≥)

Examples:

```
N .LT.5
DELTA + 7.3 .LE. B/3E7
(KAPPA + 7)/5 .NE.IOTA
1.736D-4.GT.BETA
X.GE. Y*Z**2
```

2.4.3 Logical Expressions

A logical expression consists of logical elements joined by logical operators. The value is either TRUE or FALSE. The logical operator symbols must be enclosed by periods.

The logical operator symbols are:

.NOT. Logical negation. Reverses the state of the logical quantity that follows.

.AND. Logical AND generates a logical result (TRUE or FALSE) determined by two logical elements as follows:

T .AND. T generates T
T .AND. F generates F
F .AND. T generates F
F .AND. F generates F

.OR. Logical OR generates a logical result determined by two logical elements as follows:

T .OR. T generates T
T .OR. F generates T
F .OR. T generates T
F .OR. F generates F

2.4.3.1 Construction of Logical Expression - The following rules apply to constructing logical expressions:

a. A logical expression can consist of a logical constant, a logical variable, a reference to a logical function, a relational expression, or a complex logical expression enclosed in parentheses.

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

c. Any logical expression can be enclosed in parentheses. The logical expression following the logical operator .NOT. must be enclosed in parentheses if it contains more than one quantity.

d. When two logical operators appear in sequence, they must be separated by a comma or parenthesis, unless the second operator is .NOT. In addition, when two decimal points appear together, they must be separated by a comma or parenthesis, unless one belongs to a constant and the other to a relational operator.

2.4.3.2 Hierarchy of Operations - Parentheses can be used as in normal mathematical notation to specify the order of operations. Within the parentheses, or where there are no parentheses, the order in which the operations are performed is as follows:

a. Evaluation of functions
b. **(Exponentiation)
c. Evaluation of unary minus quantities
d. * and/ (multiplication and division)
e. + and - (addition and subtraction)
f. .LT., .LE., .EQ., .NE., .GT., .GE.
g. .NOT.
h. .AND. and .OR.
i. = Replacement operator

Since .AND. and .OR. are of equal priority and are evaluated from left to right, the FORTRAN user must insert his own parentheses when necessary. The following example illustrates equivalent logical expressions according to FORTRAN (L1, L2,... are defined as LOGICAL).

Example:

```
L1.AND.L2.OR..NOT.L3.AND.L4.OR.L5
```

is equivalent to

```
(((L1.AND.L2).OR..NOT.L3) .AND.L4) .OR.L5
```

To present the foregoing expression as if it were meant to be a sum of products (instead of what FORTRAN interprets it to be) requires enclosing the product terms in parentheses.

Example:

```
(L1.AND.L2) .OR.(.NOT.L3.AND.L4) .OR.L5
```

To express the original example as if it were a product of sums requires enclosing the sum terms in parentheses.

Example:

```
L1.AND.(L2.OR..NOT.L3) .AND. (L4.OR.L5)
```

2.5 STATEMENTS

Statements specify the computations required to carry out the processes of the FORTRAN program. There are four categories of statements provided for by the FORTRAN language:

a. Arithmetic statements define a numerical calculation.

b. Control statements determine the sequence of operation in the program.

c. Input/output statements are used to transmit information between the computer and related input/output devices.

d. Specification statements define the properties of variables, functions, and arrays appearing in the source program. They also enable the user to control the allocation of storage.

CHAPTER 3
ARITHMETIC STATEMENTS

An arithmetic statement is a FORTRAN mathematical equation that defines a numerical or logical calculation. It directs the assignment of a calculated quantity to a given variable. An arithmetic statement has the form

V = E

where V is a variable (integer, real, double-precision, or logical, subscripted or nonsubscripted) or any array element name; = means replacement rather than equivalence, as opposed to the conventional mathematical notation; and E is an expression.

In some cases, the mode of the variable is different from that of the expression. In such cases, an automatic conversion takes place. The fules for the assignment of an expression, E, to a variable, V, are given in Table 3-1.

Table 3-1
Assignment Rules

V Mode	E Mode	Assignment Rule
Integer	Integer	Assign
Integer	Real	Fix and assign
Integer	Double-precision	Fix and assign
Real	Integer	Float and assign
Real	Real	Assign
Real	Double-precision	Double-precision evaluate and real assign
Double-precision	Integer	Double-precision float and assign
Double-precision	Real	Double-precision evaluate and assign
Double-precision	Double-precision	Assign
Logical	Logical	Assign

Mode conversions involving logical quantities are illegal unless the mode of both V and E is logical. Examples of an assignment statement:

```
ITEM = ITEM + 1
A(I) = B(I) + ASSIN (C (I) )
V = .FALSE.
X = A.GT.B.AND.C  .LE. G
A = B
```

CHAPTER 4
CONTROL STATEMENTS

The statements of a FORTRAN program are normally executed as written. It is frequently desirable, however, to alter the normal order of execution. Control statements give the FORTRAN user this capability. This section discusses the reasons for control statements and their use.

4.1 UNCONDITIONAL GO TO STATEMENTS

The form of the unconditional GO TO statement is

GO TO n

where n is a statement number. On execution of this statement, control is transferred to the statement identified by the statement number, n, which is the next statement to be executed.

Example:

GO TO 17

4.2 ASSIGN STATEMENT

The general form of an ASSIGN statement is

ASSIGN n TO i

where n is a statement number and i is a nonsubscripted integer variable name that appears in a subsequently executed assigned GO TO statement. The statement number, n, is the statement to which control will be transferred after the execution of the assigned GO TO statement.

Example:

ASSIGN 27 TO ITEST

4.3 ASSIGNED GO TO STATEMENT

Assigned GO TO statements have the form

GO TO i, $(n_1, n_2, \ldots, n_m)$

where i is a nonsubscripted integer variable reference appearing in a previously executed ASSIGN statement, and n_1, n_2, ..., n_m are the statement numbers which the ASSIGN statement may legally assign to i.

Examples:

```
ASSIGN 13 TO KAPPA
GO TO KAPPA, (1, 13, 72, 100, 35)
```

There is no object time checking to ensure that the assignment is a legal statement number.

4.4 COMPUTED GO TO STATEMENT

The format of a computed GO TO statement is

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

where n_1, n_2, ..., n_m are statement numbers and i is an integer variable reference the value of which is greater than or equal to 1 and less than or equal to the number of statement numbers enclosed in parentheses. If the value of i is out of this range, the statement is effectively a CONTINUE statement, however an OTS error statement is also generated.

Example:

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

If the value of ITEM is 2 at the time this GO TO statement is executed, the statement to which control is transferred is the second statement number in the series, i.e., statement 17.

4.5 ARITHMETIC IF STATEMENT

The form of the arithmetic IF statement is

IF (e) n_1, n_2, n_3

where e is an arithmetic expression and n_1, n_2, n_3 are statement numbers. The IF statement evaluates the expression in parentheses and transfers control to one of the referenced statements. If the value of the expression (e) is less than, equal to, or greater than zero, control is transferred to n_1, n_2, or n_3, respectively.

Example:

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

4.6 LOGICAL IF STATEMENT

The general format of a logical IF statement is

IF (e) s

where e is a logical expression and s is any executable statement other than a DO statement or another logical IF statement. The logical expression is evaluated, and different statements are executed depending on whether the expression is TRUE or FALSE. If the logical expression e is TRUE, statement s is executed and control is then

transferred to the statement following the IF statement (unless the statement is a GO TO statement or an arithmetic IF statement, in which cases control is transferred as indicated; or the statement s is a CALL statement, in which case control is transferred to the next statement after return from the subprogram). If the logical expression e is false, statement s is ignored and control is transferred to the statement following the IF statement.

Example:

```
IF (L1) I = I + I
IF (L.LE.k) GO TO 17
IF (LOG.AND. (.NOT.LOG1) ) IF (X) 3,5,5
```

4.7 DO STATEMENT

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

DO n i = m_1, m_2, m_3

or

DO n i = m_1, m_2

where n is a statement number representing the terminal statement or end of the range; i is a nonsubscripted integer variable known as the index; and m_1, m_2, and m_3 are unsigned nonzero integer constants or nonsubscripted integer variables, which represent the initial, final and increment values of the index.

NOTE

The quantities m_1, m_2, and m_3 must be assigned only positive values.

The range of a DO statement is the series of statements to be executed. It consists of all statements immediately following the DO, up to and including statement n. Any number of statements can appear between the DO and statement n. The terminal statement (statement n) cannot be a GO TO (of any form), an arithmetic IF, a RETURN, a STOP, a PAUSE, or a DO statement, or a logical IF statement containing any of these forms.

The index of a DO is the integer variable i which is controlled by the DO statement in such a way that its initial value is set to m_1, and is increased by m_3 each time the range of statements is executed, until a further incrementation would cause the value of m_2 to be exceeded. When i is greater than m_2, control passes to the statement following statement n. Throughout the range of the DO, the index is available for computation either as an ordinary integer variable or as the variable of a subscript. The index cannot be changed by any statement within the DO range.

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

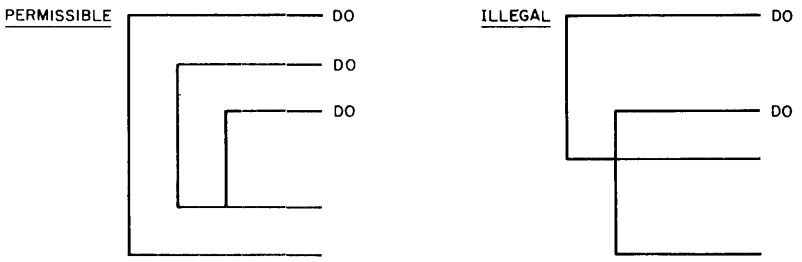
The final value is the value which the index must not exceed. When this value is reached, the DO is completed and control passes to the first executable statement following statement n.

The increment is the amount by which the index is to be increased after each execution of the range. If the increment is omitted, as in the second form of the DO statement above, its value is assumed to be 1.

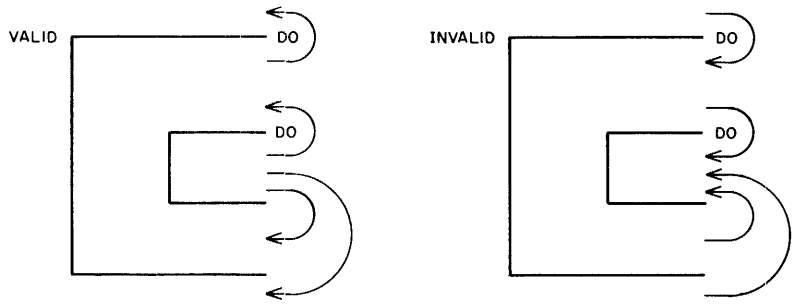
Example:

```
DO 72 I = 1, 10, 2
DO 15 K = 1, 5
DO 23 I = 1, 11, 4
```

Any FORTRAN statement can appear within the range of a DO statement, including another DO statement. When such is the case, the range of the second DO must be contained entirely within the range of the first; i.e., it is not permissible for the ranges of DOs to overlap. A set of DOs satisfying this rule is called a nest of DOs. DOs can be nested to a depth of ten. It is possible for a terminal statement to be the terminal statement for more than one DO statement. The following configuration, where brackets are used to represent the range of the DOs, indicates the permissible and illegal nesting procedures.



Transfer of control from within the range of a DO statement to outside its range is permitted at any time. The reverse is not true, however; i.e., control cannot be transferred from outside the range of a DO statement to inside its range. The following examples show both valid and invalid transfers.



4.8 CONTINUE STATEMENT

The CONTINUE statement causes no action and generates no machine coding. It is a dummy statement used for terminating DO loops when the last statement would otherwise be an illegal terminal statement (viz., GO TO, arithmetic IF, RETURN, STOP, PAUSE, or DO, or a logical IF containing any of these forms). The form consists of the single word

```
CONTINUE
```

Example:

```
      DO7K START,END
            .
            .
            .
      If(X(K))22,13,7
            .
            .
            .
7     CONTINUE
```

4.9 PAUSE STATEMENT

A PAUSE statement is a temporary halt of the program during run time. The PAUSE statement is in one of two forms:

```
PAUSE
  or
PAUSE n
```

where n is an octal integer the value of which is less than 777777_8. The integer n is typed out on the console teletype for the purpose of determining which of several PAUSE statements was encountered. Program execution is resumed, by typing control P (↑P), starting with the first statement following the PAUSE statement.

4.10 STOP STATEMENT

The STOP statement is of one of two forms:

```
STOP
  or
STOP n
```

where n is an octal integer whose value is less than 777777_8. The STOP statement is placed at the logical end of a program and causes the computer to type the integer n on the console teletype, and then to exit back to the Monitor. There must be at least one STOP statement per main program, but none are allowed in subprograms.

4.11 END STATEMENT

The END statement is placed at the physical end of a program or subprogram. The form consists of the single word

END

The END statement is used by the compiler and generates no code. It signals the compiler that the processing of the source program is complete. The compiler assumes the presence of an END statement if it fails to find one.

A control transfer type statement, a STOP statement, or a RETURN statement must immediately precede END. This will be checked by the compiler.

CHAPTER 5
I NPUT/OUTPUT STATEMENTS

The input/output (I/O) statements direct the transfer of data between the computer and I/O devices. The information thus transmitted is defined as a logical record, which can be formatted or unformatted. A logical record, or records, can be written on a device as one or more physical records. This is a function of the size of the logical record and the physical device used.

The definition of the data which comprise a physical record varies with each I/O device (Refer to Table 5-1).

Table 5-1
Physical Record Definitions

Unit/Device	Formatted Physical Record Definition	Unformatted (Binary) Physical Record Definition
Typewriter (input and output)	One line of type is terminated by a carriage return. Maximum of 72 printing characters per line	Undefined
Line printer	One line of printing. Maximum of 120 characters per line	Undefined
Cards (input and output)	One card. Maximum of 80 characters	50 words
Paper tape (input and output)	One line image of 72 printing characters	50 words
Magnetic tape	One line image of 630 characters	252 words 251
Disc/drum/Dectape	One line image of 630 characters	252 words 251

Each I/O device is identified by an integer constant which is associated with a device assignment table in the Monitor. This table may be modified at system generation time, or just before run time. For example, the statement

READ (u,f) list

requests one logical record from the device associated with slot u in the device assignment table.

The statement descriptions in this section use u to identify a specific I/O unit, f as the statement number of the FORMAT statement describing the type of data conversion, and list as a list of arguments to be input or output.

5.1 GENERAL I/O STATEMENTS

These statements cause the transfer of data between the computer and I/O devices.

5.1.1 Input/Output Argument Lists

An I/O statement which calls for the transmission of information includes a list of the quantities to be transmitted. In an input statement, this list consists of the variables to which the incoming data is to be assigned; in an output statement, the list consists of the variables the values of which are to be transmitted to the given I/O device. The order of the list must be that in which the data words exist (input) or are to exist (output) on the I/O device. Any number of items can appear in a single list. The same statement can transmit integer and real quantities. If the data to be transmitted exceeds the items in the list, only the number of quantities equal to the number of items in the list are transmitted. The remaining data is ignored. Conversely, if the items in the list exceed the data to be transmitted, succeeding superfluous records are transmitted until all items specified in the list have been transmitted.

5.1.1.1 Simple Lists - The list uses the form

$$C_1, C_2, \ldots, C_n$$

where each C_i is a variable, a subscripted variable, or an array identifier. Constants are not allowed as list items. The list reads from left to right. When an array identifier appears in the list, the entire array is to be transmitted before the next item in the list.

Examples:

```
Y, Y, Z
A, B (3), C, D (I + 1, 4)
```

5.1.1.2 DO-Implied Lists - Indexing similar to that of the DO statement can be used to control the number of times a group of simple lists is to be repeated. The list elements, thus controlled, as well as the index control itself, are enclosed in parentheses, and the entire enclosure is regarded as a single item of the I/O list.

Example:

```
W, X (3), (Y (I), Z (I,K), I = 1, 10)
```

5.1.2 READ Statement

The READ statement is used to transfer data from any input device to the computer. The general READ statement can be used to read either BCD or binary information. The form of the statement determines what kind of input is performed.

5.1.2.1 Formatted READ - The formatted READ statement has the general form

```
READ (u,f) list
     or
READ (u,f)
```

Execution of this statement causes input from device u to be converted as specified by format statement f, and the resulting values to be assigned to the items specified by list, if any.

Examples:

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

5.1.2.2 Unformatted READ - An unformatted READ statement has the general form

```
READ (u) list
     or
READ (u)
```

Execution of this statement causes input from device u, in binary format, to be assigned to the items specified by list. If no list is given, one record is read, but ignored. If the record contains more information words than the list requires, that part of the record is lost. If more elements are in the list than are in one record, additional records are read until the list is satisfied.

Examples:

```
READ (5) I,J,K
READ (8)
```

5.1.3 WRITE Statement

The WRITE statement is used to transmit information from the computer to any I/O device. The WRITE statement closely parallels the READ statement in both format and operation.

5.1.3.1 Formatted WRITE - The formatted WRITE statement has the general form

WRITE (u,f) list

or

WRITE (u,f)

Execution of this statement causes the list elements, if any, to be converted according to format statement f, and output onto device u.

5.1.3.2 Unformatted WRITE - The unformatted WRITE statement has the general form

WRITE (u) list

Execution of this statement causes output onto device u, in binary format, of all words specified by the list. If the list elements do not fill the record, the remaining part of the record is filled with blanks. If the list elements more than fill one record, successive records are written until all elements of the list are satisfied, and the last record is padded, with blanks if necessary.

Examples:

```
WRITE (1, 10) A, (B (I), (C (I,J), J=2, 10,2), I=1,5)
WRITE (2,7) A,B,C
WRITE (5) W,X(3), Y(I + 1,4),Z
```

5.2 FORMAT STATEMENTS

These statements are used in conjunction with the general I/O statements. They specify the type of conversion which is to be performed between the internal machine language and the external notation. FORMAT statements are not executed; their function is to supply information to the object program.

5.2.1 Specifying FORMAT

The general form of the FORMAT statement is

FORMAT (S_1, S_2, ..., S_n)

where S_1....S_n are data field descriptors. Breaking this format down further, the basic data field descriptor is written in the form

nkw.d

where n is a positive unsigned integer indicating the number of successive fields for which the data conversion is to be performed according to the same specification. This is also known as the repeat count. If n is equal to 1, it can be omitted. The control character, k, indicates which type of conversion is to be performed. This character can be I,E,F,G,D,P,L,A,H, or X. The nonzero integer constant, w, specifies the width of the field. The integer constant, d, indicates the number of digits to the right of the decimal point.

Six of the nine control characters listed above provide for data conversion between internal machine language and external notation.

Internal	Type	External
Integer variable	I	Decimal integer
Real variable	E	Floating-point, scaled
Real variable	F	Floating-point, mixed
Real variable	G	Floating-point, mixed/scaled
Double-precision variable	D	Floating-point, scaled
Logical variable	L	Letter T or F
Alphanumeric	A	Alphanumeric (BCD) characters

The other three control types are special purpose control characters:

Type	Purpose
P	Used to set a scale factor for use with E, F, and D conversions.
X	Provides for skipping characters in input or specifying blank characters in output.
H	Designates Hollerith fields.

Although FORMAT statements generate code, they are not executed and therefore can be placed anywhere in the source program following all specification statements. Because FORMAT statements are referenced by READ or WRITE statements, each FORMAT statement must be given a statement number.

The comma (,) and slash (/) are used as field separators. The comma is used to separate field descriptors; however, it need not follow a field specified by an H or X control character. The slash is used to specify the termination of formatted records. A series of slashes is also a field separator. Multiple slashes are the equivalent of blank records between output records, or records skipped for input records. If a series of n slashes occurs at the beginning or end of the FORMAT specifications, the number of input records skipped or blank lines inserted in output is n. If the series of n slashes occurs in the middle of the FORMAT specifications, this number is n-1. A comma cannot precede/follow a slash. An integer value cannot precede a slash.

For all field descriptors (with the exception of H and X), the field width must be specified. For those descriptors of the w.d type (paragraph 5.2.2.2), the d must be specified even if it is zero. The field width must be large enough to provide for all characters (including decimal point and sign) necessary to constitute the data value as well as blank characters needed to separate it from other data values. The data value within a field is right justified; thus, the most significant characters of the value are lost if the field specified is too small.

Successive items in the I/O list are transmitted according to successive descriptors in the FORMAT statement, until the entire I/O list is satisfied. If the list contains more items than there are descriptors in the FORMAT statement, a new record must be begun. Control is transferred to the preceding left parenthesis, where the same specifications are used again until the list is complete.

Field descriptors (except H and X) are repeated by preceding the descriptor with an unsigned, nonzero integer constant (the repeat count). A group repeat count is used to enable the repetition of a group of field descriptors or field separators enclosed in parentheses. The group count is placed to the left of the parenthesis. Two levels of parentheses (not including those enclosing the FORMAT specification) are permitted.

The field descriptors in the FORMAT must be the same type as the corresponding items in the I/O list; i.e., integer quantities require integer (I) conversion; real quantities require real (E or F) conversion, etc.

Examples:

```
READ (I, 100) I, A
    .
    .
    .
FORMAT (I7,F10.3)
FORMAT (I3, I7/E10.4,E10.4)
FORMAT (2I4, 3(I5,D10.3) )
```

5.2.2 Conversion of Numeric Data

5.2.2.1 I-Type Conversion (Field descriptor: Iw or nIw) - The number of characters specified by w is converted to a decimal integer.

On input, the number specified by w in the input field is converted to a binary integer. A minus sign indicates a negative number. A plus sign, indicating a positive number, is optional. The decimal point is illegal. If there are blanks, they must precede the sign or first digit. All imbedded blanks are interpreted as zero digits.

On output, the converted number is right justified. If the number is smaller than the field w allows, the leftmost spaces are filled with blanks. If an integer is too large the most significant digits are truncated and lost. Negative numbers have a minus sign immediately preceding their most significant digit if sufficient spaces have been reserved. No sign indicates a positive number.

Examples (b indicates blank):

Format Descriptor	Input	Internal	Output
I5	bbbbb	+00000	bbbb0
I3	-b5	-05	b-5
I8	bbb12345	+12345	bbb12345

5.2.2.2 E-Type Conversion (Field descriptor: Ew.d or nEw.d) - The number of characters specified by w is converted to a floating-point number with d spaces reserved for the digits to the right of the decimal point. The w includes field d, spaces for a sign, the decimal point, plus four spaces for the exponent (written E ± XX) in addition to space for optional sign and one digit preceding the decimal point.

The input format of an E-type number consists of an optional sign, followed by a string of digits containing an optional decimal point, followed by an exponent. Input data can be any number of digits in length, although it must fall within the range of 0 to $\pm 10^{\pm 39}$.

E output consists of a minus sign if negative (blank if positive), the digit 0, a decimal point, a string of digits rounded to d significant digits, followed by an exponent of the form E ± XX.

Examples:

Format Descriptor	Input	Internal	Output
E10.4	00.2134E03	213.4	0.2134E+03
E9.4	0.2134E02	21.34	.2134E+02
E10.3	bb-23.0321	-23.0321	-0.230E+02

5.2.2.3 F-Type Conversion (Field descriptor: Fw.d or nFw.d) - The number of characters specified by w is converted to a floating-point mixed number with d spaces reserved for the digits to the right of the decimal point.

Input for F-type conversion is basically the same as that for E-type conversion, described in paragraph 5.2.2.2.

The output consists of a minus sign if the number is negative (blank if positive), the integer portion of the number, a decimal point, and the fractional part of the number rounded to d significant digits.

Examples:

Format Descriptor	Input	Internal	Output
F6.3	b13457	13.457	13.457
F6.3	313457	313.457	13.457
F9.2	-21367.	-21367.	-21367.00
F7.2	-21367.	-21367.	1367.00

5.2.2.4 G-Type Conversion (Field descriptor: Gw.d or nGw.d) - The external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real number.

Input for G-type conversion is basically the same as that for E-type conversion, described in paragraph 5.2.2.2.

The form of the G-type output depends on the magnitude of the internal floating-point number. Comparison is made between the exponent (e) of the internal value and the number of significant digits (d) specified by the format descriptor. If e is greater than d, the E-type conversion is used. If e is less than or equal to d, the F-type conversion is used, but modified by the following formula:

F (w-d).(d-e),4X

The 4X represents four blank spaces that are always appended to the value. If the value to be represented is less than .1 , the E-type conversion is always used.

Examples:

Format Descriptor	Internal	Output
G14.6	$.12345678 \times 10^{-1}$	0.12345678E-01
G14.6	$.12345678 \times 10^{0}$	bb0.123456bbbb
G14.6	$.12345678 \times 10^{4}$	bbb1234.56bbbb
G14.6	$.12345678 \times 10^{8}$	bb0.123456E+08

5.2.2.5 D-Type Conversion (Field descriptor: Dw.d or nDw.d) - The number of characters specified by w is converted to a double-precision floating-point number with the number of digits specified by d to the right of the decimal point.

The input and output are the same as those for E-type conversion except that a D is used in place of the E in the exponent.

Examples:

Format Descriptor	Input	Internal	Output
D12.6	bb+21345D 03	21.345	0.213450D+02
D12.6	b+3456789012	3456.789012	0.345678D+04
D12.6	-12345.6D-02	-123.456	0.123456D+03

5.2.3 P-Scale Factor - Field descriptor: nP or -nP

This scale factor n is an integer constant. The scale factor has effect only on E-,F-,G-, and D-type conversions. Initially, a scale factor of zero is implied. When a P field descriptor has been processed, the scale factor established by n remains in effect for all subsequent E,F, and D descriptors within the same FORMAT statement until another scale factor is encountered.

For E, F, G, and D input conversions (when no exponent exists in the external field), the scale factor is defined as external quantity = internal quantity x 10^n.

The scale factor has no effect if there is an exponent in the external field.

The definition of scale factor for F output conversion is the same as it is for F input. For E and D output, the fractional part is multiplied by 10^n and the exponent is reduced by n.

Examples:

Format Descriptor	Input	Scale Factor	Internal	Output
-3PF6.3	123456	-3	+123456.	23.456
-3PE12.4	123456	-3	+12345.6	bb0.0001E+08
1PD10.4	12.3456	+1	+1.23456	1.2345D+00

5.2.4 Conversion of Alphanumeric Data

5.2.4.1 A-Type Conversion (7-Bit ASCII, Handled As Real Variables) (Field descriptor: Aw or nAw) - The number of alphanumeric characters specified by w is transmitted according to list specifications.

If the field width specified for A input is greater than or equal to five (the number of characters representable in two machine words), the rightmost five characters are stored internally. If w is less than five, 5 - w trailing blanks are added.

For A output, if w is greater than five, w - 5 leading blanks are output followed by five alphanumeric characters. If w is less than or equal to five, the leftmost w characters are output.

5.2.4.2 H-Field Descriptor (7-Bit ASCII) (Field descriptor: $nHa_1 a_2 a_3 \ldots a_n$) - The number of characters specified by n immediately following the H descriptor are transmitted to, or from, the external device. Blanks can be included in the alphanumeric string. The value of n must be greater than 0.

On Hollerith input, n characters read from the external device replace the n characters following the letter H.

In output mode, the n characters following the letter H, including blanks, are output.

Examples:

```
3HABC
17H THIS IS AN ERROR
16H JANUARY 1, 1966
```

(Refer to Paragraph 5.2.8 for an exception to this rule when printing a formatted record.)

5.2.5 Logical Fields, L Conversion - Field descriptor: Lw or nLw

The external format of a logical quantity is T or F. The internal format of a logical quantity is T or F. The internal format is 777777_8 for T or 0 for F.

On L input, the first nonblank character must be T or F. Leading blanks are ignored. Any other nonblank character is illegal.

For L output, if the internal value is 0, F is output. Otherwise, T is output. The F or T is preceded by w - 1 leading blanks.

5.2.6 Blank Fields, X Conversion - Field descriptor: nX

The value of n is an integer number greater than 0. On X input, n characters are read but ignored. On X output, n spaces are output.

5.2.7 FORTRAN Statements Read in at Object Time

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

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

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

The form of the FORMAT specification which is to be inserted into the array is the same as that of the source program FORMAT statement, except that the word FORMAT is omitted and the nH field descriptor cannot be used. The FORMAT specification can be inserted into the array by using a data initialization statement, or by using a READ statement together with an A format.

For example, this facility can be used to specify the format of a deck of cards to be read at object time. The first card of the deck contains the format statement,

```
1        10
(I7,F10.3)
```

Subsequent cards contain data in the general form,

```
 7       17
xx      xxxx
```

```
      DIMENSION AA (10)
   13 FORMAT (10A5)
      READ (3, 13) (AA(I),I=1,10
        .
        .
      READ (3,AA) JJ, BOB
```

With the card reader assigned to .DAT slot (logical device number) 3, the first READ placed the format statement from the first card into the array AA, and the second READ statement causes data from the subsequent cards to be read into JJ and BOB with format specifications I7 and F10.3, respectively.

5.2.8 Output of a Formatted Record

When formatted records are prepared for output, the first character of the record is replaced by a vertical form control character to effect the following vertical spacing on hard copy devices:

Character	Vertical Spacing Before Printing
Blank	One line
0	Two lines
1	Skip to first line of next page
+	No advance
All others	One line

This replacement takes place on all outputs. When the resulting record is input from a device, a different FORMAT statement must be used to compensate for the vertical form control character which will be ignored.

Examples:

Output	FORMAT (1X,F10.3)
Input	FORMAT (F10.3)

5.3 AUXILIARY I/O STATEMENTS

These statements manipulate the I/O file oriented devices. The u is an unsigned integer constant or integer variable specifying the device.

5.3.1 BACKSPACE Statement

The BACKSPACE statement has the general form

BACKSPACE u

Execution of this statement causes the I/O device identified by u, to be positioned so that the record which had been the preceding record becomes the next record. If the unit u is positioned at its initial point, execution of this statement has no effect.

5.3.2 REWIND Statement

The REWIND statement has the general form

REWIND u

Execution of this statement causes the I/O device identified by u to be positioned at its initial point.

5.3.3 ENDFILE Statement

The ENDFILE statement has the general form

ENDFILE u

Execution of this statement causes an endfile record to be written on the I/O device identified by u.

5.3.3.1 Segmented Files - A modification of AUXIO allows the user to write segmented files by using the end-of-file indicator to separate the segments. The procedure for writing segmented files is exemplified on the following page.

```
WRITE (3) (list)  ┐
.                 │
.                 } set of output operations creating segmented file on logical 3
.                 │
WRITE (3) (list)  ┘
ENDFILE 3
WRITE (3,71) (list) ┐
.                   │
.                   } set of output operations creating segmented file on logical 3
.                   │
WRITE (3,76) (list) ┘
ENDFILE 3
WRITE (3) (list)
.
.
.
```

Note that segmented files cannot be input by means of a READ statement in a FORTRAN program because the end of file will be detected as a data error. For an input operation such as this, an assembly language subroutine must be used.

Part 1
LANGUAGE

CHAPTER 6
SPECIFICATION STATEMENTS

Specification statements provide the compiler with information about the nature of the constants and variables used in the program and supply information required to allocate locations in storage for certain variables/arrays. Specification statements are nonexecutable beacuse they do not generate instructions in the object program. All specification statements in a FORTRAN source program must appear:

a. before any executable, code-generating statement and
b. before any FORMAT statements, which are nonexecutable but do generate code.

The order in which statements must appear in a source program is as follows:

1. BLOCK DATA; FUNCTION; SUBROUTINE.
2. INTEGER; REAL; LOGICAL; DOUBLE PRECISION.
3. DIMENSION.
4. COMMON.
5. EQUIVALENCE; EXTERNAL.
6. DATA.
7. Statement functions.
8. Other executable program statements and FORMAT statements.

When a statement with a legal order number is reached, any subsequent statement with a lower order number causes an I error message. (Refer to Appendix D for an explanation of FORTRAN error codes.)

6.1 TYPE STATEMENTS

The general forms of TYPE statements are

```
INTEGER a,b,c
REAL a,b,c
DOUBLE PRECISION a,b,c
LOGICAL a,b,c
```

where a, b, and c are variable names which can be dimension or function names. A TYPE statement informs the compiler that the identifiers listed are variables or functions of a specified type, i.e., INTEGER, REAL, etc. It overrides any implicit typing; i.e., identifiers which begin with the letters I,J,K,L,M, or N are implicitly of the INTEGER mode; those beginning with any other letter are implicitly of the REAL mode. The TYPE statement can be used to supply dimension information. Each variable or function name in a TYPE statement is defined to be of that specific type throughout the program; the type cannot change.

Examples:

```
INTEGER ABC,IJK,XYZ
REAL A (2,4), I,J,K
DOUBLE PRECISION ITEM, GROUP
LOGICAL TRUE, FALSE
```

All function references (statement functions, intrinsic functions, or external functions) that are not implicitly REAL or INTEGER must appear in the appropriate type statement.

Example:

```
DOUBLE PRECISION B,X,DABS,DATAN
                    .
                    .
                    .
B = DATAN (DABS (X) )
```

In this example, if DABS and DATAN had not been declared DOUBLE PRECISION, improper code would have been generated by the compiler, and no error diagnostic would have occurred.

6.1.1 Typing Double-Precision Functions

The compiler does not recognize and implicitly mode-type double-precision functions in the FORTRAN science library. Therefore, all double-precision functions must be explicitly mode-typed as double precision.

The following program is not correct.

```
DOUBLE PRECISION A
.
.
.
A=DLOG(A)
.
.
.
```

The foregoing program should be written as follows.

```
DOUBLE PRECISION A, DLOG
.
.
.
A=DLOG(A)
.
.
.
```

6.2 DIMENSION STATEMENT

The DIMENSION statement is used to declare arrays and to provide the necessary information to allocate storage for them in the object program.

The general form of the DIMENSION statement is

$$\text{DIMENSION } V\ (i_1),\ V_2(i_2),\ \ldots V_n(i_n)$$

where each V is the name of an array and each i is composed of one, two, or three unsigned integer constants separated by commas. The number of constants represents the number of dimensions the array contains; the value of each constant represents the maximum size of each dimension. The dimension information for the variable can be given in a TYPE statement, a COMMON statement, or a DIMENSION statement; however, dimensioning information should be given only once.

Example:

```
DIMENSION ITEM (150), ARRAY (50,50)
```

When arrays are passed to subprograms, they must be redeclared in the subprogram. The mode and number of dimensions must be the same as that declared by the calling program, but the size of each dimension is ignored because the array descriptor block is the calling program used.

6.3 COMMON STATEMENT

The COMMON statement provides a means for a program and its subprograms to share memory storage. The general form of the COMMON statement is:

$$\text{COMMON } /x_1/a_1/x_2/a_2/\ \ldots/x_n/a_n$$

where each x is a variable that is a COMMON block name, or it can be blank. If x_1 is blank, the first two slashes are optional. Each quantity, designated by the letter a, represents a list of variables and arrays separated by commas. The list of elements pertaining to a block name ends with a new block name, a blank COMMON block designation (two slashes), or the end of the statement.

The elements of a COMMON block, which are listed following the COMMON block name (or the blank name), are located sequentially in order of their appearance in the COMMON statement. An entire array is assigned in sequence. Block names can be used more than once in a COMMON statement, or can be used in more than one COMMON statement within the program. The entries so assigned are strung together in the given COMMON block in order of their appearance. Labeled COMMON blocks with the same name appearing in several programs or subprograms executed together must contain the same number of total words. The elements within the blocks, however, need not agree in name, mode, or order. A blank COMMON can be any length.

Examples:

```
COMMON A,B,C/XX/X,Y,Z
COMMON/A/X(3,3), Y(2,5)//Z(5,10,15)
```

The COMMON statement is a means of transferring data between programs. If one program contains the statements

```
COMMON/N/AA,BB,CC
AA=3
BB=4
CC=5
```

and another program which is called later contains the statement

```
COMMON/N/XX,YY,ZZ
```

the latter program finds the values 3, 4, and 5 in its variables XX, YY, and ZZ, respectively, because variables in the same relative positions in COMMON statements share the same locations in memory.

6.4 EQUIVALENCE STATEMENT

The EQUIVALENCE statement permits two or more entities to share the same storage location. The general format of the EQUIVALENCE statement is

$$\text{EQUIVALENCE } (k_1), (k_2), \ldots, (k_n)$$

where each k represents a list of two or more variables or subscripted variables separated by commas. Each element in the list is assigned the same memory storage location.

An EQUIVALENCE statement can lengthen the size of a COMMON block. The size can only be increased by extending the COMMON block beyond the last assignment for that block made directly by a COMMON statement. A variable cannot be made equivalent to an element of an array if it causes the array to extend past the beginning of the COMMON block.

6.4.1 Equivalencing COMMON Variables

The following rules apply to equivalencing COMMON variables:

a. Because COMMON variables occupy unique storage, two of them cannot be equivalenced together.

b. A COMMON variable that appears in an EQUIVALENCE statement cannot be the only member of its COMMON block.

c. A COMMON variable cannot be equivalenced to a variable that already appears in a preceding equivalence group.

d. All variables equivalenced to COMMON variables become COMMON variables themselves as far as the succeeding equivalence groups are concerned.

The following programs fail:

a.
```
COMMON X(10)
EQUIVALENCE (Y,X(5) )
.
.
.
```

b.
```
COMMON X(10),I
EQUIVALENCE (Y,Z),(X(5),Y)
.
.
.
```

The foregoing programs should be rewritten as follows:

a.
```
COMMON X(10),I
EQUIVALENCE (Y,X(5) )
.
.
.
```

b.
```
COMMON X(10),I
EQUIVALENCE (X(5),Y,Z)
.
.
.
```

6.5 EXTERNAL STATEMENT

An EXTERNAL statement is used to pass a subprogram name on to another subprogram. The general form of an EXTERNAL statement is:

```
EXTERNAL y,z,...
```

Example:

```
EXTERNAL ISUM,ISUB
     .
     .
     .
CALL DEBUG (ISUM,A,B)
     .
     .
     .
CALL DEBUG (ISUB,A,B)
     .
     .
     .
END
SUBROUTINE DEBUG (X,Y,Z)
     .
     .
     .
    Y=X (Z)
     .
     .
     .
RETURN
END
```

6.6 DATA STATEMENT

The DATA statement is used to set variables or array elements to initial values at the time the object program is loaded. The general form of the DATA initialization statement is:

$$\text{DATA } k_1/d_1/,k_2/d_2/,\ldots.k_n/d_n/$$

where each k is a list of variables or array elements (with constant subscripts) separated by commas, and each d is a corresponding list of constants with optional signs. The k list cannot contain dummy arguments. There must be a one-to-one correspondence between the name list and the data list, except where the data list consists of a sequence of identical constants. In such a case, the constant need be written only once, preceded by an integer constant indicating the number of repeats and by an asterisk. A Hollerith constant can appear in the data list. A double precision constant must be written explicitly in "d" format (e.g., 1.0D+01 or 1D+01, not 1.D+01).

Variable or array elements appearing in a DATA statement cannot be in blank COMMON. They can be in a labeled COMMON block and initially defined only in a BLOCK DATA subprogram.

Examples:

```
DATA A,B,C/3*2.0/
DATA X(1), X(2), X(3), X(4)/0.0, 0.1, 0.2, 0.3/,Y(1), Y(2),
2Y(3), Y(4)/1.0E2, 1.0E-2, 1.0E4, 1.0E-4/
```

CHAPTER 7
SUBPROGRAMS

A subprogram is a series of instructions which another program uses to perform complex or frequently used operations. Subprograms are stored only once in the computer, regardless of the number of times they are referred to by another program.

There are five categories of subprograms:

a. Statement Functions
b. Intrinsic or Library Functions
c. External Functions
d. External Subroutines
e. Block Data Subprograms

Functions and subroutines differ in the following two respects. Functions normally return a single value to the calling program; subroutines sometimes return more than one value. Functions are called by writing the name of the function and an argument list in a standard arithmetic expression; subroutines are called by using a CALL statement. The last category is a special purpose subprogram used for data initialization purposes.

7.1 STATEMENT FUNCTIONS

A statement function is defined by a single statement similar in form to that of an arithmetic assignment statement. It is defined internally to the program unit by which it is referenced. Statement functions must follow all specification statements and precede any executable statements of the program unit of which they are a part. The general format of a statement function is:

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

where f is a function name; the quantities designated by the letter a are nonsubscripted variables, known as dummy arguments, which are used to evaluate the function; and e is an expression.

The value of a function is a real quantity unless the name of the function begins with I, J, K, L, M, or N; in which case, it is an integer quantity, or the function type can be defined by using the appropriate specification statement.

Since the arguments are dummy variables, their use is restricted to the right side of the statement function, and any use of the same name outside this region of the FORTRAN IV program except in a mode statement will reference a different variable with the same name and mode. The number of dummy variables in any statement function must never exceed 10.

The expression of a statement function, in addition to containing nonsubscripted dummy arguments, can only contain:

a. Non-Hollerith constants
b. Variable references
c. Intrinsic function references
d. References to previously defined statement functions
e. External function references

A statement function is called any time the name of the function appears in any FORTRAN arithmetic expression. The actual arguments must agree in order, number, and type with the corresponding dummy arguments.

Execution of the statement function reference results in the computations indicated by the function definition. The resulting quantity is used in the expression which contains the function reference.

Examples:

```
A(X) = 3.2+SQRT (5.7* X**2)
SUM (A,B,C) = A+B+C
FUNC (A,B) = 2.*A/B**2.+Z
```

7.2 INTRINSIC OR LIBRARY FUNCTIONS

Intrinsic or library functions are predefined subprograms that are a part of the FORTRAN system library. The type of each intrinsic function and its arguments are predefined and cannot be changed.

An intrinsic function is referenced by using its function name with the appropriate arguments in an arithmetic statement. The arguments can be arithmetic expressions, subscripted or simple variables, constants, or other intrinsic functions (refer to Table 7-1).

Examples:

```
X = ABS (A)
I = INT (X)
J = IFIX (R)
```

Table 7-1
Intrinsic Functions

Intrinsic Functions	Definition	No. of Arguments	Symbolic Name	Type of Argument	Type of Function
Absolute value	$\|a\|$	1	ABS IABS DABS	Real Integer Double	Real Integer Double
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT INT IDINT	Real Real Double	Real Integer Integer
Remaindering*	$a_1 \pmod{a_2}$	2	AMOD MOD	Real Integer	Real Integer
Choosing largest value	Max $(a_1, a_2, \ldots)$	2	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double
Choosing smallest value	Min $(a_1, a_2, \ldots)$	2	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of sign	Sign of a_2 times $\|a_1\|$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive difference	a_1 - Min (a_1, a_2)	2	DIM IDIM	Real Integer	Real Integer
Obtain most significant part of double precision argument		1	SNGL	Double	Real
Express single precision argument in double precision form		1	DBLE	Real	Double

*The function MOD or AMOD (a_1, a_2) is defined as $a - [a_1/a_2]\, a_2$, where $[x]$ is the integer the magnitude of which does not exceed the magnitude of x and the sign of which is the same as x.

7.3 EXTERNAL FUNCTIONS

An external function is an independently written program which is executed when its name appears in another program. The basic external functions are given in Table 7-2. The general form of an external function is

```
t  FUNCTION NAME (a1,a2,...,an)
   (FORTRAN statements)
            :
   NAME = final calculation
   RETURN
   END
```

where t is either INTEGER, REAL, DOUBLE PRECISION, LOGICAL, or blank; NAME is the symbolic name of the function to be defined; and the a_1,a_2, etc., are dummy arguments which are nonsubscripted variable names, array names, or other external function names.

The first letter of the function name implicitly determines the type of function. If that letter is I, J, K, L, M, or N, the value of the function is INTEGER. If it is any other letter, the value is REAL. This determination can be overridden by placing the specific type name before the word FUNCTION.

The symbolic name of a function is one to six alphanumeric characters, the first of which must be the alphabetic name and must not appear in any nonexecutable statement of the function subprogram except in the FUNCTION statement where it is named. The function name must also appear at least once as a variable name within the subprogram. During every execution of the subprogram, the variable must be defined before leaving the function subprogram. After the variable is defined, it may be referenced or redefined. The value of this variable at the time any RETURN statement in the subprogram is encountered is called the value of function.

There must be at least one argument in the FUNCTION statement. These must be nonsubscripted variable names. If a dummy argument is an array name, an appropriate DIMENSION statement is necessary. The dummy argument names cannot appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram. The total number of dummy arguments must not exceed 10.

The function subprogram can contain any FORTRAN statements with the exception of a BLOCK DATA, SUBROUTINE, or another FUNCTION statement. It, of course, cannot contain any statement which references itself, either directly or indirectly.

A function subroutine must contain at least one RETURN statement. The general form is

```
RETURN
```

This signifies the logical end of the subprogram and returns control and the computed value to the calling program. At least one RETURN statement must appear between the last executable statement and the END statement.

An END statement, described in Section 4.11, signals the compiler that the physical end of the subprogram has been reached.

An external function is called by using its function name, followed by an actual argument list enclosed in parentheses, in an arithmetic or logical expression. The actual arguments must correspond in number, order, and type to the dummy arguments. An actual argument can be one of the following:

a. A variable name
b. An array element name
c. An array name
d. Any other expression
e. The name of an external function or subroutine

Example:

```
DIMENSION A(100), B(100)
          :
          :
RSLT = SUM (A,B)**2          } Main Program
          :
          :
END

FUNCTION SUM (X,Y)
DIMENSION X (100, Y (100)
SUM = X(1) + Y(1)
DO 10 K = 2, 100
10 SUM = SUM + X(K) + Y(K)   } Function Subprogram
RETURN
END
```

Table 7-2
External Functions

Basic External Function	Definition	No. of Arguments	Symbolic Name	Type of Argument	Type of Function
Exponential	e^a	1 1	EXP DEXP	Real Double	Real Double
Natural logarithm	$\log_e(a)$	1 1	ALOG DLOG	Real Double	Real Double

Table 7-2 (Cont)
External Functions

Basic External Function	Definition	No. of Arguments	Symbolic Name	Type of Argument	Type of Function
Common logarithm	$\log_{10}(a)$	1 1	ALOG10 DLOG10	Real Double	Real Double
Trigonometric sine	sin (a)	1 1	SIN DSIN	Real Double	Real Double
Trigonometric cosine	cos (a)	1 1	COS DCOS	Real Double	Real Double
Hyperbolic tangent	tanh (a)	1	TANH	Real	Real
Square root	$(a)^{1/2}$	1 1	SQRT DSQRT	Real Double	Real Double
Arctangent	arctan (a) arctan (a_1/a_2)	1 1 2 2	ATAN DATAN ATAN2 DATAN2	Real Double Real Double	Real Double Real Double
Remaindering*	a_1 (mod a_2)	2	DMOD	Double	Double

*The function DMOD (a_1,a_2) is defined as $a_1 - [a_1/a_2]\ a_2$, where [x] is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

7.4 SUBROUTINES

A subroutine is defined externally to the program unit which references it. It is similar to an external function in that both contain the same sort of dummy arguments, and both require at least one RETURN statement and an END statement. A subroutine, however, can have multiple outputs. The general form of a subroutine is:

SUBROUTINE NAME $(a_1,a_2, \ldots,a_n)$

or

SUBROUTINE NAME

where NAME is the symbolic name of the subroutine subprogram to be defined; and the a_1, a_2, etc., are dummy arguments (there need not be any) which are nonsubscripted variable names, array names, or the dummy name of another subroutine or external function.

The name of a subroutine consists of one to six alphanumeric characters, the first of which is alphabetic. The symbolic names of the subroutines cannot appear in any statement of the subroutine except the SUBROUTINE statement itself.

The dummy variables represent input and output variables. Any arguments used as output variables must appear on the left side of an arithmetic statement or an input list within the subprogram. If an argument is the name of an array, it must appear in a DIMENSION statement within the subroutine. The dummy argument names cannot appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram. The total number of dummy arguments must not exceed 10.

The subroutine subprogram can contain any FORTRAN subprograms with the exception of FUNCTION, BLOCK DATA, or another SUBROUTINE statement.

The logical termination of a subroutine is a RETURN statement. The physical end of the subroutine is an END statement.

A subroutine is referenced by a CALL statement, which is in the general form

CALL NAME (a_1,a_2, ...,a_n)

or

CALL NAME

where NAME is the symbolic name of the subroutine subprogram being referenced, and the a_1, a_2, etc., are the actual arguments that are being supplied to the subroutine. The actual arguments in the CALL statement must agree in number, order, and type with the corresponding arguments in the SUBROUTINE subprogram. The array sizes must be the same. An actual argument in the CALL statement can be one of the following:

a. A Holerith constant
b. A variable name
c. An array element name
d. An array
e. Any other expression
f. The name of an external function or subroutine

7.5 BLOCK DATA SUBPROGRAM

The BLOCK DATA subprogram is a special subprogram used to enter data into a COMMON block during compilation. A BLOCK DATA statement takes the form

BLOCK DATA

This special subprogram contains only DATA, COMMON, EQUIVALENCE, DIMENSION, and TYPE statements. It cannot contain any executable statements. It can be used to initialize data only in a labeled COMMON block area, not in a blank COMMON block area.

All elements of a given COMMON block must be listed in the COMMON statement, even if they do not all appear in a DATA statement. Data cannot be entered in more than one COMMON block in a single BLOCK DATA subprogram.

An END statement signifies the termination of a BLOCK DATA subprogram.

FORTRAN IV does not initialize more than one named COMMON block in any BLOCK DATA subprogram. If more than one block is stated, only the last one can be initialized with DATA statements. The following program will not work properly.

```
BLOCK DATA
COMMON /N1/I/N2/J
DATA I,J/1,2/
END
```

However, if the subprogram is divided into two BLOCK DATA programs, the problem is eliminated.

```
C   SUBPROGRAM 1
      BLOCK DATA
      COMMON /N1/I
      DATA I/1/
      END

C   SUBPROGRAM 2
      BLOCK DATA
      COMMON /N2/J
      DATA J/2/
      END
```

7.5.1 Example of BLOCK DATA Subprogram

```
BLOCK DATA
DIMENSION X(4), Y(4)
COMMON/NAME/A,B,C,I,J,X,Y
DATA A,B,C/3*2.0/
DATA X(1, X(2), X(3), X(4)/0.0, 0.1, 0.2, 0.3/Y(1), Y(2),
2Y(3), Y(4)/1.0E2, 1.0E-2, 1.0E4, 1.0E-4/
END
```

PART 2

FORTRAN IV OBJECT - TIME SYSTEM

CHAPTER 8
OBJECT-TIME SYSTEM DESCRIPTION

This chapter describes the subprograms included in the FORTRAN IV Object-Time System (OTS). The Object-Time System is a group of subprograms that process compiled FORTRAN IV statements, particularly I/O statements, at execution time. The compiler outputs calls in the form of globals to various subprograms, depending upon the content of the FORTRAN program. When the compiled program is loaded via the Linking Loader, the Loader attempts to satisfy these globals by searching the FORTRAN library. As it finds the required object-time subprograms, it brings them into core and sets up the necessary linkages.

Included in the package are programs for processing formatted and unformatted READ and WRITE statements; BACKSPACE, REWIND and ENDFILE statements; the index of computed GO TO statements; STOP and PAUSE statements; and File commands. The eight error messages output by the object-time system are described in Table 8-1.

The following information is given for each subprogram of OTS:

a. Class
b. Purpose
c. Calling sequence
d. External calls
e. Size
f. Error conditions

Table 8-1
OTS Error Messages

Error Number	Error Description	Library Routines* That May Cause Error
00-04	Not used	
05	Negative REAL Square Root Argument	SQRT
06	Negative DOUBLE PRECISION Square Root Argument	DSQRT
07	Illegal Index in Computed GO TO	.GO
10	Illegal I/O Device Number	.FR, .FW, .FS, .FX,
11	Bad input data - IOPS Mode Incorrect	.FR, .FA, .FE, .FF, .FS
12	Bad FORMAT	.FA, .FE, .FF
13	Negative or Zero REAL Logarithmic Argument	.BC, .BE,ALOG
14	Negative or Zero DOUBLE PRECISION Logarithmic Argument	.BD, .BF, .BG, .BH, DLOG,DLOG10
15	Raise zero to a $\leq$ zero power (error is recoverable and zero result is passed)	.BB, .BC, .BD, .BE, .BF, .BG, .BH

*Only those routines whose calls are generated by the compiler are listed.

8.1 OTS BINARY CODED INPUT/OUTPUT (BCDIO)

Class: Object - Time System

Purpose: The BCD input/output object-time package is designed to process the formatted READ and WRITE statements in FORTRAN IV programs and subprograms. The FORTRAN IV compiler generates all necessary object-time subroutine calls to perform input and output operations on a character-to-character basis under the control of a FORMAT statement. To permit FORMAT statements to be altered or read at execution time, the FORMAT statements are interpreted by BCDIO at execution time rather than at compile-time. This method has two advantages:

a. It provides a greater flexibility to the FORTRAN programmer.

b. It provides the ability to utilize fully the capabilities of BCDIO in machine-language programs.

To demonstrate this capability, a MACRO language program is given below. The program reads eight floating point numbers into memory with F-conversion and writes them on an output device using the E-conversion.

Example:

	Program		Comment
	.TITLE	EXMPL1	
	.GLOBL	.FP,.FR,.FE,.FF,.FW	
	.IODEV	3,4	
ENTRY	JMS*	.FP	/Initialize I/O device status table.
	JMS*	.FR	/Initialize device 3 for input
	.DSA	(3)	/under control of FORMAT statement.
	.DSA	FRMT1	/FRMT1 and read first record into line /buffer.
	LAW	-10	/Set loop counter to 8.
	DAC	COUNT	
	LAC	(ARRAY)	/Set element address to first word
	DAC	ARG1	/in the array.
LOOP1	JMS*	.FE	
ARG1	0		/Convert next line buffer field from /BCD to floating point binary and /store in ARRAY.
	ISZ	ARG1	/Increment ARRAY address by two.
	ISZ	ARG1	
	ISZ	COUNT	/Check the counter and
	JMP	LOOP1	/if not done, repeat loop.
	JMS*	.FF	/Otherwise, terminates reading.
	JMS*	.FW	/Initialize device 4 for output
	.DSA	(4)	/under control of FORMAT
	.DSA	FRMT2	/statement FRMT2.
	LAW	-10	/Set loop counter to 8.
	DAC	COUNT	
	LAC	(ARRAY)	/Set element address to first
	DAC	ARG2	/word in the array.
LOOP2	JMPS*	.FE	/Convert floating-point binary word
ARG2	0		/pair to BCD and store in line-buffer.
	ISZ	ARG2	/Increment ARRAY address by 2.
	ISZ	ARG2	/
	ISZ	COUNT	/Check count.
	JMP	LOOP2	/If not done, go to LOOP 2.
	JMPS*	.FF	/If done, output last line-buffer /and terminates writing.
	HLT		
ARRAY	.BLOCK	20	
FRMT1	.ASCII	'(8F10.5)'	
FRMT2	.ASCII	'(8E12.5)'	
COUNT	0		
	.END		

Calling Sequences:

a. To initialize a device for BCD input (output):

```
JMS*        .FR (.FW)
.DSA        address of slot number.
.DSA        address of first word of FORMAT statement or array.
```

b. To input (output) a data element:

```
JMS*        .FE
.DSA        address of element (first word)
```

c. To input (output) an entire FORTRAN array:

```
JMS*        .FA
.DSA        address of last word in the Array Descriptor Block.
```

d. To terminate the current logical record:

```
JMS*        .FF
```

All BCDIO routines utilize the FIOPS object-time package to perform all I/O data transfers between devices and the FIOPS line buffer. Device level communication is never employed.

e. External Calls:

FIOPS, OTSER, REAL ARITHMETIC

f. Size: 2773 octal locations

g. Error Conditions:

OTS ERROR 10 - Illegal I/O Device Number
OTS ERROR 11 - Bad Input Data (IOPS Mode Incorrect)
OTS ERROR 12 - Illegal FORMAT

8.2 OTS BINARY INPUT/OUTPUT (BINIO)

Class: Object - Time System

Purpose: The Binary Input/Output Object-Time package is designed to process the unformatted READ and WRITE statements in FORTRAN IV programs and subprograms. A FORMAT statement is not required, and data transfer is on a word-to-word basis instead of a character-to-character basis, regardless of data type.

- The size of the physical data record is always the standard line buffer size provided by IOPS.

  Logical data records comprise one or more physical records, the number of which is determined by the length of the I/O list associated with the WRITE statements that generate the logical records.

Each WRITE statement generates one logical record.

Each READ statement reads one logical record, regardless of the length of its I/O list. For this reason, it is the responsibility of the FORTRAN programmer to ensure that I/O lists for WRITE and READ statements are compatible.

Calling Sequences:

a. To initialize a device for binary input (output):

```
JMS*        .FS (.FX)
.DSA        DEVICE
```

b. To input (or output) an integer data element:

```
JMS*        .FI
.DSA        address of the element
```

c. To input (or output) a real data element:

```
JMS*        .FJ
.DSA        address of the element (first word)
```

d. To input (or output) a double-precision data element:

```
JMS*        .FK
.DSA        address of the element (first word)
```

e. To input (or output) a logical data element:

```
JMS*        .FL
.DSA        address of the element
```

f. To input (or output) an entire FORTRAN array:

```
JMS*        .FB
.DSA        address of the last word in the Array Descriptor Block.
```

g. To terminate the current logical record:

```
JMS*        .FG
```

The third word of each physical record contains a record of ID numbers starting with ZERO for the first record. ID is incremented by one as each physical record is generated, until the last record in the logical record has bit 0 set.

A typical WRITE statement can generate the following record for ID:

	000000	
	000001	
LOGICAL	000002	PHYSICAL RECORD
RECORD #1	000003	FOR ID (OCTAL)
	000004	

	000001
	000001
LOGICAL	000002
RECORD #2	000003
	000004

External Calls:

FIOPS, OTSER

Size: 244 octal locations

Error Conditions:

OTS ERROR 10 - Illegal I/O Device Number
OTS ERROR 11 - Illegal Input Data (IOPS Mode Incorrect)

8.3 OTS AUXILIARY INPUT/OUTPUT (AUXIO)

Class: Object - Time System

Purpose: Auxiliary Input/Output consists of the processors for the three auxiliary I/O statements in FORTRAN IV: BACKSPACE, REWIND, and ENDFILE. These statements are normally used to control Magnetic Tape Transports which are being used by unformatted READ and WRITE statements (BINIO).

a. BACKSPACE .FT:

Repositions the tape at a point just prior to the first physical record associated with the current logical record.

Example:

```
WRITE (7) A,B,C
BACKSPACE 7
READ (7) D,E,F
```

The three instructions in the above order cause the data of A, B, and C to be transferred to D, E, and F.

b. REWIND .FU

Causes the specified device to be positioned at its initial (load) point.

c. ENDFILE .FV

Issues an IOPS command to close the current file on the specified device. In the case of Magnetic Tape, this writes a file mark.

Calling Sequences:

a. To backspace one logical record:

```
JMS*        .FT
.DSA        DEVICE
```

b. To position a device at its initial point:

```
JMS*        .FU
.DSA        DEVICE
```

c. To end (close) a file:

```
JMS*        .FV
.DSA        DEVICE
```

External Calls:

FIOPS

Size: 76 octal locations

Error Conditions:

OTS ERROR 10 - Illegal I/O Device Number

8.4 OTS IOPS COMMUNICATION (FIOPS)

Class: Object - Time System

Purpose: FIOPS provides the necessary calls to IOPS required by all FORTRAN input and output statements.

Slot numbers are initialized by the .FC routine (Initialize I/O Device). Initialization of all slots is maintained in the device status table. The first time that .FC is called for any device, the appropriate .INIT call is made to IOPS. The buffer size and input/output flag are stored in the status word table. All subsequent calls to .FC for the same device number suppress another .INIT unless the input/output flag has changed, or this device number has been closed with a file command.

One line buffer is used by all FORTRAN programs. Data transfers between the line buffer and I/O devices are performed by the .FQ routine, which performs a .READ if the input/output (.FH) is ZERO or a .WRITE if .FH is ONE. A .WAIT is always performed.

The .FP routine is called at the beginning of all FORTRAN main programs. This routine sets all words in the device status table to zero, indicating that all devices are uninitialized.

Calling Sequences:

a. To ilitialize the I/O device status table:

```
JMS*        .FP
```

b. To specify input:

```
DZM*        .FH
```

c. To specify output:

```
LAC         (1)
DAC*        .FH
```

d. To select device:

```
LAC         DEVICE (address of slot number)
JMS*        .FC
```

e. To input or output the line buffer:

```
LAC         address of .DAT slot number (bits 9-17) and IOPS mode (bits 6-8)
JMS*        .FQ
```

NOTES

1. DEVICE is a cell containing the slot number.
2. The line buffer is in location .FN to .FN+377_8.
3. The standard line buffer size (for the device currently selected) is in location .FM.
4. On output, IOPS header words (.FN and .FN + 1) must be prepared by the user.

External Calls:

OTSER

Size: 540 octal locations

Error Conditions:

OTS ERROR 10 - Illegal I/O Device Number

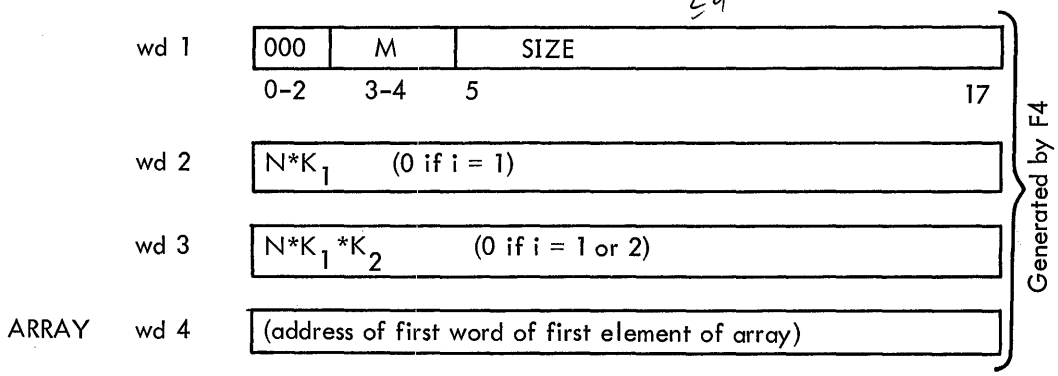
8.5 OTS CALCULATE ARRAY ELEMENT ADDRESS (.SS)

Class: Object-Time System

Purpose: To calculate the address of the first word of an array element.

Consider the array defined by DIMENSION ARRAY ($K_1,\ldots,K_i$); where i = 1, 2, or 3 is the number of dimensions in the array. The array descriptor block for this array is constructed of the following four computer words.

$\leq 4096_{10}$



where SIZE = $N*K_1\ldots*K_i$ and M (the mode number) and N (the number of words per element) are specified as follows:

ARRAY TYPE	M	N
INTEGER	00_2	1
REAL	01_2	2
DOUBLE PRECISION	10_2	3
LOGICAL	11_2	1

Consider the address A of the first word of the array element ARRAY ($k_1,\ldots,k_i$), where $1 \leq k_j \leq k_j$ for j = 1 to i. This address is given by the following formula.

$$A = WD4 + (k_1-1) * N + (k_2-1) * WD2 + (k_3-1) * WD3$$

where WD2, WD3, and WD4 stand for the contents of words 2, 3, and 4 of the array descriptor block for ARRAY as shown above.

Calling Sequence:

```
.GLOBL      .SS
JMS*        .SS
.DSA        ARRAY       /ADDRESS OF WD4
LAC  (k1                /SUBSCRIPT i
     :
     :
LAC  (k1                /SUBSCRIPT i
DAC ALOC                /RETURN WITH A IN AC
```

External Calls:

INTEGER and REAL ARITHMETIC

Size: 57 octal locations

Error Conditions:

None.

8.6 OTS COMPUTED GO TO (GO TO (.GO))

Class: Object-Time System

Purpose: To compute the index of a computed GO TO.

Calling Sequence:

```
LAC         V           /Index value in A-register
JMS*        .GO
-N                      /Number of statement addresses
STMT ADDR (1)
STMT ADDR (2)
     :
     :
STMT ADDR (N)
```

External Calls:

OTSER

Size: 26 octal locations

Error Conditions:

OTS ERROR 7 if the index is illegal (equal to or less than zero).

8.7 OTS STOP (STOP (.ST))

Class: Object-Time System

Purpose: To process the STOP statement and return control to the monitor.

Calling Sequence:

```
LAC     (Octal number to be printed)
JMS*    .ST
```

External Calls:

SPMSG (.SP)

Size: 13 octal locations

Error Conditions:

None.

8.8 OTS PAUSE (PAUSE (.PA))

Class: Object-Time System

Purpose: To process the PAUSE statement. After receiving a ↑P (Control P) from the keyboard, control is returned to the program.

Calling Sequence:

```
LAC     (Octal number to be printed)
JMS*    .PA
```

External Calls:

SPMSG (.SP)

Size: 14 octal locations

Error Conditions:

None.

8.9 OTS OCTAL PRINT (SPMSG (.SP))

Class: Object-Time System

Purpose: To print the octal number coded with STOP and PAUSE. If no number is given, zero (0) is assumed.

Calling Sequence:

```
LAC     (Octal integer to be printed)
JMS*    .SP
.DSA    (Control return) /pause only
LAC     1st Character
LAC     2nd Character
LAC     3rd Character
LAC     4th Character
LAC     5th Character
LAC     6th Character
```

External Calls:

None.

Size: 74 octal locations

Error Conditions:

None.

8.10 OTS ERRORS (OTSER (.ER))

Class: Object-Time System

Purpose:

a. To announce an error on the teletype:

```
JMS*    .ER
.DSA    Error number
```

b. If bit 0 of the error number is a 1, the error is recoverable and program control is returned to the calling program at the first location following the error number.

c. If bit 0 of the error number is a 0, the error is unrecoverable and program control is transferred to the monitor by means of the .EXIT function.

d. In the case of recoverable errors, the AC and link are restored to their original contents prior to returning control to the caller.

e. If the error is a bad format statement (unrecoverable), the current 5/7 ASCII word pair of the erroneous format statement is printed in addition to the error number.

Calling Sequence:

```
                JMS*        .ER
                .DSA        Error number, octal
ERROR #12       LAC         Note word 1
 only           LAC         Note word 2
```

Words 1 and 2 are the current 5 characters (in 5/7 ASCII of the bad format statement (ERROR #12)).

External Calls:

None.

Size: 117 octal locations

Error Conditions:

None.

8.11 ADDITIONS TO THE FORTRAN IV SUBROUTINE LIBRARY

8.11.1 File Commands (FILE)

Class: External Subroutine

Purpose: To provide the device-independent .IOPS commands SEEK, ENTER, CLOSE, FSTAT, RENAM, and DLETE. These commands are used to allow the FORTRAN IV Object-Time System to communicate with .IOPS file-oriented devices.

a. SEEK finds and opens a named input file.

b. ENTER initiates and opens a named output file.

c. CLOSE terminates an input or an output file and must be used if SEEK or ENTER has been used.

d. FSTAT checks for the presence of a named file.

e. RENAM checks for the presence of a file and renames it if found.

f. DLETE checks for the presence of a file and deletes it if found.

NOTE

BACKSPACE, REWIND, and ENDFILE commands should never be used with a device that is operating in the file-oriented mode using the above subroutines.

Calling Sequence:

a. To seek a named file:

```
CALL SEEK (N,A)
```

where N = device number
A = array name containing the 9-character 5/7 ASCII file name and extension.

The file array has the following format for the named file FILNAM EXT:

```
DIMENSION FILEN (2)
DATA FILEN(1), FILEN(2)/5HFILNA,4HMEXT/
```

To use this named file for input on .DAT slot 1:

```
CALL SEEK (1,FILEN)
```

b. To enter a named file:

```
CALL ENTER (N,A)
```

where N and A are the same as for SEEK.

c. To close a named file:

```
CALL CLOSE (N)
```

where N is the same as for SEEK.

d. To check for the presence of a named file:

```
CALL FSTAT (N, A, I)
```

where N and A are the same as for SEEK and I = 0
(.FALSE.) if file not found and I = -1 (.TRUE.) if file found and action complete.

e. To rename a file A and call it B:

```
CALL RENAM (N, A, B, I)
```

where N, A(B is the same as A), and I are the same as for FSTAT.

f. To delete a named file:

```
CALL DLETE (N, A, I)
```

where N, A, and I are the same as for FSTAT.

NOTE

In Hollerith constants when the filename or extension does not contain the maximum number of characters, the filler character is a space.

External Calls:

FIOPS, .DA, .SS, .SEEK, .ENTER, .CLOSE, .FSTAT, .RENAM, .DLETE

Size: 333 octal locations

Error Conditions:

.OTS Error 10 if I/O device number is illegal

.IOPS Error 13 if file not found on SEEK

.IOPS Error 14 if directory full on ENTER

8.11.2 Clock Handling (TIME)

Class: External Subroutine

Purpose: To provide the ability to record elapsed time in minutes and seconds on a 60-cycle machine.

Calling Sequence:

```
CALL TIME (IMIN, ISEC, IOFF)
```

This call causes the clock to be started and the elapsed time to be recorded as minutes and seconds in IMIN and ISEC. To stop the clock, set IOFF to non-zero.

Only one call to TIME or TIME 10 can be active at any point in the user program.

Example:

```
    CALL TIME (IM, IS, IOF)
A       :
        :
    IOF = 1
    WRITE (4,100) IM, IS
```

This sequence causes the time taken to execute the code at A to be output.

External Calls:

.DA, .TIMER

Size: 53 octal locations

Error Conditions:

None.

8.11.3 Clock Handling (TIME10)

Class: External Subroutine

Purpose: To provide the ability to record elapsed time in minutes, seconds, and tenths of seconds on a 60-cycle machine.

Calling Sequence:

```
CALL TIME10 (IMIN, ISEC, ISEC10, IOFF)
```

This call causes the clock to be started and the elapsed time to be recorded as minutes, seconds, and tenths of seconds in IMIN, ISEC, and ISEC10, respectively. To stop the clock, set IOFF to non-zero. Only one call at TIME10 or TIME can be active at any point in the user program.

Example: See TIME.

External Calls:

.DA, .TIMER

Size: 66 octal locations

Error Conditions:

None.

8.11.4 Adjustable Dimensioning (ADJ1)

Class: External Subroutine

Purpose: To provide dimension adjustment on a 1-dimension array.

Calling Sequence:

```
DIMENSION B(1)
      :
CALL ADJ1 (B,A)
      :
```

where B is the array with storage beginning at A. A must be an array element (such as C(200)) with sufficient storage beyond A to allow for all the entries of array B. The dimensions or type of array A do not have to agree with array B.

B cannot be a dummy argument in a subroutine, but A can be a dummy argument.

Example:

```
DIMENSION A(300), B(1), C(1)
      :
CALL ADJ1 (B,A(101))
CALL ADJ1 (C,A(201))
      :
```

After the calls to ADJ1, the arrays B and C can be referenced as if they had been dimensioned as (100) each. It is not necessary to make further calls to ADJ1.

External Calls:

.DA

Size: 17 octal locations

Error Conditions:

None.

8.11.5 Adjustable Dimensioning (ADJ2)

Class: External Subroutine

Purpose: To provide dimension adjustment for a 2-dimension array.

Calling Sequence:

```
DIMENSION B(1,1)
      :
CALL ADJ2 (B,A, NR)
```

where NR is the number of rows to appear in array B.

Refer to ADJ1 for comments on B and A.

Example:

```
DIMENSION A(300), B(1,1), C(1,1)
      :
CALL ADJ2 (B,A (1), 10)
CALL ADJ2 (C,A (101), 20)
      :
```

After the calls to ADJ2, the arrays B and C can be referenced as if they had been dimensioned (10,10) and (20,10), respectively. It is not necessary to make further calls to ADJ2.

External Calls:

DA, .AD

Size: 36 octal locations

Error Conditions:

None.

8.11.6 Adjustable Dimensioning (ADJ3)

Class: External Subroutine

Purpose: To provide dimension adjustment for the 3-dimension array.

Calling Sequence:

```
DIMENSION B (1,1,1)
      :
CALL ADJ3 (B,A,NR,NC)
```

where NR and NC are the number of rows and columns, respectively, to appear in array B.
Refer to ADJ1 for comments on B and A.

Example:

See ADJ1 and ADJ2

External Calls:

.DA, .AD

Size: 41 octal locations

Error Conditions:

None.

Part 2
OBJECT-TIME SYSTEM

PART III

THE SCIENCE LIBRARY

CHAPTER 9
SCIENCE LIBRARY DESCRIPTION

This chapter describes mathematical routines in the Science Library. Most of the descriptive material is listed in Table 9-1; in cases where detailed calculations or algorithms are involved, a reference (Δ) is made in column 1 to detailed descriptions following the table. Information given in Table 9-1 for each routine includes the routine name; mnemonic; calling sequence; function; mode; errors; accuracy and timing (where available); storage requirements; and external calls. Routines are categorized as Intrinsic Functions, External Functions, Sub-Functions, or part of the Arithmetic Package and are listed in the table accordingly.

9.1 INTRINSIC FUNCTIONS

Intrinsic Functions are predefined subprograms that are part of the FORTRAN library. The type of each Intrinsic Function and its arguments are predefined and cannot be changed. Intrinsic Functions are referenced in a FORTRAN program by writing the function name and the desired arguments in an appropriate FORTRAN statement. Example:

```
X = ABS (A)
```

9.2 EXTERNAL FUNCTIONS

External Functions are independently written programs that are executed each time their name appears in a FORTRAN program. Each External Function accepts one or more numerical arguments and computes a single result. SIN, COS, and ALOG are examples of external functions. All basic External Functions supplied with the FORTRAN system are described in Table 9-1.

9.3 SUB-FUNCTIONS

Sub-Functions are called by Intrinsic and External Functions, but are not directly accessible to the user via FORTRAN. For example, the Sub-Function .EB is called by the External Function SIN, and performs the actual computation of the sine.

9.4 THE ARITHMETIC PACKAGE

The Arithmetic Package contains all arithmetic routines required for integer, real, and double-precision arithmetic. Both EAE and non-EAE versions are available, depending on the hardware.

9.5 ACCUMULATORS

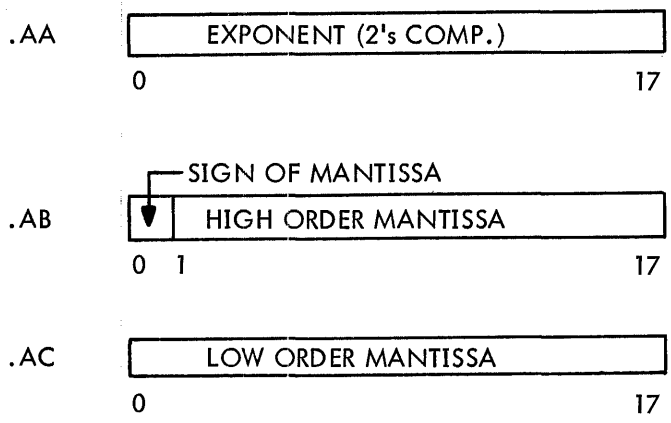
There are three accumulators referred to in the CALLING SEQUENCE column of the table. These include the A-register, the floating accumulator, and the held accumulator.

9.5.1 A-Register

The A-register is the standard hardware accumulator and is used in some of the computations that involve integer values.

9.5.2 Floating Accumulator

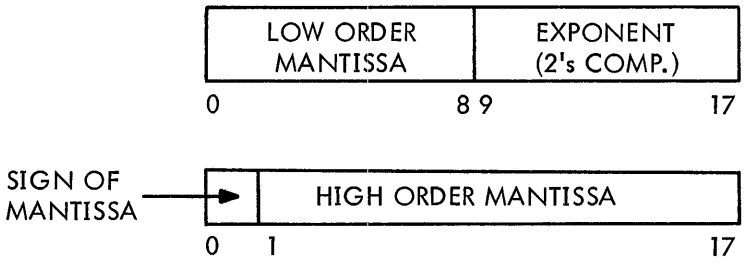
The floating accumulator is a software accumulator that is included in the REAL ARITHMETIC package. It is a 3-word accumulator, .AA being the label of the first word, .AB the second, and .AC the third. Numbers are stored in this accumulator in the following format:



NOTE

Negative mantissae are indicated with a change of sign.

Used by both the single and double-precision routines, this format is also that of double-precision numbers. Single-precision numbers have a different format and must be converted before and after use in the floating accumulator. The format of single-precision numbers is show in the following illustration.



9.5.3 Held Accumulator

The held accumulator has the same format as the floating accumulator and is used as temporary storage by some routines. The labels of the three words are CE01, CE02, and CE03.

9.6 CALLING SEQUENCES

The MACRO calling sequences (given in the third column of Table 9-1) assume, in some cases where there are two arguments, that the appropriate accumulator has been loaded with the first argument. If the first argument is an integer value, it can be loaded into the A-register with a LAC instruction. If the first argument is a real or double-precision value, the routines .AG and .AO, respectively, are used to load the floating accumulator. The DAC instruction can be used to store the result of routines that return with an integer value in the A-register. The routines .AH and .AP are used to store the result of routines that return with real or double-precision values in the floating accumulator.

In calling sequences that use the .DSA pseudo operation to define the symbolic address of arguments, 400000 must be added to the address field if indirect addressing is involved.

FORTRAN library routines that are used in MACRO programs must be declared with a .GLOBL pseudo operation in the MACRO program. The number and type of arguments in the calling program and the FORTRAN library routine must agree.

The following example shows a section of a MACRO main program that uses the FORTRAN External Function SIN.

```
        .TITLE
        .GLOBL    SIN, .AH
            .
            .
            .
        JMS*      SIN
        JMP       .+2           /JUMP AROUND ARGUMENT
        .DSA      A             /+400000 IF INDIRECT
        JMS*      .AH           /STORE IN REAL FORMAT AT X
        .DSA      X
            .
            .
            .
X       .DSA      0
        .DSA      0
```

Table 9-1
The Science Library

Routine Name	Mnemonic	Calling Sequence	Function	Mode	Errors	Accur. Bits	Storage (Octal)	External Calls
INTRINSIC FUNCTIONS								
Exponentiation:								
Integer Base, Integer Exponent	.BB	LAC ARG1 (base) JMS* .BB LAC ARG2 (exp)	I**K	I=I**I	None	N.A.	45	INTEGER
Real Base, Integer Exponent	.BC	JMS* SUBR .DSA ADDR of ARG2 (exp)	A**K	R=R**I	#13, if base ≤0	26	44	.EE,.EF,REAL
DP Base, Integer Exponent	.BD		A**K	D=D**I	#14, if base ≤0	32	46	.DE,.DF,DOUBLE
Real Base, Real Exponent	.BE		A**B	R=R**R	#13, if base ≤0	26	20	.EE,.EF,REAL
Real Base, DP Exponent	.BF		A**B	D=R**D	#13, if base ≤0	26	21	.EE,.DF,DOUBLE
DP Base, Real Exponent	.BG		A**B	D=D**R	#14, if base ≤0	32	22	.DE,.DF,DOUBLE
DP Base, DP Exponent	.BH		A**B	D=D**D	#14, if base ≤0	32	21	.DE,.DF,DOUBLE
Absolute Value:								
Real Absolute Value	ABS		|A|	R=ABS(R)	None	N.A.	16	.DA,REAL
Integer Absolute Value	IABS		|I|	I=IABS(I)	None	N.A.	14	.DA
DP Absolute Value	DABS	JMS* SUBR JMP .+2 .DSA ADDR of ARG	|A|	D=DABS(D)	None	N.A.	16	.DA,DOUBLE
Truncation:								
Real to Real Truncation	AINT		Sign of A times largest integer ≤A	R=AINT(R)	None	N.A.	15	.DA,REAL
Real to Integer Truncation	INT			I=INT(R)	None	N.A.	13	.DA,REAL
DP to Integer Truncation	IDINT			I=IDINT(D)	None	N.A.	13	.DA,REAL,DOUBLE
Remaindering:								
Real Remaindering	AMOD		Note 1	R=AMOD(R,R)	None	N.A.	27	.DA,REAL
Integer Remaindering	MOD		Note 1	I=MOD(I,I)	None	N.A.	24	.DA,INTEGER
DP Remaindering	DMOD		Note 1	D=DMOD(D,D)	None	N.A.	30	.DA,DOUBLE
Transfer of Sign:		JMS* SUBR JMP .+3 .DSA ADDR of ARG1 .DSA ADDR of ARG2						
Real Transfer of Sign	SIGN		Sign of A1 → Sign of A2	R=SIGN(R,R)	None	N.A.	26	.DA,REAL
Integer Transfer of Sign	ISIGN			I=SIGN(I,I)	None	N.A.	20	.DA
DP Transfer of Sign	DSIGN			D=SIGN(D,D)	None	N.A.	26	.DA,DOUBLE
Positive Difference:								
Real Positive Difference	DIM		A1-MIN(A1,A2)	R=DIM(R,R)	None	N.A.	22	.DA,REAL
Integer Positive Difference	IDIM		I1-MIN(I1,I2)	I=IDIM(I,I)	None	N.A.	15	.DA,INTEGER
Conversion:								
Integer to Real Conversion	FLOAT		A←I	R=FLOAT(I)	None	N.A.	11	.DA,REAL
Real to Integer Conversion	IFIX	JMS* SUBR JMP .+2 .DSA ADDR of ARG	I←A	I=IFIX(R)	None	N.A.	13	.DA,REAL
DP to Real Conversion	SNGL		A←B	R=SNGL(D)	None	N.A.	27	.DA,DOUBLE
Real to DP Conversion	DBLE		A←B	D=DBLE(R)	None	N.A.	11	.DA,REAL

NOTES: 1. Remaindering is defined as Ai - [Ai/A2]A2, where [Ai/A2] is the integer whose magnitude does not exceed the magnitude of Ai/A2 and whose sign is the same as Ai/A2.

Table 9-1 (Cont)
The Science Library

Routine Name	Mnemonic	Calling Sequence	Function	Mode	Errors	Accur. Bits	Storage (Octal)	External Calls
INTRINSIC FUNCTIONS (Cont)								
Maximum/Minimum Value:								
Integer Maximum/Minimum	IMNMX	JMS* MAX0,MIN0, AMAX0, or AMIN0 JMP .+n+1					107	INTEGER, REAL
Integer to Integer Max.	MAX0	.DSA ADDR of ARG1 .DSA ADDR of ARG2 . . . . .DSA ADDR of ARGn	Max. Value	I=MAX0(I1,...,In)	None	N.A.		
Integer to Interger Min.	MIN0		Min. Value	I=MIN0(I1,...,In)	None	N.A.		
Integer to Real Max.	AMAX0		Max. Value	R=AMAX0(I1,...,In)	None	N.A.		
Integer to Real Min.	AMIN0		Min. Value	R=AMIN0(I1,...,In)	None	N.A.		
Real Maximum/Minimum	RMNMX	JMS* AMAX1,AMIN1, MAX1, or MIN2 JMP .+n+1					120	INTEGER,REAL
Real to Real Max.	AMAX1	.DSA ADDR of ARG1 .DSA ADDR of ARG2 . . . . .DSA ADDR of ARGn	Max. Value	R=AMAX1(R1,...,Rn)	None	N.A.		
Real to Real Min.	AMIN1		Min. Value	R=AMIN1(R1,...,Rn)	None	N.A.		
Real to Integer Max.	MAX1		Max. Value	I=MAX1(R1,...,Rn)	None	N.A.		
Real to Integer Min.	MIN1		Min. Value	I=MIN1(R1,...,Rn)	None	N.A.		
DP Maximum/Minimum	DMNMX	JMS* DMAX1 or DMIN1 JMP .+n+1					106	DOUBLE
DP Maximum	DMAX1	.DSA ADDR of ARG1 . . . . .DSA ADDR of ARGn	Max. Value	D=DMAX1(D1,...,Dn)	None	N.A.		
DP Minimum	DMIN1		Min. Value	D=DMIN1(D1,...,Dn)	None	N.A.		
EXTERNAL FUNCTIONS								
Square Root:								
Real Square Root (1)	SQRT		$X^{1/2}$	R=SQRT(R)	#5,ARG < 0	26	66	.DA,.ER,REAL
DP Square Root (1)	DSQRT		$X^{1/2}$	D=DSQRT(D)	#6,ARG < 0	34	70	.DA,.ER,DOUBLE
Exponential:								
Real Exponential (2)	EXP		e^X	R=EXP(R)	#13,ARG $\leq$ 0	26	13	.DA,.EF,.ER,REAL
DP Exponential (2)	DEXP	JMS* SUBR	e^X	D=DEXP(D)	#14,ARG $\leq$ 0	34	13	.DA,.DF,.ER,DOUBLE
Natural Logarithm:		JMP .+2						
Real Natural Logarithm (3)	ALOG	.DSA ADDR of ARG	$\text{Log}_e X$	R=ALOG(R)	#13,ARG < 0	26	20	.DA,.EE,.ER,REAL
DP Natural Logarithm (3)	DLOG		$\text{Log}_e X$	D=DLOG(D)	#14,ARG < 0	32	21	.DA,.DE,.ER,DOUBLE
Common Logarithm:								
Real Common Logarithm (3)	ALOG10		$\text{Log}_{10} X$	R=ALOG10(R)	#13,ARG < 0	26	20	.DA,.EE,.ER,REAL
DP Common Logarithm (3)	DLOG10		$\text{Log}_{10} X$	D=DLOG10(D)	#14,ARG < 0	32	21	.DA,.DE,.ER,DOUBLE
Sine:								
Real Sine (4)	SIN		Sin (X)	R=SIN(R)	None	26	13	.DA,.EB,REAL
DP Sine (4)	DSIN		Sin (X)	D=SIN(D)	None	34	13	.DA,.DB,DOUBLE
Cosine:								
Real Cosine (4)	COS		Cos (X)	R=COS(R)	None	26	20	.DA,.EB,REAL
DP Cosine (4)	DCOS		Cos (X)	D=COS(D)	None	34	21	.DA,.DB,DOUBLE

Table 9-1 (Cont)
The Science Library

Routine Name	Mnemonic	Calling Sequence	Function	Mode	Errors	Accur. Bits	Storage (Octal)	External Calls
EXTERNAL FUNCTIONS (Cont)								
Arctangent:								
Real Arctangent (5)	ATAN	JMS* ATAN or DATAN JMP .+2 .DSA ADDR or ARG	$\tan^{-1}$ (a)	R=ATAN(2)	None	26	13	.DA,.ED,REAL
DP Arctangent (5)	DATAN		$\tan^{-1}$ (a)	D=DATAN(D)	None	34	13	.DA,.DD,DOUBLE
Real Arctangent (x/y) (5)	ATAN2	JMS* ATAN2 or DATAN2 JMP .+3 .DSA ADDR of ARG1 .DSA ADDR of ARG2	$\tan^{-1}$ (x/y)	R=ATAN2(R,R)	None	26	44	.DA,.ED,REAL
DP Arctangent (x/y) (5)	DATAN2		$\tan^{-1}$ (x/y)	D=DATAN2(D,D)	None	34	46	.DA,.DD,DOUBLE
Hyperbolic Tangent (6)	TANH	JMS* TANH JMP .+2 .DSA ADDR of ARG	tanh (a)	R=TANH(R)	None	26	47	.DA,.EF,REAL
SUB-FUNCTIONS								
Sine Computation:								
Real Sine (4)	.EB		Sin (a)	R=.EB(R)	None	19	102	.EC,REAL
DP Sine (4)	.DB		Sin (a)	D=.DB(D)	None	28	120	.DC,DOUBLE
ArctangentComputation:								
Real Arctangent (5)	.ED		$\tan^{-1}$ (a)	R=.ED(R)	None	26	67	.EC,REAL
DP Arctangent (5)	.DD	JMS* SUBR NOTE Enter with argument in floating accumulator. Returns with result in floating accumulator.	$\tan^{-1}$ (a)	D=.DD(D)	None	34	146	.DC,DOUBLE
Logarithm (Base 2) Computation:								
Real Log (7)	.EE		$\log_2 a$	R=.EE(R)	#13,ARG $\leq$ 0	26	71	.ER,REAL
DP Log	.DE		$\log_2 a$	D=.DE(D)	#14,ARG $\leq$ 0	32	101	.ER,DOUBLE
Exponential Computation:								
Real Exponential (2)	.EF		e^X	R=.EF(R)	None	26	116	REAL
DP Exponential (2)	.DF		e^X	D=.DF(D)	None	34	137	DOUBLE
Polynomial Evaluation:								
Real Polynomial Evaluation (8)	.EC	JMS* .EC or .DC CAL PLIST : : : :	$x=\sum_{i=0}^{n} C_{2i+1} Z^{2i+1}$	R=.EC(R_2,R_1, ...R_n)	None	N.A.	44	REAL
DP Polynomial Evaluation (8)	.DC	PLIST -N /-No. of terms +1 C_n /last term C_{n-1} /next to last : C_1 /2nd term C_0 /1st term	$x=\sum_{i=0}^{n} C_{2i+1} Z^{2i+1}$	D=.DC(D_2,D_1, ...D_n)	None	N.A.	47	DOUBLE

Table 9-1 (Cont)
The Science Library

Routine Name	Mnemonic	Calling Sequence	Function	Mode	Errors	Accur. Bits	Storage (Octal)	External Calls
SUB-FUNCTIONS (Cont)								
General Get Argument	.DA	Routine that calls: JMS* SUBR; JMP .+n+1; .DSA ARG1; .DSA ARG2; ... DSA ARGn. Calling Routine: SUBR CAL 0; JMS* .DA; JMP .+n+1; (address of ARG1); (address of ARG2); ...; (address of ARGn)	N.A.	N.A.	None	N.A.	47	None
ARITHMETIC PACKAGE								
Integer Arithmetic:	INTEGE	ARG1 A-Register / ARG2					Note 2	
Multiplication	.AD	Multiplicand / Multiplier	I*J	I=I*I	None			
Division	.AE	Dividend / Divisor	I/J	I=I/I	None			
Reverse Division	.AF	Divisor / Dividend	J/I	I=J/I	None			
Subtraction	.AY	Minuend / Subtrahend	I-J	I=I-I	None			
Reverse Subtraction	.AZ	Subtrahend / Minuend	J-I	I=I-I	None			
Double Precision Arithmetic:	DOUBLE	ARG1 FL.ACC. / ARG2					203	REAL
Load	.AO	/ Address	N.A.	D=.AO(D)	None	N.A.		
Store	.AP	Value / Address	N.A.	D=.AP(D)	None	N.A.		
Add	.AQ	Augend / Addend	A+B	D=D-D	None			
Subtract	.AR	Minuend / Subtrahend	A-B	D=D-D	None			
Reverse Subtract	.AU	Subtrahend / Minuend	B-A	D=D-D	None			
Multiply	.AS	Multiplicand / Multiplier	A*B	D=D*D	None			
Divide	.AT	Dividend / Divisor	A/B	D=D/D	None			
Reverse Divide	.AV	Divisor / Dividend	B/A	D=D/D	None			
Real Arithmetic (Includes Floating):	REAL	ARG1 FL.ACC. / ARG2					Note 3	
Load	.AG	/ Address	N.A.	R=.AG(R)	None	N.A.		
Store	.AH	Value / Address	N.A.	R=.AH(R)	None	N.A.		
Add	.AI	Augend / Addend	A+B	R=R+R	None			
Subtract	.AJ	Minuend / Subtrahend	A-B	R=R-R	None			
Reverse Subtract	.AM	Subtrahend / Minuend	B-A	R=R-R	None			
Multiply	.AK	Multiplicand / Multiplier	A*B	R=R*R	None			
Divide	.AL	Dividend / Divisor	A/B	R=R/R	None			
Reverse Divide	.AN	Divisor / Dividend	B/A	R=R/R	None			

(Double Precision and Real Arithmetic: JMS* SUBR; .DSA ARG2)
(Integer Arithmetic: LAC ARG2)

NOTES: 2. 114_8 for EAE, 160_8 for non EAE (PDP-15); 117_8 for EAE, 202_8 for non EAE (PDP-9).
3. 1022_8 for EAE, 747_8 for non EAE (PDP-15); 1034_8 for EAE, 757_8 for non EAE (PDP-9).

Table 9-1 (Cont)
The Science Library

Routine Name	Mnemonic	Calling Sequence			Function	Mode	Errors	Accur. Bits	Storage (Octal)	External Calls
ARITHMETIC PACKAGE (Cont)										
Floating Arithmetic		A-Register	FL.ACC.							
Float	.AW	Integer	F.P. No.		A←I	R=.AW(I)	None	N.A.		
Fix	.AX		F.P. No.	JMS* SUBR	I←A	I=.AX(R)	None	N.A.		
Negate	.BA				A←-A	R=.BA(R)	None	N.A.		
		FL.ACC.	HELD ACC.							
Multiply	.CA	Multiplicand	Multiplier		A*B	R=R*R	None			
Divide	.CI	Divisor	Dividend		A/B	R=R/R	None			
Add	.CC	Augend	Addend	JMS* SUBR	A+B	R=R+R	None			
Normalize	.CD	Value			N.A.	R=.CD(R)	None	N.A.		
Hold	.CF	Value			N.A.	R=.CF(R)	None	N.A.		
Round & Sign	.CH	Value			N.A.	R=.CH(R)	None	N.A.		
Sign Control	.CG	Value	Value		Note 4	R=.CG(R)	None	N.A.		
Short Get Argument	.CB	CAL 0 JMS* .CB CAL 0 .DSA 0			N.A.	R=.CB(R)	None	N.A.		
Short Get Argument	.CB									

NOTES: 4. The sign of the result (The exclusive OR of the sign bits of .AB and (CE02) is stored in .CE). The sign of .AB is saved in CE05.

When the above MACRO program is loaded, the Linking Loader attempts to satisfy the globals by searching the Science Library. The External Function SIN and the REAL ARITHMETIC package are loaded. The references to these routines in the MACRO program must be indirect (as indicated in the example) because only the transfer vectors are given in the main program.

9.7 SCIENCE LIBRARY ALGORITHM DESCRIPTIONS

9.7.⚠1 SQUARE ROOT (SQRT, DSQRT)

A first-guess approximation of the square root of the argument is obtained as follows.

If the exponent (EXP) of the argument is odd:

$$P_0 = .5^{(\frac{EXP-1}{2})} + ARG^{(\frac{EXP-1}{2})}$$

If the exponent (EXP) of the argument is even:

$$P_0 = .5^{(\frac{EXP}{2})} + ARG^{(\frac{EXP}{2}-1)}$$

Newton's iterative approximation is then applied three times.

$$P_{i+1} = \frac{1}{2} \quad (P_i + \frac{ARG}{P_i})$$

9.7.⚠2 EXPONENTIAL (EXP, DEXP, .EF, .DF)

The function e^x is calculated as $2^{x \log_2 e}$, where $x \log_2 e$ will have an integral portion (I) and a fractional portion (F). Then

$$e^x = (2^I)(2^F)$$

where $$2^F = \left(\sum_{i=0}^{n} C_i F^i \right)^2$$ and n = 6 for EXP and .EF, or n = 8 for DEXP and .DF.

The values of C are:

$C_0 = 1.0$

$C_1 = 0.34657359$

$C_2 = 0.06005663$
$C_3 = 0.00693801$
$C_4 = 0.00060113$
$C_5 = 0.00004167$
$C_6 = 0.00000241$
$C_7 = 0.00000119$
$C_8 = 0.000000518$

9.7.3 NATURAL AND COMMON LOGARITHMS (ALOG, ALOG10, DLOG, DLOG10)

The exponent of the argument is saved as one greater than the integral portion of the result. The fractional portion of the argument is considered to be a number between 1 and 2. Z is computed as follows.

$$Z = \frac{X - \sqrt{2}}{X + \sqrt{2}}$$

Then, $\log_2 X = \frac{1}{2} + \left(\sum_{i=0}^{n} C_{2i+1} Z^{2i+1} \right)$

where n = 2 for ALOG, and n = 3 for DLOG. The values of C are as follows:

ALOG and ALOG10	DLOG and DLOG10
$C_1 = 2.8853913$	$C_1 = 2.8853900$
$C_3 = 0.96147063$	$C_3 = 0.96180076$
$C_5 = 0.59897865$	$C_5 = 0.57658434$
	$C_7 = 0.43425975$

Finally,

$\log_e X = (\log_2 X)(\log_e 2)$, for ALOG and DLOG,

and

$\log_{10} X = (\log_2 X)(\log_{10} 2)$, for ALOG10 and DLOG 10.

9.7.4 SINE AND COSINE (SIN,COS,DSIN,DCOS, .EB, .DB)

The argument is converted to quarter circles by multiplying by $2/\pi$. The low two bits of the integral portion determine the quadrant of the argument and produce a modified value of the fractional portion (Z) as follows.

Low 2 Bits	Quadrant	Modified Value (Z)
00	I	F
01	II	1-F
10	III	-F
11	IV	-(1-F)

Z is then applied to the following polynomial expression.

$$\sin X = \left(\sum_{i=0}^{n} C_{2i+1} Z^{2i+1} \right)$$

where n=4 for REAL routines, and n=6 for DP routines. The values of C are as follows.

REAL ROUTINES	DP ROUTINES
$C_1 = 1.570796318$	$C_1 = 1.5707932680$
$C_3 = -0.645963711$	$C_3 = -0.6459640975$
$C_5 = 0.079689677928$	$C_5 = 0.06969262601$
$C_7 = -0.00467376557$	$C_7 = -0.004681752998$
$C_9 = 0.00015148419$	$C_9 = 0.00016043839964$
	$C_{11} = -0.000003595184353$
	$C_{13} = 0.000000054465285$

The argument for COS and DCOS routines is adjusted by adding $\pi/2$. The sin subfunction is then used to compute the cosine according to the following relationship:

$$\cos x = \sin \left(\frac{\pi}{2} + x\right)$$

9.7.⚠5 ARCTANGENT (ATAN, DATAN, ATAN2, DATAN2, .ED, .DD)

For X less than or equal to 1, Z = X, and:

$$\text{arctangent } X = \left(\sum_{i=0}^{n} C_{2i+1} Z^{2i+1} \right)$$

where n = 7 for REAL routines and n = 3 for DP routines. For X greater than 1, Z = 1/X, and

$$\text{arctangent } X = \frac{\pi}{2} - \left(\sum_{i=0}^{n} C_{2i+1} \, Z^{2i+1} \right)$$

where n = 8 for REAL routines and n = 3 for DP routines. The values of C are as follows.

REAL ROUTINES	DP ROUTINES
$C_1 = 0.9992150$	$C_1 = 0.9999993329$
$C_3 = -0.3211819$	$C_3 = -0.3332985605$
$C_5 = 0.1462766$	$C_5 = 0.1994653599$
$C_7 = -0.0389929$	$C_7 = -0.1390853351$
	$C_9 = 0.0964200441$
	$C_{11} = -0.0559098861$
	$C_{13} = 0.0218612288$
	$C_{15} = -0.0040540580$

9.7. (6) HYPERBOLIC TANGENT (TANH)

$$\tanh |X| = \left(1 - \frac{2}{1+e^{2|X|}} \right)$$

e^X, calculated as $2^{x \log_2 e}$, where $x \log_2 e$ will have an integral portion (I) and a fractional portion (F), then

$$e^X = (2^I)(2^F)$$

where

$$2^F = \left(\sum_{i=0}^{n} C_i F^i \right)^2 \quad \text{and } n = 6$$

The values of C are as follows.

$C_1 = 1.0$	$C_5 = 0.00060113$
$C_2 = 0.34657359$	$C_6 = 0.00004167$
$C_3 = 0.06005663$	$C_7 = 0.00000241$
$C_4 = 0.00693801$	

9.7.⑦ LOGARITHM, BASE 2 (.EE, .DE)

The exponent of the argument is saved as one greater than the integer portion of the result. The fractional portion of the argument is considered to be a number between 1 and 2. Z is computed as follows.

$$Z = \frac{X - \sqrt{2}}{X + \sqrt{2}}$$

Then,

$$\log_2 X = \frac{1}{2} + \left(\sum_{i=0}^{n} C_{2i+1} Z^{2i+1} \right)$$

where n = 2 for .EE and n = 3 for .DE. The values of C are as follows:

.EE	.DE
$C_1 = 2.8853913$	$C_1 = 2.8853900$
$C_3 = 0.96147063$	$C_3 = 0.96180076$
$C_5 = 0.59897865$	$C_5 = 0.57658434$
	$C_7 = 0.43425975$

9.7.⑧ POLYNOMIAL EVALUATOR (.EC, .DC)

The polynomial is evaluated as follows:

$$X = Z\,(C_0 + Z^2\,(C_1 \ldots + Z^2\,(C_n Z^2 + C_{n-1})))$$

Part 3
SCIENCE LIBRARY

APPENDIX A

FORTRAN IV, ADDITIONAL INFORMATION

The FORTRAN language used in this manual is essentially the language of USASI Standard FORTRAN (X3.9-1966). The following features are modified to allow the compiler to operate in 8192 words of core storage:

a. All references to complex arithmetic are illegal.
b. The size of arrays in subprograms is not adjustable to the size specified by the calling program.
c. Blank COMMON is treated as name COMMON.
d. The implied DO feature is not legal in a DATA statement.

There are two versions of the FORTRAN IV compiler: F4 and F4A. F4 is the basic compiler; F4A is an abbreviated version of the compiler that allows DECtape input and output in an 8K system. F4A operates under control of the Keyboard Monitor only, and is called by typing F4A rather than F4 on the teletype. The F4A version does not provide for EQUIVALENCE, EXTERNAL, ASSIGN, and Assigned GO TO statements, or the following options available in the F4 version:

O	Object code listing
S	Symbol table printout

In paper tape systems, the FORTRAN compiler, along with necessary I/O device handlers and an appropriate version of the I/O Monitor, are punched on a tape in absolute format, referred to as a system tape. At the beginning of the system tape is a Bootstrap Loader. The system tape can be loaded by setting the starting address of the Loader (17720 for 8K systems, 37720 for 16K) on the console address switches, pressing I/O RESET, and then pressing the READIN switch.

In larger systems with a bulk storage device such as DECtape, the Monitor accepts direct keyboard commands to load the compiler in a device-independent environment. This feature enables use of READ (I,f) or READ (I) statements where the value of I is undefined at compile and load times. If such statements are used, it is important to clear unused positive .DAT slots before loading to avoid loading device handlers that are not required.

Either the DDT or Linking Loader utility program must be used to load user object programs for execution. Refer to the appropriate System User's Guide for operating procedures.

APPENDIX B

FORTRAN IV AND MACRO LINKAGE

B.1 LINKING FORTRAN IV PROGRAMS WITH MACRO SUBPROGRAMS

There are two essential elements in a MACRO subprogram that is linked to FORTRAN IV. One is the declaration of the name of the subprogram (as used in the F4 program) in a .GLOBL statement within the subprogram. The second is leaving open registers in the subprogram for the transfer vectors of the arguments used in the FORTRAN calling sequence. The number of open registers must agree with the number of arguments given in the calling sequence.

For example, consider a FORTRAN program and a MACRO subprogram in which one positive, single-precision, floating-point number is read by the FORTRAN program, negated in the MACRO subprogram, and written out from the FORTRAN program.

FORTRAN IV PROGRAM:

```
C            TEST MACRO SUBPROGRAM
C            READ A NUMBER (A)
1            READ (1,100) A
100          FORMAT (E12.4)
C            NEGATE THE NUMBER AND PUT IT IN B
             CALL MIN (A,B)
C            WRITE OUT THE NUMBER (B)
             WRITE (2,100) B
             STOP
             END
```

MACRO-9 SUBPROGRAM:

```
             .TITLE MIN
             .GLOBL MIN,.DA
MIN          0                    /ENTRY/EXIT
             JMS*       .DA       /USE THE F4 GENERAL GET ARGUMENT
                                  /SUBPROGRAM TO LOAD THE ARGUMENTS
             JMP        .+2+1     /JUMP AROUND REGISTERS LEFT FOR
                                  /ARGUMENT ADDRESS EST†
```

† .DA uses the address .+N+1 to calculate the number of argument addresses to be passed.

```
MIN1        .DSA     Ø          /ARG 1
MIN2        .DSA     Ø          /ARG2
            LAC*     MIN1       /PICK UP FIRST WORD OF A
            DAC*     MIN2       /STORE IN FIRST WORD OF B
            ISZ      MIN1       /BUMP THE POINTER TO SECOND WORD
            ISZ      MIN2       /OF A AND B
            LAC*     MIN1       /PICK UP SECOND WORD OF A
            TAD      (400000    /SIGN BIT = 1
            DAC*     MIN2       /STORE IN SECOND WORD OF B
            JMP*     MIN        /EXIT
            .END
```

Since A is a single-precision, floating-point number, two machine words are required and must be accounted for in the subprogram. Thus, MIN1 and MIN2 (which contain the addresses of A and B) must be incremented to get to the second word of each number. FORTRAN expands the CALL statement as follows:

```
     CALL MIN (A,B)
ØØØ13          JMS*      MIN          /(EXIT TO MACRO SUBPROGRAM)
ØØØ14          JMP       $ØØØ14       /(ENTRY FROM MACRO SUBPROGRAM)
ØØØ15          .DSA      A
ØØØ16          .DSA      B
$ØØØ14=ØØØ17
```

When the program is loaded, the address (plus relocation factor) of A is stored in location 00015 (plus relocation factor) and the address of B is stored in 00016 (plus relocation factor). When .DA is called from the MACRO subprogram, it stores the addresses in MIN1 and MIN2 (plus relocation factor). Thus, MIN1 must be referenced indirectly to get the value of A (a direct reference gets the address of A).

The subroutine .DA allows one level of indirection. All FORTRAN arguments are referred to by the 15-bit address of their first word. This leaves bits 0 through 2 free for flags. By convention, FORTRAN uses bit 0 to indicate to .DA that the word specifying the argument contains the 15-bit address of a word which contains the 15-bit address of the first word of the argument. The resulting argument word in the called MACRO subroutine always contains a direct reference to an argument (the 15-bit address of the first word of the argument).

In the case of unsubscripted array names used as arguments in a FORTRAN CALL statement or function reference, the argument is represented by the 15-bit address of the fourth word of the array descriptor block. (Refer to Paragraph 8.5 OTS for an explanation of the contents of an array descriptor block, as well as the calling sequence of .SS and the algorithm for determining the array element address.)

In the foregoing example a MACRO subroutine was used instead of a FORTRAN SUBROUTINE subprogram. There is no difference in the calling procedures used by FORTRAN to call SUBROUTINE and FUNCTION subprograms. However, for FUNCTION subprograms FORTRAN expects a value to be returned in the A-register (LOGICAL or INTEGER functions) or in the floating accumulator (REAL or DOUBLE-PRECISION functions).

B.2 LINKING MACRO PROGRAMS WITH FORTRAN IV SUBPROGRAMS

There are two forms of FORTRAN IV subprograms: subroutines and external functions. The main difference between the two is the method of returning arguments to the calling program: subroutines return the argument directly to the calling program, while functions return arguments through accumulators.

The MACRO program setup for a FORTRAN IV subroutine is basically that described for FORTRAN IV Science Library routines in Part III of this manual. The name of the subroutine to be called must be declared as a global; there must be a jump around the argument addresses, and the number and type (integer, real, double precision) of arguments in the calling program, and the subroutine must agree.

An example of a calling routine:

```
TITLE

.GLOBL          SUBROT
JMS*            SUBROT
JMP             .+N+1           /JMP AROUND ARGUMENTS IGNORED BY .DA
.DSA            ARG1            /FIRST ARG ADDR†
.DSA            ARG2            /2ND ARG ADDR†
  .
  .
  .
.DSA            ARGN            /NTH ARG ADDR†
  .
  .
```

When the FORTRAN IV subroutine is compiled, the compiler generates code for .DA, the General Get Argument Routine, which transfers the arguments from the MACRO calling program to the FORTRAN IV subroutine. .DA expects to find the calling sequence just described for the calling program. The following is an example of an expansion of the beginning of a FORTRAN IV subroutine.

```
          C                       TITLE SUBROT
                                  SUBROUTINE SUBROT (A,B)
000000                              CAL         0
000001                              JMS*        .DA
000002                              JMP         $000002
000003                              .DSA        A
00004                               .DSA        B
$000002=000005
```

The simplest method of passing arguments between the main program and the subroutine is to use one of the calling arguments as output. For example, if the value of D is to be calculated in the subroutine, use D as one of the calling arguments. "D=" generates DAC* D, which stores the value calculated for D by the subroutine in location D in the calling program.

†Bit 0 of each address can be set to 1 to indicate indirect references.

The MACRO program setup for a FORTRAN IV External Function is identical to that for linkage with subroutines, except that some provision must be made for storage of the values calculated and stored in the accumulator. In the case of integers, the value is returned in the A-register. The value is returned in the floating accumulator for real and double-precision numbers. The simplest method of storing the values is to use the FORTRAN IV routines furnished in the library for this purpose. .AH stores real values, and .AP stores double-precision values. Since the A-register is the standard hardware accumulator, a DAC instruction stores integer values.

B.3 LINKING MACRO PROGRAMS WITH FORTRAN IV LIBRARY ROUTINES

Refer to Part III of this manual, The Science Library, for a complete description of the linkage to library routines and the conventions for representing floating-point variables in FORTRAN. (INTEGER variables are in 2's complement notation, logical truth is 777777 and logical falsity is 000000 in unsigned octal representation.)

B.4 MORE ILLUSTRATIVE EXAMPLES

B.4.1 A New Dimension Adjustment Routine

The present versions of the OTS routines ADJ1, ADJ2, and ADJ3 do not alter the size of the array being adjusted (Refer to Paragraph 8.11.4 through 8.11.6). If only the array name of an adjusted array is given in a READ or WRITE argument list, FORTRAN uses this size information; therefore, undesired results can occur. A new routine (ADJ) can be loaded with a user program which completely handles all cases of dimension adjustment, although it occupies 72 octal locations. (ADJ3 occupies 41 octal locations.) Consider the following programs:

```
C    PROGRAM 1
      DIMENSION A(4,3,2)
      .
      .
      .
C    MAKE ARRAY A ACT LIKE IT
C    WAS DIMENSIONED A (2,3,4)
      CALL ADJ(A,A(1,1,1),2,3,4)

C    PROGRAM 2
      DIMENSION A(3,2)
      .
      .
      .
C    ADJUST ARRAY A TO BE A (2,3)
      CALL ADJ (A,A(1,1),2,3,0)
C    THE LAST ARGUMENT MUST BE 0
      .
      .
      .
```

```
          C    PROGRAM 3
               DIMENSION A(2)
               .
               .
               .
          C    ADJUST ARRAY A TO BE A(1)
               CALL ADJ(A,A(1),1,Ø,Ø)
          C    THE LAST 2 ARGUMENTS MUST BE ZERO
          C    THE NO. OF SUBSCRIPTS IS NOT ADJUSTABLE


          .TITLE ADJ
/
/SUBROUTINE TO PERFORM DIMENSION ADJUSTMENT
/
/MACRO-9 CALLING SEQUENCE
/         .GLOBL ADJ
/         JMS* ADJ
/         JMP .+6
/         .DSA ARRAY           /ADDRESS OF WD4
/         .DSA B     /NEW WD4
/         .DSA K1    /ADDRESS OF NEW MAXIMUM 1ST SUBSCRIPT
/         .DSA K2    /ADDRESS OF NEW MAXIMUM 2ND SUBSCRIPT
/         .DSA K3    /ADDRESS OF NEW MAXIMUM 3RD SUBSCRIPT
/
          .GLOBL ADJ,.DA,.AD
ADJ       Ø
          JMS* .DA   /GET ARGUMENTS
          JMP .+5+1 /# OF ARGUMENTS = 5
ARRAY     Ø
B         Ø
K1        Ø
K2        Ø
K3        Ø
          LAC (LAC* B         /INITIALIZE SUBSCRIPT POINTER
          DAC C
          LAC B      /SET NEW STARTING ADDRESS
          DAC* ARRAY
          LAW -3
          DAC CTR#   /MAXIMUM OF 3 SUBSCRIPTS
          TAD ARRAY
          DAC ARRAY /POINT TO FIRST WORD
          DAC ARRAYP#          /OF ARRAY DESCRIPTOR BLOCK
          LAC* ARRAY           /ARRAY TYPE IN BITS 3-4
          AND (60000           /ZERO OUT ARRAY SIZE
          DAC* ARRAY           /SAVE CLEAN ARRAY TYPE
          RTL
          RTL
          RTL
          TAD (1     /ADD 1 FOR # OF WORDS
          AND (3     /AND TREAT LOGICAL
          SNA        /AS 1 WORD PER ARRAY ELEMENT
          LAC (1
LOOP      ISZ C      /POINT TO NEXT SUBSCRIPT
          JMS* .AD   /MULTIPLY INTEGERS
C         LAC* K1    /PROGRAM MODIFIED
          SNA        /IS SUBSCRIPT PRESENT
          JMP D      /RAN OUT OF SUBSCRIPTS
          DAC SIZE# /UPDATE SIZE
          ISZ CTR    /ARE WE FINISHED?
          SKP
          JMP E      /YES
```

```
          ISZ ARRAYP            /STORE INTO ARRAY
          DAC* ARRAYP           /DESCRIPTOR BLOCK
          JMP LOOP  /OFFSET WORDS (2,3)
D         DZM* ARRAYP           /ZERO THE REST
          ISZ ARRAYP            /OF THE OFFSET WORDS

          ISZ CTR   /ARE WE FINISHED
          JMP LOOP  /NO
E         LAC SIZE  /FINISHED
          AND (17777            /PACK SIZE
          XOR* ARRAY            /ARRAY DESCRIPTOR BLOCK
          DAC* ARRAY
          JMP* ADJ  /RETURN
          .END
```

B.4.2 A Function to Read the AC Switches

It is very often desirable to use the AC switches to alter the sequence of instructions executed in a FORTRAN program. The following program can be used as a function in an arithmetic IF statement to conditionally branch.

```
          .TITLE ITOG
/
/SUBROUTINE TO READ AC SWITCHES
/
/MACRO-9 CALLING SEQUENCE
/         .GLOBL ITOG
/         JMS* ITOG
/         JMP .+2    /JUMP OVER ARGUMENT
/         .DSA (MASK           /ADDRESS OF MASK
/                              /RETURN WITH MASKED ACS IN AC
           .GLOBL ITOG,.DA
ITOG      Ø          /INTEGER FUNCTION
          JMS* .DA   /GET ARGUMENTS
          JMP .+1+1  /1 ARGUMENT
MASK      Ø          /MASK ADDRESS
          LAS        /LOAD AC FROM SWITCHES
          AND* MASK  /MASK AC
          JMP* ITOG  /RETURN WITH MASKED AC SWITCHES
          .END
```

B.4.3 A Routine to Read an Array in Octal

The present version of the Object-Time System does not read octal FORMATTED information. A MACRO subroutine which reads octal information (REDAR) is as follows:

```
          .TITLE REDAR
/
/SUBROUTINE TO READ ARRAY IN OCTAL
/
/MACRO-9 CALLING SEQUENCE
/         .GLOBL REDAR
/         JMS* REDAR
/         JMP .+5
/         .DSA SLOT /ADDRESS OF SLOT #
/         .DSA FORMAT          /ADDRESS OF FORMAT STATEMENT ADDRESS
/         .DSA DIGITS          /ADDRESS # OF DIGITS
/         .DSA ARRAY           /ADDRESS OF ARRAY DESCRIPTOR
/                              /BLOCK WORD 4
/
          .GLOBL REDAR,.DA,.FR,.FE,.FF
REDAR     Ø
          JMS* .DA  /GET ARGUMENTS
          JMP .+4+1 /#ARGUMENTS = 4
SLOT      Ø
FORMAT    Ø
DIGITS    Ø
ARRAY     Ø
          LAC SLOT
          DAC A
          LAC* FORMAT
          DAC B
          JMS* .FR  /FORMATED WRITE
A         XX        /ADDRESS DAT SLOT #
B         XX        /ADDRESS OF FORMAT STATEMENT
          LAW -3
          TAD ARRAY
          DAC SLOT  /ADDRESS OF ARRAY DESCRIPTOR BLOCK WORD 1
          LAC* SLOT /PICK UP PACKED SIZE OF ARRAY
          AND (17777          /CLEAN OFF MODE #
          SNA
          JMP E     /NO ELEMENTS IN ARRAY
          CMA
          DAC SLOT
          ISZ SLOT  /COUNTER FOR # WORDS IN ARRAY
          LAC* DIGITS         /#DIGITS IN EACH WORD
          AND (7    /CLEAN ARGUMENT
          SZA
          SAD (7
          JMP E     /Ø OR 7 DIGITS ILLEGAL
          CMA
          TAD (1
          DAC C     /INITIALIZE LAW INSTRUCTION
          LAC* ARRAY
          DAC ARRAY /POINTER TO FIRST WORD OF ARRAY
          XX        /LAW -DIGITS
          DAC DIGITS
          CLA       /INITIALIZE DIGIT PACK
          DAC TEMP# /STORE DIGIT PACK
          JMS* .FE  /READ DIGIT
          .DSA FORMAT         /DIGIT READ INTO FORMAT
          LAC TEMP  /LOAD DIGIT PACK
```

```
            CLL
            RTL       /MULTIPLY BY 8
            RAL
            TAD FORMAT          /ADD DIGIT
            ISZ DIGITS          /COUNT DIGITS
            JMP D     /GO BACK FOR MORE
            DAC* ARRAY          /STORE VALUE IN ARRAY ELEMENT
            ISZ ARRAY /POINT TO NEXT ARRAY WORD
            ISZ SLOT  /COUNT ARRAY WORDS
            JMP C     /READ ANOTHER WORD
E           JMS* .FF  /END OF READ
            JMP* REDAR          /EXIT
            .END
```

B.4.4 A FORTRAN Program Using the Foregoing Programs

This FORTRAN program uses the preceding three MACRO programs to read in an array from the teletype in octal and type it in decimal. The teletype should be assigned to .DAT slot 4. Note how the arguments are specified. Because the array J is never referenced with subscripts at object time, it can be altered at object time to have more than one subscript, although this fact is academic. Notice that EQUIVALENCE performs the array element calculation at compile time.

```
C FORTRAN PROGRAM TO READ AN ARBITRARY INTEGER ARRAY IN OCTAL
C AND WRITE IT IN DECIMAL
        DIMENSION J(2000)
C USE EQUIVALENCE TO GET J(1) WITHOUT USING .SS
        EQUIVALENCE (J(1),K)
C I CONTAINS ADDRESS OF FORMAT
C STATEMENT + 1 TO MOVE OVER JMP INSTRUCTION
        ASSIGN 1 TO I
        I=I+1
1       FORMAT(6I1,1X,6I1,1X,6I1,1X,6I1,1X,6I1,1X,6I1,1X,6I1,1X,
        16I1)
C TO SIMULATE FORMAT(Ø6,1X,Ø6,1X,Ø6,1X,Ø6,1X,Ø6,1X,Ø6,1X,
C Ø6,1X,Ø6)
C WRITE SOMETHING TO SHOW INFORMATION NEEDED
2       WRITE(4,3)
3       FORMAT(/19H READ K1 K2 K3(3I4))
C READ IN DIMENSION INFORMATION
        READ(4,4) K1,K2,K3
4       FORMAT(3I4)
C ADJUST ARRAY J TO THE PROPER SIZE
        CALL ADJ(J,K,K1,K2,K3)
C READ IN ARRAY IN OCTAL
5       CALL REDAR(4,I,6,J)
C WRITE OUT ARRAY
        WRITE(4,6) J
6       FORMAT(8I7)
C WAIT FOR ↑P
        PAUSE
C IF AØS17-Ø READ IN IDENTICAL ARRAY TYPE
        IF (ITOG(1)) 2,5,2
        END
```

APPENDIX C

CHAINING FORTRAN IV PROGRAMS

Chaining is a method of program segmentation that allows for multiple core overlap of executable code and certain types of data areas. FORTRAN programs can thus be divided into segments and executed separately, with intersegment communication of data accomplished through common storage. Common areas of core are reserved by means of the blank COMMON statement.

Transfer of control from one chain segment to another can be specified in a FORTRAN source program with the statement

```
CALL CHAIN (N)
```

where N is the segment number to be called. The chain number (N) is established at chain-build time (refer to the CHAIN section of the applicable System User's Guide). N can be greater than or less than (but not equal to) the current chain number. Only variables and arrays named in blank COMMON statements are retained from one chain segment to another. Blank common size should be the same for all chain segments.

NOTE

Use of a CALL CHAIN (N) statement rather than a STOP statement immediately preceding the END statement causes an I error during compilation (illegal statement preceding the END statement). The I error should be ignored; it is a warning only. The CHAIN subroutine never returns control to the statement following the CALL CHAIN (N) statement (control is transferred to the beginning of the chain which is called).

```
C          TEST CHAIN PROGRAM
C
C          CHAIN JOB SEGMENT 1
           COMMON A,B,C
           DIMENSION ARRAY (10,10)
           READ (4,5) ARRAY
                    .
                    .
                    .
           CALL CHAIN (2)
           END
```

```
C          CHAIN JOB SEGMENT 2
           COMMON A,B,C
           DIMENSION TABLE (30)
                    .
                    .
                    .
           CALL CHAIN (3)
           END

C          CHAIN JOB SEGMENT 3
           COMMON A,B,C
           DIMENSION A LIST (5,5)
                    .
                    .
                    .
C          WRITE (4,6) A LIST
           FORMAT (E10.3)
           STOP
           END
```

APPENDIX D

FORTRAN IV ERROR LIST

The errors shown in Table D-1 apply to all versions of F4 and F4A (refer to Table 8-1 of this manual for a list of object-time errors).

Table D-1
Compilation Errors†

Error Code	Meaning	Explanation
X	Syntax error	Statement cannot be recognized as a properly constructed FORTRAN IV statement.
V	Variable/constant mode error	Illegal mode mixing. Missing constant, variable or exponent, or illegal matching of constants or variables in a DATA statement. (See Note.)
N	Statement number error	Phase error, number more than five digits, no statement number where one is required, statement should not be labeled, or doubly defined statement numbers.
S	Argument/subscript error	Missing argument or subscript, illegal use of subscripts, illegal construction of subscripted variable, more than three subscripts or stated number of subscripts does not agree with declared number.
F	FORMAT statement error	Illegal FORMAT specification or illegal construction of FORMAT statement. (Refer to Paragraph D.1.)
I	Character/statement/term error	Illegal character, unrecognizable statement, illegal statement for program type, statement out of order or improper statement preceding END statement.
D	DO loop error	Illegal DO construction or illegal statement terminating DO LOOP.
T	Table overflow	Symbol/constant/arg (I)/OP(I) table limits exceeded. (Refer to Paragraph D.2.)
L	Nesting error	Illegal nesting or DO nesting too deep.

†Occasionally FORTRAN IV prints out the line after the error line.

Table D-1 (Cont)
Compilation Errors†

Error Code	Meaning	Explanation
M	Magnitude error	Program exceeds 8192 words, maximum number of dummy arguments or EQUIVALENCE classes exceeded, or constant/variable exceeds specified limits. (Refer to Paragraph D.3.)
C	COMMON/EQUIVALENCE/ DIMENSION/DATA statement error	Illegal construction of statement, illegal EQUIVALENCE relationships, illegal COMMON declaration or non-common storage declared in BLOCK DATA subprogram.
E	FUNCTION/SUBROUTINE/ EXTERNAL/CALL statement error	Illegal use of FUNCTION/SUBROUTINE name, out of order, or illegal variable for EXTERNAL declaration. (Refer to Paragraph D.4.)
H	Hollerith error	Hollerith data illegal in this statement or illegal use of Hollerith constant.

†Occasionally FORTRAN IV prints out the line after the error line.

NOTE

Hollerith constants and alphanumeric information read in under A format are stored as REAL variables. Only nA1, nA2, ..., nA5 is allowed in reading alphanumeric information into REAL variables and arrays. If an integer variable is used to store Hollerith constants in a DATA statement, a V error occurs in compilation.

D.1 TECHNIQUES FOR AVOIDING F ERRORS

a. The following ASCII characters are ignored:

! (041)	> (076)
" (042)	? (077)
& (046)	@ (100)
' (047)	[(133)
: (072)	\ (134)
; (073)	] (135)
< (074)	↑ (136)

If these characters are counted in the number preceding H, an F error occurs. The following statements fail because the FORMAT statement causes an F error.

```
  WRITE (4,1)
1 FORMAT (5H  WHO?)
```

The following sequence allows the user to type out the question mark:

```
C READ IN A QUESTIONMARK
      READ (4,2) QSTMK
2     FORMAT (A1)
      .
      .
      .
      WRITE (4,1) QSTMK
1     FORMAT (4H WHO,A1)
```

b. An F error occurs if "/," or ",/" occurs in a FORMAT statement. Omitting the comma in such instances prevents the F error from occurring.

D.2 TECHNIQUES FOR AVOIDING T ERRORS

a. A maximum of 14 arguments is allowed in the argument-operator table. An implied DO configuration as a parenthesized element in a READ or WRITE statement is completely stored in the argument-operator table before any code is generated. An array element is stored as 2+n arguments, where n=1, 2, or 3 is the number of subscripts for the array. A simple variable is stored as one argument. The DO information for each loop is stored as three or four arguments, depending on whether the DO increment is implied (1) or given explicitly. The following statement compiles:

```
WRITE(4,10)  (I0(1,K),(I0(J,K),J=1,10),K=1,10)
          14=    4    +   4    + 3       + 3
```

The following statement gives a T error:

```
WRITE(4,10)      (K,I0(1,K),(I0(J,K),J=1,10),K=1,10)
              15= 1+4     +       4       +3        +3
```

This statement can be rewritten as an explicit DO loop.

```
      DO 1 K=1, 10
    1 WRITE (4,10) K, I0(1,K) , (I0(J,K) ,J=1,10)
                             7= 4  +       3
```

Each execution of the WRITE statement starts at the beginning of the FORMAT statement.

b. A maximum of 14 addresses is allowed in computed GO TO statements. The following statement gives a T error:

```
1 GO TO (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16), I
```

This statement can be rewritten as follows:

```
1 J=I-14
IF (J.GT.0) GO TO (15,16),J
GO TO (1,2,3,4,5,6,7,8,9,10,11,12,13,14), I
```

c. A maximum of 14 items is allowed in a DATA statement list.

The following statement gives a T error:

```
DATA I,I,I,I,I,I,I,I,I,I,I,I,I,I,I/15*1/
```

The foregoing statement can be rewritten as follows:

```
DATA I,I,I,I,I,I,I,I,I,I,I,I,I,I/14*1/,I/1/
```

D.3 TECHNIQUES FOR AVOIDING M ERRORS

a. No program unit may exceed one core bank (only 8192 computer words are addressable). All non-COMMON storage in a FORTRAN program is included in the program size. To avoid M errors, break long programs up into subroutines, and put large arrays in COMMON.

b. The size of arrays is limited to 8192 computer words (one core bank). The size of an array in computer words can be determined by the DIMENSION statement in which it occurs and by its mode type. For example, consider the following array:

```
DIMENSION ARRAY (5,10,15)
     (SIZE=n*5*10*15)
```

where n is 1 for LOGICAL and INTEGER arrays, 2 for REAL arrays, and 3 for DOUBLE-PRECISION arrays.

c. FORTRAN IV does not compile dummy variables in excess of 10 for any function, statement function, or subroutine. Every violation of this constraint causes an M error. In the case of statement functions, there is no way of avoiding this without writing smaller statement functions and combining them. For external functions and subroutines, the use of named COMMON is suggested as an alternative. For example, compare the following alternative programs.

Standard Program

a. Main Program

```
CALL FOR (A,B,C)
:
:
```

b. Subroutine

```
SUBROUTINE FOR (A,B,C)
:
:
```

Modified Program

a. Main Program

```
COMMON /DARG/D,E,F
:
:
D = A
E = B
F = C
CALL FOR
:
:
```

b. Subroutine

```
SUBROUTINE FOR
.
.
COMMON /DARG/A,B,C
.
.
```

COMMON is initialized to the proper values before calling the subroutine.

D.4 TECHNIQUE FOR AVOIDING AN E ERROR

a. A dummy function reference in a CALL statement causes an E error. The following program fails to compile:

```
SUBROUTINE O(F)
.
.
CALL F
.
.
```

The foregoing situation can be avoided by the following technique:

```
SUBROUTINE O(F)
.
.
DUMMY=F(DUMMY)
.
.
```

The contents of DUMMY should be ignored at all times. This essentially calls the SUBROUTINE whenever the statement is reached.

APPENDIX E
SYMBOL TABLE SIZES (F4 V5A)

The following symbol table sizes are for 8K systems with the full complement of skip IOTs in the skip chain.

NOTE

Handlers listed are for DAT slots -11, -12, and -13, respectively.

F4

a. PRB, TTA, PPC - 171 symbols (decimal)

b. DTC, TTA, PPC - 21 symbols (decimal)

F4A

a. PRB, TTA, PPC - 368 symbols (decimal)

b. DTC, TTA, PPC - 239 symbols (decimal)

c. DTB, TTA, DTB - 18 symbols (decimal)

INDEX

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections are published monthly by Software Information Service in the "Digital Software News for 18-Bit Computers".

These newsletters contain information applicable to software available from Digital's Program Library (see title page for address). Software products and documents are usually shipped only after the Program Library receives a specific request from a user.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it.

Please complete the card below to receive information on DECUS membership or to place your name on the newsletter mailing list.

Please send

☐ DECUS membership information,

or add my name to the

☐ DECUSCOPE non-membership list.

And, send me

☐ "Digital Software News for 18-Bit Computers"

Name _____

Company _____

Address _____

City _____ State _____ Zip _____

..... 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

DECUS
Digital Equipment Corporation
146 Main Street
Maynard, Mass. 01754

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? _____

_____

_____

_____

_____

How can this manual be improved? _____

_____

_____

_____

_____

DEC also strives to keep its customers informed of current DEC software and publications. Thus, the following periodically distributed publications are available upon request. Please check the appropriate boxes for a current issue of the publication(s) desired.

☐ **Software Manual Update,** a quarterly collection of revisions to current software manuals.

☐ **User's Bookshelf,** a bibliography of current software manuals.

☐ **Program Library Price List,** a list of currently available software programs and manuals.

Please describe your position. _____

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