decsystem10

FORTRAN IV (F40)

digital

decsystem10

FORTRAN IV (F40) PROGRAMMER'S REFERENCE MANUAL

The information in this document reflects the software as of Version 27 of the FORTRAN Compiler.

Additional copies of this manual may be ordered from: Software Distribution Center, Digital Equipment Corporation, Maynard, Ma 01754 Order Code: DEC-10-LFLMA-B-D

DIGITAL EQUIPMENT CORPORATION ● MAYNARD, MASSACHUSETTS

1st Printing March 1967
2nd Printing (Rev) November 1967
3rd Printing (Rev) September 1968
4th Printing April 1969
5th Printing June 1969
6th Printing September 1969
7th Printing (Rev) February 1970
Update Pages October 1970
Update Pages February 1971
Update Pages October 1971
Update Pages May 1972
Update Pages June 1974
8th Printing (Rev) January 1975

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this manual.

The software described in this document is furnished to the purchaser under a license for use on a single computer system and can be copied (with inclusion of DIGITAL's copyright notice) only for use in such system, except as may otherwise be provided in writing by DIGITAL.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright©1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975
by Digital Equipment Corporation

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDP	DIGITAL	INDAC	PS/8
COMPUTER LAB	DNC	KA10	QUICKPOINT
COMSYST	EDGRIN	LAB-8	RAD-8
COMTEX	EDUSYSTEM	LAB-8/e	RSTS
DDT	FLIP CHIP	LAB-K	RSX
DEC	FOCAL	OMNIBUS	RTM
DECCOMM	GLC-8	OS/8	RT-11
DECTAPE	IDAC	PDP	SABR
DIBOL	IDACS	PHA	TYPESET 8
			UNIBUS

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

FIGURES

TABLES

PREFACE

This is a reference manual describing the specific statements and features of the FORTRAN IV (F40) language for the DECsystem-10. It is written for the experienced FORTRAN programmer who is interested in writing and running FORTRAN programs alone or in conjunction with MACRO-10 programs in the time-sharing environment. Familiarity with the basic concepts of FORTRAN programming on the part of the user is assumed. DECsystem-10 FORTRAN conforms to the requirements of the USA Standard FORTRAN (1966).

FORTRAN IV (F40) is one of two FORTRAN compilers offered by Digital Equipment Corporation. The languages implemented by these two compilers are slightly different. The other FORTRAN compiler is called FORTRAN-10 and is described in the DECsystem-10 FORTRAN-10 Language Manual (DEC-10-LFORA-C-D). The FORTRAN-10 Language Manual contains a complete description of the FOROTS object time system which is used by both compilers.

INTRODUCTION TO THE FORTRAN SYSTEM

The FORTRAN compiler translates source programs written in the FORTRAN language into the machine language of the DECsystem-10. This translated version of the FORTRAN program exists as a retrievable, relocatable binary file on some storage device. All relocatable binary filenames have the extension .REL if they reside on a directory-oriented device (disk or DECtape).

In order for the FORTRAN program to be processed, LINK-10 must load the relocatable binary file into core memory. Also loaded are any relocatable binary files found in the FORTRAN library (FORLIB) which are necessary for the program's execution. Within the FORTRAN source program, the library files may be called explicitly, such as SIN, in the statement

```
X=SIN(Y)
```

or implicitly, such as FLOUT., the floating-point to ASCII conversion routine, which is implied in the following statements.

```
        PRINT    3,X
3       FORMAT(1X,F4.2)
```

A FORTRAN main program and its FORTRAN and/or MACRO-10 subprograms may be compiled or assembled separately and then linked together by LINK-10 at load time. The core image may then be saved on a storage device. When saved on a directory storage device, these files have the extension .SAV.

The monitor acts as the interface between the user and the computer so that all users are protected from one another and appear to have system resources available to themselves. Several user programs are loaded into core at once and the monitor schedules each program to run for a certain length of time. The monitor directs data flow between I/O devices and user programs, making the programs device independent, and overlaps I/O operations concurrently with computations.

In a multiprogramming system, all jobs reside in core and the scheduler decides which of these jobs should run. In a swapping system, jobs can exist on an external storage device (usually disk) as well as in core. The scheduler decides not only which job is to run but also when a job is to be swapped out onto the disk or brought back into core.

The number of users that can be handled by a given size time-sharing configuration is further increased by using the reentrant user-programming capability. This means that a sequence of instructions may be entered by more than one user job at a time. Therefore, a single copy of a reentrant program may be shared by a number of users at the same time to increase system economy. The FORTRAN compiler and operating system are both reentrant.

SECTION I

The DECsystem-10 FORTRAN IV (F40) Language

The seven chapters of this section deal with the DECsystem 10 FORTRAN IV (F40) language. Included in these chapters are the language elements of FORTRAN and the six categories of FORTRAN statements (arithmetic, control, input/output, specification, subprogram, and file control).

CHAPTER 1

INTRODUCTION TO THE FORTRAN LANGUAGE

The term FORTRAN (FORmula TRANslation) is used interchangeably to designate both the FORTRAN language and the FORTRAN translator or compiler. The FORTRAN language is composed of mathematical-form statements constructed in accordance with precisely formulated rules. FORTRAN programs consist of meaningful sequences of FORTRAN statements intended to direct the computer to perform the specified operations and computations.

The FORTRAN compiler is itself a computer program that examines FORTRAN statements and tells the computer how to translate the statements into machine language. The compiler runs in a minimum of 12K of core. The program written in FORTRAN language is called the source program. The resultant machine language program is called the object program.

1.1 LINE FORMAT

Each line of a FORTRAN program consists of three fields: statement number field, line continuation field, and statement field. A typical FORTRAN program is shown in Figure 1-1.

1.1.1 Statement Number Field

A statement number consists of from one to five digits in columns 1-5. Leading zeros and all blanks in this field are ignored. Statement numbers may be in any order and must be unique. Any statement referenced by another statement must have a statement number. For source programs prepared on a teletypewriter, a horizontal tab may be used to skip to the statement field with from 0 through 5 characters in the label field. This is the only place a tab is not treated as a space.

1.1.2 Line Continuation Field

If a FORTRAN statement is so large that it cannot conveniently fit into one statement field, the statement fields of up to 19 additional lines may be used to specify the complete statement. Any line which is not continued, or the first line of a sequence of continued lines, must have a blank or zero in column 6. Continuation lines must have a character other than blank or zero in column 6. If a continuation line is desired when a TAB is used in the statement number field, a digit from 1 to 9 must immediately follow the TAB.

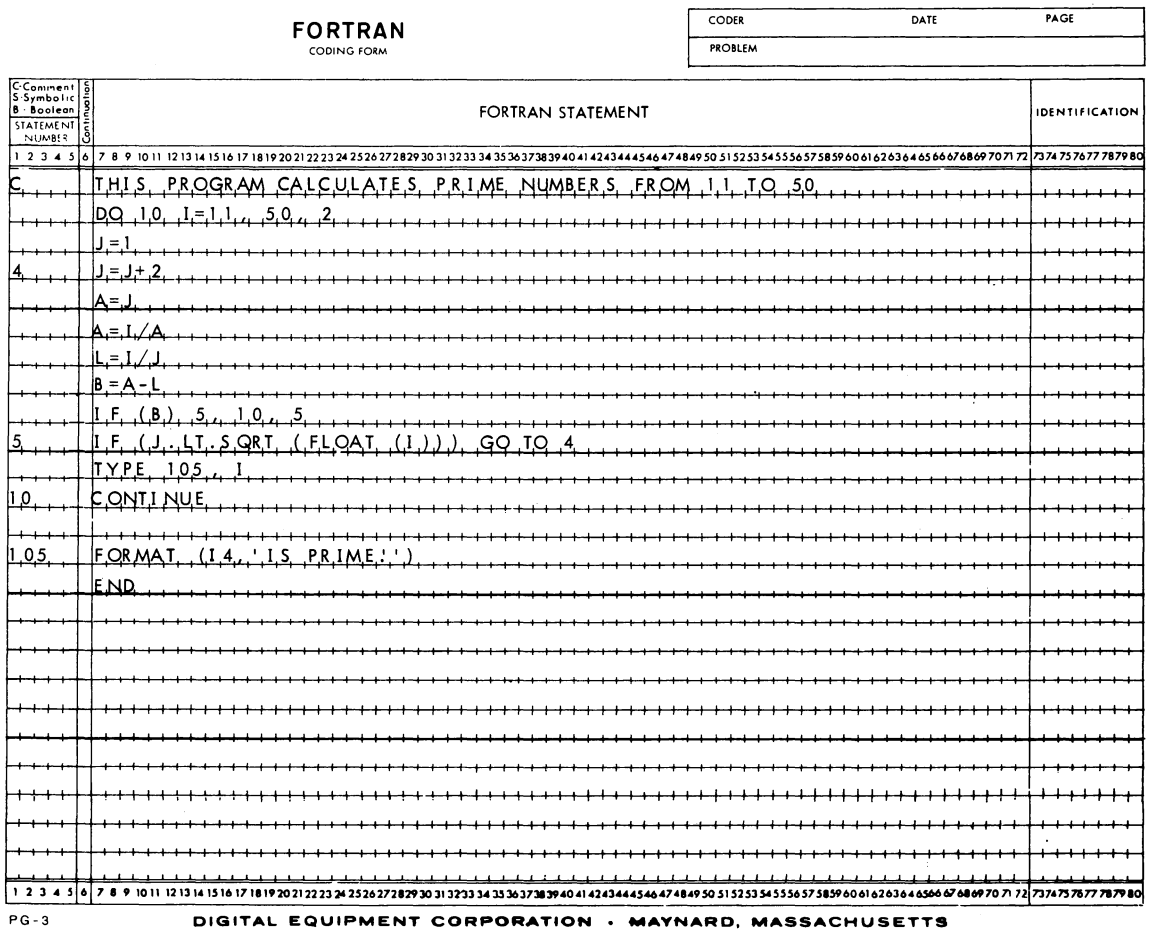


Figure 1-1 Typical FORTRAN Coding Form

1.1.3 Statement Field

Any FORTRAN statement, as described in later sections, may appear in the statement field (columns 7-72). Except for alphanumeric data within a FORMAT statement, DATA statement, or literal constant, blanks (spaces) and TABS are ignored and may be used freely for appearance purposes. Thus the following statements are equivalent.

END (tab) FILE (tab) 2
END (space) FILE (space) 2
ENDFILE2

1.1.4 Comment Line

Any line that starts with one of the characters $ * / or the letter C in column 1 is interpreted as a line of comments. Comment lines are printed onto any listings requested but are otherwise ignored by the compiler. Columns 2-72 may be used in any format for comment purposes. A comment line must not immediately precede a continuation line.

As an aid for program debugging, the letter D in column 1 causes the line to be interpreted as a comment unless the /I switch appears in the command string. (Refer to Table 12-1 for Compiler Switch options.) If the /I switch is present, the letter D in column 1 is interpreted as a space and the line is compiled as a program statement.

1.2 CHARACTER SET

The following characters are used in the FORTRAN language:

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

NOTE

ASCII characters greater than Z (132_8) are replaced by the error character "↑". See Chapter 13 for the internal representation of these characters.

CHAPTER 2

CONSTANTS, VARIABLES, AND EXPRESSIONS

The rules for defining constants and variables and for forming expressions are described in this chapter.

2.1 CONSTANTS

Seven types of constants are permitted in a FORTRAN source program: integer or fixed point, real or single precision floating point, double precision floating point, octal, complex, logical, and literal.

2.1.1 Integer Constants

An integer constant consists of from one to eleven decimal digits written without a decimal point. A negative constant must be preceded by a minus sign. A positive constant may be preceded by a plus sign.

Examples:
```
3
+10
−528
8085
```

An integer constant must fall within the range $-2^{35}+1$ to $2^{35}-1$. When used for the value of a subscript, the value of the integer constant is taken as modulo 2^{18}.

2.1.2 Real Constants

Real constants are written as a string of decimal digits including a decimal point. A real constant may consist of any number of digits but only the leftmost 9 digits appear in the compiled program. Real constants may be given a decimal scale factor by appending an E followed by a signed integer constant. The field following the letter E must not be blank, but may be zero.

Examples:
```
15.
 0.0
  .579
−10.794
 5.0E(i.e., 5000.)
 5.0E+3(i.e., 5000)
 5.0E−3(i.e., 0.005)
```

A real constant has precision to eight digits. The magnitude must lie approximately within the range 0.14×10^{-38} to 1.7×10^{38}. Real constants occupy one word of DECsystem-10 storage.

2.1.3 Double Precision Constants

A double precision constant is specified by a string of decimal digits, including a decimal point, which are followed by the letter D and a signed decimal scale factor. The field following the letter D must not be blank, but may be zero.

Examples: 24.671325982134D0
3.6D2 (i.e., 360.)
3.6D−2 (i.e., .036)
3.0D0

Double precision constants have precision to 16 digits. The magnitude of a double precision constant must lie approximately between 0.14×10^{-38} and 1.7×10^{38}. Double precision constants occupy two words of DECsystem-10 storage.

2.1.4 Octal Constants

A number preceded by a double quote represents an octal constant. An octal constant may appear in an arithmetic or logical expression or a DATA statement. Only the digits 0-7 may be used and only the last twelve digits are significant. A minus sign may precede the octal number, in which case the number is negated. A maximum of 12 octal digits are stored in each 36-bit word.

Examples: "7777
"−31563

2.1.5 Complex Constants

FORTRAN provides for direct operations on complex numbers. Complex constants are written as an ordered pair of real constants separated by a comma and enclosed in parentheses.

Examples: (.70712, −.70712)
(8.763E3,2.297)

The first constant of the pair represents the real part of the complex number, and the second constant represents the imaginary part. The real and imaginary parts may each be signed. The enclosing parentheses are part of the constant and always appear, regardless of context. Each part is internally represented by one single precision floating point word. They occupy consecutive locations of DECsystem-10 storage.

FORTRAN arithmetic operations on complex numbers, unlike normal arithmetic operations, must be of the form:

$$A \pm B = a_1 \pm b_1 + i(a_2 \pm b_2)$$

$$A*B = (a_1 b_1 - a_2 b_2) + i(a_2 b_1 a_1 b_2)$$

$$A/B = \frac{(a_1 b_1 + a_2 b_2) + i(a_2 b_1 - a_1 b_2)}{b_1^2 + b_2^2} \quad \frac{}{b_1^2 + b_2^2}$$

where $A = a_1 + ia_2$, $B = b_1 + ib_2$, and $i = \sqrt{-1.}$

2.1.6 Logical Constants

The two logical constants, .TRUE. and .FALSE., have the internal values −1 and 0, respectively. The enclosing periods are part of the constant and always appear.

Logical constants may be entered in DATA or input statements as signed octal integers (−1 and 0). Logical quantities may be operated on in either arithmetic or logical statements. Only the sign is tested to determine the truth value of a logical variable.

2.1.7 Literal Constants

A literal constant may be in either of two forms

A. A string of alphanumeric and/or special characters enclosed in single quotes; two adjacent single quotes within the constant are treated as one single quote.

B. A string of characters in the form

$$nHx_1x_2...x_n$$

where $x_1x_2...x_n$ is the literal constant, and n is the number of characters following the H.

Literal constants may be entered in DATA statements or input statements as a string of up to five 7-bit ASCII characters per variable (10 characters if the variable is double precision or complex). Literal constants may be operated on in either arithmetic or logical statements.

NOTE

Literal constants used as subprogram arguments will have a zero word as an end-of-string indicator.

Examples: CALL SUB ('LITERAL CONSTANT')
'DON"T'
5HDON'T
A = 'FIVE' + 42
B = (5HABCDE.AND."376)/2

2.2 VARIABLES

A variable is a quantity whose value may change during the execution of a program. Variables are specified by name and type. The name of a variable consists of one or more alphanumeric characters, the first one of which must be alphabetic. Only the first six characters are interpreted as defining the variable name. The type of variable (integer, real, logical, double precision, or complex) may be specified explicitly by a type declaration statement or implicitly by the IMPLICIT statement. If the variable is not specified in this manner, then a first letter of I, J, K, L, M or N indicates a fixed point (integer) variable; any other first letter indicates a floating point (real) variable. Variables of any type may be either scalar or array variables. When used in a subscript or as an index to a DO Statement, the value of the integer variable is taken as modulo 2^{18}.

2.2.1 Scalar Variables

A scalar variable represents a single quantity.

Examples: A
G2
POPULATION

2.2.2 Array Variables

An array variable represents a single element of an n dimensional array of quantities. The variable is denoted by the array name followed by a subscript list enclosed in parentheses. The subscript list is a sequence of integer expressions, separated by commas. The expressions may be of any form or type providing they are explicitly changed to type integer when each is completely evaluated. Each expression represents a subscript, and the values of the expressions determine the array element referred to. For example, the row vector A_i would be represented by the subscripted variable A(J), and the element, in the second column of the first row of the square matrix A, would be represented by A(1,2). Arrays may have any number of dimensions.

Examples: Y(1)
STATION (K)
A (3* K+2, I, J−1)

The three arrays above (Y, STATION, and A) would have to be dimensioned by a DIMENSION, COMMON, or type declaration statement prior to their first appearance in an executable statement or in a DATA or NAMELIST statement. (Array dimensioning is discussed in Chapter 6).

1-Dimensional Array A(10)

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

CONSECUTIVE STORAGE LOCATIONS

2-Dimensional Array B(5,5)

1	B(1,1)	6	B(1,2)	11	B(1,3)	16	B(1,4)	21	B(1,5)
2	B(2,1)	7	B(2,2)	12	B(2,3)	17	B(2,4)	22	B(2,5)
3	B(3,1)	8	B(3,2)	13	B(3,3)	18	B(3,4)	23	B(3,5)
4	B(4,1)	9	B(4,2)	14	B(4,3)	19	B(4,4)	24	B(4,5)
5	B(5,1)	10	B(5,2)	15	B(5,3)	20	B(5,4)	25	B(5,5)

B(3,1) IS THE THIRD STORAGE WORD IN SEQUENCE
B(3,4) IS THE EIGHTEENTH STORAGE WORD IN SEQUENCE

3-Dimensional Array C(5,5,5)

												101	C(1,1,5)	106	C(1,2,5)	111	C(1,3,5)	116	C(1,4,5)	121	C(1,5,5)
												102	C(2,1,5)	107	C(2,2,5)	112	C(2,3,5)	117	C(2,4,5)	122	C(2,5,5)
						76	C(1,1,4)	81	C(1,2,4)	86	C(1,3,4)	91	C(1,4,4)	96	C(1,5,4)	118	C(3,4,5)	123	C(3,5,5)		
						77	C(2,1,4)	82	C(2,2,4)	87	C(2,3,4)	92	C(2,4,4)	97	C(2,5,4)	119	C(4,4,5)	124	C(4,5,5)		
				51	C(1,1,3)	56	C(1,2,3)	61	C(1,3,3)	66	C(1,4,3)	71	C(1,5,3)	98	C(3,5,4)	120	C(5,4,5)	125	C(5,5,5)		
				52	C(2,1,3)	57	C(2,2,3)	62	C(2,3,3)	67	C(2,4,3)	72	C(2,5,3)	99	C(4,5,4)						
		26	C(1,1,2)	31	C(1,2,2)	36	C(1,3,2)	41	C(1,4,2)	46	C(1,5,2)	73	C(3,5,3)	100	C(5,5,4)						
		27	C(2,1,2)	32	C(2,2,2)	37	C(2,3,2)	42	C(2,4,2)	47	C(2,5,2)	74	C(4,5,3)								
1	C(1,1,1)	6	C(1,2,1)	11	C(1,3,1)	16	C(1,4,1)	21	C(1,5,1)	48	C(3,5,2)	75	C(5,5,3)								
2	C(2,1,1)	7	C(2,2,1)	12	C(2,3,1)	17	C(2,4,1)	22	C(2,5,1)	49	C(4,5,2)										
3	C(3,1,1)	8	C(3,2,1)	13	C(3,3,1)	18	C(3,4,1)	23	C(3,5,1)	50	C(5,5,2)										
4	C(4,1,1)	9	C(4,2,1)	14	C(4,3,1)	19	C(4,4,1)	24	C(4,5,1)												
5	C(5,1,1)	10	C(5,2,1)	15	C(5,3,1)	20	C(5,4,1)	25	C(5,5,1)												

C(1,3,2) is the 36th storage word in sequence.

C(1,1,5) is the 101st storage word in sequence.

Figure 2-1
Array Storage

Arrays are stored in increasing storage locations with the first subscript varying most rapidly and the last subscript varying least rapidly. For example, the 2-dimensional array B(I, J) is stored in the following order: B(1,1), B(2,1), . . . , B(I,1), B(1,2), B(2,2), . . . , B(I,2), . . . , B(I,J).

2.3 EXPRESSIONS

Expressions may be either numeric or logical. To evaluate an expression, the object program performs the calculations specified by the quantities and operators within the expression.

2.3.1 Numeric Expressions

A numeric expression is a sequence of constants, variables, and function references separated by numeric operators and parentheses in accordance with mathematical convention and the rules given below.

The numeric operators are +, −, *, /, **, denoting, respectively, addition, subtraction, multiplication, division, and exponentiation.

In addition to the basic numeric operators, function references are also provided to facilitate the evaluation of functions such as sine, cosine, and square root. A function is a subprogram which acts upon one or more quantities, called arguments, to produce a single quantity called the function value. Function references are denoted by the identifier, which names the function (such as SIN, COS, etc.), followed by an argument list enclosed in parentheses:

```
identifier(argument, argument, . . . , argument)
```

At least one argument must be present. An argument may be an expression, an array identifier, a subprogram identifier, or an alphanumeric string.

Function type is given by the type of the identifier which names the function. The type of the function is independent of the types of its arguments. (See Chapter 7, Section 7.4.1.1.)

A numeric expression may consist of a single element (constant, variable, or function reference):

```
2.71828
Z(N)
TAN(THETA)
```

Compound numeric expressions may be formed by using numeric operations to combine basic elements:

```
X+3.
TOTAL/A
TAN(PI*M)
(X+3.) −(TOTAL/A) *TAN (PI*M)
```

Compound numeric expressions must be constructed according to the following rules:

A. With respect to the numeric operators +, −, *, /, any type of quantity (logical, octal, integer, real, double precision, complex or literal) may be combined with any other, with one exception: a complex quantity cannot be combined with a double precision quantity.

   The resultant type of the combination of any two types may be found in Table 2-1. The conversions between data types will occur as follows:

   (1) A literal constant will be combined with any integer constant as an integer and with a real or double word as a real or double word quantity. (Double word refers to both double precision and complex.)

   (2) An integer quantity (constant, variable, or function reference) combined with a real or double word quantity results in an expression of the type real or double word respectively; e.g., an integer variable plus a complex variable will result in a complex subexpression. The integer is converted to floating point and then added to the real part of the complex number. The imaginary part is unchanged.

   (3) A real quantity (constant, variable, or function reference) combined with a double word quantity results in an expression that is of the same type as the double word quantity.

   (4) A logical or octal quantity is combined with an integer, real, or double word quantity as if it were an integer quantity in the integer case, or a real quantity in the real or double word case (i.e., no conversion takes place).

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

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

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

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

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

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

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

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

X*−Y

is improper. Use of parentheses yields the correct form:

X*(−Y)

By use of the foregoing rules, all permissible numeric expressions may be formed. As an example of a typical numeric expression using numeric operators and a function reference, the expression for one of the roots of the general quadratic equation:

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

would be coded as:

(−B+SQRT(B**2−4.*A *C))/(2.*A)

Table 2-1
Types of Resultant Subexpressions

+, −, *, /		Type of Quantity				
		Real	Integer	Complex	Double Precision	Logical, Octal, or Literal
Type of Quantity	Real	Real	Real	Complex	Double Precision	Real
	Integer	Real	Integer	Complex	Double Precision	Integer
	Complex	Complex	Complex	Complex	Not Allowed	Complex
	Double Precision	Double Precision	Double Precision	Not Allowed	Double Precision	Double Precision
	Logical, Octal, or Literal	Real	Integer	Complex	Double Precision	Logical, Octal, or Literal

2.3.2 Logical Expressions

A logical expression consists of constants, variables, function references, and arithmetic expressions, separated by logical operators or relational operators. Logical expressions are provided in FORTRAN to permit the implementation of various forms of symbolic logic. Logical masks may be represented by using octal constants. The result of a logical expression has the logical value TRUE (negative) or FALSE (positive or zero) and therefore, only uses one word.

2.3.2.1 Logical Operators - The logical operators, which include the enclosing periods and their definitions, are as follows, where P and Q are expressions:

.NOT.P	Has the value .TRUE. only if P is .FALSE., and has the value .FALSE. only if P is .TRUE.
P.AND.Q	Has the value .TRUE. only if P and Q are both .TRUE., and has the value .FALSE. if either P or Q is .FALSE.
P.OR.Q	(Inclusive OR) Has the value .TRUE. if either P or Q is .TRUE., and has the value .FALSE. only if both P and Q are .FALSE.
P.XOR.Q	(Exclusive OR) Has the value .TRUE. if either P or Q but not both are .TRUE., and has the value .FALSE. otherwise.
P.EQV.Q	(Equivalence) Has the value .TRUE. if P and Q are both .TRUE. or both .FALSE., and has the value .FALSE. otherwise.

Logical expressions are evaluated by combining the full word values of P and Q (only the high-order part if P and Q are double precision, only the real part if P and Q are complex) using the appropriate logical operator. The result is .TRUE. if it is arithmetically negative and FALSE if it is arithmetically positive or zero.

Logical operators may be used to form new variables, for example,

```
X = Y.AND.Z
E = E.XOR."400000000000
```

2.3.2.2 Relational Operators - The relational operators are as follows:

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

The enclosing periods are part of the operator and must be present.

Mixed expressions involving integer, real, and double precision types may be combined with relationals. The value of such an expression will be .TRUE. (-1) or .FALSE. (0).

The relational operators .EQ. and .NE. may also be used with COMPLEX expressions. (Double word quantities are equal if the corresponding parts are equal.)

A logical expression may consist of a single element (constant, variable, function reference, or relation):

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

Single elements may be combined through use of logical operators to form compound logical expressions, such as:

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

Any logical expression may be enclosed in parentheses and regarded as an element:

```
(T.XOR.S).AND.(R.EQV.Q)
CALL PARITY ((2.GT.Y.OR.X.GE.Y).AND.NEVER)
```

Any logical expression may be preceded by the unary operator .NOT. as in:

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

No two logical operators may appear in sequence, except in the case where .NOT. appears as the second of two logical operators, as in the example above. Two decimal points may appear in sequence, as in the example above, or when one belongs to an operator and the other to a constant.

When the precedence of operators is not given explicitly by parentheses, it is understood to be as follows (in order of decreasing precedence):

```
**
*,/
+,-
.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.
.NOT.
.AND.
.OR.
.EQV., .XOR.
```

For example, the logical expression:

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

is interpreted as

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

CHAPTER 3

THE ARITHMETIC STATEMENT

3.1 GENERAL DESCRIPTION

One of the key features of FORTRAN is the ease with which arithmetic computations can be coded. Computations to be performed by FORTRAN are indicated by arithmetic statements, which have the general form:

```
A=B
```

where A is a variable, B is an expression, and = is a replacement operator. The arithmetic statement causes the FORTRAN object program to evaluate the expression B and assign the resultant value to the variable A. Note that the = sign signifies replacement, not equality. Thus, expressions of the form:

```
A=A+B and
A=A*B
```

are quite meaningful and indicate that the value of the variable A is to be replaced by the result of the expression to the right of the = sign.

```
Examples:   Y=1*Y
            P=.TRUE.
            X(N)=N*ZETA(ALPHA*M/PI)+(1.,-1.)
```

Table 3-1 indicates which type of expression may be equated to each type of variable in an arithmetic statement. D indicates that the assignment is performed directly (no conversion of any sort is done); R indicates that only the real part of the variable is set to the value of the expression (the imaginary part is set to zero); C means that the expression is converted to the type of the variable; and H means that only the high-order portion of the evaluated expression is assigned to the variable.

The expression value is made to agree in type with the assignment variable before replacement occurs. For example, in the statement:

```
THETA=W*(ABETA+E)
```

if THETA is an integer and the expression is real, the expression value is truncated to an integer before assignment to THETA.

Table 3-1
Allowed Assignment Statements

Variable	Expression				
	Real	Integer	Complex	Double Precision	Logical, Octal,or Literal Constant
Real	D	C	R,D	H,D	D
Integer	C	D	R,C	H,C	D
Complex	D,R,I	C,R,I	D	H,D,R,I	D,R,I
Double Precision	D,H,L,	C,H,L	R,D,H,L	D	D,H,L
Logical	D	D	R,D	H,D	D

D - Direct Replacement

C - Conversion between integer and floating point

R - Real only

I - Set imaginary part to 0

H - High order only

L - Set low order part to 0

CHAPTER 4

CONTROL STATEMENTS

FORTRAN compiled programs normally execute statements sequentially in the order in which they were presented to the compiler. However, the following control statements are available to alter the normal sequence of statement execution: GO TO, IF, DO, PAUSE, STOP, END, CALL, RETURN. CALL and RETURN are used to enter and return from subroutines.

4.1 GO TO STATEMENT

The GO TO statement has three forms: unconditional, computed, and assigned.

4.1.1 Unconditional GO TO Statements

Unconditional GO TO statements are of the form:

GO TO n

where n is the number of an executable statement. Control is transferred to the statement numbered n. An unconditional GO TO statement may appear anywhere in the source program, except as the terminal statement of a DO loop.

4.1.2 Computed GO TO Statements

Computed GO TO statements have the form:

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

where n_1,n_2, . . . ,n_k are statement numbers, and i is an integer expression.

This statement transfers control to the statement numbered n_1,n_2, . . . ,n_k if i has the value 1, 2, . . . ,k, respectively. If i exceeds the size of the list of statement numbers or is less than one, execution will proceed to the next executable statement. Any number of statement numbers may appear in the list. There is no restriction on other uses for the integer variable i in the program.

In the example

GO TO (20,10,5),K

the variable K acts as a switch, causing a transfer to statement 20 if K=1, to statement 10 if K=2, or to statement 5 if K=3.

A computed GO TO statement may appear anywhere in the source program, except as the terminal statement of a DO loop.

4.1.3 Assigned GO TO Statement

Assigned GO TO statements have two equivalent forms:

GO TO k

and

GO TO k, (n_1, n_2, n_3, ...)

where k is a variable or array element and n_1, n_2, ...,n_k are statement numbers. Any number of statement numbers may appear in the list. Both forms of the assigned GO TO have the effect of transferring control to the statement whose number is currently associated with the variable k. The second form of the assigned GO TO statement passes control to the next executable statement if k is not associated with one of the statement numbers in the list. This association is established through the use of the ASSIGN statement, the general form of which is:

ASSIGN i TO k

where i is a statement number and k is a variable or array element. If more than one ASSIGN statement refers to the same integer variable name, the value assigned by the last executed statement is the current value.

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

An assigned GO TO statement may appear anywhere in the source program, except as the terminal statement of a DO loop.

4.2 IF STATEMENT

IF statements have two forms in FORTRAN: numerical and logical.

4.2.1 Numerical IF Statements

Numerical IF statements are of the form:

IF (expression) n_1,n_2,n_3

where n_1,n_2,n_3 are statement numbers. This statement transfers control to the statement numbered n_1,n_2,n_3 if the value of the numeric expression is less than, equal to, or greater than zero, respectively. All three statement numbers must be present. The expression may not be complex.

Examples:
```
IF (ETA) 4,7,12
IF (KAPPA-L (10)) 20, 14, 14
```

4.2.2 Logical IF Statements

Logical IF statements have the form:

IF (expression)S

where S is a complete statement. The expression must be logical. S may be any executable statement other than a DO statement or another logical IF statement (see Chapter 2, Section 2.3.2). If the value of the expression is .FALSE. (positive or zero), control passes to the next sequential statement. If value of the expression is .TRUE. (negative), statement S is executed. After execution of S, control passes to the next sequential statement unless S is a numerical IF statement or a GO TO statement; in these cases, control is transferred as indicated. If the expression is .TRUE. (negative) and S is a CALL statement, control is transferred to the next sequential statement upon return from the subroutine.

Numbers are present in the logical expression:

```
IF (B)Y=X*SIN(Z)
W=Y**2
```

If the value of B is .TRUE., the statements Y=X*SIN(Z) and W=Y**2 are executed in that order. If the value of B is .FALSE., the statement Y=X*SIN(Z) is not executed.

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

NOTE

Care should be taken in testing floating point numbers for equality in IF statements as rounding may cause unexpected results.

4.3 DO STATEMENT

The DO statement simplifies the coding of iterative procedures. DO statements are of the form:

DO n i=m_1,m_2,m_3

where n is a statement number, i is a nonsubscripted integer variable, and m_1,m_2,m_3 are any integer expressions. If m_3 is not specified, it is understood to be 1.

The DO statement causes the statements which follow, up to and including the statement numbered n, to be executed repeatedly. This group of statements is called the range of the DO statement. The integer variable i of the DO statement is called the index. The values of m_1,m_2, and m_3 are called, respectively, the initial, limit, and increment values of the index.

A zero increment (m_3) is not allowed. The increment m_3 may be negative if $m_1 \geq m_2$. If $m_1 \leq m_2$, the increment m_3 must be positive. The index variable can assume legal values only if $(m_2 - m_i) * m_3 \geq 0$. (m_i is the current value of the index variable m_1.)

Examples: Form	Restriction
DO 10 I=1,5,2	
DO 10 I=5,1,-1	
DO 10 I=J,K,5	J≤K
DO 10 I=J,K,-5	J≥K
DO 10 L=I,J,-K	I≤J,K<0 or I≥J,K>0
DO 10 L=I,J,K	I≤J,K>0 or I≥J,K<0

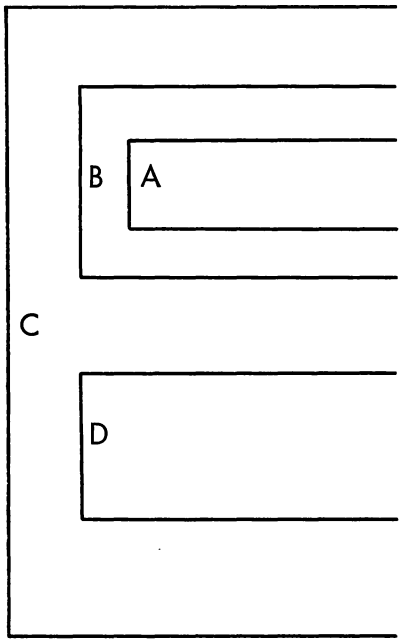
Initially, the statements of the range are executed with the initial value assigned to the index. This initial execution is always performed, regardless of the values of the limit and increment. After each execution of the range, the increment value is added to the value of the index and the result is compared with the limit value. If the value of the index is not greater than the limit value, the range is executed again using the new value of the index. When the increment value is negative, another execution will be performed if the new value of the index is not less than the limit value.

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

The range of a DO statement may include other DO statements, provided that the range of each contained DO statement is entirely within the range of the containing DO statement. When one DO loop is completely contained in another, it is said to be nested. DO loops can be nested to any depth. A transfer into the range of a DO statement from outside the range is not allowed.

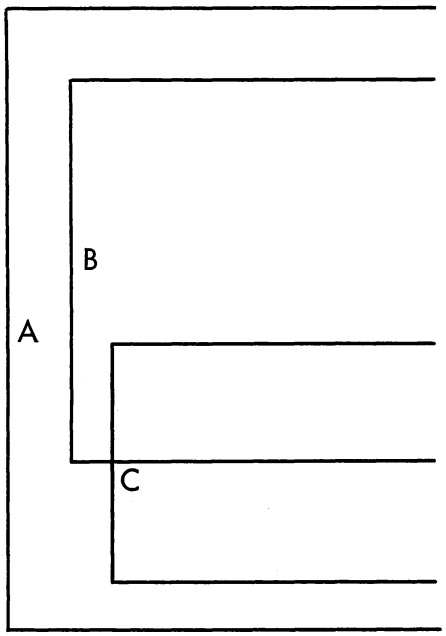
More than one DO loop within a nest of DO loops can end on the same statement. This terminal statement is considered to belong to the innermost DO loop that ends on the terminal statement. The statement label of such a terminal statement cannot be used in any GO TO or arithmetic IF statements except those that occur within the DO loop to which the terminal statement belongs.

Valid DO Loop Nest



Control must not pass from within loop A or loop B into loop D, or from loop D into loop A or loop B.

Invalid DO Loop Nest



Loop C is not fully within the range of loop B even though it is within the range of loop A.

Figure 4-1
Nested DO Loops

Within the range of a DO statement, the index is available for use as an ordinary variable. After a transfer from within the range, the index retains its current value and is available for use as a variable. The value of the index variable becomes undefined when the DO loop it controls is satisfied. The values of the initial, limit, and increment variables for the index and the index of the DO loop, may not be altered within the range of the DO statement.

The range of a DO statement must not end with a GO TO type statement or a numerical IF statement. If an assigned GO TO statement is in the range of a DO loop, all the statements to which it may transfer must be either in the range of the DO loop or all must be outside the range. A logical IF statement is allowed as the last statement of the range. In this case, control is transferred as follows. The range is considered ended when, and if, control would normally pass to the statement following the entire logical IF statement.

As an example, consider the sequences:

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

Statement 5 is executed four times whether the statement Y(K) = X(K) is executed or not. Statement 6 is not executed until statement 5 has been executed four times.

Examples:
```
DO 22 L = 1,30
DO 45 K = 2,LIMIT,-3
DO 7 X = T,MAX,L
```

4.4 CONTINUE STATEMENT

The CONTINUE statement has the form:

```
CONTINUE
```

This statement is a dummy statement, used primarily as a target for transfers, particularly as the last statement in the range of a DO statement. For example, in the sequence:

```
DO 7 K = START,END

.
.
.

IF (X(K))22,13,7

.
.
.

7 CONTINUE
```

a positive value of X(K) begins another execution of the range. The CONTINUE provides a target address for the IF statement and ends the range of the DO statement.

4.5 PAUSE STATEMENT

The PAUSE statement enables the program to incorporate operator activity into the sequence of automatic events. The PAUSE statement assumes one of three forms:

```
PAUSE
PAUSE n
PAUSE 'xxxxx'
```

where n is an unsigned string of six or less octal digits, and 'xxxxx' is a literal message.

Execution of the PAUSE statement causes the message or the octal digits, if any, to be typed on the user's teletypewriter . Program execution may be resumed (at the next executable FORTRAN statement) from the console by typing "G", followed by a carriage return. Program execution may be terminated by typing "X," followed by a carriage return.

Example:
```
PAUSE 167
PAUSE 'NOW IS THE TIME'
```

4.6 STOP STATEMENT

The STOP statement has the forms:

```
STOP      or
STOP n
```

where n is an unsigned string of one to five octal digits.

The STOP statement terminates the program and returns control to the monitor system. (Termination of a program may also be accomplished by a CALL to the EXIT or DUMP subroutines.) Use of the STOP statement implies a call to the EXIT subroutine.

4.7 END STATEMENT

The END statement has the form:

```
END
```

The END statement informs the compiler to terminate compilation and must be the physically last statement of the program. The END statement implies a STOP statement in a main program or a RETURN statement in a subroutine or a function. The END statement is implied by an end-of-file.

CHAPTER 5

DATA TRANSMISSION STATEMENTS

Data transmission statements are used to control the transfer of data between computer memory and either peripheral devices or other locations in computer memory. These statements are also used to specify the format of the output data. Data transmission statements are divided into the following four categories.

A. Nonexecutable statements that enable conversions between internal form data within core memory and external form data (FORMAT), or specify lists of arrays and variables for input/output transfer (NAMELIST).

B. Statements that specify transmission of data between computer memory and I/O devices: READ, WRITE, PRINT, PUNCH, TYPE, ACCEPT.

C. Statements that control magnetic tape unit mechanisms: REWIND, BACKSPACE, END FILE, UNLOAD, SKIP RECORD.

D. Statements that specify transmission of data between series of locations in memory: ENCODE, DECODE.

5.1 NONEXECUTABLE STATEMENTS

The FORMAT statement enables the user to specify the form and arrangement of data on the selected external medium. The NAMELIST statement provides for conversion and input/output transmission of data without reference to a FORMAT statement.

5.1.1 FORMAT Statement

FORMAT statements may be used with any appropriate input/output medium or ENCODE/DECODE statement. FORMAT statements are of the form:

$$n \text{ FORMAT } (S_1,S_2,\ldots S_n / S_1^1 S_2^1,\ldots,S_n^1 / \ldots)$$

where n is a statement number, and each S is a data field specification.

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

Slashes are used to specify unit records, which must be one of the following:

A. A tape or disk record with a maximum length corresponding to a line buffer (135 ASCII characters).

B. A punched card with a maximum of 80 characters.

C. A printed line with a maximum of 72 characters for a terminal and either 120 or 132 characters for the line printer.

During transmission of data, the object program scans the designated FORMAT statement. If a specification for a numeric field is present (see Section 5.2.1 of this chapter) and the data transmission statement contains items remaining to be transmitted, transmission takes place according to the specifications. This process ceases and execution of the data transmission statement is terminated as soon as all specified items have been transmitted. Thus, the FORMAT statement may contain specifications for more items than are specified by the data transmission statement. Conversely, the FORMAT statement may contain specifications for fewer items than are specified by the data transmission statement.

The following types of field specifications may appear in a FORMAT statement: numeric, numeric with scale factors, logical, alphanumeric. The FORMAT statement also provides for handling multiple record formats, formats stored as data, carriage control, skipping characters, blank insertion, and repetition. If an input list requires more characters than the input device supplies for a given unit record, blanks are supplied.

5.1.1.1 Numeric Fields - Numeric field specification codes designate the type of conversion to be performed. These codes and the corresponding internal and external forms of the numbers are listed in Table 5-2.

The conversions are specified by the forms:

1. Dw.d
2. Ew.d
3. Fw.d
4. Iw
5. Ow
6. Gw.d (for real or double precision)
   Gw (for integer or logical)
   Gw.d,Gw.d (for complex)

respectively. The letter D, E, F, I, O, or G designates the conversion type; w is an integer specifying the field width, which may be greater than required to provide for blank columns between numbers; d is an integer specifying the number of decimal places to the right of the decimal point or, for G conversion, the number of significant digits. (For D, E, F, and G input, the position of the decimal point in the external field takes precedence over the value of d in the format.)

For example,

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

could be used to output the line,

bbb32bbbb—17.60bbb.5962547681D+03

on the output listing.

The G format is the general format code that is used to transmit real, double precision, integer, logical, or complex data. The rules for input depend on the type specification of the corresponding variable in the data list. The form of the output conversion also depends on the individual variable except in the case of real and double-precision data. In these cases the form of the output conversion is a function of the magnitude of the data being converted. The following table shows the magnitude of the external data, M, and the resulting method of conversion.

Table 5-1
Magnitude of Internal Data

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

The field width w should always be large enough to include spaces for the decimal point, sign, and exponent. In all numeric field conversions if w is not large enough to accommodate the converted number, the excess digits on the left will be lost; if the number is less than w spaces in length, the number is right-adjusted in the field.

Table 5-2
Numeric Field Codes

Conversion Code	Internal Form	External Form
D	Binary floating point double precision	Decimal floating point with D exponent
E	Binary floating point	Decimal floating point with E exponent
F	Binary floating point	Decimal fixed point
I	Binary integer	Decimal integer
O	Binary integer	Octal integer

Table 5-2 (Cont)
Numeric Field Codes

Conversion Code	Internal Form	External Form
G	One of the following: single precision binary floating point, binary integer, binary logical, or binary complex	Single precision decimal floating point integer, logical (T or F), or complex (two decimal floating point numbers), depending upon the internal form

5.1.1.2 Numeric Fields with Scale Factors - Scale factors may be specified for D, E, F, and G conversions. A scale factor is written nP where P is the identifying character and n is a signed or unsigned integer that specifies the scale factor.

For F type conversions (or G type, if the external field is decimal fixed point), the scale factor specifies a power of ten so that

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

For D, E, and G (external field not decimal fixed point) conversions, the scale factor multiplies the number by a power of ten, but the exponent is changed accordingly leaving the number unchanged except in form. For example, if the statement:

FORMAT (F8.3,E16.5)

corresponds to the line

bb26.451bbbb-0.41321E-01

then the statement

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

might correspond to the line

bbb2.645bbb-41.32157E-03

In input operations, F type (and G type, if the external field is decimal fixed point) conversions are the only types affected by scale factors.

When no scale factor is specified, it is understood to be zero. However, once a scale factor is specified, it holds for all subsequent D, E, F, and G type conversions within the same format unless another scale factor is encountered. The scale factor is reset to zero by specifying a scale factor of zero. Scale factors have no effect on I and O type conversions.

5.1.1.3 Logical Fields - Logical data can be transmitted in a manner similar to numeric data by use of the specification:

Lw

where L is the control character and w is an integer specifying the field width. The data is transmitted as the value of a logical variable in the input/output list.

If on input, the first nonblank character in the data field is T or F, the value of the logical variable will be stored as true or false, respectively. If the entire data field is blank or empty, a value of false will be stored.

On output, w minus 1 blanks followed by T or F will be output if the value of the logical variable is true or false, respectively.

5.1.1.4 Variable Field Width - The D, E, F, G, I, and O conversion types may appear in a FORMAT statement without the specification of the field width (w) or the number of places after the decimal point (d). In the case of input, omitting the w implies that the numeric field is delimited by any character which would otherwise be illegal in the field, in addition to the characters −, +, ., E, D, and blank provided they follow the numeric field. For example, input according to the format

10 FORMAT (2I, F, E, O)

might appear on the input medium as

−10,3/15.621−.0016E−10,777.

In this case, commas delimit the numeric fields, blanks may also be used as field delimiters. On output, omitting the w has the following effect:

Format	Becomes
D	D25.16
E	E15.7
F	F15.7
G	G15.7 or G25.16
I	I15
O	O15

5.1.1.5 Alphanumeric Fields - Alphanumeric data can be transmitted in a manner similar to numeric data by use of the form Aw, where A is the control character and w is the number of characters in the field. The alphanumeric characters are transmitted as the value of a variable in an input/output list. The variable may be of any type. For the sequence:

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

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

Although w may have any value, the number of characters transmitted is limited by the maximum number of characters which can be stored in the space allotted for the variable. This maximum depends upon the variable type. For a double precision variable the maximum is ten characters; for all other variables, the maximum is five characters. If w exceeds the maximum, the leftmost characters are lost on input and replaced with blanks on output. If, on input, w is less than the maximum, blanks are filled in to the right of the given characters until the maximum is reached. If, on output, w is less than the maximum, the leftmost w characters are transmitted to the external medium. Since for complex variables each word requires a separate field specification, the maximum value for w is 5. For example,

```
      COMPLEX C
      ACCEPT 1, C
    1 FORMAT (2A5)
```

could be used to transmit ten alphanumeric characters into complex variable C.

5.1.1.6 Alphanumeric Data Within Format Statements - Alphanumeric data may be transmitted directly into or from the format statement by two different methods: H-conversion, or the use of single quotes.

In H-conversion, the alphanumeric string is specified by the format nH. H is the control character and n is the number of characters in the string counting blanks. For example, the format in the statement below can be used to print PROGRAM COMPLETE on the output listing.

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

The statement

```
      FORMAT (16HPROGRAM COMPLETE)
```

causes ROGRAM COMPLETE to be printed.

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

The same effect is achieved by merely enclosing the alphanumeric data in quotes. The result is the same as in H-conversion; on input, the characters between the quotes are replaced by input characters, and, on output, the characters between the quotes (including blanks) are written as part of the output data. A quote character within the data is represented by two successive quote marks. For example, referring to:

```
      FORMAT (' DON''T')
```

with an output statement would cause DON'T to be printed. Referring to

```
FORMAT ('DON''T')
```

causes ON'T to be printed. The first character referenced by the FORMAT statement for output is interpreted as a carriage control character (see 5.1.1.13). TAB characters in FORMAT statements are converted to single blanks at runtime by the FORTRAN object time system.

5.1.1.7 Mixed Fields - An alphanumeric format field may be placed among other fields of the format. For example, the statement:

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

can be used to output the line:

```
bbb22bFORCE=bb17.68901
```

The separating comma may be omitted after an alphanumeric format field, as shown above.

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

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

could be used in the transmission of three complex quantities.

5.1.1.9 Repetition of Field Specifications - Repetition of a field specification may be specified by preceding the control character D, E, F, I, O, G, L, or A by an unsigned integer giving the number of repetitions desired. For example:

```
FORMAT (2E12. 4,3I5)
```

is equivalent to:

```
FORMAT (E12.4,E12.4,I5,I5,I5)
```

5.1.1.10 Repetition of Groups - A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number. For example:

```
FORMAT (2I8,2(E15.5,2F8.3))
```

is equivalent to:

FORMAT (2I8,E15.5,2F8.3,E15.5,2F8.3)

5.1.1.11 Multiple Record Formats - To handle a group of input/output records where different records have different field specifications, a slash is used to indicate a new record. For example, the statement:

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

is equivalent to

FORMAT (3O8)

for the first record and

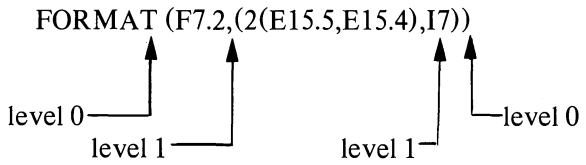
FORMAT (I5,2F8.4)

for the second record.

The separating comma may be omitted when a slash is used. When n slashes appear at the end or beginning of a format, n blank records may be written on output or records skipped on input. When n slashes appear in the middle of a format, n-1 blank records are written or n-1 records skipped.

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

Thus, the statement



causes the format

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

to be used on the first record, and the format

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

to be used on succeeding records.

As a further example, consider the statement

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

The first record has the format

```
F7.2
```

and successive records have the format

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

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

As an example, consider the sequence:

```
      DIMENSION SKELETON (2)
      READ 1, (SKELETON(I), I = 1,2)
    1 FORMAT (2A5)
      READ SKELETON,K,X
```

The first READ statement enters the ASCII string into the array SKELETON. In the second READ statement, SKELETON is referred to as the format governing conversion of K and X.

5.1.1.13 Carriage Control - The first character of each ASCII record controls the spacing of the line printer or terminal. This character is usually set by beginning a FORMAT statement for an ASCII record with 1Ha, where a is the desired control character. The line spacing actions, listed below, occur before printing:

FORTRAN Character	Printer Character	Octal Value	Effect	Printer Channel
space	LF	012	Skip to next line with form feed after 60 lines	8
0 zero	LF,LF	012	Skip a line	8
1 one	FF	014	Form feed - go to top of next page	1

FORTRAN Character	Printer Character	Octal Value	Effect	Printer Channel
+ plus			Suppress skipping - overprint the line	
* asterisk	DC3	023	Skip to next line with no form feed	5
- minus	LF,LF,LF	012	Skip two lines	8
2 two	DLE	020	Space 1/2 of a page	2
3 three	VT	013	Space 1/3 of a page	7
/ slash	DC4	024	Space 1/6 of a page	6
. period	DC2	022	Triple space with a form feed after every 20 lines printed	4
, comma	DC1	021	Double space with a form feed after every 30 lines printed	3

NOTE: Printer control characters DLE, DC1, DC2, DC3, and DC4 affect only the line printer.

A $ (dollar sign) as a format field specification code suppresses the carriage return at the end of the terminal or line printer line.

5.1.1.14 Spacing - Input and output can be made to begin at any position within a FORTRAN record by use of the format code

Tw

where T is the control character and w is an unsigned integer constant specifying the character position in a FORTRAN record where the transfer of data is to begin. When the output is printed, w corresponds to the (w-1)th print position. This is because the first character of the output buffer is a carriage control character and is not printed. It is recommended that the first field specification of the output format be 1x, except where a carriage control character is used.

For example,

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

would cause the following line to be printed

For input, the statements

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

cause the first 34 characters of the input data to be skipped, and the next 5 characters would replace the characters M, O, N, T, and H in storage. If an input record containing

```
ABCbbbXYZ
```

is read with the format specification

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

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

5.1.1.15 Blank or Skip Fields - Blanks may be introduced into an output record or characters skipped on an input record by use of the specification nX. The control character is X; n is the number of blanks or characters skipped and must be greater than zero. For example, the statement

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

may be used to output the line

```
bSTEPbbb28bbbbbbbbbbY=b-3.872
```

5.1.2 NAMELIST Statement

The NAMELIST statement, when used in conjunction with special forms of the READ and WRITE statements, provides a method for transmitting and converting data without using a FORMAT statement or an I/O list. The NAMELIST statement has the form

$$NAMELIST/X_1/A_1,A_2,\ldots,A_i/X_2/B_1,B_2,\ldots,B_i\ldots/X_m/C_1,C_2,\ldots C_n$$

where the X's are NAMELIST names, and the A's, B's, and C's are variable or array names.

Each list or variable mentioned in the NAMELIST statement is given the NAMELIST name immediately preceding the list. Thereafter, an I/O statement may refer to an entire list by mentioning its NAMELIST name. For example:

```
NAMELIST/FRED/A,B,C/MARTHA/D,E
```

states that A, B, and C belong to the NAMELIST name FRED, and D and E belong to MARTHA.

The use of NAMELIST statements must obey the following rules:

A. A NAMELIST name may not be longer than six characters; it must start with an alphabetic character; it must be enclosed in slashes; it must precede the list of entries to which it refers; and it must be unique within the program.

B. A NAMELIST name may be defined only once and must be defined by a NAMELIST statement. After a NAMELIST name has been defined, it may only appear in READ or WRITE statements. The NAMELIST name must be defined in advance of the READ or WRITE statement.

C. A variable used in a NAMELIST statement cannot be used as a dummy argument in a subroutine definition.

D. Any dimensioned variable contained in a NAMELIST statement must have been defined in a DIMENSION statement preceding the NAMELIST statement.

5.1.2.1 Input Data For NAMELIST Statements - When a READ statement refers to a NAMELIST name, the first character of all input records is ignored. Records are searched until one is found with a $ or & as the second character immediately followed by the NAMELIST name specified. Data is then converted and placed in memory until the end of a data group is signaled by a $ or & either in the same record as the NAMELIST name, or in any succeeding record as long as the $ or & is the second character of the record. Data items must be separated by commas and be of the following form:

$$V=K_1,K_2,\ldots,K_n$$

where V may be a variable name or an array name, with or without subscripts. The K's are constants which may be integer, real, double precision, complex (written as (A, B) where A and B are real), or logical (written as T for true and F for false). A series of J identical constants may be represented by J*K where J is an unsigned integer and K is the repeated constant. Logical and complex constants must be equated to logical and complex variables, respectively. The other types of constants (real, double precision, and integers) may be equated to any other type of variable (except logical or complex), and will be converted to the variable type. For example, assume A is a two-dimensional real array, B is a one-dimensional integer array, C is an integer variable, and that the input data is as follows:

```
$FRED A(7,2)=4, B=3,6*2.8, C=3.32$
↑
Column 2
```

A READ statement referring to the NAMELIST name FRED will result in the following: the integer 4 will be converted to floating point and placed in A(7,2). The integer 3 will be placed in B(1) and the floating point number 2.8 will be placed in B(2), B(3),..., B(7). The floating point number 3.32 will be converted to the integer 3 and placed in C.

5.1.2.2 Output Data For NAMELIST Statements - When a WRITE statement refers to a NAMELIST name, all variables and arrays and their values belonging to the NAMELIST name will be written out, each according to its type. The complete array is written out by columns. The output data will be written so that:

A. The fields for the data will be large enough to contain all the significant digits.

B. The output can be read by an input statement referencing the NAMELIST name.

For example, if JOE is a 2x3 array, the statements

```
NAMELIST/NAM1/JOE,K1,ALPHA
WRITE (u,NAM1)
```

generate the following form of output.

```
Column 2
↓
$NAM1
JOE = -6.75,        .234E-04,          68.0,
      -17.8,           0.0,        -.197E+07,
 K1 = 73.1,       ALPHA=3,$
```

5.2 DATA TRANSMISSION STATEMENTS

The data transmission statements accomplish input/output transfer of data that may be listed in a NAMELIST statement or defined in a FORMAT statement. When a FORMAT statement is used to specify formats, the data transmission statement must contain a list of the quantities to be transmitted. The data appears on the external media in the form of records.

5.2.1 Input/Output Lists

The list of an input/output statement specifies the order of transmission of the variable values. During input, the new values of listed variables may be used in subscript or control expressions for variables appearing later in the list. For example:

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

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

The transmission of array variables may be controlled by indexing similar to that used in the DO statement. The list of controlled variables, followed by the index control, is enclosed in parentheses. For example,

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

is equivalent to:

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

As in the DO statement, the initial, limit, and increment values may be given as integer expressions:

```
READ 5, N, (GAIN(K),K=1,M/2,N)
```

The indexing may be compounded as in the following:

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

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

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

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

```
READ 11, MASS
```

Entire arrays may also be designated for transmission by referring to a NAMELIST name (see description of NAMELIST statement).

5.2.2 Input/Output Records

All information appearing on external media is grouped into records.The maximum amount of information in one record and the manner of separation between records depends upon the medium. For punched cards, each card constitutes one record; on a terminal a record is one line, and so forth. The amount of information contained in each ASCII record is specified by the FORMAT reference and the I/O list. For magnetic tape binary records, the amount of information is specified by the I/O list.

Each execution of an input or output statement initiates the transmission of a new data record. Thus, the statement

```
READ 2, FIRST, SECOND, THIRD
```

is not necessarily equivalent to the statements

```
READ 2, FIRST
READ 2, SECOND
READ 2, THIRD
```

since, in the second case, at least three separate records are required, whereas, the single statement

```
READ 2, FIRST, SECOND, THIRD
```

may require one, two, three, or more records depending upon FORMAT statement 2.

If an input/output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another input/output statement without repositioning the record.

If an input/output list requires more than one ASCII record of information, successive records are read.

5.2.3 PRINT Statement

The PRINT statement assumes one of two forms:

```
PRINT f, list
PRINT f
```

where f is a format reference.

The data is converted from internal to external form according to the designated format. If the data to be transmitted is contained in the specified FORMAT statement, the second form of the statement is used.

```
Examples:  PRINT 16, T, (B(K), K=1, M)
           PRINT F106, SPEED, MISS
```

In the second example, the format is stored in array F106.

5.2.4 PUNCH Statement

The PUNCH statement assumes one of two forms:

```
PUNCH f, list
PUNCH f
```

where f is a format reference.

Conversion from internal to external data forms is specified by the format reference. If the data to be transmitted is contained in the designated FORMAT statement, the second form of the statement is used.

```
Examples:  PUNCH 12, A, B(A), C(B(A))
           PUNCH 7
```

5.2.5 TYPE Statement

The TYPE statement assumes one of two forms:

```
TYPE f, list
TYPE f
```

where f is a format reference.

This statement causes the values of the variables in the list to be read from memory and listed on the user's teletypewriter. The data is converted from internal to external form according to the designated format. If the data to be transmitted is contained in the designated FORMAT statement, the second form of the statement is used.

```
Examples:  TYPE 14, K, (A(L), L=1, K)
           TYPE FMT
```

5.2.6 WRITE Statement

The WRITE statement assumes one of the following forms:

```
WRITE (u,f) list
WRITE (u,f)
WRITE (u,N)
WRITE (u) list
WRITE (u#R,f) list
```

where u is a unit designation, f is a format reference, N is a NAMELIST name, and R is a record number where I/O is to start.

The first form of the WRITE statement causes the values of the variables in the list to be read from memory and written on the unit designated in ASCII form. The data is converted to external form as specified by the designated FORMAT statement.

The second form of the WRITE statement causes information to be read directly from the specified format and written on the unit designated in ASCII form.

The third form of the WRITE statement causes the names and values of all variables and arrays belonging to the NAMELIST name, N, to be read from memory and written on the unit designated. The data is converted to external form according to the type of each variable and array.

The fourth form of the WRITE statement causes the values of the variables in the list to be read from memory and written on the unit designated in binary form.

The fifth form of the WRITE statement causes the variables in the list to be written in the specified record of the file on the disk unit designated. Either a pound sign (#) or a single quote (') can be used to separate the unit and the record. This allows a programmer to access fixed-length records directly, and eliminates the sequential writing of data to access one or more records within

the file. The file must first be defined properly by an OPEN statement (see Chapter 8). Output begins when the random WRITE specifying the record to which the writing is desired is given in the correct format.

5.2.7 READ Statement

The READ statement assumes one of the following forms:

```
READ f, list
READ f
READ (u,f) list
READ (u,f)
READ (u,N)
READ (u) list
READ (u#R,f) list
READ (u,f,END=C, ERR=d) list
READ (u,f,END=C) list
READ (u,f,ERR=d) list
```

where f is a format reference, u is a unit designation, N is a NAMELIST name, R is a record number where I/O is to start, C is a statement number to which control is transferred upon encountering an end-of-file, and d is the statement number to which control is transferred upon encountering an error condition on the input data.

The first form of the READ statement causes information to be read from cards and put in memory as values of the variables in the list. The data is converted from external to internal form as specified by the referenced FORMAT statement.

Example: READ 28,Z1,Z2,Z3

The second form of the READ statement is used if the data read from cards is to be transmitted directly into the specified format.

Example: READ 10

The third form of the READ statement causes ASCII information to be read from the unit designated and stored in memory as values of the variables in the list. The data is converted to internal form as specified by the referenced FORMAT statement.

Example: READ (1,15)ETA,P1

The fourth form of the READ statement causes ASCII information to be read from the unit designated and transmitted directly into the specified format.

Example: READ (N,105)

The fifth form of the READ statement causes data of the form described in the discussion of input data for NAMELIST statements to be read from the unit designated and stored in memory as values of the variables or arrays specified.

Example: READ (2,FRED)

The sixth form of the READ statement causes binary information to be read from the unit designated and stored in memory as values of the variables in the list.

Example: READ (M)GAIN,Z,AI

The seventh form of the READ statement causes information to be read from the specified record in a disk file into the variables of the list. This allows random access of fixed-length records in a disk file. The file from which records are to be read is defined by the OPEN statement (see Chapter 8).

```
Example:  DOUBLE PRECISION FIL
          DIMENSION A(6)
          DATA FIL/'FILE.ONE'/
          OPEN (UNIT=4,ACCESS='RANDOM',FILE=FIL,DIRECTORY='11,23')
          READ (4#54,5)A
```

This example reads the 54th record from FILE.ONE on the disk area belonging to programmer [11,23] into the list variables A(1) through A(6).

The eighth form of the READ statement causes control to be transferred if an end-of-file or error condition is encountered on the input data. The arguments END=c and ERR=d are optional and if both are included, either may appear first. If an end-of-file is encountered, control transfers to the statement specified by END=c. If an END parameter is not specified, I/O on that device terminates and the program halts with an error message to the user's TTY. If an error on input is encountered, control transfers to the statement specified by ERR=d. If an ERR=d parameter is not specified, the program halts with an error message to the user's TTY.

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

5.2.8 REREAD Statement

The reread feature allows a FORTRAN program to reread information from the last used input file. The format used during the reread need not correspond to the original read format, and the information may be read as many times as desired.

A. To reread from an input device, the following coding would be used:

```
READ (16,100)A
      .
      .
      .
REREAD 105,A
```

The REREAD 105,A statement causes the last input device used to be reread according to format statement 105. The original read format and a subsequent reread format need not be the same.

B. The reread feature cannot be used until an input from a file has been accomplished. If the feature is used prematurely, an error message will be generated.

C. Information may be reread as many times as desired using either the same or a new format statement each time.

D. The reread feature must be used with some forethought and care since it rereads from the last input file used, i.e.:

The following example will reread from the file on Device No. 10, not Device No. 16:

```
READ (16,100)A
      .
      .
      .
READ (10,200)B
      .
      .
      .
REREAD 110,A
```

5.2.9 ACCEPT Statement

The ACCEPT statement assumes one of two forms:

```
ACCEPT f, list
ACCEPT f
```

where f is a format reference.

This statement causes information to be input from the user's teletypewriter and put in memory as values of the variables in the list. The data is converted to internal form as specified by the format. If the transmission of data is directly into the designated format, the second form of the statement is used.

```
Examples: ACCEPT 12,ALPHA,BETA
          ACCEPT 27
```

5.3 DEVICE CONTROL STATEMENTS

Device control statements and their corresponding effects are listed in Table 5-3.

Table 5-3
Device Control Statements

Statement	Effect
BACKSPACE u	Backspaces designated tape one ASCII record or one logical binary record.
END FILE u	Writes an end-of-file.
REWIND u	Rewinds tape on designated unit.
SKIP RECORD u	Causes skipping of one ASCII record or one logical binary record.
UNLOAD u	Rewinds and unloads the designated tape.

5.4 ENCODE AND DECODE STATEMENTS

ENCODE and DECODE statements transfer data, according to format specifications, from one section of user's core to another. No peripheral equipment is involved. DECODE is used to change data in ASCII format to data in another format. ENCODE changes data of another format into data in ASCII format.

The two statements are of the form

```
ENCODE(c,f,v) L(1),...,L(N)
DECODE(c,f,v) L(1),...,L(N)
```

where

c = the number of ASCII characters
f = the format statement number or array name
v = the starting address of the ASCII record referenced
L(1),...,L(N) = the list of variables.

A slash cannot appear in the FORMAT statement referenced by an ENCODE or DECODE statement.

Example: Assume the contents of the variables to be as follows:

A(1) contains the floating-point binary number 300.45
A(2) contains the floating-point binary number 3.0
J contains the binary integer value 1.
B is a four-word array of indeterminate contents
C contains the ASCII string 12345

```
      DO 2 J = 1,2
      ENCODE (16,10,B) J, A(J)
10    FORMAT (1X,2HA(,I1,4H) = ,F8.2)
      TYPE 11,B
11    FORMAT (4A5)
 2    CONTINUE
      DECODE (5, 12, C) B
12    FORMAT (3F1.0, 1X, F1.0)
      TYPE 13,B
13    FORMAT (4F5.2)
      END
```

Array B can contain 20 ASCII characters. The result of the ENCODE statement after the first iteration of the DO loop is:

B(1)	A(1)
B(2)	=
B(3)	300.4
B(4)	5

Typed as

A(1) = 300.45

The result after the second iteration is:

B(1)	A(2)
B(2)	=
B(3)	3.0
B(4)	

Typed as

A(2) = 3.0

The result of the DECODE statement is to extract the digits 1, 2, and 3 from C, convert them to floating point binary values, and store them in B(1), B(2), and B(3) and then skip the next character (4), extract the digit 5 from C, convert it to a floating point binary value, and store it in B(4).

CHAPTER 6

SPECIFICATION STATEMENTS

Specification statements allocate storage and furnish information about variables and constants to the compiler. Specification statements may be divided into three categories, as follows:

A. Storage specification statements: DIMENSION, COMMON, and EQUIVALENCE.

B. Data specification statements: DATA and BLOCK DATA.

C. Type declaration statements: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, SUBSCRIPT INTEGER, and IMPLICIT.

By extending the USA Standard in regard to specification statements, DECsystem-10 FORTRAN allows the following statements to be used anywhere in the program, provided that the variables they specify appear in executable statements only after the particular specification statement. The specification statement must not appear in the range of a DO loop.

DIMENSION statement
EXTERNAL statement (described in Chapter 7)
COMMON statement
EQUIVALENCE statement
Type declaration statements
DATA statement

A sample program that incorporates these statements follows.

```
      DOUBLE PRECISION D
      DIMENSION Y(10), D(5)
      Y(1) = -1.0
      INTEGER XX(5)
      Y(2) = ABS(Y(1))
      DATA XX/1,2,3,4,5,/
      DO 10I = 3,7
10    Y(I) = XX(I-2)
      COMMON Z
      Z=Y(1)*Y(2)/(Y(3) + Y(5))
      END
```

Only IMPLICIT statements and arithmetic function definition statements (described in Chapter 7) must appear in the program before any executable statement.

In addition, arrays must be dimensional before being referenced in a NAMELIST, EQUIVALENCE, or DATA statement. DOUBLE PRECISION and COMPLEX arrays must be declared before they are dimensioned.

6.1 STORAGE SPECIFICATION STATEMENTS

6.1.1 DIMENSION Statement

The DIMENSION statement is used to declare identifiers to be array identifiers and to specify the number and bounds of the array subscripts. The information supplied in a DIMENSION statement is required for the allocation of memory for arrays. Any number of arrays may be declared in a single DIMENSION statement. The DIMENSION statement has the form

DIMENSION $S_1,S_2,\ldots,S_n$

where S is an array specification.

Each array variable appearing in the program must represent an element of an array declared in a DIMENSION statement, unless the dimension information is given in a COMMON or TYPE statement. Dimension information may appear only once for a given variable.

Each array specification gives the array identifier and the minimum and maximum values which each of its subscripts may assume in the following form:

identifier(min/max, min/max, . . . ,min/max)

The minima and maxima must be integers. The minimum must not exceed the maximum. For example, the statement

DIMENSION EDGE(-1/1,4/8)

specifies EDGE to be a two-dimensional array whose first subscript may vary from -1 to 1 inclusive, and the second from 4 to 8 inclusive.

Minimum values of 1 may be omitted. For example,

NET(5, 10)

is interpreted as:

NET(1/5,1/10)

Examples: DIMENSION FORCE(-1/1,0/3,2,2,-7/3)
DIMENSION PLACE(3,3,3),JI(2,2/4),K(256)

Arrays may also be declared in the COMMON or type declaration statements in the same way:

```
COMMON X(10,4),Y,Z
INTEGER A(7,32),B
DOUBLE PRECISION K(-2/6,10)
```

6.1.1.1 Adjustable Dimensions - Within either a FUNCTION or SUBROUTINE subprogram, DIMENSION and TYPE statements may use integer variables in an array specification, provided that the array name and variable dimensions are dummy arguments of the subprogram. The actual array name and values for the dummy variables are given by the calling program when the subprogram is called. The variable dimensions may not be altered within the subprogram (i.e., typing the array DOUBLE PRECISION or COMPLEX after it has been dimensioned) and must be less than or equal to the explicit dimensions declared in the calling program.

Example:

```
      SUBROUTINE SBR(ARRAY, M1,M2,M3,M4)
      DIMENSION  ARRAY (M1/M2,M3/M4)
                 .
                 .
                 .
                 .
      DO 27 L=M3,M4
      DO 27 K=M1,M2
                 .
                 .
                 .
                 .
27    ARRAY(K,L)=VALUE
                 .
                 .
                 .
      END
```

The calling program for SBR might be:

```
DIMENSION  A1(10,20),A2(1000,4)
           .
           .
           .
           .
CALL SBR(A1,5,10,10,20)
           .
           .
           .
           .
```

```
CALL SBR(A2,100,250,2,4)
      .
      .
      .
      .
      .
END
```

6.1.2 COMMON Statement

The COMMON statement causes specified variables or arrays to be stored in an area available to other programs. By means of COMMON statements, the data of a main program and/or the data of its subprograms may share a common storage area.

The common area may be divided into separate blocks which are identified by block names. A block is specified as follows:

```
/block identifier/identifier, identifier, . . ,identifier
```

The identifier enclosed in slashes is the block name. The identifiers which follow are the names of the variables or arrays assigned to the block and are placed in the block in the order in which they appear in the block specification. A common block may have the same name as a variable in the same program.

The COMMON statement has the general form

```
COMMON/BLOCK1/A,B,C/BLOCK2/D,E,F/. . .
```

where BLOCK1, BLOCK2, . . . are the block names, and A, B, C, . . . are the variables to be assigned to each block. For example, the statement

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

indicates that the elements X, Y, and T are to be placed in block R in that order, and that U, V, W, and Z are to be placed in block C.

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

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

have the same effect as the statement

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

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

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

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

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

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

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

as its first COMMON statement, and a subprogram has

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

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

Common blocks may be any length provided that no program attempts to enlarge a given common block declared by a previously loaded program.

Array names appearing in COMMON statements may have dimension information appended if the arrays are not declared in DIMENSION or type declaration statements. For example,

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

specifies the dimensions of the array T while entering T in blank common. Variable dimension array identifiers may not appear in a COMMON statement, nor may other dummy identifiers. Each array name appearing in a COMMON statement must be dimensioned somewhere in the program containing the COMMON statement.

6.1.3 EQUIVALENCE Statement

The EQUIVALENCE statement causes more than one variable within a given program to share the same storage location. The EQUIVALENCE statement has the form

$$\text{EQUIVALENCE}(V_1,V_2,\ldots),(V_k,V_{k+1},\ldots),\ldots$$

where the V's are variable names.

The inclusion of two or more references in a parenthetical list indicates that the quantities in the list are to share the same memory location. For example,

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

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

The relation of equivalence is transitive; e.g., the two statements,

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

have the same effect.

The subscripts of array variables must be integer constants.

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

6.1.4 EQUIVALENCE and COMMON

Identifiers may appear in both COMMON and EQUIVALENCE statements provided the following rules are observed.

A. No two quantities in common may be set equivalent to one another.

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

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

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

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

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

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

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

6.2 DATA SPECIFICATION STATEMENTS

The DATA statement is used to specify initial or constant values for variables. The specified values are compiled into the object program, and become the values assumed by the variables when program execution begins.

6.2.1 DATA Statement

The data to be compiled into the object program is specified in a DATA statement. The DATA statement has the form

DATA list/d_1,d_2,../ ,list/d_k,d_{k+1},.../ ,...

where each list is in the same form as an input/output list, and the d's are data items for each list.

Indexing may be used in a list provided the initial, limit, and increment (if any) are given as constants. Expressions used as subscripts must have the form

c_1*i$\pm c_2$

where c_1 and c_2 are integer constants and i is the induction variable. If an entire array is to be defined, only the array identifier need be listed. Variables in COMMON may appear on the lists only if the DATA statement occurs in a BLOCK DATA subprogram. (See Chapter 7, Section 7.6)

The data items following each list correspond one-to-one with the variables of the list. Each item of the data specifies the value given to its corresponding variable with no implied type conversion. Thus, integer variables can only be defined numerically by integer constants, real variables by real constants, double precision variables by double precision constants, and so forth. Refer to Section 2.1 for definitions of the various constants. Data items may be numeric constants, alphanumeric strings, octal constants, or logical constants. For example,

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

specifies the value .5 for ALPHA and the value .16 for BETA.

Alphanumeric data is packed into words according to the data word size in the manner of A conversion; however, excess characters are not permitted. The specification is written as nH followed by n characters or is imbedded in single quotes. Double precision variables must have at least six characters assigned to them in DATA statements.

Octal data is specified by the letter O or the character ", followed by a signed or unsigned octal integer of one to twelve digits.

Logical constants are written as .TRUE., .FALSE., T, or F. For example:

```
DATA NOTE,K/4HFOOT, O-7712/
DATA QUOTE/'QUOTE'/
```

Any item of the data may be preceded by an integer followed by an asterisk. The integer indicates the number of times the item is to be repeated. For example,

```
DATA(A(K),K=1,20)/61E2,  19*32E1/
```

specifies 20 values for the array A; the value 6100 for A(1); the value 320 for A(2) through A(20). To cause an array or part of an array to be initialized to blanks, the blank areas must be specified explicitly in the DATA statement. For example,

```
DATA(A(I),I=1,10)/'12345',9*' '/
```

causes the first word of A to contain 12345 in ASCII and the next nine words of the array to be blank.

6.2.2 BLOCK DATA Statement

The BLOCK DATA statement has the form:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram. Data may be entered into labeled or blank common.

The first statement of the subprogram must be the BLOCK DATA statement. The subprogram may contain only the declarative statements associated with the data being defined.

Example:

```
BLOCK DATA
COMMON/R/S,Y/C/Z,W,V
DIMENSION Y(3)
DOUBLE PRECISION X
COMPLEX Z
DATA Y/1E-1,2*3E2/,X,Z/11.877D0,(-1.41421,1.41421)/
END
```

Data may be entered into more than one block of common in one subprogram.

6.3 TYPE DECLARATION STATEMENTS

The type declaration statements INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, IMPLICIT, and SUBSCRIPT INTEGER are used to specify the type of identifiers appearing in a program. An identifier may appear in only one type statement. Type statements may be used to give dimension specifications for arrays.

The explicit type declaration statements have the general form

```
type identifier,identifier,identifier...
```

where type is one of the following:

```
INTEGER,REAL,DOUBLE PRECISION,COMPLEX,LOGICAL,
SUBSCRIPT INTEGER
```

In addition, for the sake of compatibility the following types have been made equivalent:

```
INTEGER is equivalent to INTEGER*4 and SUBSCRIPT INTEGER
REAL is equivalent to REAL*4
```

DOUBLE PRECISION is equivalent to REAL*8
LOGICAL is equivalent to LOGICAL*1 and LOGICAL*4
COMPLEX is equivalent to COMPLEX*8

The listed identifiers are declared by the statement to be of the stated type. Fixed point variables in a SUBSCRIPT INTEGER statement must fall between -2^{27} and 2^{27}.

6.3.1 IMPLICIT Statement

The IMPLICIT statement has the form

IMPLICIT $type_1(a_1,a_2,\ldots),\ldots,type_2(a_3,a_4,\ldots)$

where type represents INTEGER, REAL, LOGICAL, COMPLEX, DOUBLE PRECISION, or one of the equivalent types listed in Section 6.3 (except SUBSCRIPT INTEGER) and $a_1,a_2,\ldots$ represent single alphabetic characters, each separated by commas, or a range of characters (in alphabetic sequence) denoted by the first and last characters of the range separated by a minus sign (e.g., (A-D)).

This statement causes any program variable which is not mentioned in a type statement, and whose first character is one of those listed in the IMPLICIT statement, to be classified according to the type appearing before the list in which the character appears. As an example, the statement

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

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

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

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

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

A. Identifiers beginning with I, J, K, L, M, or N are assigned integer type.

B. Identifiers not assigned integer type are assigned real type.

If the program contains an IMPLICIT statement, this statement will override throughout the program the implicit state initially set by the compiler. No program may contain more than one IMPLICIT declaration for the same letter.

CHAPTER 7

SUBPROGRAM STATEMENTS

FORTRAN subprograms may be either internal or external. Internal subprograms are defined and may be used only within the program containing the definition. The arithmetic function definition statement is used to define internal functions.

External subprograms are defined separately from (i.e., external to) the programs that call them, and are complete programs which conform to all the rules of FORTRAN programs. They are compiled as closed subroutines; i.e., they appear only once in the object program regardless of the number of times they are used. External subprograms are defined by means of the statements FUNCTION and SUBROUTINE.

7.1 DUMMY IDENTIFIERS

Subprogram definition statements contain dummy identifiers, representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram definition and indicate the sort of arguments that may appear and how the arguments are used. The dummy identifiers are replaced by the actual arguments when the subprogram is executed.

7.2 LIBRARY SUBPROGRAMS

The standard FORTRAN library for the DECsystem-10 includes built-in functions, FUNCTION subprograms, and SUBROUTINE subprograms, listed and described in Chapter 10. Built-in functions are open subroutines; that is, they are incorporated into the object program each time they are referred to by the source program. FUNCTION and SUBROUTINE subprograms are closed subroutines; their names derive from the types of subprogram statements used to define them.

7.3 ARITHMETIC FUNCTION DEFINITION STATEMENT

The arithmetic function definition statement has the form:

```
identifier(identifier,identifier,...)=expression
```

This statement defines an internal subprogram. The entire definition is contained in the single statement. The first identifier is the name of the subprogram being defined.

Arithmetic function subprograms are single-valued functions with at least one argument. The type of the function is determined by the type of the function identifier.

The identifiers enclosed in parentheses represent the arguments of the function. These are dummy identifiers; they may appear only as scalar variables in the defining expression. Dummy identifiers have meaning and must be unique only within the defining statement. Dummy identifiers must agree in order, number, and type with the actual arguments given at execution time.

Identifiers, appearing in the defining expression, which do not represent arguments are treated as ordinary variables. The defining expression may include external functions or other previously defined arithmetic statement functions.

All arithmetic function definition statements must precede the first executable statement of the program.

Examples: SSQR(K)=K*(K+1)*(2*K+ 1)/6
ACOSH(X)=(EXP(X/A)+EXP(−X/A))/2

In the last example above, X is a dummy identifier and A is an ordinary identifier. At execution time, the function is evaluated using the current value of the quantity represented by A.

7.4 FUNCTION SUBPROGRAMS

A FUNCTION subprogram is a single-valued function that may be called by using its name as a function name in an arithmetic expression, such as FUNC(N), where FUNC is the name of the subprogram that evaluates the corresponding function of the argument N. A FUNCTION subprogram begins with a FUNCTION statement and ends with an END statement. It returns control to the calling program by means of one or more RETURN statements.

7.4.1 FUNCTION Statement

The FUNCTION statement has the form:

FUNCTION identifier(argument,argument,. . .)

This statement declares the program which follows to be a FUNCTION subprogram. The identifier is the name of the function being defined. This identifier must not be used as a dummy argument or appear in any nonexecutable statement in the program other than as a scalar variable in a TYPE statement. It must appear as a scalar variable and be assigned a value during execution of the subprogram which is the function value.

Arguments appearing in the list enclosed in parentheses are dummy arguments representing the function argument. The arguments must agree in number, order, and type with the actual arguments used in the calling program. FUNCTION subprogram arguments may be expressions, alphanumeric strings, array names, statement labels preceded by an asterisk (*) or dollar sign ($), or subprogram names.

Dummy arguments may appear in the subprogram as scalar identifiers, array identifiers, subprogram identifiers, or an asterisk (*) or dollar sign ($), denoting statement labels in the calling program. A function must have at least one dummy argument. Dummy arguments representing array names must appear within the subprogram in a DIMENSION statement, or one of the type statements that provide dimension information. Dimensions given as constants must equal the dimensions of the corresponding arrays in the calling program. In a DIMENSION statement, dummy identifiers may be used to specify adjustable dimensions for array name arguments. For example, in the statement sequence:

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

The dimensions of array A are specified by the dummies M and N, while the dimension of array B is given as a constant. The various values given for M and N by the calling program must be those of the actual arrays which the dummy A represents. The arrays may each be of different size but must have two dimensions. The arrays are dimensioned in the programs that use the function.

Dummy dimensions may be given only for dummy arrays. In the example above the array C must be given absolute dimensions, since C is not a dummy identifier. A dummy identifier may not appear in an EQUIVALENCE statement in the FUNCTION subprogram.

A function must not modify any arguments which appear in the FORTRAN arithmetic expression calling the function. Modification of implicit arguments from the calling program, such as variables in COMMON and DO loop indexes, is not allowed. The only FORTRAN statements not allowed in a FUNCTION subprogram are SUBROUTINE, BLOCK DATA, and another FUNCTION statement.

7.4.1.1 Function Type — The type of the function is the type of identifier used to name the function. This identifier may be typed, implicitly or explicitly, in the same way as any other identifier. Alternatively, the function may be explicitly typed in the FUNCTION statement itself by preceding the word FUNCTION with one of the types or equivalent types described in Section 6.3. For example:

```
INTEGER FUNCTION
REAL FUNCTION
COMPLEX FUNCTION
LOGICAL FUNCTION
DOUBLE PRECISION FUNCTION
REAL*8 FUNCTION
```

Thus, the statement

```
COMPLEX FUNCTION HPRIME(S,N)
```

is equivalent to the statements

```
FUNCTION HPRIME(S,N)
COMPLEX HPRIME
```

Examples:

```
FUNCTION MAY(RANGE, EP, YP, ZP)
COMPLEX FUNCTION COT(ARG)
DOUBLE PRECISION FUNCTION LIMIT(X,Y)
FUNCTION WORK (A,$,C)
```

7.5 SUBROUTINE SUBPROGRAMS

A SUBROUTINE subprogram may be multivalued and can be referred to only by a CALL statement. A SUBROUTINE subprogram begins with a SUBROUTINE statement and returns control to the calling program by means of one or more RETURN statements.

7.5.1 SUBROUTINE Statement

The SUBROUTINE statement has the form:

```
SUBROUTINE identifier(argument,argument,...)
```

This statement declares the program which follows to be a SUBROUTINE subprogram. The first identifier is the subroutine name. This identifier cannot be used as a dummy argument or appear in any nonexecutable statement in the program other than as a scalar variable in a TYPE statement. The subroutine name can, however, be used as a scalar variable in any executable statement in the program. The arguments in the list enclosed in parentheses are dummy arguments representing the arguments of the subprogram. The dummy arguments must agree in number, order, and type with the actual arguments used by the calling program.

SUBROUTINE subprograms may have expressions, alphanumeric strings, array names, statement labels, and subprogram names as arguments. The dummy arguments may appear as scalar, array, subprogram identifiers, or an asterisk (*) or dollar sign ($) denoting a statement label in the calling program. Dummy arguments representing statement labels can be used only in connection with the RETURN statement.

Dummy identifiers which represent array names must be dimensioned within the subprogram by a DIMENSION or type declaration statement. As in the case of a FUNCTION subprogram, either constants or dummy identifiers may be used to specify dimensions in a DIMENSION statement. The dummy arguments must not appear in an EQUIVALENCE or COMMON statement in the SUBROUTINE subprogram.

A SUBROUTINE subprogram may use one or more of its dummy identifiers to represent results. The subprogram name is not used for the return of results. A SUBROUTINE subprogram need not have any argument at all.

```
Examples:   SUBROUTINE FACTOR(COEFF,N,ROOTS)
            SUBROUTINE RESIDU(NUM,N,DEN,M,RES)
            SUBROUTINE SERIES
            SUBROUTINE TYPE(A,$,B,*)
```

The only FORTRAN statements not allowed in a function subprogram are FUNCTION, BLOCK DATA, and another SUBROUTINE statement.

7.5.2 CALL Statement

The CALL statement assumes one of two forms:

```
CALL identifier
CALL identifier (argument,argument,. . .,argument)
```

The CALL statement is used to transfer control to SUBROUTINE subprogram. The identifier is the subprogram name.

The arguments may be expressions, array identifiers, alphanumeric strings, subprogram identifiers, or statement labels of the calling program preceded by an asterisk (*), dollar sign ($), or ampersand (&). Arguments may be of any type, but must agree in number, order, type, and array size (except for adjustable arrays, as discussed under the DIMENSION statement) with the corresponding arguments in the SUBROUTINE statement of the called subroutine. Unlike a function, a subroutine may produce more than one value and cannot be referred to as a basic element in an expression.

A subroutine may use one or more of its arguments to return results to the calling program. If no arguments at all are required, the first form is used.

```
Examples:   CALL EXIT
            CALL SWITCH(SIN,2.LE.BETA,X**4,Y)
            CALL TEST(VALUE,123,275)
            CALL TYPE(A,$10,B,*20,&30)
```

The identifier used to name the subroutine is not assigned a type and has no relation to the types of the arguments. Arguments which are constants or formed as expressions must not be modified by the subroutine.

7.5.3 RETURN Statement

The RETURN statement has one of two forms:

```
RETURN
RETURN i
```

where i is an integer constant or an integer variable. The value of i must be positive, and specifies that the return is to the i -th argument of the referencing statement (where the i -th argument is a statement number preceded by a $ or *). If i=0, the return is the same as with the first form of the RETURN statement.

This statement returns control from a subprogram to the calling program. Normally, the last statement executed in a subprogram is a RETURN statement. Any number of RETURN statements may appear in a subprogram. For purposes of debugging functions and subroutines originally written as main programs, the RETURN statement has been made equivalent to the STOP statement in a main program.

7.6 BLOCK DATA SUBPROGRAMS

A BLOCK DATA subprogram is a data specification subprogram and is used to enter initial values into variables in COMMON for use by FORTRAN subprograms and MACRO-10 main programs (see Chapter 9). No executable statements may appear in a BLOCK DATA subprogram.

7.6.1 BLOCK DATA Statement

The BLOCK DATA statement has the form:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram and it must be the first statement of the subprogram (see Chapter 6, Section 6.2.2).

7.7 EXTERNAL STATEMENT

FUNCTION and SUBROUTINE subprogram names may be used as the actual arguments of subprograms. Such subprogram names must be distinguished from ordinary variables by their appearance in an EXTERNAL statement. The EXTERNAL statement has the form:

```
EXTERNAL identifier, identifier, . . .,identifier
```

This statement declares the listed identifiers to be subprogram names. Any subprogram name given as an argument to another subprogram must have previously appeared in an external declaration in the calling program (i.e., as an identifier in an EXTERNAL or CALL statement or as a function name in an expression).

Example:

```
EXTERNAL SIN, COS
        .
        .

CALL TRIGF(SIN, 1.5,ANSWER)
        .
        .
        .
CALL TRIGF(COS,.87,ANSWER)
        .
        .
        .
END


SUBROUTINE TRIGF(FUNC,ARG,ANSWER)
        .
        .
        .
ANSWER = FUNC(ARG)
        .
        .
        .
RETURN
END
```

To reference external variables from a MACRO-10 program by name, place the variables in named COMMON. Use the name of the variable as the name of the COMMON block:

```
COMMON /A/A/B/B(13)/C/C(6,7)
```

CHAPTER 8

FILE CONTROL STATEMENTS

File control statements are used to set up files and establish parameters for I/O operations and to terminate I/O operations.

The OPEN and CLOSE file control statements are described in this chapter.

8.1 OPEN AND CLOSE STATEMENTS

Both the OPEN and CLOSE statements are unique to DECsystem-10 FORTRAN. They both use the same format and have the same options and arguments.

The OPEN statement enables the user to explicitly define all of the important aspects of each desired data transfer operation. It provides an extensive list of required and optional arguments which define in detail:

A. the name and location of the data file
B. the type of access required
C. the data format within the file
D. the protection code[1] to be assigned to an output data file
E. the disposition of the data file
F. data file record, block and file sizes
G. a data file version identifier
H. error modes

In addition, a DIALOG argument is provided which permits the user to establish a dialog mode of operation when the OPEN statement containing it is executed. In a dialog mode, interactive communication between the user's terminal and program is established. This enables the user, during program execution, to define, redefine, or defer the values of the optional arguments contained by the current OPEN statement.

The OPEN statement has the general form:

```
OPEN(argument,argument,...)
```

[1]Refer to Chapter 6 of the DECsystem-10 Monitor Calls Manual (DEC-10-MRRD-D) for a description of file access protection codes.

The CLOSE statement is used in the termination of an I/O operation to dissociate the I/O device being used from the active file and file-related information, and to restore the core occupied by I/O buffers and other transfer-related operations. All required device dependent termination functions are also performed on the execution of a CLOSE statement, including release of the unit.

Once a CLOSE statement has been executed, another OPEN statement is required to regain access to the closed file.

The CLOSE statement has the general form:

CLOSE(argument,argument,...)

8.1.1 Options for OPEN and CLOSE Statements

The options and their arguments which may be used in both OPEN and CLOSE statements are listed below.

A. UNIT — This option is required. It defines the FORTRAN I/O unit number to be used. FORTRAN devices are identified by assigned decimal numbers within the range 1 to 63 (unless redefined by the installation). UNIT may, however, be assigned an integer variable or constant. This argument has the form:

UNIT= An integer variable or constant

NOTE
FORTRAN logical device assignments are described in Section 13.2.1 (Table 13-3).

B. DEVICE — This option may specify either the physical or logical name of the I/O device involved. (A logical name always takes precedence over a physical name.) The DEVICE argument may specify I/O devices located at remote stations, as well as logical devices. If this option is omitted, the first logical name u (where u is the decimal unit number) is tried. If this is not successful, the standard (default) device is attempted. The DEVICE argument has the form:

DEVICE= A literal constant or variable

C. ACCESS — ACCESS describes the type of input and/or output statements and the file access mode to be used in a specified data transfer operation. ACCESS may be assigned any one of the following six names, each of which indicates a specific type of I/O operation.

1. 'SEQIN' — The specified data file is to be read in sequential access mode.

2. 'SEQOUT' — The specified data file is to be written in a sequential access mode.

3. 'SEQINOUT' — The specified data file may first be read then written (READ/WRITE sequence) record-by-record in a sequential access mode. When SEQINOUT is specified, a WRITE/READ sequence is illegal unless the file has been removed.

4. 'RANDOM' — The specified data file may be either read or written, one record at a time. In a random access mode of operation, the relative position of each record is independent of the previous READ or WRITE statement; all records accessed must have a fixed logical record length. This argument is required for random access operations. A disk device must be specified when RANDOM is used.

5. 'RANDIN' — This argument enables the user to establish a special, read-only random access mode with a named file. In a RANDIN mode, the user may read the named file simultaneously with other users who have also established a RANDIN mode and with the owner of the file. The use of RANDIN enables a data base to be shared by more than one user at a time.

6. 'APPEND' — The record specified by a corresponding WRITE statement is to be added to the logical end of a named file. The modified file must then be closed and reopened in order to be read.

The ACCESS argument has the form:

```
                'SEQIN'
                'SEQOUT'
                'SEQINOUT'
ACCESS=         'RANDOM'
                'RANDIN'
                'APPEND'
                variable (set to literal)
```

D. MODE

This option defines the character set of an external file or record. Use of this argument is optional. If it is not given, one of the following is assumed:

ASCII for a formatted I/O file transfer

Binary for an unformatted I/O file transfer

One of the following character set specifications must be used with the MODE argument:

'ASCII' Specifies an ASCII character set.

'BINARY' Specifies data formatted as a FORTRAN binary data file.

'IMAGE' Specifies an image (I) mode data transfer for the associated READ or WRITE statements. IMAGE is an unformatted binary mode.

'DUMP' The data file to be transferred is to be handled in a DUMP mode of operation.

NOTE

Refer to the DECsystem-10 Monitor Calls Manual for a detailed description of these data modes.

The MODE argument has the form:

MODE= 'ASCII'
'BINARY'
'IMAGE'
'DUMP'
variable (set to literal)

E. DISPOSE

This option specifies an action to be taken regarding a file at close time. When used, DISPOSE must be either an ASCII variable or one of the following literals:

'SAVE' Leave the file on the device.

'DELETE' If the device involved is either a DECtape or disk, remove the file. Otherwise, take no action.

'PRINT' If the file is on the disk, queue it for printing. Otherwise, take no action.

'LIST' Same as "PRINT", but file is deleted after printing.

'PUNCH' Paper tape punch output.

'RENAME' Change filename.

If the DISPOSE argument is not given, the argument DISPOSE='SAVE' is assumed. The DISPOSE argument has the form:

DISPOSE= 'SAVE'
'DELETE'
'PRINT'
'LIST'
'PUNCH'
'RENAME'
variable (set to literal)

F. FILE

This option specifies the name of the file involved in the data transfer operation. FILE must be either an ASCII literal or a double precision, complex, or single precision variable. Single precision variables are assumed to contain a 1 to 5 character file specification; double precision variables permit 10-character file specifications. The format is a 1 to 6 character filename optionally followed by a period and a 0 to 3 character extension. Any excess characters in either the name or extension are ignored. If the period and extension are omitted, the extension .DAT is assumed. If just the extension is omitted, the null extension is assumed.

If a file name is not specified or is zero, a default name is generated which has the form

FORxx.DAT

where xx is the FORTRAN logical unit number (decimal) or is the logical unit name for the default statements ACCEPT, PRINT, PUNCH, READ, or TYPE. The FILE argument has the form:

FILE= An ASCII literal or variable (set to literal)

G. PROTECTION

This option specifies a protection code to be assigned the data file being transferred. The protection code determines the level of access that three possible classes of users (i.e., owner, member, or other) will have to the file. PROTECTION may be a 3-digit octal literal or a variable. If the argument is assigned a zero value or is not given, the default protection code established for the DECsystem-10 installation is used. The PROTECTION argument has the form:

PROTECTION= 3-digit octal or integer variable

H. DIRECTORY

This option is used for disk files only. It specifies the location of the user file directory (UFD) or the sub-file directory (SFD) which contains the file specified in the OPEN statement. A directory identifier may consist of either

1. the user's project-programmer number, which identifies the UFD (for example, 10,7), or

2. a UFD/SFD directory path specification. A path specification lists the UFD and the names of the SFD's which form a path to the desired SFD. For example, the following path specification identifies the path leading to SFD 1234:

```
10,7,SFDA,SFDB,1234
```

NOTE

Refer to the DECsystem-10 Monitor Calls Manual for a complete description of directories and multilevel directory structures.

The DIRECTORY argument has the form:

DIRECTORY= A literal or variable containing a UFD name or directory path specification

The user may also establish an array containing the directory specification as its elements and reference the array in the DIRECTORY argument. Single precision arrays permit 5-character directory names to be used; double precision arrays permit 6-character names to be used. A zero (0) entry must be used to terminate a directory path specification given in an array.

Examples of the use of single and double precision arrays in an OPEN statement DIRECTORY specification follow:

1. Single Precision Array

```
OPEN(UNIT=5,DIRECTORY=PATH,...)
```

where PATH and its elements are

```
DIMENSION PATH (5)
PATH (1)="10          project number
PATH (2)="7           programmer number
PATH (3)='SFDA'     } names of subfile
PATH (4)='SFDB'     } directories (SFD's)
PATH (5)=0
```

2. Double Precision Array

OPEN(UNIT=5,DIRECTORY=PATH,...)

where PATH and its elements are

DOUBLE PRECISION PATH (5)

```
PATH (1)"000010000007  project, pro-
                       grammer numbers=
                       UFD
PATH (2)'SFDABC'   }
PATH (3)'MY AREA'  }   names of sub-file
PATH (4)'YOURIT'   }   directories (SFD's)
PATH (5) 0
```

The elements of a directory specification may then be either a literal or a single or double precision array.

The following is an example of a literal specification:

```
DIRECTORY='10,7,SFD1,SFD2,SFD3'
          project-      sub-file
          programmer    directory
          number        path
```

Whenever the specification is an array, the required project and programmer numbers may be specified either as one word with the project number in the left half and the programmer number in the right half, or as the right halves of separate sequential word locations.

I. BUFFER COUNT This option enables the user to specify the number of I/O buffers to be assigned to a particular device. If this argument is not given or is assigned a value of zero, the monitor default is assumed. This argument has the form:

BUFFER COUNT= An integer constant or variable

J. FILE SIZE This option is used for disk operations only. It enables the user to estimate the number of words that an output file is going to contain. The use of FILE SIZE allows a user to ensure at the start of a program that enough space is available for its execution. If the size specified is found to be too small during program execution, the monitor allocates additional space according to the normal monitor algorithms. The value assigned to the FILE SIZE argument may be either an integer constant or variable.

This argument has the form:

FILE SIZE= An integer constant or variable

K. VERSION

This option is used for disk operations only. It enables the user to assign a 12-digit octal version number to a file when it is output. The quantity assigned to the VERSION argument may be either an octal constant or variable. This argument has the form:

VERSION= An octal constant or integer variable

L. BLOCK SIZE

This option enables the user to specify a physical storage block size for devices other than disk or DECtape. The value assigned the BLOCK SIZE argument may be either an integer constant or variable. The size specified must be greater than or equal to 3 and less than or equal to 4095. This argument has the form:

BLOCK SIZE= An integer constant or variable

M. RECORD SIZE

This option enables the user to force all logical records to be a specified length. If a logical record exceeds the specified length, it is truncated. If it is less than the specified length, nulls are added to pad the record to its full size. The RECORD SIZE argument is required whenever a random access mode is specified. The value assigned to this argument may be either an integer constant or variable and may be expressed as the numbers of words or characters, depending on the mode of the file being described. This argument has the form:

RECORD SIZE= An integer constant or variable

N. ASSOCIATE VARIABLE

This option is for disk random access operations only. It stores the number of the record to be accessed next if the program being executed were to continue to access files sequentially from the specified random access file. This argument has the form:

ASSOCIATE VARIABLE= An integer variable

O. PARITY

This option is for magnetic tape operations only. It permits the user to specify the type of parity to be observed (odd or even) during the transfer of data. This option has the form:

PARITY= 'ODD'
'EVEN'
variable (set to literal)

P. DENSITY

This option is for magnetic tape operations only. It permits the user to specify any of three possible bit-per-inch (bpi) tape density parameters for magnetic tape transfer operations. This option has the form:

	DENSITY=	'200' '556' '800' variable (set to literal)

Q. DIALOG

The use of this option in an OPEN statement enables the user to supersede or defer, at execution time, the values previously assigned to the arguments of the statement. The value assigned to DIALOG may be null, a literal, or an array. This argument has the form:

DIALOG= Literal, array, or null

Whenever DIALOG is assigned a null value, it establishes a user/program dialogue mode when the OPEN statement containing it is executed. During a dialogue mode, FOROTS outputs the following messages at the user's terminal.

ENTER FILE SPECIFICATIONS FOR LOGICAL UNIT XX

FOROTS then types the existing file specifications defined by the current OPEN statement.

Once the message and defined file specifications are output, the user may enter any desired changes. Only the arguments that are to be changed need to be entered.

Whenever a literal or an array is assigned to DIALOG, it must contain in ASCII the file specification information or indicate where to request dialog information.

8.1.2 Summary of OPEN/CLOSE Statement Options

The options permitted and required in the OPEN and CLOSE statements and the type of value required by each are summarized in Table 8-1.

Table 8-1
OPEN/CLOSE Statement Arguments

Argument	Values Required
UNIT=	Integer variable or constant
MODE=	Literal constant or variable
DIRECTORY=	Literal or array
FILE SIZE=	Integer constant or variable
BUFFER COUNT=	Integer constant or variable
ASSOCIATE VARIABLE=	Integer variable
ACCESS=	'SEQIN', 'SEQOUT', 'SEQINOUT' 'RANDIN', 'RANDOM', 'APPEND', or variable
FILE=	Literal constant or variable
DIALOG=	Literal array or null
BLOCK SIZE=	Integer constant or variable
VERSION=	Octal constant or variable
DEVICE=	Literal constant or variable
PROTECTION=	Octal constant or integer variable
DISPOSE=	Literal constant or variable
RECORD SIZE=	Integer constant or integer variable
PARITY=	Literal constant or variable
DENSITY=	Literal constant or variable

CHAPTER 9

SUMMARY OF DECsystem-10 FORTRAN STATEMENTS

CONTROL STATEMENTS

General Form	Section References
ASSIGN i to m	4.1.3
CALL name (a_1, a_2, ...)	7.5.2
CONTINUE	4.4
DO i m=m_1, m_2, m_3	4.3
GO TO i	4.1.1
GO TO m	4.1.3
GO TO m, (i_1, i_2, ...)	4.1.3
GO TO (i_1, i_2, ...),m	4.1.2
IF (e_1)i_1, i_2, i_3	4.2.1
IF (e_2)s	4.2.2
PAUSE	4.5
PAUSE j	4.5
PAUSE 'h'	4.5
RETURN	7.5.3
RETURN i	7.5.3
STOP	4.6
END	4.7

DATA TRANSMISSION STATEMENTS

General Form	Section References
ACCEPT f	5.2.9
ACCEPT f, list	5.2.9
BACKSPACE unit	5.3
DECODE (n,f,v) list	5.4
END FILE unit	5.3
ENCODE (n,f,v) list	5.4
FORMAT (g)	5.1.1
PRINT f	5.2.3
PRINT f, list	5.2.3
PUNCH f	5.2.4
READ f	5.2.7
READ f, list	5.2.7
READ (unit, f)	5.2.7
READ (unit, f) list	5.2.7
READ (unit) list	5.2.7
READ (unit, $name_1$)	5.2.7
READ (unit #R,f) list	5.2.7
READ (unit, f, END=c, ERR=d) list	5.2.7
READ (unit, f, END=c) list	5.2.7
READ (unit, f, ERR=d) list	5.2.7
REREAD f, list	5.2.8
REWIND unit	5.3
SKIP RECORD unit	5.3

DATA TRANSMISSIONS STATEMENTS (Cont)

General Form	Section References
TYPE f	5.2.5
TYPE f, list	5.2.5
WRITE (unit, f)	5.2.6
WRITE (unit, f) list	5.2.6
WRITE (unit) list	5.2.6
WRITE (unit, $name_1$)	5.2.6
WRITE (unit #R, f) list	5.2.6
UNLOAD unit	5.3

SPECIFICATION STATEMENTS

General Form	Section References
BLOCK DATA	6.2.2
COMMON a(n_1,n_2,...),b(n_3,n_4,...),...	6.1.2
COMMON /blk1/a, b/blk2/c, d/...	6.1.2
COMPLEX a(n_1, n_2,...), b(n_3, n_4,...),...	6.3
DATA t,u,.../k_1,k_2,k_3,.../ v,w,.../k_4, k_5, k_6,.../...	6.2.1
DIMENSION a(n_1, n_2,...), b(n_1, n_2,...),...	6.1.1
DOUBLE PRECISION a(n_1, n_2,...), b(n_3, n_4,...),...	6.3
EQUIVALENCE (a(n_1,...), b(n_2,...),...),... (c(n_3,...), d(n_4,...),...),...	6.1.3

SPECIFICATIONS STATEMENTS (Cont)

General Form	Section References
EXTERNAL y,z,...	7.7
IMPLICIT $type_1(1_1-1_2), type_2(1_3-1_4),...$	6.3.1
INTEGER $a(n_1, n_2,...), b(n_3, n_4,...),...$	6.3
LOGICAL $a(n_1, n_2,...), b(n_3, n_4,...),...$	6.3
NAMELIST $/name_1/a,b,.../name_2/c,d,...$	5.1.2
REAL $a(n_1, n_2,...)$ $b(n_3, n_4, ...),...$	6.3
SUBSCRIPT INTEGER $a(n_1, n_2,...),b(n_3,...),...$	6.3

ARITHMETIC STATEMENT FUNCTION DEFINITION

General Form	Section Reference
name (a,b,...)=e	7.3

FILE CONTROL STATEMENTS

General Form	Section Reference
OPEN (argument,argument,...)	8.1
CLOSE (argument,argument,...)	8.1

NOTES:

$a_1, a_2,...$	are expressions
a,b,c,d	are variable names

blk1, blk2	are block names
c	is the statement number to which control is transferred upon encountering an end-of-file
d	is the statement number to which control is transferred upon encountering an error condition on the input data.
e	is an expression
e_1	is a noncomplex expression
e_2	is a logical expression
f	is a format number
g	is a format specification
'h'	is an alphanumeric
$i,i_1,i_2,\ldots$	are statement numbers
j	is an integer constant
$k_1,k_2,\ldots$	are constants of the general form j*k where k is any constant
$l_1,l_2,\ldots$	are letters
list	is an input/output list
m	is an integer variable name
m_1,m_2,m_3	are integer expressions
$n_1,n_2,\ldots$	are dimension specifications
n	are the number of ASCII characters
name	is a subroutine or function name
$name_1,name_2$	are NAMELIST names
#R	is a record number where I/O begins
s	is a statement (not DO or logical IF)
t,u,v,w	are variable names or input/output lists

$type_1, type_2, \ldots$	are type specifications
unit	is an integer variable or constant specifying a logical device number
v	is the starting address of the ASCII record referenced
y,z	are external subprogram names

SECTION II

THE OBJECT TIME SYSTEM

The four chapters of this section contain information on FORLIB, SUBPROGRAM calling sequences, accumulator usage, compiler switches and diagnostic messages, and FORTRAN user programming.

Complete information on the FOROTS object time system may be found in the DECsystem-10 FORTRAN-10 Language Manual (DEC-10-LFORA-C-D).

CHAPTER 10

FORLIB

FORLIB is a single file which contains all of the programs in tne FORTRAN library. It is composed of three groups of programs:

(1) The FORTRAN Object Time System.

(2) Science Library.

(3) FORTRAN Utility Subprograms.

There are two forms of FORLIB, one for the KA-10 and the other for the KI-10. The KA-10 library will run on the KI-10, but will not take advantage of the speed of the KI-10. The KI-10 library will not run on the KA-10 because of the hardware differences. Also, the library used must match the compiler used, i.e., KA-10 compiled code must use the KA-10 FORLIB and the KI-10 compiled code must use the KI-10 FORLIB.

10.1 THE FORTRAN OBJECT TIME SYSTEM

The system programs in the FORTRAN object time system act as the interface between the user's program and the DECsystem-10. All of these programs are invisible to the user's program. The FORTRAN object time system is loaded automatically from FORLIB and resides in the user's core area along with the user's main programs and any library functions and subroutines that his programs reference.

10.1.1 FOROTS

FOROTS is the main program of the FORTRAN object time system and is loaded whenever a FORTRAN main program is in core. The primary functions of FOROTS are

A. FORMAT statement processing,

B. Core management, and

C. Control of I/O devices at runtime.

10.1.1.1 FORMAT Processing - FOROTS assumes that all FORMAT statements are syntactically correct since the syntax of each statement is checked by the compiler. FOROTS scans the FOR-

MAT statements and performs the indicated I/O operations. FOROTS invokes the required conversion routine to actually do data conversion. The conversion routine that is used is a function of the conversion indicated in the FORMAT statement and of the data type of the element in the I/O list.

10.1.1.2 I/O Device Control - FOROTS executes the required carriage control of output devices that are physical listing devices (LPT, TTY) and stores the carriage control character at the beginning of each line if the output is going to a retrievable medium for deferred listing. When listings are deferred, the appropriate switch in PIP can be used to list the file and execute the required carriage control.

10.1.1.3 Additional Functions of FOROTS - FOROTS is responsible for the following:

A. Control of REREAD and ENCODE/DECODE features.

B. Interaction with EOFTST and READ (unit,f,END=C)list to handle end-of-file testing.

C. Control of the assignment of devices to software channels.

D. Control of the handling of filenames for I/O associated with directory devices.

E. Control of the opening and closing of data files.

10.2 SCIENCE LIBRARY AND FORTRAN UTILITY SUBPROGRAMS

The Science Library and FORTRAN Utility Subprograms extend the capabilities of the FORTRAN language. These subprograms are called explicitly by the user. The subprograms include the built-in FORTRAN math functions and the user-called utility subroutines which provide optional I/O capabilities and control of and information about the program's environment. The optional I/O capabilities and environmental control are achieved by the subroutines from interactions with the FORTRAN Operating System.

10.2.1 FORTRAN Library Functions

This section contains descrptions of all standard function subprograms provided with the FORTRAN library for the DECsystem-10. These functions are called by using the function mnemonic as a function name in an arithmetic expression. The function mnemonics in Table 10-1 have the types specified unless their types are explicitly or implicitly changed. (Refer to Section 6.3, "Type Declaration Statements" and Section 6.3.1, "IMPLICIT Statement.")

Table 10-1
FORTRAN Library Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function	External Calls
Absolute value:						
Real	ABS	|arg|	1	Real	Real	
Integer	IABS	|arg|	1	Integer	Integer	
Double precision	DABS	|arg|	1	Double	Double	
Complex to real	CABS	$c=(x^2+y^2)^{1/2}$	1	Complex	Real	SQRT
Conversion:						
Integer to real	FLOAT*		1	Integer	Real	
Real to integer	IFIX*	Result is largest integer $\leq a$	1	Real	Integer	
Double to real	SNGL		1	Double	Real	
Real to double	DBLE		1	Real	Double	
Integer to double	DFLOAT		1	Integer	Double	
Complex to real (obtain real part)	REAL		1	Complex	Real	
Complex to real (obtain imaginary part)	AIMAG		1	Complex	Real	
Real to complex	CMPLX	$c=Arg_1+i*Arg_2$	2	Real	Complex	
Truncation:						
Real to real	AINT	Sign of arg * largest integer $\leq$ arg	1	Real	Real	
Real to integer	INT*		1	Real	Integer	
Double to integer	IDINT		1	Double	Integer	
Remaindering:						
Real	AMOD	The remainder when Arg 1 is divided by Arg 2	2	Real	Real	ERROR.,TRAPS
Integer	MOD		2	Integer	Integer	
Double precision	DMOD		2	Double	Double	

*These functions are not used on the KI-10 because they are unnecessary.

Table 10-1(Cont)
FORTRAN Library Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function	External Calls
Maximum Value:	AMAX0	$\{Max(Arg_1,Arg_2,...)\}$	$\{\geq 2\}$	Integer	Real	FLOAT
	AMAX1			Real	Real	
	MAX0			Integer	Integer	
	MAX1			Real	Integer	IFIX
	DMAX1			Double	Double	
Minimum Value:						
	AMIN0	$\{Min(Arg_1,Arg_2,...)\}$	$\{\geq 2\}$	Integer	Real	FLOAT
	AMIN1			Real	Real	
	MIN0			Integer	Integer	
	MIN1			Real	Integer	IFIX
	DMIN1			Double	Double	
Transfer of sign:						
Real	SIGN	$\{Sgn(Arg_2)*\|Arg_1\|\}$	2	Real	Real	
Integer	ISIGN		2	Integer	Integer	
Double precision	DSIGN		2	Double	Double	
Positive Difference:						
Real	DIM	$\{Arg_1-Min(Arg_1,Arg_2)\}$	2	Real	Real	
Integer	IDIM		2	Integer	Integer	
Exponential:						
Real	EXP	$\{e^{Arg}\}$	1	Real	Real	ERROR.
Double	DEXP		1	Double	Double	
Complex	CEXP		1	Complex	Complex	EXP,SIN,COS, ALOG,ERROR.

Table 10-1 (Cont)
FORTRAN Library Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function	External Calls
Logarithm:						
Real	ALOG	$\log_e(Arg)$	1	Real	Real	ERROR.
	ALOG10	$\log_{10}(Arg)$	1	Real	Real	ERROR.
Double	DLOG	$\log_e(Arg)$	1	Double	Double	
	DLOG10	$\log_{10}(Arg)$	1	Double	Double	
Complex	CLOG	$\log_e(Arg)$	1	Complex	Complex	ALOG,ATAN2, SQRT,ERROR.
Square Root:						
Real	SQRT	$(Arg)^{1/2}$	1	Real	Real	ERROR.
Double	DSQRT	$(Arg)^{1/2}$	1	Double	Double	
Complex	CSQRT	$c=(x+iy)^{1/2}$	1	Complex	Complex	SQRT
Sine:						
Real (radians)	SIN	sin (Arg)	1	Real	Real	
Real (degrees)	SIND		1	Real	Real	
Double (radians)	DSIN		1	Double	Double	
Complex	CSIN		1	Complex	Complex	SIN,SINH,COSH, ALOG,EXP
Cosine:						
Real (radians)	COS	cos (Arg)	1	Real	Real	
Real (degrees)	COSD		1	Real	Real	
Double (radians)	DCOS		1	Double	Double	
Complex	CCOS		1	Complex	Complex	SIN,SINH,COSH, ALOG,EXP

Table 10-1 (Cont)
FORTRAN Library Function

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function	External Calls
Hyperbolic:						
Sine	SINH	sinh (Arg)	1	Real	Real	EXP,ERROR.
Cosine	COSH	cosh (Arg)	1	Real	Real	EXP,ERROR.
Tangent	TANH	tanh(Arg)	1	Real	Real	EXP
Arc - sine	ASIN	asin (Arg)	1	Real	Real	ATAN,SQRT ERROR.
Arc - cosine	ACOS	acos (Arg)	1	Real	Real	ATAN,SQRT, ERROR.
Arc tangent						
Real	ATAN	atan (Arg)	1	Real	Real	
Double	DATAN	atan (Arg)	1	Double	Double	
quotient of two arguments	ATAN2	atan (Arg_1/Arg_2)	2	Real	Real	ATAN,ERROR., TRAPS
	DATAN2	atan (Arg_1/Arg_2)	2	Double	Double	DATAN,ERROR.
Complex Conjugate	CONJG	Arg=X+iY,C=X - iY	1	Complex	Complex	
Random Number	RAN	result is a random number in the range of 0 to 1.0.	1	Integer Real, Double, or Complex	Real	

10.2.2 FORTRAN Library Subroutines

This section contains descriptions of all standard subroutine subprograms provided within the FORTRAN library for the DECsystem-10. These subprograms are closed subroutines and are called with a CALL statement.

Table 10-2
FORTRAN Library Subroutines

Subroutine Name	Effect
DATE	Places today's date as left-justified ASCII characters into a dimensioned 2-word array. CALL DATE (array) where array is the 2-word array. The date is in the form dd-mmm-yy where dd is a 2-digit day (if the first digit is 0, it is converted to a blank), mmm is a 3-letter month (e.g., MAR), and yy is a 2-digit year. The date is stored in ASCII code, left-justified in the two words.
DUMP	Causes particular portions of core to be dumped and is referred to in the following form: CALL DUMP (L_1,U_1,F_1,...,L_n,U_n,F_n) where L_i and U_i are the variable names which give the limits of core memory to be dumped. Either L_i or U_i may be upper or lower limits. F_i is a number indicating the format in which the dump is to be performed: 0=octal, 1=real, 2=integer, and 3=ASCII. If F is not 0,1,2,3, the dump is in octal. If F_n is missing, the last section is dumped in octal. If U_n and F_n are missing, an octal dump is made from L to the end of the job area. If L_n, U_n, and F_n are missing, the entire job area is dumped in octal. The dump is terminated by a call to EXIT.

Table 10-2 (Cont)
FORTRAN Library Subroutines

Subroutine Name	Effect
ERRSET	Allows the user to control the typeout of execution-time arithmetic error messages, ERRSET is called with one argument in integer mode. CALL ERRSET(N) Typeout of each type of error is suppressed after N occurrences of that error message. If ERRSET is not called, the default value of N is 2.
EXIT	Returns control to the monitor and, therefore, terminates the execution of the program.
ILL	Sets the ILLEG flag. If the flag is set and an illegal character is encountered in floating point/double precision input, the corresponding word is set to zero. CALL ILL
LEGAL	Clears the ILLEG flag. If the flag is set and an illegal character is encountered in floating point/double precision input, the corresponding word is set to zero. CALL LEGAL
PDUMP	Is referred to in the following form: CALL PDUMP(L_1,U_1,F_1,...,L_n,U_n,F_n) where the arguments are the same as those for DUMP. PDUMP is the same as DUMP except that control returns to the calling program after the dump has been executed.
RELEAS	Closes out I/O on a device initialized by the FORTRAN object time system and returns it to the uninitialized state. CALL RELEAS (unit*)

*For explanation, see page 9-6.

Table 10-2 (Cont)
FORTRAN Library Subroutines

Subroutine Name	Effect
SAVRAN	SAVRAN is called with one argument in integer mode. SAVRAN sets its argument to the last random number (interpreted as an integer) that has been generated by the function RAN.
SETRAN	SETRAN has one argument which must be a non-negative integer $<2^{31}$. The starting value of the function RAN is set to the value of this argument, unless the argument is zero. In this case, RAN uses its normal starting value.
SLITE(i)	Turns sense lights on or off. i is an integer expression. For$\leq$i$\leq$36 sense light i will be turned on. If i=0, all sense lights will be turned off.
SLITET(i, j)	Checks the status of sense light i and sets the variable j accordingly and turns off sense light i. If i is on, j is set to 1; and if i is off, j is set to 2.
SSWTCH(i, j)	Checks the status of data switch i($0\leq i\leq 35$) and sets the variable j accordingly. If i is set down, j is set to 1; and, if i is up, j is set to 2.
TIME	Returns the current time in its argument(s) in left-justified ASCII characters. If TIME is called with one argument, CALL TIME(X) the time is in the form hh : mm where hh is the hours (24-hour time) and mm is the minutes. If a second argument is requested, CALL TIME(X,Y) the first argument is returned as before and the second has the form ss.t where ss is the seconds and t is the tenths of a second.

CHAPTER 11

INTERACTING WITH NON-FORTRAN PROGRAMS AND FILES

11.1 CALLING SEQUENCES

The standard procedures for writing DECsystem-10 subroutine calls are described below:

1. Procedure

   A. The calling program must load the right half of accumulator (AC) 16 with the address of the first argument in the argument list.

   B. The left half of AC 16 must be set to zero.

   C. The subroutine is then called by a PUSHJ instruction to AC 17.

   D. The returns will be made to the instruction immediately after the PUSHJ 17 instruction.

2. Restrictions

   A. Skip returns are not permitted.

   B. The contents of the pushdown stack located before the address specified by AC 17 belongs to the calling program; it cannot be read by the called subprogram.

   C. FOROTS assumes that it has control of the stack; therefore the user must not create his own stack. The FOROTS stack is initialized by

      JSP 16,RESET.

      or the library routine CALL RESET.

11.2 ACCUMULATOR USAGE

The specific functions performed by accumulators (AC) 17, 16, 0, and 1 are listed below:

1. Pushdown pointer

   AC 17 is always maintained as a pushdown pointer. Its right half points to the last location in use on the stack and its left half contains the negative of the number of

(words-1) allocated to the unused remainder of the stack. (A trap occurs when something is pushed into the next to last location. The trap instruction may itself be a PUSHJ on the KI10 processor which uses the last location.) A positive left half is not permitted.

2. Argument list pointer

   AC 16 is used as the argument pointer. The called subprogram does not need to preserve its contents. The calling program cannot depend on getting back the address of the argument list passed to the callee. AC 16 cannot point to the AC's or to the stack.

3. Temporary and value return registers

   AC's 0 and 1 are used as temporary registers and for returning values. The called subprogram does not need to preserve the contents of AC's 0 and 1 (even if not returning a value). The calling program must never depend on getting back the original contents of the data passed to the called subprogram.

4. Returning values

   At the option of the designer of a called subprogram, a subroutine may pass back results by modifying the arguments, returning a single precision value in AC 0 or a double precision or complex value in AC's 0 and 1. A combination of the above may be used. However, two single precision values cannot be returned in AC's 0 and 1 since FORTRAN would not be able to handle it.

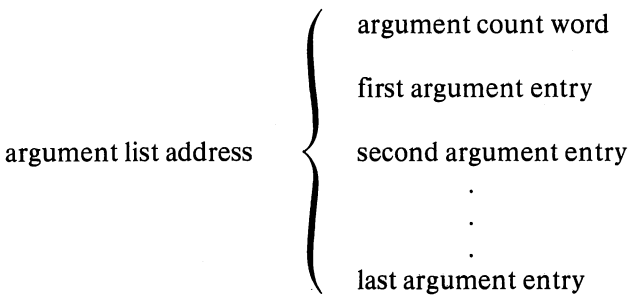
5. Preserved AC's

   FORTRAN FUNCTION subprograms preserve AC's 2-15; SUBROUTINE subprograms do not.

   The design of the called subprogram cannot depend on the contents of any of the AC's being set up by the calling subprogram, except for AC's 16 and 17. Passing information must be done explicitly by the argument list mechanism. Otherwise, the called subprograms cannot be written in either FORTRAN or COBOL.

11.3 ARGUMENT LISTS

The format of the argument list is the following:



The format of the argument count word is the following:

bits 0-17	These contain -n, where n is the number of argument entries.
bits 18-35	These are reserved and must be 0.

The format of an argument entry is the following (each entry is a single word):

bits 0-8	Reserved for future DEC development (set to 0 for now).
bits 9-12	Argument type code.
bit 13	Indirect bit if desired.
bits 14-17	Index field, must be 0 for present.
bits 18-35	Address of the argument.

The following restrictions should be observed.

1. Neither the argument lists nor the arguments themselves can be on the stack. This restriction is imposed so that the stack can be moved. The same restriction applies to any indirect argument pointers.

2. The called program may not modify the argument list itself. The argument list may be in a write-protected segment.

Note that the argument count word is at position -1 with respect to the contents of AC 16. This word is always required even if the subroutine does not handle a variable number of arguments. A subroutine which has no arguments must still provide an argument list consisting of two words (i.e., the argument count word with a 0 in it and zero argument word).

Example:

```
MOVEI     16,1+ [EXP-3B17,A,B,C]                ;SETUP ARG LIST

PUSHJ     17,SUB                                ;CALL SUBROUTINE
. . .                                           ;RETURN HERE
. . .
;SUBROUTINE TO SET THIRD ARG TO SUM OF FIRST TWO ARGS

SUB;      MOVE      T,@0 (16)                   ;GET FIRST ARG
          ADD       T,@1 (16)                   ;ADD SECOND ARG
          MOVEM     T,@2 (16)                   ;SET THIRD ARG
          POPJ      17,                         ;RETURN TO CALLER
```

11.4 CONVERTING EXISTING MACRO-10 LIBRARIES FOR USE WITH FORTRAN

To conveniently convert existing MACRO-10 programs so that they will still load and execute correctly when called from FORTRAN, the user can

1. transfer the initial entry sequence for a routine to

```
entry:  CAIA

        PUSH 17, CEXIT.
```

2. change all returns to POPJ 17,0

These are the functions performed by the HELLO and GOODBY macros. These macros (available in the file FORPRM.MAC, part of the FOROTS release) were successfully used in converting the library routines to run FORTRAN.

In addition, since the FORTRAN compiler uses the indirect bits on argument lists (note that this permits shared, pure code argument lists), it is essential that code which accesses parameters takes this into account. Specifically, sequences that obtained the values of parameters through the use of operations such as

```
HRRZ R,1(16)
```

to pick up the address of the second argument should be changed to

```
MOVEI R,@1(16)
```

This latter operation will work when interfacing to either FORTRAN IV (F40) or FORTRAN-10.

11.5 MIXING FORTRAN-10 AND FORTRAN IV (F40) COMPILED PROGRAMS

Starting with Version 1A of LINK-10, use of the switch /MIXFOR will permit loading of FORTRAN-10 and FORTRAN IV (F40) programs. This is achieved by modifying the code while it is loaded.

This introduces extra code that results in a degradation of the execution of programs so loaded. This feature is provided as a convenience for conversion. It is not intended that it be used for other than conversion assistance.

CHAPTER 12

FORTRAN IV (F40) COMPILER AND DIAGNOSTICS

12.1 RUNNING THE FORTRAN IV (F40) COMPILER

The command to run the FORTRAN IV (F40) compiler is:

.R F40

A command to the compiler is of the general form:

object file name, listing filename=source file name(s)

EXAMPLE:

*MYPROG.REL,MYPROG.LST=MYPROG.F4

or

*MYPROG,MYPROG=MYPROG

The FORTRAN IV (F40) compiler will insert the default extensions if they have been omitted.

The switches to the compiler are shown in Table 12-1.

12.2 MONITOR COMMANDS TO RUN THE FORTRAN IV (F40) COMPILER

Compilation of FORTRAN source program files can be performed by use of the COMPILE, LOAD, EXECUTE, and DEBUG commands.

COMPILE COMMAND

The COMPILE command produces relocatable binary files (REL files) for the specified source program files.

FORM: .COMPILE filename.ext
EX: .COMPILE MAIN.F4

LOAD COMMAND

The LOAD command translates the specified source files if necessary, loads the REL files generated into core and begins execution of the program.

FORM: .LOAD filename.ext
EX: .LOAD MAIN.F4

EXECUTE COMMAND

The EXECUTE command translates the specified files if necessary, loads the REL files generated into core and begins execution of the program.

FORM: .EXECUTE filename.ext
EX: .EXECUTE MAIN.F4

DEBUG COMMAND

The DEBUG command translates the specified source files if necessary, loads the REL files generated and prepares for debugging.

FORM: .DEBUG filename.ext
EX: .DEBUG MAIN.F4

Table 12-2 lists the FORTRAN Compiler Diagnostics.

Table 12-1
FORTRAN Compiler Switch Options

Switch	Meaning
A†	Advance magnetic tape reel by one file.
B†	Backspace magnetic tape reel by one file.
C†	Generate a CREF-type cross-reference listing. (DSK:CREF.TMP assumed if no list-dev specified) Complement: Do not produce cross-reference information (standard procedure).
E	Print an octal listing of the binary program produced by the compiler in addition to the symbolic listing output. Complement: Do not produce octal listing (standard procedure).
I	Translate the letter D in column 1 as a space and treat the line as a normal FORTRAN statement. Complement: Translate the letter D in column 1 as a comment character and treat the line as a comment (standard procedure).
M	Include MACRO coding in the output listing.

†Switches A through C and T, W, and Z must immediately follow the device name or filename.ext to which the individual switch applies.

Table 12-1 (Cont)
FORTRAN Compiler Switch Options

Switch	Meaning
	Complement: Eliminate the MACRO coding from the output listing (standard procedure).
N	Suppress output of error messages on the terminal.
	Complement: Output error messages on TTY (standard procedure).
S	If the compiler is running on the KA-10, produce code for execution on the KI-10 and vice-versa.
T†	Skip to the logical end of the magnetic tape reel.
W†	Rewind the magnetic tape reel.
Z†	Zero the DECtape directory.

12.3 DIAGNOSTICS

Table 12-2
FORTRAN Compiler Diagnostics
(Command Errors)

Message	Meaning
?BINARY OUTPUT ERROR dev:filename.ext	An output error has occurred on the device specified for the binary program output.
?CANNOT FIND dev:filename.ext	Filename.ext cannot be found on this device.
?DEVICE INPUT ERROR for command string	Device error occurred while attempting to read Monitor command file.
IMPROPER IO FOR DEVICE dev:	An input device specified for output (or vice versa) or an illegal data mode was specified (e.g., binary output to TTY).
ILLEGAL MEMORY REFERENCE AT loc COMPILATION TERMINATED	An illegal memory reference has occurred and compilation has stopped. The current output files will be closed and the next source files read.

†Switches A through C and T, W, and Z must immediately follow the device name or filename.ext to which the individual switch applies.

Table 12-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
I-8 ARGUMENT TYPE DOESN'T AGREE WITH FUNCTION SPEC	The actual arguments for a function do not agree in type with the dummy arguments in the specification of the function.
I-9 THIS FUNCTION REQUIRES MORE ARGUMENTS	Not enough arguments were supplied for a function.
I-10 SUBPROGRAM NAME ALREADY IN USE	A subprogram name has appeared in another statement as a scalar or array variable, arithmetic function statement name, or COMMON block name. (See Section 7.5)
I-11 DUMMY ARGUMENT IN DATA STATEMENT	Dummy arguments may not appear in DATA statements. (See Section 6.2.1)
I-12 NOT A SCALAR OR ARRAY	The variable defining the starting address for an ENCODE/DECODE statement must be a scalar or an array. (See Section 5.4) The I/O unit name of a READ/WRITE statement is not a scalar or array. (See Sections 5.2.6, 5.2.7) An attempt to ASSIGN a label number to a variable that is not a scalar or array. (See Section 2.2) An attempt to GO TO through a variable that is not a scalar or array. (See Section 4.1)
I-13 ILLEGAL USE OF DUMMY ARGUMENT	Dummy arguments may be used with functions or subprograms only. (See Section 7.4.1, 7.5.1)
I-14 ILLEGAL DO LOOP PARAMETER	The DO index must be a non-subscripted integer variable while the initial, limit, and increment values of the index must be an integer expression; the index may not be zero. (See Section 4.3)
I-15 I/O VARIABLES MUST BE SCALARS OR ARRAYS	Referencing data in an I/O statement other than scalars or arrays is illegal. (See Section 5.2)
I-16 A CONFLICT EXISTS WITH A COMMON DECLARATION	The function named used was previously declared a scalar variable in a COMMON statement.

Table 12-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
S-1 ILLEGAL NAME OR DELIMITER OR KEY CHARACTER	A variable name doesn't start with an alphabetic character, or a delimiter such as the left parenthesis that begins a format is missing, or a key character such as the letter D in BLOCK DATA is missing.
S-2 STATEMENT KEYWORD NOT RECOGNIZED	A statement keyword such as ERASE was not recognized possibly due to misspelling (e.g., ERASC 16).
S-3 ILLEGAL FIELD SPECIFICATION	The field width or decimal specification in a FORMAT statement must be integer. The number of Hollerith characters in an H specification must be equal to the number specified. (See Sections 5.1.1.1, 5.1.1.6)
S-4 SCALAR VARIABLE - MAY NOT BE SUBSCRIPTED	An undimensioned variable (a scalar variable) is being illegally subscripted (see Section 2.2.1) or a scalar variable is subscripted in an ENCODE/DECODE statement (see Section 5.4)
S-5 ILLEGAL TYPE SPECIFICATION	The type of constant specified is illegal or misspelled. (See Section 2.1)
S-6 ARGUMENT IS NOT SINGLE LETTER	Arguments in parentheses must be single letters in IMPLICIT statement. (See Section 6.3.1)
S-7 'NAMELIST' NOT FOLLOWED BY "/"	The first character following NAMELIST must be /. (See Section 5.1.2)
S-8 ILLEGAL CHARACTER IN LABEL	A non-numeric character was detected in the label field of the statement, possibly because tabs or spaces are missing.
S-9 MISSING COMMA OR SLASH IN SPECIFICATION STATEMENT	A specification statement (see Chapter 9) requires a comma or slash and it is missing.
S-10 ILLEGAL ARITHMETIC "IF" - TOO MANY LABELS	An arithmetic "IF" statement must have no more or less than three statement labels to transfer to. Special optimization will occur if two of the labels are the same, or one or more labels refer to the next statement.
S-11 A NUMBER WAS EXPECTED	Only arrays which are subprogram arguments can have adjustable dimensions. (See Section 6.1.1.1)

Table 12-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
S-12 IMPLICIT TYPE RANGE OVERLAPS PREVIOUS SPECIFICATION	An implicit type range encompasses a character that has already been given an implicit type.
S-13 ATTEMPT TO USE AN ARRAY OR FUNCTION NAME AS A SCALAR	Variables may be either scalar or array but not both. Variables appearing in a DIMENSION statement must be subscripted when used. (See Section 2.2) Function names must be followed by at least one argument enclosed in parentheses (See Section 7.4).
S-14 ARRAY NOT SUBSCRIPTED	See S-13.
S-15 ILLEGAL USE OF AN ARITHMETIC FUNCTION NAME	Arithmetic function definition statement name is being used without arguments (i.e., as a scalar) in an arithmetic expression. (See Section 7.3)
S-16 MULTIPLE RETURN ILLEGAL WITHOUT STATEMENT LABEL ARG	A dollar sign ($) or an asterisk (*) must have appeared in the argument list of this subprogram to represent the position of a statement label argument in the call.
S-17 INCORRECT PAREN COUNT OR MISSING IMPLIED DO INDEX	The number of left and right parentheses does not match, or an undefined index variable was used in defining a DO loop (see Section 5.2.1), or the number of implied DO loops and the number of matching parentheses differ in a DATA statement. (See Section 6.2.1)
S-18 INVALID INDEX IN DO-LOOP OR IMPLIED DO-LOOP	The index of a DO statement must be a non-subscripted integer variable and must not be zero. (See Section 4.3) The index is not used as a subscript in a DATA list. (See Section 6.2.1)
S-19 EQUIVALENCE REQUIRES TWO OR MORE ELEMENTS	The EQUIVALENCE statement must have more than one argument because it causes variables to share the same location. (See Section 6.1.3)
S-20 ILLEGAL DEFINITION OF AN ARITHMETIC STATEMENT FUNCTION	The statement function continues past its recognized end point.
S-21 MISSING COMMA IN INPUT/OUTPUT LIST	An input/output list continues past its recognized end point.
S-22 STATEMENT CONTINUES PAST RECOGNIZED END POINT	A statement other than those mentioned above continued past its recognized end point.

Table 12-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
S-23 ILLEGAL COMPLEX CONSTANT	The parentheses of the complex constant enclose a logical, Hollerith, or complex constant.
S-24 NULL STATEMENT ILLEGAL	A line appears with a statement label but no statement.
O-1 BLOCK DATA NOT SEPARATE PROGRAM	Block Data must exist as a separate program. (See Sections 6.2.2, 7.6)
O-2 SUBROUTINE IS NOT A SEPARATE PROGRAM	A subroutine following a main program or another subroutine subprogram may have no statement between it and the preceding program's END statement and must begin with a SUBROUTINE statement. The previous program must have been terminated properly. (See Section 7.5)
O-3 STATEMENT OUT OF PLACE	The IMPLICIT specification statement and any arithmetic function definition statement must appear before any executable statement. (See Chapter 6)
O-4 EXECUTABLE STATEMENTS ILLEGAL IN BLOCK DATA	Block DATA statements cannot contain executable statements.
A-1 MINIMUM VALUE EXCEEDS MAXIMUM VALUE	Minimum value of an array exceeds the maximum value specified. (See Section 6.1.1)
A-2 ATTEMPT TO ENTER A VARIABLE INTO COMMON TWICE	A variable name may appear in COMMON statement only once. (See Section 6.1.2)
A-3 ATTEMPT TO EQUIVALENCE A SUBPROGRAM NAME OR DUMMY ARGUMENT	An identifier defined as a subprogram name may not appear in EQUIVALENCE statements in the defining program. Dummy argument identifiers of a subprogram may not appear in EQUIVALENCE statements in that subprogram. (See Sections 6.1.3,7.1)
A-4 NOT A CONSTANT OR DUMMY ARGUMENT	Only constant and dummy arguments may be used as arguments in dimension statements. (See Section 7.4.1)
A-5 CAUTION ** COMMON VARIABLE PASSED AS ARGUMENT	The variable may be multiply defined in the called subprogram (See Sections 7.4.1, 7.5.1).
M-1 TOO MANY SUBSCRIPTS	An array variable appears with more subscripts than specified. (See Sections 2.2.2, 6.1.1)
M-2 WRONG NUMBER OF SUBSCRIPTS	An array variable appears with too few subscripts. (See Sections 2.2.2, 6.1.1)
M-3 CONSTANT OVERFLOW	Too many significant digits in the formation of a constant or the exponent is too large. (See Section 2.1)

Table 12-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
M-4 ILLEGAL 'IF' ARGUMENT	Logical IF or DO statement adjacent to a logical IF statement, or illegal expression within a logical IF statement. (See Sections 4.2.2, 4.3)
M-5 ILLEGAL CONVERSION IMPLIED	Attempt to mix double precision and complex data in the same expression. (See Section 2.3.1)
M-6 LABEL OUT OF RANGE OR ARRAY TOO LARGE	Illegal statement label (See Section 1.1.1) or array size is greater than $2^{18}-1$.
M-7 UNTERMINATED HOLLERITH STRING	A missing single quote or fewer than n characters following an "nH" specification. (See Section 5.1.1.6)
M-8 SYSTEM ERROR - NO MORE SPACE FOR RECURSIVE STORAGE	The compiler's work roll is too small to hold the parts of all the subexpressions this statement implies. Break this statement or reassemble the compiler with a larger work roll parameter (WORLEN= 150_8 at present).
M-9 TOO MUCH DATA - WRONG ARRAY SIZE OR LITERAL TOO LONG	The list of DATA constants defines more words than the list of DATA variables specifies. This may be due to an array of the wrong size in the list of DATA variables, or definition of an integer, real, or logical DATA variable with a Hollerith constant of more than five characters.
M-10 ILLEGAL DO LOOP CLOSE	Illegal statement terminating a DO loop. (See Section 4.3)
M-11 MORE DATA NEEDED - LITERAL TOO SHORT OR TYPE CONVERSION EXPECTED	The list of DATA constants defines fewer words than the list of DATA variables specifies. This may be due to a double precision or complex DATA variable defined with a Hollerith constant of less than six characters, or a double precision DATA variable defined with a real constant.
M-12 NON-INTEGER PARAMETER IN 'DO' STATEMENT	DO statement parameters must be integers. (See Section 4.3)
M-13 NON-INTEGER SUBSCRIPT	Array subscripts must be integer constants, variables, or expressions. (See Section 4.3)
M-14 ILLEGAL COMPARISON OF COMPLEX VARIABLES	The only comparison allowed of complex variables is .NE. or .EQ. (See Sections 2.2, 2.3)
M-15 TOO MANY CONTINUATION CARDS	More than 19 continuation cards. (See Section 1.1.2)

Table 12-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
M-16 NON-INTEGER I/O UNIT OR CHARACTER COUNT	The I/O unit variable of a READ/WRITE statement, or the character count variable of an ENCODE/DECODE statement, is not an integer variable. (See Sections 5.2.6, 5.2.7, 5.4)
M-17 SYSTEM ERROR-ROLL OUT OF RANGE	Compiler error. Report this message and its circumstances via a Software Trouble Report.
M-18 SYSTEM ERROR - NO MORE SPACE FOR RECURSIVE CALLS	The compiler's exit roll is too small to hold the return addresses for all the recursive subroutine calls this statement requires to be compiled. Break up the statement or reassemble the compiler with a larger exit roll parameter (EXLEN1=201_8 at present).
M-19 ILLEGAL USE OF STATEMENT LABEL	A GO TO or IF statement transfers to itself.
M-20 ILLEGAL RECURSIVE CALL	The statement function called itself. Recursive calls are illegal in the FORTRAN language.
M-21 DO LOOP INDEX ALREADY BEING USED IN OPEN LOOP	The index variable is already active as a DO loop index variable.
M-22 ATTEMPTING TO MODIFY ACTIVE DO LOOP INDEX	An attempt to modify a variable is being made to an active DO loop index, which cannot be modified.
M-23 INTEGER EXPRESSION EXPECTED	The expression must compute to an integer value.
M-24 ILLEGAL REPEAT COUNT	The repeat count for an item in a DATA statement is zero or negative.
M-25 DUPLICATE ARGUMENT IN OPEN/CLOSE STATEMENT	An argument has been specified twice in the same OPEN or CLOSE statement.
M-26 NOT A RECOGNIZED ARGUMENT TO OPEN/CLOSE	An invalid argument has been specified in an OPEN or CLOSE statement. Note that argument names cannot be abbreviated. Any attempt to do so will generate this error message.
M-27 UNIT IS A REQUIRED ARGUMENT FOR OPEN/CLOSE	An OPEN or CLOSE statement was given without a 'UNIT=' in the argument list.
EXCESSIVE COUNT	The number specified is greater than the maximum possible number of characters in a statement.
OPEN DO LOOPS	The list of statements are specified in DO statements but not defined.
UNDEFINED LABELS	The list of labels that do not appear in the label field.

FORTRAN also accepts the following codes in 7-bit ASCII — horizontal tab (011), line feed (012), form feed (014), and carriage return (015).

13.2 FORTRAN INPUT/OUTPUT

In addition to the arithmetic functions, the DECsystem-10 FORTRAN library (FORLIB) contains several subprograms which control FORTRAN I/O operations at runtime. The I/O subprograms are compatible with the DECsystem-10 monitor.

Table 13-2
DECsystem-10 FORTRAN Standard Peripheral Devices

Name	Mnemonic	Input/Output		Buffer Size In Words	Operation
		Formatted	Unformatted		
Card Punch	CDP	Yes	Yes	26	WRITE
Card Reader	CDR	Yes	Yes	28	READ
Disk (includes disk packs and drums)	DSK	Yes	Yes	128	READ/WRITE
DECtapes	DTA	Yes	Yes	127	READ/WRITE
Line Printer	LPT	Yes	No	26	WRITE
Magtape	MTA	Yes	Yes	128	READ/WRITE
Plotter	PLT	Yes	Yes	36	WRITE
Paper Tape Punch	PTP	Yes	Yes	33	WRITE
Paper Tape Reader	PTR	Yes	Yes	33	READ
Pseudo Terminal	PTY	Yes	No	17	READ/WRITE
Terminal-User	TTY	Yes	No	17	READ/WRITE
Terminal-Console	CTY	Yes	No	17	READ/WRITE

13.2.1 Logical and Physical Peripheral Device Assignments

Logical and physical device assignments are controlled by either the user at runtime or a table called DEVTB. The first entry in DEVTB. is the length of the table. Each entry after the first is a sixbit ASCII device name. The position in the table of the device name corresponds to the FORTRAN logical number for that device. For example, in Table 13-3, magnetic tape 0 is the 16th entry in DEVTB. Therefore, the statement

```
WRITE (16, 13)A
```

refers to magnetic tape 0. The last five entries in DEVTB. correspond to the special FORTRAN statements READ, ACCEPT, PRINT, PUNCH, and TYPE. Any device assignments may be changed by reassembling DEVTB.

If the user gives the Monitor command

```
ASSIGN DSK 16
```

prior to the running of his program, a file named FOR16.DAT would be written on the disk. Similarly, the monitor command

```
ASSIGN LPT 16
```

causes output to go to the line printer.

The FORTRAN programmer can make logical assignments such that each device has its own unique file as intended, but each can be on the disk. In order to use the devices available, the programmer can make assignments at runtime and assign the disk to those not available.

For example, the FORTRAN logical device numbers, e.g., 1=DSK, 2=CDR, 3=LPT, are used in the file name. The written file names are FOR01.DAT, FOR02.DAT, etc. The same is true for READ. For example, a WRITE(3,1)A,B,C, in the FORTRAN program generates the file name FOR03.DAT on the disk if the disk has been assigned LPT or 3 prior to running the program. (Note: REREAD rereads from the file belonging to the device last referenced in a READ statement, not FOR-6.DAT, as usual.) The programmer must, of course, realize his own mistake in assigning the disk as the TTY in the case that FOROTS tries to type out error messages or PAUSE messages.

More than one disk file may be accessed, without making logical assignments at runtime, by using logical device numbers 1, and 20 through 24 in the FORTRAN program. Logical device number 19 refers to logical device FORTR which must be assigned at runtime and accesses file name FORTR.DAT to maintain compatibility with the past system of default file name FORTR.DAT. In all cases when the operating system fails to find a file specified, an attempt will be made to read from file FORTR.DAT as before.

Table 13-3
FORTRAN Logical Device Assignments

Device/Function	Default Filename	FORTRAN Logical Unit Number	Use
Standard Devices*			
0	FORxx.DAT	00	ILLEGAL
DSK		01	DISK
CDR		02	Card Reader
LPT		03	Line Printer
CTY		04	Console Teletype
TTY		05	User's Teletype
PTR		06	Paper Tape Reader
PTP		07	Paper Tape Punch
DIS		08	Display
DTA1		09	DECtape
DTA2		10	
DTA3		11	
DTA4		12	
DTA5		13	
DTA6		14	
DTA7		15	DECtape
MTA0		16	Magnetic Tape
MTA1		17	
MTA2		18	
FORTR		19	Assignable Device
DSK		20	DISK
DSK		21	
DSK		22	
DSK		23	
DSK		24	
DEV1		25	Assignable Devices
DEV2		26	
DEV3		27	
DEV4		28	
DEV5		29	
DEV63	FOR63.DAT	63	DISK
Default Devices (inaccessible to the user)			
REREAD	Current file	−6	REREAD statement
CDR	FORCDR.DAT	−5	READ statement
TTY	FORTTY.DAT	−4	ACCEPT statement
LPT	FORLPT.DAT	−3	PRINT statement
PTP	FORPTP.DAT	−2	PUNCH statement
TTY	FORTTY.DAT	−1	TYPE statement

*The total number of standard devices permitted is an installation parameter.

13.2.2 ASCII Data Files

Each record within an ASCII data file consists of a set of contiguous 7-bit characters; each set is terminated by a vertical paper-motion character (i.e., form feed, vertical tab, or line feed). All ASCII records start on a word boundary; the last word in a record is padded with nulls, if necessary, to insure that the record also ends on a word boundary. Logical records may be split across physical blocks. There is no implied maximum length for logical records.

NOTE

On sequential input, FOROTS does not require conformation to word boundaries: it reads whatever it sees. However, any file that is written by FOROTS will conform to the foregoing format requirements.

13.2.3 FORTRAN Binary Data Files

Each logical record in a FORTRAN binary data file contains data which may be referred to by either a READ or WRITE statement in the program being executed. A logical record is preceded by a control word and may have one or more control words embedded within it. In FORTRAN binary data files, there is no relationship between logical records and physical device block sizes. There is no implied maximum length for logical records.

13.2.3.1 Format of Binary Files - A FOROTS binary file may contain three forms of Logical Segment Control Words (LSCW). These LSCW's give FOROTS the ability to distinguish ASCII files from binary files.

	LSCW	
START	001+	the number of words in the segment (exclusive of any "END" LSCW's)
CONTINUE	002+	indicates that the segment of a disk block boundary continues
END	003+	number of words in the preceding segment including LSCW's

If the access specified for a file (through the OPEN statement ACCESS=parameter is 'SEQIN', 'SEQOUT', or 'SEQINOUT', all three LSCW's may appear in a record. If the access specified is 'RANDIN', 'APPEND', or 'RANDOM', all records are of the same length, and there are no CONTINUE LSCW's.

13.2.4 Mixed Mode Data Files

FOROTS permits files containing both ASCII and binary data records to be read. Mixed files may be accessed in either sequential or random access mode. Logical ASCII and binary records have the same format as described in the preceding paragraphs.

13.2.5 Image Mode Files

The image data transfer mode is a buffered mode in which data is transferred in a blocked format consisting of a word count located in the right half of the first data word of the buffer followed by the number of 36-bit data words. The devices which permit image data transfers, and the form in which the data is read or written, are listed below.

Device	Data Forms
Card Punch	In image mode, each buffer contains three 12-bit bytes. Each byte corresponds to one card column. Since there is room for 81 columns in the buffer (3 X 27) and there are only 80 columns on a card, the last word contains only two bytes of data; the third byte is thrown away. Image causes exactly one card to be punched for each output. The CLOSE punches the last partial card and then punches an EOF card.
Card Reader	All 12 punches in all 80 columns are packed into the buffer as 12-bit bytes. The first 12-bit byte contains column 1. The last word of the buffer contains columns 79 and 80 as the left and middle bytes, respectively. Cards are not split between two buffers.
Disk	Data is written on the disk exactly as it appears in the buffer. Data consists of 36-bit words.
Magnetic Tape	Data appears on magnetic tape exactly as it appears in the buffer. No processing or checksumming of any kind is performed by the service routine. The parity checking of the magnetic tape system is sufficient assurance that the data is correct. Normally, all data, both binary and ASCII, is written with odd parity and at 800 bits per inch unless changed by the installation.
Paper Tape Punch	Binary words taken from the output buffer are split into 6-bit bytes and punched with the eighth hole punched in each frame. There is no format control or checksumming performed by the I/O routine. Data punched in this mode is read back by the paper tape reader in the same mode.

Paper Tape Reader	Characters not having the eighth hole punched are ignored. Characters are truncated to six bits and packed six to the word without further processing. This mode is useful for reading binary tapes having arbitrary blocking format.
Plotter	Six 6-bit characters per word are transmitted to the plotter exactly as they appear in the buffer.

13.3 RANDOM ACCESS PROGRAMMING

In random access programming, data is obtained from (or placed into) storage, where the time required for this access is independent of the location of the data most recently obtained from (or placed into) storage. Random access programming allows a programmer to access any record within a file with a single READ or WRITE statement independent of the location of the previously accessed record within that file. For example, a programmer may read or write only the 10th record in a file if he wishes. Random I/O is desirable when only a few records in a large file are to be accessed, or when a file is to be read or written in a non-sequential manner, as in a sort.

Random access applies only to data files on the disk with fixed-length record sizes. Any fixed-length record file (formatted or unformatted) which has been written on the disk with FORTRAN or with PIP using the A switch may be read or rewritten non-sequentially.

13.3.1 How to Use Random Access

A programmer may directly access fixed-length records in a disk file by defining the structure of the file with an OPEN statement (see Chapter 8) and then specifying the record he wishes to access with a READ or WRITE statement.

I/O begins when the random WRITE or READ is specified in the correct format. (See Sections 5.2.6 and 5.2.7.)

13.3.2 Restrictions

A number of restrictions are imposed in random access programming:

A. Mixed formatted and unformatted files are not accessible randomly.

B. Before random I/O is performed through a READ or WRITE statement, the file must be properly defined through execution of an OPEN statement.

C. All FORTRAN data files must be created by FORTRAN or PIP with the A switch.

D. The records within the file must be of a fixed length.

E. Random access is used for disk files only.

F. Access to files is controlled by the file protection scheme in effect at each installation. (See the DECsystem-10 Monitor Calls Manual for a discussion of file access privileges.)

APPENDIX A

LIMITATIONS IN THE FORTRAN IV (F40) COMPILER

The FORTRAN IV (F40) compiler is intended to fill the need for a small and reliable FORTRAN compiler. Its purpose is to provide an alternative to the larger FORTRAN-10 compiler when optimization is not important or when there is a need for repeated compilation for debugging purposes. With the addition to the FORTRAN IV (F40) compiler of the OPEN and CLOSE file control statements, the user has full access to the FOROTS object time system from an F40 compiled program.

There are, however, some continuing problems with the F40 compiler. These problems lead to incorrectly compiled code under certain infrequent circumstances. Below is a description of known limitations in version 27 of the FORTRAN IV (F40) compiler.

A.1 CODE GENERATION ERRORS

1. A partial result in a CALL statement may be handled incorrectly. For example:

   ```
   CALL FUNC(INT+1, REAL*(INT+1))
   ```

   will cause the second argument to be compiled incorrectly.

2. Nested subscripts in simple assignment and I/O statements may be compiled incorrectly. For example:

   ```
   R(I(J,K),K) = INT
   ```

   does not produce the same results as

   ```
   M = I(J,K)
   R(M,K) = INT
   ```

3. Subscripts repeated in an I/O statement will cause incorrect code for the second occurrence of the subscript. For example, in the statement

   ```
   WRITE (IJ(N), 7) (B(K),K=1,5), L(N)
   ```

   the array reference L(N) fails.

4. Implied DO loops repeated in I/O statements may cause bad code. For example, in the statement

   ```
   READ (1,1) (A(L,J),J=1,5), A(2,5), (A(K,J),K=5,10)
   ```

the second loop involving K will fail.

5. A double precision function which computes a single precision result may fail to reset the second word of its result. For example:

```
DOUBLE PRECISION DFNC, D
DFNC(X) = X+1
D = DFNC(Y)
```

will set up AC 0 but not AC 1.

6. In general, any long, mixed-mode arithmetic statement may produce bad code, usually due to improper register usage.

A.2 ERROR CONDITIONS WHICH DO NOT GENERATE CORRECT ERROR MESSAGES

1. If a name is entered into COMMON and then declared as a line function, the function will take precedence. For example:

```
COMMON FINE
FINE(X) = X + 1
```

2. A repeated formal name in a statement function will generate the wrong error message:

```
FOO (X,Y,Z,Y) = X + Y * Z
  ↑
STATEMENT KEYWORD NOT RECOGNIZED
```

APPENDIX B

SUMMARY OF DDT FUNCTIONS

This appendix gives a summary of functions performed by the Dynamic Debugging Technique (DDT), which is used in conjunction with FORTRAN programming. It is not intended to teach DDT to the user, but rather to serve as a reference guide for users who are already familiar with DDT.

B.1 TYPE-OUT MODES

The following are used to set the type-out mode:

	Type	Sample Output(s)
Symbolic instructions	$S	ADD 4, TAG+1 ADD 4, 4002
Numeric, in current radix	$C	69. 105
Floating point	$F	0.125E−3
7-bit ASCII text	$T	PQRST
SIXBIT text	$6T	TSRQPO
RADIX50	$5T	4 DDTEND
Halfwords, two addresses	$H	4002,,4005 X+1,,X+4
Bytes (of n bits each)	$NO	$80 COULD YIELD 0,14,237,123,0

B.2 ADDRESS MODES

The following are used to set the address mode for typeout of symbolic instructions and half words (see examples above):

Relative to symbolic address	$R	TAG+1
Absolute numeric address	$A	4005

B.3 RADIX CHANGE

The following is used to change the radix of numeric type-outs

to n (for n≥2):	$NR	$2R COULD YIELD 1101011000000100000000000011100101100

B.4 PREVAILING VS. TEMPORARY MODES

The following are used in prevailing vs. temporary modes:

To set a temporary type-out or address mode or a temporary radix as shown in the commands above, type	$	$C $10R
To set a prevailing type-out or address mode on a prevailing radix, in the commands above, substitute	$$	$$C $$10R
To terminate temporary modes and revert to prevailing modes, type a carriage return	↵	
Initial prevailing (and temporary) modes are	$$S $$R $$8R	

B.5 STORAGE WORDS

The following are used to examine storage words:

To open and examine the contents of any address in current type-out mode	adr/	LOC/	254020,,DDTEND
To open a word, but inhibit the type out of contents	adr!	LOC!	
To open and examine a word as a number in the current radix	adr[	LOC[	254020,,3454

To open and examine a word as a symbolic instruction	adr]	LOC]	JRST @DDTEND
To retype the last quantity typed (particularly used after changing the current type-out mode)	;	$60;	25,40,20,00,34,54
		$6T;	5%0 <L

B.6 RELATED STORAGE WORD

The following are used to examine related storage words:

To close the current open word (making any modification typed in) and to open the following related words, examining them in the current type-out mode:	
To examine ADR+1	↓ (line feed)
To examine ADR−1	↑ (or backspace, on the Teletype Model 37)
To examine the contents of the location specified by the address of the last quantity typed, and to set the location pointer to this address	→| (TAB)
To examine the contents of address of last quantity typed, but not change the location pointer	\ (backslash)
To close the currently open word, without opening a new word, and revert to permanent type-out modes	↵ (carriage return)

B.7 ONE-TIME ONLY TYPEOUTS

The following typeouts occur only one time:

To repeat the last typeout as a number in the current radix	=

To represent the address of the search mask register	$M
To represent the address of the saved flags, etc.	$I
To represent the pointers associated with the nth breakpoint	$nB

B.11 ARITHMETIC OPERATORS

The following arithmetic operators are permitted in forming expressions:

Two's complement addition	+
Two's complement subtraction	−
Integer multiplication	*
Integer division (remainder discarded)	' (apostrophe)

B.12 FIELD DELIMITERS IN SYMBOLIC TYPE-INS

The following are field delimiters:

To delimit op-code name	one or more spaces	JRST SUBRTE
To delimit accumulator field	, (comma)	
To delimit two halfwords	left,,right	−6,,BEGIN−1
To delimit index register	( )	
To indicate indirect addressing	@	

B.13 BREAKPOINTS

The following are used for breakpoints:

To set a specific breakpoint n(1<n<8)	adr$nB	CAR$8B
To set the next unused breakpoint	adr$B	303$B
To set a breakpoint with automatic proceed	adr$$nB adr$$B	CAR$$8B 303$$B

To set a breakpoint which will automatically open and examine a specified address,x	x,,adr$nB x,,adr$B x,,adr$$nB x,,adr$$B	AC3,,Z+6$5B AC4,,ABLE$B AC3,,Z+6$$5B AC4,,ABLE$$B
To remove a specific breakpoint	0$nB	0$8B
To remove all breakpoints	$B	$B
To check the status of breakpoint n	$nB/	
To proceed from a breakpoint	$P	$P
To set the proceed count and proceed	n$P	25$P
To proceed from a breakpoint and thereafter proceed automatically	$$P n$$P	$$P 25$$P

B.14 CONDITIONAL BREAKPOINTS

The following are used for conditional breakpoints:

To insert a conditional instruction (INST), or call a conditional routine, when breakpoint n is reached	$nB+1/ $2B+1/0	INST CAIE 3,100

If the conditional instruction does not cause a skip, the proceed counter is decremented and checked. If the proceed count ≤ 0, a break occurs

If the conditional instruction or subroutine causes one skip, a break occurs.

If the conditional instruction or subroutine causes two skips, execution of the program proceeds.

B.15 STARTING THE PROGRAM

The following commands are used to start the program:

To start at the starting address in JOBSA	$G	$G
To start, or continue, at a specified address	adr$G	LOC$G
To execute an instruction	inst$X	JRST 2, @JOBOPC$X returns to program after ↑C and DDT commands

B.16 SEARCHING

The following commands are used for searching:

To set a lower limit (a), an upper limit (b), a word to be searched for (c), and search for that word	a<b>c$W	200<250>0$W
To set limits and search for a not-word	a<b>c$N	351<731>0$N
To set limits and search for an effective address	a<b>c$E	401<471>LOC+6$E
To examine the mask used in searches (initially contains all ones)	$M/	$M/ −1
To insert another quantity n in the mask	n$M	777000777777$M

B.17 UNUSED FUNCTIONS

The following is unused:

$U

B.18 ZEROING MEMORY

The following are used for zeroing memory:

To zero memory, except DDT, locations 20–137, and the symbol table	$$Z
To zero memory locations FIRST through LAST inclusive	FIRST<LAST $$Z

B.19 SPECIAL CHARACTERS

The following special characters are used in DDT typeouts:

Breakpoint stops		
Break caused by conditional break instruction	>	
Break because proceed counter ≤0	>>	
Undefined symbol cannot be assembled	U	
Half-word type-outs	left,,right	401,,402
Unnormalized floating-point number	#1.234E+27	#1.234E+27
To indicate an integer is decimal. The decimal point is printed	$10R 77=63.	
Illegal command	?	
If all eight breakpoints have been assigned	?	
RUBOUT echo	XXX	

B.20 PAPER TAPE COMMANDS

The following commands are available only in EDDT:

To punch a RIM10B loader	$L	
To punch checksummed data blocks where ADR1 is the first, and ADR2 is the last location of the data	ADR1<ADR2	TAPE
To punch data as above, except that more than two consecutive locations containing zeros are not punched.	ADR1<ADR2	$ TAPE
	(TAPE is ↑R)	
To punch a one-word block to cause a transfer to adr after the preceding program has been loaded from paper tape	adr$J	

INDEX

DECsystem-10
FORTRAN IV (F40)
Programmer's Reference Manual
DEC-10-LFLMA-B-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form

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

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or
Country

If you do not require a written reply, please check here. ☐

---------- 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

Software Communications
P. O. Box F
Maynard, Massachusetts 01754