digital

H. Bergkvist

fortran IV

language manual

pdp15

digital equipment corporation

PDP-15
FORTRAN IV LANGUAGE
PROGRAMMER'S REFERENCE MANUAL

For additional copies, order No. DEC-15-GFWA-D from Program Library, Digital Equipment Corporation, Maynard, Massachusetts 01754 Price $5.00

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Edition August 1971

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

ILLUSTRATIONS

TABLES

PREFACE

This manual describes the elements, syntax and use of the FORTRAN IV language as implemented for the PDP-15 computer. Three versions of the PDP-15 compiler are available; their use is governed by the hardware/software configuration of the system on which FORTRAN is to be run. The most comprehensive version of PDP-15 FORTRAN IV is described in this manual. See Appendix C for overall outlines and descriptions, and tabularized descriptions of the differences between the various versions of the FORTRAN IV compilers and their associated libraries.

All versions of PDP-15 FORTRAN IV are based on USASI Standard FORTRAN (X3.9-1966); the following features were added to the PDP-15 version of FORTRAN IV:

ENCODE/DECODE Statements, data-directed Input/Output, multiple entries and returns from subroutines.

double integer constants and variables, part-word notation for Arithmetic statements;

direct access input/output statements.

END and ERR input-output options

octal format descriptors

The following standard features are not available:

complex arithmetic

adjustable arrays (done via subroutine in this version)

One additional difference from the standard is that the blank COMMON and labeled COMMON are treated the same.

A companion manual, "PDP-15 FORTRAN IV OPERATING ENVIRONMENT", order code DEC-15-GFZA-D, describes the system software facilities needed to support the various versions of the PDP-15 FORTRAN IV compiler and hardware features which affect the FORTRAN programmer. Included in this manual are descriptions of the FORTRAN IV Object Time System (OTS) and Science Library.

CHAPTER 1
BASIC ELEMENTS OF A FORTRAN-IV PROGRAM

A FORTRAN-IV source program is a sequence of symbolic statements which are translated by the FORTRAN-IV compiler into an object program; that is, a program which may be executed by a computer. A statement, the basic unit of expression in a FORTRAN source program, may represent computer instructions and program data or may provide the compiler with instructions required in the translating process, such as the size of an array, the number of times a loop is to be executed, or whether the program is a subroutine to be called by other programs.

A FORTRAN statement consists of a command portion which characterizes its function and may, in addition, require arguments. For example, the GO TO statement, which transfers control from one statement to another, requires an argument specifying the source-program statement which is the target of the transfer. The five functional categories into which all FORTRAN-IV statements fall are given below.

Category	General Function
Assignment Statements	Assign values to symbolic representations
Control Statements	Govern the sequence in which operations are performed
Specification Statements	Describe data the object program will process
Subprogram Statements	Establish subprograms
Data Transmission Statements	Govern the transfer of information between the computer and peripheral devices (I/O)

The format of each statement and the arguments required for each are described in detail in subsequent chapters.

1.1 THE CHARACTER SET

The character set from which FORTRAN statements may be constructed consists of the 26 letters (A-Z), the 10 digits (0-9), and the following special characters:

[	Left bracket	-	Minus
]	Right bracket	*	Asterisk
:	Colon	/	Slash
;	Semi-colon	(	Left parenthesis
#	Sharp sign	)	Right parenthesis
'	Single quote	,	Comma
	Blank	.	Decimal point
=	Equals	$	Dollar sign
+	Plus	"	Quotes

Other characters may appear only in a text string (Hollerith constant).

1.2 PROGRAM STRUCTURE

A FORTRAN source program's beginning is simply the first statement encountered. Its end must be denoted by an END statement consisting of the characters END.

Comments may precede the body of the program or be inserted between statements by means of a comment line which begins with the character C and is followed by text.

The over-all rule for ordering statements is that non-executable statements (no machine code generated) must precede executable statements. The precise order in which statements may appear is given in Table 1-1 below.

Table 1-1
Sequence Rules for FORTRAN Statements

Order	Statements
1	BLOCK DATA; FUNCTION; SUBROUTINE
2	IMPLICIT
3	INTEGER; REAL; LOGICAL; DOUBLE PRECISION; DOUBLE INTEGER
4	DIMENSION
5	COMMON
6	EQUIVALENCE; EXTERNAL
7	DATA
8	Statement functions
9	All other

The form in which statements and comments are entered is governed by an 80-character line on a standard FORTRAN coding form, as shown in Figure 1-1. If the source program is input in card form, the columns on the coding sheet correspond to card columns. If paper tape is used, the columns refer to characters.

Each line in a program is organized into the following fields, some of which may be blank:

Field Name	Columns	Contents
Statement number	1-5	A decimal number from 1-9999 identifying the statement. May be in any order.
Line continuation field	6	If non-blank, indicates that the statement portion is a continuation of the preceding line.
Statement field	7-72	FORTRAN statement or portion thereof.
Identification field	73-80	Ignored by the compiler, filled at the user's discretion.

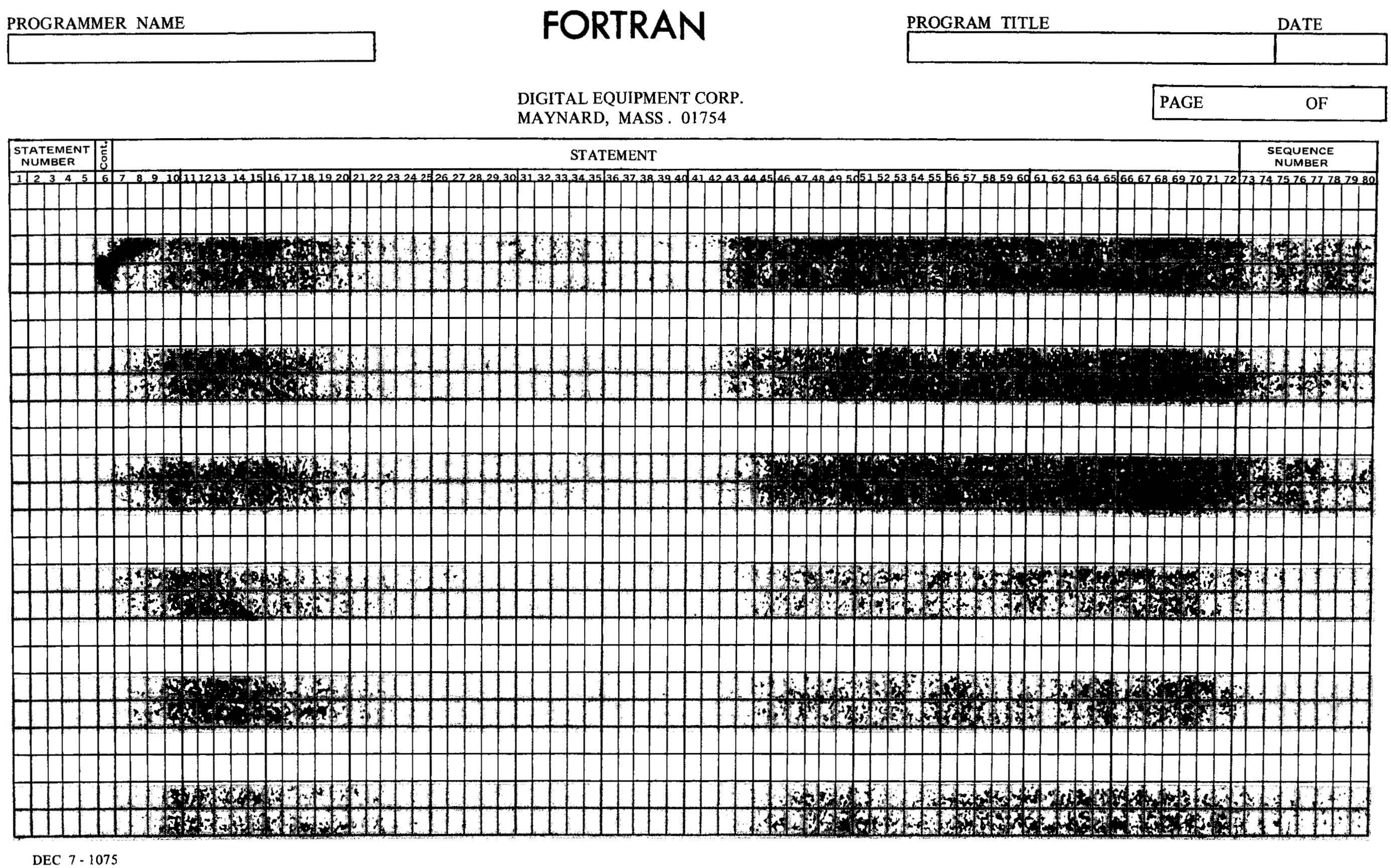
A comment line is indicated by a C in column 1; comment text may be placed anywhere in columns 2-72.

With the exception of the DO statement, which must be on one line, any statement may have as many continuation lines as desired. Any statement except the Arithmetic statement and the Arithmetic IF statement may be broken at any point. Continuation rules for these two statements are described in Chapters 2 and 3, respectively. In general, blanks may be freely imbedded within statements to improve their legibility.

For non-card input, the first character is equivalent to the first column and a line is terminated by a carriage return. A statement field may begin with the seventh character or may be indicated by a TAB followed by an alphabetic character. A continuation line may begin with the sixth character or may be indicated by a TAB followed by a numeric character.

1.3 EXPRESSING DATA VALUES

Program data may be expressed in a variety of ways in a FORTRAN-IV program. The basic units, constants and variables, represent single values - a constant has the same value throughout program execution, a variable has whatever value it is currently assigned. New values may be computed from known values of these units via expressions, which are composed of constants, variables and FORTRAN operators which indicate the computation to be performed.


PROGRAMMER NAME

FORTRAN

PROGRAM TITLE

DATE

DIGITAL EQUIPMENT CORP.
MAYNARD, MASS. 01754

PAGE OF

STATEMENT NUMBER	Cont.	STATEMENT	SEQUENCE NUMBER
1 2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80

DEC 7 - 1075

Figure 1-1 DEC FORTRAN-IV Coding Form

1.3.1 Constants

A constant is, as indicated above, a value which does not change from one execution of a program to another. Six types of constants, each representing a different PDP-15 internal data format, may appear in a FORTRAN-IV program. These are: INTEGER, DOUBLE INTEGER, REAL, DOUBLE-PRECISION, LOGICAL, and HOLLERITH. The form of each is described below.

INTEGERS - An integer constant represents a single word of PDP-15 storage. Its value may be expressed in the source program as a decimal or octal number.

A decimal integer consists of one to six decimal digits with no decimal point. A negative quantity is indicated by a minus sign. A positive quantity may optionally be preceded by a plus sign. All of the following are legal decimal constants:

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

Leading zeroes are ignored; thus -0010 is equivalent to -10. The magnitude of the integer must be less than or equal to 131071_{10} (i.e., $2^{17}-1$).

An octal integer is indicated by a sharp sign (#) followed by one to six octal integers. The following are legal octal integers:

```
#777
#1
#20137
```

DOUBLE INTEGERS - A double integer represents two words of storage (1 sign bit and 35 bits of magnitude) and has a range between $-34,359,738,367_{10}$ and $+34,359,738,367_{10}$. The notation D, preceding an octal integer, indicates that it is double integer, as in: #D7777777. Note that an octal integer value not preceded by D may be assigned to a double integer variable (i.e., DI = #130000). However, the value of the assigned integer must NOT exceed the maximum size permitted integers (i.e., 377777_8). If an assigned octal integer does exceed the maximum permitted value, its most significant digits will be truncated before it is assigned to the double integer variable regardless of the fact that the double integer would accept its original value. Decimal integers whose absolute values exceed 131071_{10} are taken as double integers. The following are examples of legal double integers:

```
#D400000
 141520
#D0
#D400000000
 400000
```

REAL CONSTANTS - A real constant is a string of decimal digits with a decimal point, optionally followed by a decimal exponent. It may be a whole number (i.e., 10.), a fraction (i.e., .10), or a mixed format number (i.e., 10.10). The programmer may supply any number of digits in a real constant but only the leftmost seven are significant. Thus,

    10.111550

and

    10.111552

are equivalent.

A plus or minus sign may precede the constant - plus being optional for positive quantities. All of the following are valid:

```
325.
0.0
+325.
-9.8
999999.0
999999999.
```

If a decimal exponent is present, it is indicated by the letter E immediately following the constant. The decimal point may be omitted if immediately followed by an exponent. The exponent itself, which immediately follows the E, is an optionally signed one- or two-digit number indicating the appropriate power of 10, as in:

```
5.E-3  (i.e., 0.005)
5.0E3  (i.e., 5000.)
5E2    (i.e., 500.)
```

The adjusted absolute value of the exponent cannot exceed 75. Thus, the constant .99999E75 is legal, but 999.999E73 is not.

A real constant occupies two words of PDP-15 storage in the following arrangement.

Low order mantissa	Exponent (2's complement)
0 8	9 17

Sign	High order mantissa
0	1 17

The mantissa and its sign are represented in signed magnitude form.

DOUBLE-PRECISION CONSTANTS - A double-precision constant is interpreted like a real constant but with greater accuracy (nine-digit). It is written as a string of decimal digits, including a decimal point immediately followed by the letter D and a signed decimal exponent no greater than 75. (Plus is optional.) The field following the D may not be blank but may be zero. For example:

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

Double-precision constants are stored in three words arranged as follows.

Exponent (2's complement)
0 ... 17

Sign of mantissa	High order mantissa
0	1 ... 17

Low order mantissa
0 ... 17

The mantissa and its sign are represented in signed magnitude form.

LOGICAL CONSTANTS - There are two logical constants - .TRUE., which is stored as 777777_8, and .FALSE., which is stored as 0. Logical quantities may be operated upon both by arithmetic and logical operators yielding, respectively, arithmetic and logical results.

HOLLERITH CONSTANTS - A Hollerith constant is a string of 1 to 5 characters. They are packed in 7-bit ASCII in two words of storage with the rightmost bit of the second word always zero. A Hollerith constant may be used in CALL and DATA statements and, if the programmer exercises caution, in an Arithmetic statement. There are four forms for writing a Hollerith constant:

(1) nH characters

where n is the number of characters (1 to 5). Examples of this format are:

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

When the above notation is used, the string is stored as a real constant.

(2) 'characters'

Examples of this are:

'A'
'A$CD'

If quotation marks are to be included in the character string itself, this may be indicated by having two single quotes in sequence, as in:

'A''''CD'

which stores the string A''CD. This and the following forms of string constants are stored as unsigned double integers and may be used wherever a double integer may. Double quotes may be used instead of single quotes.

(3) "characters"

Examples of this form are:

"A BC"

which yields string A BC and

"A" " " "B"

which yields string A" "B.

(4) $characters$

Examples are:

A
$A$$CD$

where the second example yields the string A$CD.

Blanks within a Hollerith constant are considered as characters. Thus,

'AB'

is a three-character string.

1.3.2 Variables

The term variable refers to a symbolic name which represents a location in memory and to the values which are stored there during program execution. A variable name in FORTRAN is a string of from one to six characters, the first of which must be alphabetic. Thus, ALPHA, MAX, A34, and A are legal variable names while 2A and MAXIMUM are not.

The kind of value which may be associated with a given variable name must be specified so that appropriate storage is allocated. This specification is referred to as the mode of the variable, where mode is INTEGER, DOUBLE INTEGER, REAL, DOUBLE PRECISION, or LOGICAL.

The implicit mode assumptions of the compiler are that all variables beginning with the letters I through N are integers and all others are real. For any modes other than integer and real, the programmer is responsible for establishing a variable's mode. The programmer may also establish a different set of mode assumptions via the IMPLICIT statement or explicitly declare a variable's mode via one of the FORTRAN-IV mode-declaration statements. Chapter 5 describes these statements in detail.

A variable may also name an array, an ordered set of data whose elements are referred to by means of subscripted variables. A subscripted variable has the form:

V(n)

where n is a list of from one to three expressions which yield positive (non-zero) integer values.

The variable name is, in effect, the name of the entire array. For example, the subscripted variable:

A(3)

refers to the third element of a one-dimensional array named A. Arrays in FORTRAN-IV may have up to three dimensions; consequently, subscripted variables (also referred to as array elements) may have up to three subscripts as in:

A(1,2,2)

which represents the value located in the first row, second column, and second plane of a three-dimensional array named A.*

A variable which represents an array must be assigned adequate storage to contain all elements. To ensure this, the programmer must provide dimensioning information giving the maximum value each of the array's subscripts can obtain. This may be done via several of the specification statements described

---

*Arrays are stored in column order in ascending absolute storage locations. For example, a 2 by 2 by 2 array is stored in the following sequence:

A(1,1,1)
A(2,1,1)
A(1,2,1)
A(2,2,1)
A(1,1,2)
A(2,1,2)
A(1,2,2)
A(2,2,2)

in Chapter 5. Note that when an array has been defined to have a certain number of dimensions, all references to it must contain that number of subscripts. Note also that an array must be of a given mode; i.e., each element is of the same specified mode.

1.3.3 Expressions

The term expression may broadly refer to the whole range of value descriptions which can be made in FORTRAN. This includes the primary units discussed so far (constants and variables, function references discussed in Chapter 5), and combinations which relate several units via FORTRAN operators. The value of this latter type of expression is, in reality, the result of the computations represented by its operators.

Two types of compound expression - arithmetic and logical - may be constructed in FORTRAN. Either type may be enclosed in parentheses and function as a primary unit (or operand) in another expression.

ARITHMETIC EXPRESSIONS

An arithmetic expression is any configuration which yields a numeric value. It may be a single arithmetic unit or combination of arithmetic operands and the arithmetic operators given below.

Operator	Operation
+	Addition (or unary plus)
-	Subtraction (or unary minus)
*	Multiplication
/	Division
**	Exponentiation

An operand may be a constant, variable, function reference, or a parenthesized expression.

The following are examples of legal arithmetic expressions:

```
2.71828
XYZ
A+B*C
(A+B)*C
```

Precedence of Operations

Arithmetic operations are carried out according to the following rules of precedence:

(1) function reference

(2) ** (exponentiation)

(3) unary minus

(4) * (multiplication), / (division)

(5) + (addition), - (subtraction)

At the same precedence level, operations are carried out from left to right. For example, the expression:

```
-I + J/2 * 10 + SQRT(A) ** 3
(2)  (3)        (1)
     (4)
```

is evaluated as follows:

(1) the square root of A is raised to a power of 3;

(2) the value of I is complemented;

(3) J is divided by 2 and

(4) the result multiplied by 10.

The remaining operations [(2) + (4) + (1)] are carried out from left to right.

When an expression enclosed in parentheses appears within an expression, it is evaluated before being used as an operand, thus overriding the rules of precedence. Some examples are:

Regular Precedence	With Parentheses
4 + 2 ** 2 = 8	(4 + 2) ** 2 = 36
8 - 4 * 2 = 0	(8 - 4) * 2 = 8
-10 + 4 = -6	-(10 + 4) = -14
18/2 * 3 = 27	18/(2 * 3) = 3
-1 ** 2 = -1	(-1) ** 2 = 1

Mode of Expressions

Expressions, like variables, have modes. In the case of an expression, however, the mode determines its accuracy. When an expression is composed of operands of the same mode, this mode applies to the entire expression. For example, an expression consisting of integer constants or variables is in integer mode, and so on. Different mode operands may also be used to form expressions. All legal combinations and the resultant mode are given in Table 1-2.

Table 1-2
Modes of Mixed Expressions

A plus sign represents any of the operators (+, -, *, or /).

Expression	Mode
I+I	Integer
R+R	Real
I+DI DI+I DI+DI	Double integer
R+D D+R D+D	Double precision
I**I	Integer
R**I or DI R**R	Real
R**D D**I or DI D**R D**D	Double precision
I**DI DI**I DI**DI	Double integer

LOGICAL EXPRESSIONS

Logical expressions consist of any configuration which yields a logical value (i.e., .TRUE. or .FALSE.). This may be a combination of arithmetic expressions and relational operators, a logical constant or variable, or a combination of logical operands and logical operators.

An expression using relational operators has the form:

A operator B

where A and B are arithmetic expressions and operator is one of those listed below.

Relational Expression	Relation
A .LT. B	A less than B
A .LE. B	A less than or equal to B
A .EQ. B	A equal to B
A .NE. B	A not equal to B
A .GT. B	A greater than B
A .GE. B	A greater than or equal to B

An expression has the value .TRUE. if the relation expressed is true; otherwise, it has the value .FALSE. For example, assuming a variable A with the value 10 and a variable B with the value 20:

A .LT. B has the value .TRUE.

while

A .GE. B has the value .FALSE.

The following mode combinations are permitted in a relational expression:

Mode	May be Related to
Integer	Integer, double integer
Double integer	Double integer, integer
Real	Real, double precision
Double precision	Double precision, real

Logical operators can combine logical or integer operands. The operators and their meanings are given below (T indicates a value of .TRUE. for logical operands and non-zero for integers, F, .FALSE. or zero).

Logical Operator	Meaning	Examples	
		Expression	Result
.NOT.	logical negation	.NOT. T	F
		.NOT. F	T
.AND.	logical and	T .AND. T	T
		T .AND. F	F
		F .AND. T	F
		F .AND. F	F
.OR.	inclusive or	T .OR. T	T
		T .OR. F	T
		F .OR. T	T
		F .OR. F	F
.XOR.	exclusive or	T .XOR. T	F
		T .XOR. F	T
		F .XOR. T	T
		F .XOR. F	F

Logical expressions are, like arithmetic expressions, evaluated according to precedence rules. These are:

(1) relationals

(2) .NOT.

(3) .AND.

(4) .OR.

(5) .XOR.

Thus, T .XOR. F .AND. F yields the value .TRUE.

The arithmetic operands of a relational expression are evaluated before the relation is tested. At the same level of precedence, operations are carried out from left to right. In addition, logical expressions can be parenthesized, thus overriding precedence. For example:

F .AND. F .XOR. T = .TRUE.

but

F .AND. (F .XOR. T) = .FALSE.

Following .NOT. a compound expression must be parenthesized, as in:

.NOT. (F .AND. T)

which has the value .TRUE.

CHAPTER 2
ASSIGNMENT STATEMENTS

An assignment statement permits the programmer to assign a value to a symbolic name. Two FORTRAN-IV statements, the Arithmetic statement and the ASSIGN statement, perform this function. In the case of the Arithmetic statement, the symbolic name identifies a variable or an array element and the value is a constant data value or the result of a computation. For the ASSIGN statement, the name is a symbolic address label which may be referred to by a GO TO or arithmetic IF statement (Chapter 3) and the value is a statement number within the source program. For either statement, the value assigned to a particular name may be changed by subsequent assignment statements; other statements or expressions which refer to them will automatically operate on the most recent value assigned.

2.1 THE ARITHMETIC STATEMENT

General Form	var = value or array (i) = value
Where	value = any FORTRAN constant or expression
Examples	COUNT = 1 TABLE (COUNT) = 100 COUNT = COUNT + 1 TABLE (COUNT) = 200
Effect	The value to the right of the equal sign is assigned to the variable or array element to the left

Note that the equal sign in an Arithmetic statement indicates replacement rather than equivalence; this permits constructions such as COUNT = COUNT + 1. If an Arithmetic statement requires a continuation line, the = sign must appear on the first line.

If an expression of one mode is assigned to a variable of another mode, the expression is converted before assignment. That is, integers may be converted to real, real to double-precision, and so on. There are, however, situations in which the value obtained will be meaningless. For example, if the integer variable I is assigned the value of the double-integer variable J, when J = 100, the assignment will be as expected. When J = 10000000, however, an unpredictable value assignment will result.

Conversions between logical and integer obey the following convention. Any non-zero integer is .TRUE. (777777), zero is .FALSE. (000000).

In addition to the basic Arithmetic statement form, the programmer may use a part-word notation of the form:

    [m : n]

where m and n are integer constants indicating a range from 0 to 35 ($0 \le m \le n \le 35$). This construction may optionally follow any variable, array element, or parenthesized expression in the value portion of an Arithmetic statement (to the right of =) and/or the variable or array element being assigned. In the former case, the expression will be of type integer if $(n-m) \le 16$ and type double integer if $(n-m) \ge 17$; its value is bits m through n of the actual value (right adjusted). For example, the statement:

```
A=#2300[6:11]
```

assigns A the value 23, and

```
A=#2300[6:8]
```

assigns A the value 2. If A were a double integer, the statement

```
A=#2300[0:29]
```

would assign A the value 23. Note that #2300 is represented internally as 002300.

If this notation is used to the left of the equal sign, it indicates that only bits m through n of the variable are to be replaced by the value of the right hand side. For example, if the integer variable IVAR had previously been assigned the octal value 77, the statement:

```
IVAR[9:11]=#1
```

would make the new value of IVAR the octal integer 177. Also, the statements:

```
IVAR=100
IVAR[9:11]=IVAR+1
```

leave the value of IVAR unchanged (i.e., 100). The programmer must be careful not to specify a double integer range (n > 17) for an integer variable. For example:

```
A=#D77000000[19:35]
```

yields the single integer value 0.

Note that only the first two words of a double-precision floating variable (the exponent and first-order mantissa) may be manipulated via this notation.

2.2 THE ASSIGN STATEMENT

General Form	ASSIGN n TO label
Examples	ASSIGN 27 TO ITEST ASSIGN 10 TO LOOP
Effect	The symbolic label (a variable of type integer) represents the specified statement number in an assigned GO TO or arithmetic IF statement

The ASSIGN statement provides a symbolic addressing capability for the two control statements mentioned above, GO TO and arithmetic IF. The statement number assigned must be that of an executable statement. Note that the integer variable is a symbolic label only within the context of an ASSIGN and its associated statements and may function as an integer variable elsewhere in the source program. Before this variable appears in such an expression, however, it must be redefined by an Arithmetic statement. For example, the statement:

```
ASSIGN 20 TO IVAR
```

does not assign the value 20 to IVAR but the memory location associated with the source-program statement 20. The sequence:

```
ASSIGN 20 TO IVAR
IVAR=20
NEWVAR=IVAR+1
```

is permitted, but IVAR may not be used as a label until redefined by another ASSIGN statement.

CHAPTER 3
CONTROL STATEMENTS

Statements in a FORTRAN-IV program are normally executed in the sequence in which they appear. The user may alter this sequence in two basic ways - by invoking a subprogram (Chapter 5) which returns control to the normal sequence after execution, or via one of the control statements described in this chapter. These are:

a. the GO TO statement, which transfers control to a specified statement, thus originating a new sequence of execution;

b. the DO statement, which establishes an iterative sequence of statements within the normal sequence;

c. the IF statement, which specifies conditions for the execution of a statement in sequence or transfer to a new sequence; and

d. PAUSE and STOP, which, respectively, interrupt and halt program execution.

3.1 THE GO TO STATEMENT

Three forms of the GO TO statement are described below - unconditional, computed, and assigned. All of these forms transfer control to a statement in the source program; the difference between them is the manner in which that statement is specified. Any GO TO statement may appear at any point in the executable portion of the source program except as the terminal statement of a DO loop (3.2).

3.1.1 The Unconditional GO TO Statement

General Form	GO TO n
Where	n = the number of an executable statement
Example	GO TO 27
Effect	Control is transferred to statement n

The simplest form of GO TO statement, the unconditional, is a direct branch to another location in the source program. Program execution proceeds from this point in the usual sequence.

3.1.2 The Computed GO TO Statement

General Form	GO TO $(n_1,n_2,\ldots,n_k)$,i
Where	n = the number of an executable statement i = an integer variable
Example	GO TO (3,17,25,50,66),ITEM
Effect	Control is transferred to the statement whose number is the n_ith in the list. If ITEM = 2, control passes to statement 17

A maximum of 64 numbers may be listed in a computed GO TO statement. The value of the integer variable i must fall within the range from 1 to the number of statement numbers listed. If the value falls outside of this range, an OTS error statement is generated and control passes to the next statement in sequence.

3.1.3 The Assigned GO TO Statement

General Form	GO TO label or GO TO label,$(n_1,n_2,\ldots,n_k)$
Where	label = an integer variable assigned a statement number value $n_1 \ldots n_k$ = statement numbers which the ASSIGN statement may legally assign to label
Examples	ASSIGN 13 TO KAPPA GO TO KAPPA GO TO KAPPA,(1,13,100) GO TO KAPPA,(1,72,100)
Effect	Control is transferred to the location specified by label

The assigned GO TO statement permits symbolic addressing of statements and execution-time modification of control transfer, for example:

```
ASSIGN 30 TO LOOP
       :
20 GO TO LOOP
       :
30 ASSIGN 45 TO LOOP
       :
GO TO 20
```

In this sequence, the same statement which branched to statement 30 will next branch to statement 45.

3.2 THE DO STATEMENT

General Form	DO n i=m_1,m_2,m_3 DO n i=m_1,m_2 or DO n i=m_1,m_2,-m_3
Where	n = a statement number i = an integer variable m_1,m_2,m_3 = variables or constants
Examples	DO 10 I=2,10,2 (iterations for I=2,4,6,8,10) J=1 DO 1,I=5,1,-J (I=5,4,3,2,1) DO 2 I=1,5 (I=1,2,3,4,5)
Effect	Statements following the DO up to and including statement n are executed repeatedly for values of i starting with m_1, incremented (or decremented) by m_3 until i has surpassed the limit m_2. If m_3 is not present, an increment of one is assumed.

The series of statements which are executed as the result of a DO statement are called the range of the DO. The variable i is called the index. The values m_1, m_2, and m_3 are, respectively, the initial, limit, and increment values of the index.

If the increment variable, m_3, is preceded by a minus sign, it is actually a decrement. For a constant m_3, this is simply something of the form -3. For a variable m_3, the value of the variable itself must be positive and may be preceded by a minus sign. Thus, J=1 and -J as an increment is correct. J=-1 with J as an increment is invalid.

The initial (m_1) and limit (m_2) values of a DO statement may be positive, negative, or zero provided the difference between them is less than 131072. Positive values for m_1 and m_2 may be expressed as positive integers or as variables assigned positive values. Explicit minus signs, however, may not precede the integer (constant or variable) given within a DO statement for initial and limit index values.

Negative initial and limit values must be expressed as a variable whose assigned value is negative. For example, the statements:

```
DO 10 I=2, -10, -2
and
DO 10 I=2, -A, -2
```

are both incorrect since a minus sign is not permitted before the integer constants or variables given for limit values. The series:

```
A= -2
DO 10 I=2,A,-2
```

is correct.

Loop termination, as indicated in the model, occurs when m_3 has a value beyond the limit. For a positive increment, this occurs when I is greater than m_2; for a negative increment when I is less than m_2. For example, the loop initiated by:

```
DO 10 I=1,100,2
```

will not be executed when I=101. The loop initiated by:

```
DO 10 I=100,1,-2
```

will be terminated when I=0.

It is the programmer's responsibility to ensure that the limit value specified will ultimately be reached.

```
J = 10
K = 100
M = 10
MINUSJ = -J
MINUSK = -K
```

The statements:

```
DO 10 I=J,K,-M
DO 10 J=J,MINUSK,M
```

specify infinite loops which are not detected by the compiler.

The statements:

```
DO 10 I = MINUSJ,K,M
DO 10 I = J,MINUSK,-M
```

specify finite loops.

The range of a DO may contain any statement with one exception. That is, the terminal statement may not be a GO TO, RETURN, STOP, PAUSE, or numerical IF statement. A logical IF statement is permitted provided that it does not include any of the statements given above.

3.2.1 Execution of a DO Range

The processing of a simple DO range is shown below.

```
   DO 10 I=1,10,1
   ARRAY(I) = TAB(I*2)
10 TAB (I*2) = 0
```

The range of the DO here consists of the two Arithmetic statements, which use the index variable as a subscript index. The range may have any number of statements and the index variable may be used as an ordinary variable provided that its value is not changed. The statement I=I*2, for example, would be illegal within the range given above, but TAB(I)=I is valid.

The exit from the range of the DO to the next statement in sequence is referred to as the normal exit. In this case, the value of the index variable becomes undefined. Exit may also be accomplished by the occurrence of a control statement within the range, leaving the index variable with its current value available for use as a variable. Control may also be transferred from outside the range of a DO to any statement within. For example:

```
   DO 20 I=1,100
10 IF (TAB(I) .EQ. 0) GO TO 50
20 CONTINUE
   :
   :
50 TAB(I)=TAB(I+1)
   GO TO 10
```

Here, a table is consolidated by replacing a zero entry with the next entry. Control is transferred out of the DO loop to move the entry and returned to check for zero. Note that the above example permits branching into the range of a DO which standard FORTRAN does not permit. PDP-15 FORTRAN considers statement 50 and the following GO TO as the "extended range" of the DO.

3.2.2 Nested DO Statements

When the range of a DO statement contains another DO statement (and its range), it is referred to as nesting. Nesting may occur to a depth of 9.* The ranges of nested DO's must not overlap; that is, the range of an inner DO must be contained entirely within the outer DO statement as shown in Figure 3-1. They may, however, end on the same statement.

Execution of nested DO's proceeds as follows. Each time the outermost DO is executed for one of its index values, the DO within it is executed for all of its index values. If this range contains another DO, that range is executed completely for each of the values of the second-level DO. For example, using the legal nesting example above, when range 1 is executed first for I=1, range 2 is initiated with K=2 and range 3 is initiated with J=1 and iterated (until J>5) five times. Then range 2 is iterated with K=3 and range 3 iterated until J>5. This process continues until K>10. That is, range 2 is repeated 9 times and each of these times, range 3 is repeated five times (a total of 45 times) while I of range 1 is still equal to 1. At this point range 1 is repeated for I=2 and the whole procedure recurs - range 2 is done 9 times, range 3, 45 times. When I >10 for range 1, range 2 will have been performed 90 times and range 3, 450 times.

---

*This restriction includes any implied DO in an input-output list (Chapter 6).

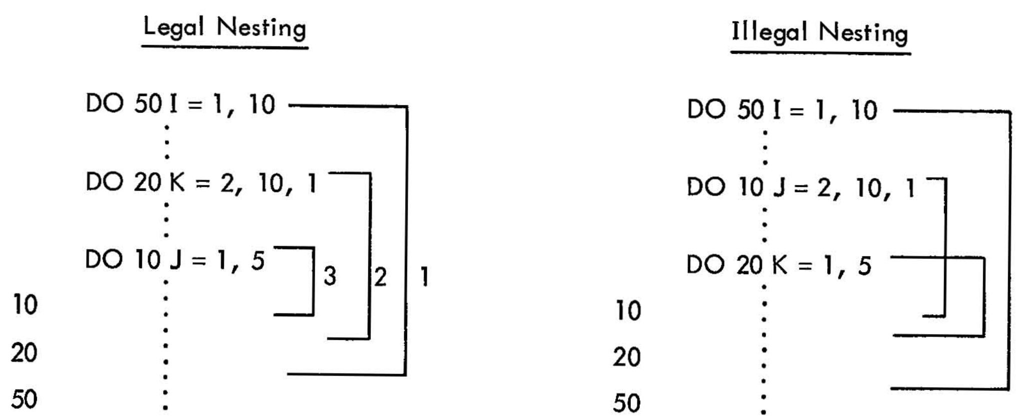


Figure 3-1 Rules for Nested DO Statements

3.2.3 The CONTINUE Statement

The CONTINUE statement is a dummy statement, which does not generate code or cause any action. It consists simply of the text:

```
CONTINUE
```

The CONTINUE statement may appear anywhere within a FORTRAN program but is especially useful for terminating DO loops when the last statement would otherwise be one of the illegal terminal statements listed previously. For example:

```
      DO 10 K=START, END
            :
    7 PAUSE
   10 CONTINUE
```

Here, the user can interrupt program execution at every iteration of the loop although the PAUSE statement cannot be the terminal statement.

3.3 THE IF STATEMENT

An IF statement causes control to be transferred or a statement to be executed contingent on the value of a test expression. Two forms of the IF statement are available - arithmetic and logical. They differ both in general form as well as in type of expression tested.

3.3.1 The Arithmetic IF Statement

General Form Where	IF(expr)n_1,n_2,n_3 expr = an arithmetic or logical expression n = a statement number or symbolic label established by an ASSIGN statement

Examples	IF(COUNT)10,20,30
	ASSIGN 20 TO MID
	ASSIGN 30 TO FIN
	IF(A(I)*B)10,MID,FIN

Effect	The parenthesized expression is evaluated. Control is transferred to: n_1 if expr <0 n_2 if expr =0 n_3 if expr >0

As shown in the model, the Arithmetic IF statement transfers control to one of three statements according to the value of the expression given. Thus, if COUNT=3, control in the first example, would be transferred to statement 30.

If an Arithmetic IF statement requires a continuation line, the line must be broken at a comma.

```
IF(E)10
10,101,102
```

will not compile. (The desired result is IF(E)100,101,102.)

Logical values .TRUE. and .FALSE. have the following decimal values:

a. .TRUE. = -1

b. .FALSE. = 0

Since logical values have specific arithmetic values, logical expressions may be used in place of arithmetic expressions in FORTRAN statements. For example, the statement:

```
IF (A.GT.B.AND.A.LT.C) 3,4,999
```

is equivalent to the two statements:

```
IF (A.GT.B.AND.A.LT.C) GO TO 3
GO TO 4
```

Note that in the above example the branch to 999 will never be executed. Logical expressions, however, do not always yield the values 0 or -1; for example, in the statement:

```
IF (I.XOR.J) 1,2,3
```

the branch is made to statement:

a. 2 if I = J,

b. 3 if I ≠ J but both have the same sign,

c. 1 if the signs of I and J are different.

3.3.2 The Logical IF Statement

General Form	IF(expr)s
Where	expr = any expression s = any executable statement except a DO or logical IF

Examples	IF(L1 .LE. L2)GO TO 17
	IF(L1)IF(X)3,5,5
	IF(L .AND. (.NOT. L1))A=A+1
	IF(I-J)A=A+1

Effect	If the expression is .TRUE. (or non-zero), statement s is executed; if .FALSE. (zero), the statement is ignored

Unless the statement executed as the result of a logical IF statement transfers control (i.e., GO TO), control continues in the normal sequence with the statement following the IF. For example, in the statement

```
      IF(L1)IF(X)3,5,5
   10 A=B
```

when L1 is .TRUE., the numeric IF is executed and control transferred to statement 3 or 5. When L1 is .FALSE., control passes directly to statement 10.

Non-logical (arithmetic) expressions are permitted within logical IF statements. In such cases, non-zero values are regarded as being logically .TRUE. and zeros as being logically .FALSE. For example, the statement:

```
      IF (X-3.0) GO TO 5
```

causes a branch to statement 5 if X ≠ 3.0 (i.e., x -3.0 ≠ 0).

3.4 EXECUTION CONTROL

3.4.1 The PAUSE Statement

General Form	PAUSE or PAUSE n
Where	n = an octal interger $\leq 777777_8$
Examples	PAUSE PAUSE 100
Effect	Execution is suspended and the number, if any, is printed

The PAUSE statement interrupts program execution, but maintains the current state of all values. Execution may be resumed by typing CTRL P (↑P) on the console teletype. The integer n, when supplied, is printed on the console teletype and may be used to identify which of several PAUSE statements was encountered.

3.4.2 The STOP Statement

General Form	STOP or STOP n
Where	n = an octal integer $\leq 777777_8$
Examples	STOP STOP 20
Effect	Control returns to the MONITOR after n is printed

A STOP statement is used to signify the logical end of a program. If several occur (the logical end depending on processing results), they may be numbered to indicate where the program ended.

For example, the statements:

```
10 IF (COUNT .GE. 100) STOP 10
   .
   .
   .
50 IF (COUNT .GE. 100) STOP 50
```

make program termination dependent on the value of the variable count at two different points.

CHAPTER 4
SPECIFICATION STATEMENTS

Specification statements provide the compiler with information regarding the data mode, size, and, if desired, initial values of variables in the source program. All specification statements must precede the executable portion of the program (see Table 1-1 in Chapter 1).

Data mode specification is accomplished either explicitly via the statements INTEGER, DOUBLE INTEGER, REAL, DOUBLE PRECISION, and LOGICAL: or implicitly according to the initial character in the variable name. The FORTRAN-IV compiler contains implicit mode assumptions, but the user may override these via an IMPLICIT statement. In addition, the EXTERNAL statement specifies a subprogram name which will appear in a subprogram call.

Variable size is implicit in the mode assigned to a scalar variable but must be specified in the case of arrays so that the compiler can allocate adequate storage space. The most common way of declaring the size of an array is the DIMENSION statement. The programmer may also control the way in which memory is allocated via the COMMON and EQUIVALENCE statements.

Initial values may be assigned within a program via the DATA statement. A set of initial values may also be obtained at run time by using a BLOCK DATA subprogram.

4.1 MODE SPECIFICATION

Any data mode may be specified in a mode-declaration statement as described below. If INTEGER and REAL are the only data modes used in a program, the programmer need not have any mode-specification statements since he may use the compiler's implicit mode assumptions.

4.1.1 Mode-Declaration Statement

General Form	m $a_1, a_2, \ldots a_n$
Where	m = INTEGER, DOUBLE INTEGER, REAL, DOUBLE PRECISION, or LOGICAL a = variable name, array name with dimensions, or function name
Examples	INTEGER A,B,CYZ LOGICAL TTAB(10,10),T,F REAL XYZ
Effect	Elements in the argument list are declared to be of the given mode. An array is, in addition, allocated storage to the dimensions given

A mode-declaration statement overrides any implicit mode assumptions. Thus, the statement:

```
REAL ITAB,J
```

overrides the basic compiler assumption that ITAB and J are INTEGER. This rule also applies to any mode assumption specified in an IMPLICIT statement (4.1.2). An item may be assigned a mode only once in a given program. Note that any function which has not been assigned a mode in the definition statement and which does not have an implicit mode must appear in a mode declaration statement. Note also that arguments must have the appropriate mode, as in:

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

This declaration ensures the proper working of the external and intrinsic functions (Section 5.1.3) DATAN and DABS.

4.1.2 The IMPLICIT Statement

General Form	IMPLICIT $m_1(l_1), m_2(l_2), \ldots m_n(l_n)$
Where	m = INTEGER, DOUBLE INTEGER, REAL, DOUBLE PRECISION, or LOGICAL l = a list of one or more alphabetic characters and/or consecutive ranges of alphabetic characters (e.g., A-G)
Examples	IMPLICIT REAL (A-E,N,X-Z),INTEGER(F-L)
Effect	Establishes a new assumption for mode of non-declared variables

The IMPLICIT statement governs the implicit mode assumptions for a single source program. After the occurrence of the IMPLICIT statement shown above, for example, all variables beginning with F, G, H, I, J, K, or L will be assumed INTEGER while those that begin with A, B, C, D, E, N, X, Y, and Z are assumed REAL. In this case, the compiler will not assume that all letters not specified INTEGER are REAL. Only those listed as INTEGER are REAL. The initial mode assumption may be stated as:

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

4.1.3 The EXTERNAL Statement

General Form	EXTERNAL $a_1, a_2, \ldots a_n$
Where	a = name of a subprogram
Examples	EXTERNAL ISUM,ISUB ⋮ CALL DEBUS(ISUM,A,B) ⋮ CALL DEBUG(ISUB,A,B)
Effect	The listed symbols are defined as the names of subprograms for use as arguments of other subprograms

A statement function (Section 5.1.1) may be passed as an argument to a subprogram without being declared EXTERNAL. If, however, a subprogram requires the name of a FUNCTION or SUBROUTINE subprogram as an argument, the calling program must declare the name in an EXTERNAL statement.

The transmission of arguments to subprograms is discussed more fully in Chapter 5. In brief, a subprogram uses dummy symbols in statements which obtain values when called; these values must all be defined in the calling program. If these values are program variables, they are already defined within the calling program. The EXTERNAL statement ensures that subprogram names are also defined.

4.2 STORAGE ALLOCATION

4.2.1 The DIMENSION Statement

General Form	DIMENSION $a_1(l_1), a_2(l_2), \ldots a_n(l_n)$
Where	a = array name l = a list of from one to three integer constants giving the maximum value of each dimension of the array (i.e., column, row, plane)
Examples	DIMENSION A(100),B(50,50)
Effect	Consecutive storage is allocated; the sum of the dimensions and the array type determine the amount

Each array specification in a DIMENSION statement gives the maximum value which each of its subscripts may assume. Array dimensions may, instead, be given within a type-declaration or COMMON statement using the same notation, as in:

```
COMMON ARRAY(10,4),Y,Z
INTEGER A(10,10,10),B
```

Dimensioning information should appear only once in a given program. An array which will be passed as an argument to a subprogram must be declared in both the subprogram and the calling program; with dummy array names and with real array names, respectively (see Chapter 5).

4.2.2 The COMMON Statement

General Form	COMMON/b_1/vlist$_1$/b_2/vlist$_2$/...
Where	b = blank or a label vlist = a list of variable and arrays
Examples	COMMON/BLKA/X(3,3),Y(2,5) COMMON A,B,C/XX/X,Y,Z//D,E
Effect	The items in each vlist are allocated to the specified block of memory where they may be shared by other programs

The COMMON statement allows the data of a main program and/or its subprograms to share a common storage area. A common area may be divided into separate blocks. In this case, they are distinguished from one another by a label which is a unique variable name. One block may be left unlabeled; this area is referred to as blank COMMON. If the first block referred to in a COMMON statement is a blank COMMON block, the slashes may be omitted, as shown in the second example above. A blank COMMON area is otherwise denoted by two consecutive slashes.

Items are assigned to a COMMON block in the order in which they appear in a COMMON statement. All items to be stored in a given block need not be listed at once, however, but may be given later in the same statement, as in:

```
COMMON/BLK1/A,B,C/BLK2/X,Y,Z/BLK1/M,N,O
```

or in a subsequent COMMON statement as in:

```
COMMON/BLK1/COUNT/BLK2/T/BLK1/W,F
COMMON/BLK3/ARRAY(10)/BLK1/G,R
```

Entries are linked sequentially so the items assigned to BLK1 will be A, B, C, M, N, O, COUNT, W, F, G, and R, in that order.

The sharing of COMMON blocks is made possible by the fact that storage is allocated at the same location for blocks of the same name in all programs executed together. For example, if PROG1 contains the statement:

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

and SUBPROG1 has:

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

the variables A and D will share the same location in blank COMMON as will the variables B and E; likewise, COMMON block R will store X and U, Y and V, and Z and W at the same locations. A COMMON block may be of any length but no program may attempt to enlarge a block declared by a previously compiled program.

4.2.3 The EQUIVALENCE Statement

General Form	EQUIVALENCE $(I_1),(I_2),\ldots,(I_n)$
Where	I = a list of two or more variables or array elements with constants as subscripts
Examples	EQUIVALENCE(RED,BLUE,GREEN,COLOR) EQUIVALENCE(RED,BLUE),(BLUE,GREEN,COLOR)
Effect	The elements of each I are assigned to the same storage location

Note in the second example that the relation of equivalence, once applied to a variable, holds for subsequent equivalences.

Variables located in a COMMON block may be made equivalent to other variables but not to one another. Quantities placed in a COMMON block via an EQUIVALENCE statement which change the size can only occur at the current end of the block.

For example, the statements

```
COMMON/BLK1/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(A,Y)
```

cause BLK1 to extend from X to A(4), arranged as follows:

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

Note that if, for example, A(1) were previously made equivalent to a variable, M, that Y, A(1), and M would all share the same location.

In the following example, three arrays of different dimensions occupy the same storage locations.

```
DIMENSION A(100),C(50),D(200)
EQUIVALENCE(A,C,D)
```

Here, of course, only the first 50 locations apply to all three, as shown.

```
location 1 A(1), C(1), D(1)
   :          :
location 51 A(51), D(51)
   :          :
location 101 D(101)
   :          :
location 200 D(200)
```

In the following sample program, arrays containing different numbers of subscripts are made equivalent using the EQUIVALENCE statement.

```
C       EQUIVALENCE TWO ARRAYS SUCH THAT THE FOLLOWING
C       PAIRS OF SUBSCRIPTED VARIABLES WILL OCCUPY THE SAME
C       MEMORY LOCATIONS;
C       A(1) AND B(1,1)
C       A(2) AND B(2,1)
C
C
        DIMENSION A(2),B(2,2)
        EQUIVALENCE (A,B)
C
C
C       FOLLOWING DATA STATEMENT WILL SET B(1,1) AND B(2,1) TO
C       THE VALUES 3.0 AND 4.0 AS WELL.
C
C
        DATA A(1),A(2)/3.0,4.0/
C
C
        STOP
        END
```

All variables equivalenced to COMMON variables are treated as COMMON variables with regard to subsequent EQUIVALENCE statements. EQUIVALENCE statements which would require extension of the beginning of a COMMON block are not allowed, as, for example:

```
COMMON/BLK1/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(X,A(3))
```

4.3 THE DATA STATEMENT

General Form	DATA $vlist_1/clist_1/, vlist_2/clist_2/, \ldots vlist_n/clist_n/$
Where	vlist = a list of variables, array elements, or array identifiers (no dummy variables permitted)
	clist = a list of constants with optional signs or a construction of the form n*c where n is the number of variables for which c is to be assigned
Examples	DATA A,B,C/1,2,3/ DATA A,B,C,D,E,F/1,2,3,3*0/
Effect	Each listed constant is assigned as the value of the corresponding variable

Any type of constant may appear in a "clist". A double-precision constant must be written explicitly in D format (e.g., 1.0D+01 or 1D+01 - not 1.D+01). The mode of the variable and the constant to be assigned must agree with one exception - an integer constant may be assigned to a double-integer variable.

The values specified in a DATA statement are compiled into the object program and become the values assumed by the listed variables when program execution begins. DATA statements may be given in BLOCK DATA subprograms described below. Note that COMMON variables may not be initialized using DATA except in the context of a BLOCK DATA subprogram; in this context the COMMON statement itself may also include data as in:

```
COMMON/B1/X,Y(5),I;X,I,Y(3)/2.0,3,5.0/
```

Array identifiers without subscripts may be specified in the list of variables in a DATA statement. The occurrence of an unsubscripted array identifier is equivalent to the occurrence of all of the elements of that array listed in ascending order. Note that in respect to the limitations of the number of variables in the variable list permitted each version of FORTRAN (refer to Appendix C) an unsubscripted array identifier counts as one variable. For example:

```
DIMENSION I (3),A(200)
DATA I,J,A,B/1,2,3,4,201*3.14/
```

4.4 BLOCK DATA SUBPROGRAMS

A BLOCK DATA subprogram is used to enter data into a single labeled COMMON block at run time; it is identified by a statement consisting of the text:

```
BLOCK DATA
```

Between this and the END line is the body of the subprogram which may contain only DATA, COMMON, EQUIVALENCE, DIMENSION, and TYPE statements.

A BLOCK DATA subprogram may not be used to initialize variables in a blank COMMON block area. When a given labeled block is initialized in this manner, all elements of it must be listed in a COMMON statement within the subprogram even if they do not appear in a DATA statement.

More than one COMMON block may be stated in a COMMON statement but only the last one can be initialized with DATA statements. The following program, for example, will not give a compilation error but will not function properly.

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

Two BLOCK DATA subprograms are required to initialize the two blocks, as shown below:

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

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

A more extensive example of a BLOCK DATA subprogram is given below.

```
BLOCK DATA
DIMENSION X(4),Y(4)
COMMON/NAME/A,B,C,I,J,X,Y
DATA A,B,C/3*2.0/
DATA X(1),X(2),X(3),X(4)/0.0,0.1,0.2,0.3/,
1Y(1),Y(2),Y(3),Y(4)/1.0,#-21,1.0E-4,0.2/
END
```

CHAPTER 5
SUBPROGRAM STATEMENTS

A subprogram is a program which is invoked by name from other programs whenever the operations it performs are required. It is a convenient and efficient means for encoding frequently used or complex operations since the statements appear only once in the object program regardless of the number of times they are used. In addition, a subprogram may be designed to handle a variety of different values which may be transmitted as arguments whenever the subprogram is invoked. The process of establishing a subprogram is referred to as subprogram definition; the statements to be executed are referred to as the body of the subprogram; the process of invoking the subprogram and transmitting arguments is referred to as a subprogram call.

Two basic types of subprogram* may be defined and called in FORTRAN-IV - functions and subroutines. A function is a subprogram which always returns a result - a single numeric value. This value is, by convention, represented in an expression by the function call. A subroutine may return several or no values; when results are obtained, they are assigned to variables in the calling program. The programmer may call a variety of predefined subprograms from the science library. (Refer to "Operating Environment" manual DEC-15-GFZA-D for a description of the science library.)

The transmission of arguments to a function and to or from a subroutine requires the use of dummy variables in the definition. That is, those variables in the subprogram which are to acquire calling values (real arguments) are listed in parentheses following the subprogram name in the subprogram definition statement. For example, if a subroutine were designed to merge two arrays (one with 500 elements and the other with 100) into a third array, the arrays might be referred to by dummy variables A, B, and C in the subroutine statements which refer to them, such as:

```
IF(A(I) .GE. B(I))C(I)=A(I)
```

These dummy variables would appear in the definition statements as follows:

```
SUBROUTINE MERGE(A,B,C)
```

---

*BLOCK DATA subprograms, described in Chapter 4, are a third type of subprogram in that they may by compiled separately.

When the subroutine is called, the user indicates the actual arrays to be merged, as in:

```
CALL MERGE(FILA,FILB,FILC)
```

The statement shown above then becomes:

```
IF(FILA(I) .GE. FILB(I))FILC(I)=FILA(I)
```

Note that the real arguments must appear in the same order as the corresponding dummy variables in the subprogram definition. Real arguments must also agree in mode with the dummy arguments to which they correspond. When arrays are involved, DIMENSION statements must be given for the real arrays in the calling program and for the corresponding dummy arrays in the body of the subprogram. In the above example, the following DIMENSION statements would be required:

Calling Program	Subprogram
DIMENSION FILA(500),FILB(100),FILC(600)	DIMENSION A(500),B(100),C(600)

Figures 5-1 and 5-2 show a main program and a subprogram which it invokes, respectively.

5.1 FUNCTIONS

Two types of function may be defined in FORTRAN-IV - statement and external. A statement function is defined within another program via a form of the Arithmetic statement and may be called only by the program in which it is defined (except when passed as an argument to a subprogram). An external function, defined via the FUNCTION statement, is an independent subprogram which may be called by any program. Both types of function must have names which conform to the rules for variable names and require at least one argument. A function call may appear only within an expression and, like other elements of an expression, must have an associated mode.

5.1.1 Statement Functions

General Form	$f(a_1,a_2,...a_n)$=e
Where	f = function name a = dummy argument e = an expression
Examples	SUM(A,B,C)=A+B+C FUNC(A,B)=2.*A/B A(X)=3.2+SQRT(X)
Effect	Defines function $f(a_1,a_2,...a_n)$ to have the value of expression e

```
C         THIS MAIN PROGRAM CALLS SUBR TO MAKE ALL ELEMENTS OF
C         THE ARRAY ICOL POSITIVE RELATIVE TO THE
C         SMALLEST ELEMENT OF THE ARRAY.  THE ELEMENTS OF THE
C         ARRAY ARE ASSUMED > OR = TO ZERO.
C
C
C
C         A COLUMN OF DATA IS READ INTO ICOL VIA LOGICAL 1.  THEN THE
C         SMALLEST ARRAY ELEMENT IS FOUND.  THE ARRAY NAME AND SMALLEST
C         ELEMENT IS PASSED TO SUBR.  SUBR SUBTRACTS THIS
C         SMALLEST ELEMENT FROM EVERY ELEMENT OF THE ARRAY.
C
C
          DIMENSION ICOL(20)
C
C
C         READ IN COLUMN OF DATA
C
99        READ(1,2)ICOL
2         FORMAT(I6)
C
C
C         FOLLOWING LOOP FINDS SMALLEST ARRAY ELEMENT AND
C         REMEMBERS ITS SUBSCRIPT IN JMIN.
C
C
C
          JMIN=1
          DO 4 I=1, 19
          IF(ICOL(JMIN)-ICOL(I+1))4,4,6
6         JMIN=I+1
4         CONTINUE
C
C
C         CALL SUBROUTINE TO PERFORM THE SUBTRACTION OF ICOL(JMIN)
C         FROM EVERY ELEMENT OF ICOL.
C
C
          CALL SUBR(ICOL,ICOL(JMIN))
C
C
C         LIST MODIFIED ELEMENTS OF ICOL VIA LOGICAL 2.
C
C
          WRITE(2,5)ICOL
5         FORMAT(1X,I6)
C
C
          GO TO 99
          END
```

Figure 5-1 Main Program Sample

```
C      SUBROUTINE SUBTRACTS VALUE OF ONE ELEMENT FROM ALL THE
C      ELEMENTS OF A ONE DIMENSIONAL INTEGER ARRAY.
C      CALLING SEQUENCE IS:
C      CALL SUBR(ARRAY,ARRAY ELEMENT)
C      DUMMY VARIABLES OF SUBROUTINE SUBPROGRAM ARE IDUMA,IDUMAE
C      WHERE IDUMA CORRESPONDS TO ARRAY NAME AND IDUMAE
C      CORRESPONDS TO ARRAY ELEMENT TO BE SUBTRACTED.
C      IDUMA MUST BE A 20 ELEMENT SINGLY DIMENSIONED INTEGER
C      ARRAY.
C
C
C      DECLARE SUBROUTINE AND ITS DUMMY VARIABLES
C
C
C
       SUBROUTINE SUBR (IDUMA,IDUMAE)
C
C
C
C
C      DIMENSION IDUMA
C
C
       DIMENSION IDUMA(20)
C
C
C
C      FOLLOWING LOOP SUBTRACTS VALUE OF ARRAY ELEMENT PASSED
C      TO SUBR FROM EVERY ARRAY ELEMENT OF THE ARRAY PASSED TO
C      SUBR.
C
C      SET ELEMENT INTO TEMPORARY.  CANNOT USE IDUMAE IN LOOP SINCE
C      ELEMENTS OF THE PASSED ARRAY ARE ALL MODIFIED
C
C
C
C
       IDECR=IDUMAE
C
C
C
C
       DO 1 I=1,20
1      IDUMA(I)=IDUMA(I)-IDECR
       RETURN
       END
```

Figure 5-2 Subprogram Sample

A statement function definition is a single non-executable statement in a FORTRAN source program. It may not precede any specification statement but must precede any executable statement of the program in which it appears.

The expression which defines a function may include dummy variables, ordinary variables, non-Hollerith constants, and previously defined external or statement functions. The function name may be associated with a data mode by any of the conventions specified for variables. The dummy variables used to define the function may appear in a specification statement but in no other context. Up to 10 dummy arguments may be used in a single definition.

To call a statement function, the programmer simply includes the name, with real arguments as required, in an arithmetic expression. For example, assume the definition:

```
OFFSET(A,B)=A+B+100
```

where A and B are dummy variables. The function might be called as follows:

```
TAB(I)=100+OFFSET(CT1,CT2)
```

which yields the same result as:

```
TAB(I)=200+CT1+CT2
```

A statement function, in effect, permits the programmer to extend the symbolic language available to him. By assigning a mnemonic name to a frequently used calculation, he obtains a more readable source program. For example, one might define the integer function percent as:

```
PERCENT(I,J)=(I*100)/J
```

This function is used in the following statements to express the values in TAB1 as a percentage of each of the values in TAB2.

```
    DO 10 K=1,10
    TOT=TAB2(K)
    DO 10 I=1,100
    VAL=TAB1(I)
    J = 100*(K-1)+I
 10 TAB3(J)=PERCENT(VAL,TOT)
```

5.1.2 External Functions

General Form	m FUNCTION $f(a_1, a_2, \ldots a_n)$
Where	m = an optional mode specification f = function name a = dummy argument
Example	INTEGER FUNCTION TOT(A,B) DIMENSION A(100) TOT=0 DO 10 I=1,B 10 TOT=TOT+A(I) RETURN END
Effect	Defines an external function

An external function is an independently written program which is executed when its name appears in an expression in another program. The optional mode specification (m) may be INTEGER, DOUBLE INTEGER, REAL, DOUBLE PRECISION, or LOGICAL. If no mode is given the function is associated with a mode in the same manner as a variable.

The function name must conform to the rules for variable names and must appear at least once as a variable within the body of the function; that is, the function name must be defined before control is returned to the calling program. Only non-subscripted variable names may appear as arguments in an external function. If an array name is used, an argument must appear in a DIMENSION statement within the subprogram. At least one argument must be specified for all functions.

The body of a function may contain any FORTRAN statement with the exception of BLOCK DATA, SUBROUTINE, FUNCTION, or a statement containing any reference to itself. It must contain at least one RETURN statement, which signifies the logical end of the subprogram. When a RETURN statement is executed, control is returned to the calling program. The function name must have been assigned a value before the occurrence of a RETURN.

An external function is called when its name and arguments appear in an arithmetic or logical expression. Real arguments may be variables, array elements, array names, any other expression, or the name of an external function or subroutine. A sample external function is given below.

```
   INTEGER FUNCTION CODE(J)
   COUNT=1
   DO 20 I=10,100,10
   IF(J .LT. I)GO TO 30
20 COUNT=COUNT+1
30 CODE=COUNT
   RETURN
   END
```

This function encodes values of J as follows: If J has a value between 0 and 9, it is encoded as one; if its value is between 10 and 19, it is encoded as 2 and so on up to 100. It is invoked by the following statements:

```
      DO 30 I=1,100
      VAL=TAB(I)
      IF(VAL .LT. 0)VAL=ABS(J)
   30 TAB(I)=CODE(VAL)
```

Negative numbers are avoided by calling the library function ABS which returns the absolute value of its argument. A table, TAB, is converted here to contain the codes associated with its original values.

5.1.3 DEC Library Functions

Two types of predefined functions are part of the FORTRAN Science Library - intrinsic and external. Both types are called in the same manner as a programmer-defined external function, as in:

```
V=2*SQRT(C)
```

and

```
J=2+IFIX(R)
```

where SQRT is an external library function and IFIX is an intrinsic library function. In each case, the value of the expression will, if possible, be converted to the mode of the variable. (All library functions are listed and described in the "Operating Environment" manual DEC-15-GFZA-D).

5.2 SUBROUTINES

A subroutine is defined external to the program which invokes it in a manner similar to an external function. The important distinctions between the two are: a subroutine need not have any arguments at all but will accept numerical, logical, or Hollerith constants as arguments; a subroutine need not return any values to the caller but may return several; and a subroutine is called via an explicit CALL statement.

5.2.1 Subroutine Definition

General Form	SUBROUTINE name(a_1,a_2,...a_n) or SUBROUTINE name : (FORTRAN statements) : RETURN END
Where	a = dummy argument
Example	SUBROUTINE STORE(A,B) DIMENSION A(100),B(100) DO 10 I=1,100 10 A(I)=B(I) RETURN END
Effect	Defines an external subroutine

A subroutine name must conform to the rules for variable names and may not appear within any statement in the body of the subroutine. A subroutine's arguments may represent any FORTRAN expression (this includes any constant, variable, array element, or an external subroutine or function name). Dummy arguments may represent values supplied by the calling program or values returned by the subroutine. In the latter case, the dummy variable must appear on the left side of an Arithmetic statement or in an input list within the subroutine body.

For example, if the function CODE given in Section 5.1.2 were defined as a subroutine rather than as a function, it would have the following form:

```
      SUBROUTINE CODE(J,C)
      C=1
      DO 20 I=10,100,10
      IF(J .LT. I)GO TO 30
   20 C=C+1
   30 RETURN
      END
```

The statements which invoked the function would be modified to invoke the subroutine as follows:

```
      DO 30 I=1,100
      VAL=TAB(I)
      IF(J .LT. 0)J=ABS(J)
      CALL CODE(VAL,TAB(I))
```

An array name argument must appear in a DIMENSION statement. EQUIVALENCE, COMMON, and DATA statements are permitted but may not include any dummy variables.

The logical termination of a subroutine is signalled by a RETURN statement. The physical end is indicated by an END line.

5.2.2 Subroutine Calls

General Form	CALL subr(a_1,a_2,...a_n) or CALL subr
Where	subr = the subroutine name a = a real argument
Examples	CALL COMPUTE(A,B,ANS) CALL ERROR
Effect	Control is transferred to the subroutine; real arguments, if any, are substituted for dummy variables at execution

The arguments of a CALL statement may be of any type but must agree in number, order, type, and, in the case of arrays, dimensions with the corresponding arguments in the SUBROUTINE statement of the called subroutine. When subroutine execution has been completed (a RETURN has been encountered), control returns to the statement following the CALL.

5.3 MULTIPLE ENTRIES AND RETURNS

General Form	SUBROUTINE/FUNCTION : ENTRY name(a_1,a_2,...a_n) : END
Where	name = the entry name a = a dummy argument
Examples	SUBROUTINE MOVE(A,B,C) DIMENSION A(100),B(100),C(100) DO 10 I=1,100 10 C(I)=B(I) ENTRY MOVE1(A,B) DO 20 I=1,100 20 B(I)=A(I) RETURN END
Effect	A call to the name associated with ENTRY will invoke a portion of the subprogram in which it occurs

An ENTRY statement establishes a separately callable subprogram within a subroutine or external function. Thus, as in the model, the programmer bypasses one of the operations of a subroutine or function by calling MOVE1 rather than MOVE. An ENTRY statement may not occur within a DO loop. The dummy arguments of an ENTRY may appear in the body of the subprogram prior to the entry statement only if they are also arguments of a prior ENTRY or of the SUBROUTINE or FUNCTION definition.

In a multiple-entry function, the return value is always assigned to the function name itself. For example:

```
      INTEGER FUNCTION ADD10(A)
      DIMENSION A(10)
      ADD10=0
      J=10
   10 DO 20 I=1,J
   20 ADD10=ADD10+A(I)
      RETURN
      ENTRY ADD5(A)
      DIMENSION A(5)
      ADD10=0
      J=5
      GO TO 10
      END
```

The user calls ADD5, as in

```
      SUBTOT=ADD5(TAB)
```

which is given the value assigned to ADD10 in the body of the function.

Multiple returns may be specified via the construction:

```
      RETURN I
```

where I is an integer variable for which a legal statement number will be substituted when the subroutine or function is called. The variable I appears in the real argument list preceded by @, as in:

```
      CALL COMP(X,Y,@10)
```

The use of multiple returns is illustrated below.

```
   SUBROUTINE COMPARE(A,B,I,J)
      IF(A .EQ. B)RETURN
      IF(A .LT. B)RETURN I
      RETURN J
      END
```

Here the user need not examine the result of the comparison before transferring control but control is automatically transferred by the subroutine.

CHAPTER 6
DATA TRANSMISSION STATEMENTS

Data transmission statements control the transfer of data between internal storage and peripheral devices. The numerous details involved in this process, while greatly simplified in FORTRAN-IV, require that the programmer be familiar with the way in which data are stored externally and, to varying extent, with the way data are stored internally. In general, the external data may be thought of as a continuous string of information (logical record) which consists of one or more physical records on the device being used as a particular instance. The size of a physical record varies from device to device and, while not specifically mentioned in any data transmission statement, influences the selection of a particular type of logical record. Table 6-1 describes the type of physical record, for each device, which is associated with each of the two types of logical record - formatted and unformatted.

Table 6-1
Physical Record Description for Formatted and Unformatted Records

Device	Formatted	Unformatted
Teletypewriter	one line of up to 72 characters terminated by carriage return	not applicable
Line printer	one line of up to 80, 120, or 132 characters	not applicable
Card reader	one card (up to 80 characters)	not applicable
Paper tape reader and punch	one typewritten line image	50 words
Magnetic tape	one line image of 628 characters	251 words
Disk/DECtape	one line image of 628 characters	251 words

The terms formatted and unformatted refer to the way data are stored externally since this governs the techniques used in input/output. A formatted record is an ASCII character string which is interpreted on input and constructed on output according to conversion rules defined in a FORMAT statement. An unformatted record is a string of 18-bit words which may be stored into and read from program variables specified in an input/output list in a format determined by the compiler.

PDP-15 FORTRAN-IV provides a third way of treating data called Data-Directed I/O which uses default FORMATS for output and converts data on input to conform to the program variables which will contain them. A logical record need not conform in size to the size of a physical record associated with a particular device as the input/output statements which accomplish data transmission will automatically read or write the quantity of physical records required to accommodate the logical record.

A file (a complete collection of related logical records) is identified in a source program by a logical device number, an integer constant associated with a particular type of device in the monitor Device Assignment Table* (DAT). At run time, this number is assigned to a physical unit and all program references to it interpreted accordingly.

There are two types of input/output which may be performed via FORTRAN-IV I/O statements - sequential and direct access. Sequential statements direct records to and from memory in the sequence in which they are physically recorded on the device in question. Direct access (6.3.3) statements permit the sequential transmission of logical records to and from a direct-access device where the physical records are not stored sequentially.

In addition to the statements which perform input/output or describe a file for direct access, FORTRAN-IV provides a number of auxiliary data transmission statements. The most important of these, the FORMAT statement, provides program control over the conversion of data between program-required and device-required forms. Other auxiliary statements control mechanical aspects of device control such as opening and closing files and the ENCODE and DECODE statements permit conversion between formatted and unformatted (ASCII-binary) information in memory. ENCODE and DECODE permit the programmer to build a logical record in core, and to transmit data from this record into program variables.

6.1 THE FORMAT STATEMENT

General Form	n FORMAT($s_1,s_2,\ldots s_n$)
Where	n = statement number s = field specification of the form: nkw.d n = number of successive fields involved k = type of conversion** w = field width d = number of places to the right of the decimal point
Effect	FORMAT n is established as a reference for formatted I/O operations

*Refer to the FORTRAN "Operating Environment" manual, DEC-15-GFZA-D for a detailed description of the Device Assignment Table.

**The argument k, which characterizes the type of FORMAT, is always required; others may be optional, depending on the value of k.

A FORMAT statement describes one or more records to be read or written. It may be used with or without an input/output list. Without the list, data are input to and output from the FORMAT specification itself. Otherwise, the listed variables correspond to the list of field descriptions and the appropriate conversions performed.

6.1.1 Statement Syntax

A FORMAT statement may describe one or more physical records. If more than one is described, the character (/) denotes the end of a record, as in:

$$\text{10 FORMAT } (\underbrace{s_1, s_2}_{\text{Record 1}}/\underbrace{s_1, s_2, s_3}_{\text{Record 2}})$$

When a number of slashes appear at the beginning or end of a statement, that number of records will be skipped on input and blank records written on output. When a sequence of slashes occur within the statement, one is considered to be the record indicator and the rest interpreted as above. Thus:

$$\text{FORMAT } (///s_1//s_1, s_2, s_3///)$$

specifies three skipped or blank records before record 1, one after record 1, and three after record 3. Each record description, as shown above, may consist of one or more field descriptors, a field being a consecutive series of characters within the record. Field descriptors are, as shown above, separated by commas. If a particular descriptor applies to a set of consecutive fields, it may be preceded by a number indicating the number of fields to which it applies. For example, 3s1 applies the descriptor s1 to three consecutive fields. In addition, a group of field descriptors may apply to a set of consecutive fields. In this case, the group of descriptors is enclosed in parentheses and preceded by a group count indicating the number of times it will be used. The statement:

FORMAT (s1, s2, 3(s3, s4))

divides the record into eight fields associated with descriptors as follows:

S1 - field 1
S2 - field 2
S3 - field 3 } 1)
S4 - field 4 }
S3 - field 5 } 2)
S4 - field 6 }
S3 - field 7 } 3)
S4 - field 8 }

6.1.2 Field Descriptors

The form of a field descriptor is as given in the statement model:

nkw.d

where n is an optional repeat count, k is a control character specifying conversions to be performed or special formatting functions, w is field width, and .d indicates number of decimal places to the right as required. Table 6-2 below summarizes the control characters and their meanings.

Table 6-2
Field Descriptor Control Characters

Conversion Control	Output Field	Input Field
I	decimal integer	integer or double integer variable
E	floating-point, scaled	real variable
F	floating-point, mixed	real variable
G	floating-point, mixed/scaled	real variable
D	floating-point, scaled	double-precision variable
L	T or F	logical variable
A	alphanumeric	ASCII characters
R	alphanumeric, right-adjusted	ASCII characters

Special Purpose	Meaning
P	sets scale factor for E,F,D conversions
X	skips characters on input or output blanks
H ' ' $ $ " "	designate Hollerith field
O	octal field
T	tab to specified position

Each of these descriptors is described in detail below. All conversion descriptors must specify the field width. Note that the field width must be large enough to contain all characters (including decimal point and sign) required to constitute the data value and any blank characters needed to separate it from other data values. Note that during input, an inadequately sized input field enables only the leftmost (most significant) characters of each item input up to the width specified to be input; all other input characters are lost. If during output, a variable (the value of which exceeds the format specified field width) is found; a string of asterisks (*) including any indicated decimal point is output in its place. For example, if the output field specified is I4, the integer 12345 is output as ****.

The interaction of the FORMAT descriptors with items in a standard or data-directed I/O list are described in Section 6.3. An output FORMAT for a printing device must allow for the fact that the first character of a record is used as a carriage control indicator. Carriage control is described later in this chapter; it is ignored in the examples given below to clarify the conversion descriptors illustrated.

I-TYPE CONVERSION

General Form:	Iw or nIw		
	Examples (b indicates blank)		
Descriptor	Input	Internal	Output
I5	bbbbb	+00000	bbbb0
I3	-b5	-05	b-5
I8	bbb12345	+12345	bbb12345
I13	bbb1234567890	+1234567890	bbb1234567890

I-conversion descriptors are used for both single and double integer variables. On input, the numbers specified by w are converted to a binary integer before being stored. A minus sign is required for negative numbers; plus is optional. The field may not contain a decimal point. Blanks may precede the sign or first digit of a number; embedded blanks are interpreted as zeros.

On output, a binary integer is converted to a string of numeric characters and right-justified in the field. A minus sign is provided for negative numbers. If the number does not fit within the field, a string of asterisks is output instead.

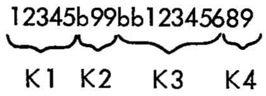
The following examples illustrate the use of the I descriptor. Program variables K1, K2, K3, and K4 will obtain values via the input FORMAT and their values will be output via the output FORMAT. The two format statements are given below - for input:

```
10 FORMAT (I5,I3,I8,I2)
```

and for output:

```
15 FORMAT (1X,I4,I2,I4,I1)
```

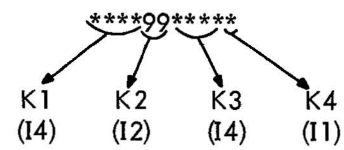
Note that the output format specification contains a carriage control indicator (1X). The input data is the string below:



which will be interpreted in the fields shown and will have the following internal values (in octal):

K1 = 030071
K2 = 000143
K3 = 361100
K4 = 000131

Using the output FORMAT given to output these variables gets the following printout:



Only the second value, as shown, fits into the allotted field width. Unpredictable results may be obtained when a number is too large for an internal integer but fits in the field width. For example, the integer 131073 (stored internally as 400001) is printed out -131071 if the output field width is large enough.

E-TYPE CONVERSION

General Form:	Ew.d or nEw.d		
	Examples		
Format Descriptor	Input	Internal	Output
E11.4	00.2134E03	213.4	b0.2134E+03
E9.2	0.2134E02	21.34	b0.21E+02
E10.3	bb-23.0321	-23.032	-0.230E+02

The input format of an E-type number is a string of digits optionally preceded by a sign. A decimal point and an exponent may be included in the string. The number itself should be restricted to 11 digits to ensure that the number, if given as an integer, will not exceed $2^{35}-1$ in magnitude. Results are otherwise unpredictable. The input string is converted to a floating-point number with d spaces reserved for digits to the right of the decimal point.

E-type output of a floating-point number consists of a minus sign for negative numbers followed by the digit zero, a decimal point, d significant digits, and an exponent of the form E±nn, as shown in the model. The field width must be at least d+7 characters long to accommodate this notation.

E-conversion is illustrated in the following examples.

Example 1.

```
Input format:       10 FORMAT(E6.0,E9.3,E10.3,E16.4)
Output format:      20 FORMAT(1X,E6.0,E9.3,E10.3,E16.4)
Input data:         -1.E2b-0.12E-01-0.123E-00-123456.7891E+01
                    E6.0   E9.3     E10.3      E16.4
Program variables:  R1     R2       R3         R4
```

They are stored internally, each in two words of storage, as follows:

```
R1  000007
    710000
R2  227772
    704467
R3  704467
    662775
R4  773716
    374025
```

Output of these variables using the given format will give the printout:

```
*.*****.*******-0.123E+00bbbbb-0.1235E+07
 R1     R2      R3          R4
(E6.0) (E9.3) (E10.3)     (E16.4)
```

Note that the output field was not large enough to accommodate the E-format of output, for R1, -0.E+03[W=7], or R2, -0.120E-01(W=10).

Example 2.

```
Input format:       10 FORMAT(E12.5,E7.2,E10.2,E12.4)
Output format:      20 FORMAT(1X,E12.5/1X,E12.5/1X,E20.9/1X,E30.10)
(Note the carriage control character preceding each output value.)
Input data:         123456.000000012345-.01E+03bb+100.0E-2
                         R1        R2       R3       R4
Output:             b0.12346E+06
                    b0.12345E+03
                    bbbb-0.100000000E+02
                    bbbbbbbbbbbbbbb0.1000000000E+01
```

F-TYPE CONVERSION

General Form:	Fw.d or nFw.d		
	Examples		
Format Descriptor	Input	Internal	Output
a) F6.3	b13457	13.457	13.457
b) F6.3	313457	313.457	**.***
c) F9.2	-21367.	-21367.	-21367.00

F-type conversion is the same as that described for E-type conversion on input. On output, however, the number is written with a minus sign if negative, an integer portion, a decimal point and the fractional part rounded to d significant digits. Note in the example an instance (item b) where the variable exceeds the output field width. It is important, on output, to have a field width large enough for a leading zero for values less than 1.0. For example, the descriptor F4.2 cannot output the value -.12 which must be printed as -0.12.

G-TYPE CONVERSION

General Form:	Gw.d or nGw.d	
	Examples	
Format Descriptor	Internal	Output
G14.6	$.12345678 \times 10^{-1}$	bb0.123457E-0
G14.6	$.12345678 \times 10^{0}$	bb0.123457bbbb
G14.6	$.12345678 \times 10^{4}$	bbb1234.57bbbb
G14.6	$.12345678 \times 10^{8}$	bb0.123457E+08

G-type conversion on input is the same as for E and F described above. On output, the format used depends on the magnitude of the internal data. If the exponent of the normalized value is greater than the number of decimal places specified in the output format (d), E-type output format is used. Otherwise, a modified F-type notation is used; that is:

F(w-4).(d-e),4X [e = value of exponent]

where 4X indicates four blanks to be appended to the value. The programmer must be sure to provide a field width sufficient to include these. E format is always used for values less than 0.1.

Some examples of G-conversion follow.

Example

Input format: 10 FORMAT(G16.5,G16.5,G16.4;G16.4)

Output format: 20 FORMAT(1X,G16.5/1X,G16.5/1X,G16.4/1X,G16.4)

Input data:

```
.12345E+5
.12345E+6
.12345E+4
.12345E+5
```

Output:

```
bbbbbb12345.bbbb
bbbbb0.12345E+06
bbbbbbb1234.bbbb
bbbbbb0.1234E+05
```

D-TYPE CONVERSION

General Form:	Dw.d or nDw.d		
Examples			
Format Descriptor	Input	Internal	Output
D13.6	bb+21345D03	21.345	b0.213450D+02
D13.6	b+3456789012	3456.789012	b0.345689E+04
D12.6	-12345.6D-02	-123.456	*.**********

In D-type conversion, an input field conforming to the format of an E-type input field is converted to a double-precision floating-point number. The output format is also the same as for E-type output, with the exception that the E is replaced by a D.

Example 1.

Input format: 10 FORMAT(D30.12)

Output format: 20 FORMAT(1X,D30.12)

Input data:

```
34359738367.*
34359738370.
34359738371.
.34359738367*
-34359738367.0000000000D+02
-34359.738367D+2*
```

Output:

```
bbbbbbbbbbbb0.343597383680D+11*
bbbbbbbbbbbb0.200000000000D+01
bbbbbbbbbbbb0.299999999986D+01
bbbbbbbbbbbb0.343597383680D+00*
bbbbbbbbbbb-0.243597383669D+03
bbbbbbbbbbb-0.343597383657D+07*
```

*Starred values have significant digits within the limit (34359738367) representable without truncation. Others obtain meaningless values.

Example 2.

Input format: 10 FORMAT(D15.9,D13.3,D20.10,D16.4)

Output format: 20 FORMAT(1X,D15.9,D13.3,D20.10,D16.4)

Program variable: D1,D2,D3,D4

Input data:

```
-1234.56789D+12-1.00D0bbbbb34359738367
\______D1_____/\___D2____/\___D3___/\D4
```

Output:

```
*.*************bbbb0.100D+01bbbb0.3435973836D+03bbbbbb0.0000D+00
\______D1_____/\_____D2_____/\__________D3_________/\______D4____/
```

P-SCALE FACTOR

General Form:	nP or -nP			
Examples				
Format Descriptor	Input	Scale Factor	Internal	Output
1PD10.4	12.3456	+1	+1.23456	1.2345D+00
-3PF6.3	123.456	-3	123456	123.456

The control character P indicates a scale factor (n or -n) to be applied to E-, F-, G-, and D-type conversions, for which a scale factor of zero is assumed if P is not present. When a scale factor P occurs in a statement, it applies to all subsequent conversions of the type to which it applies within that statement unless another explicit scale factor is encountered.

If an exponent appears in the external field, the scale factor is not used for input conversions. Otherwise, the internal value will be the external value times 10^{-n}.

On output, for D and E conversions, the fractional part of the number is multiplied by 10^n and the exponent is reduced by n. For G-type output, the scale factor is used only if the magnitude of the number is such that E-type output notation is to be used. For F-type output conversion, the external value = internal value times 10^n.

Example 1.

Input format: 10 FORMAT(1PF12.4/-3PE8.3/3pE12.5/-3PF15.5)

Output format: 20 FORMAT(1X,F12.4/1X,E8.3/1X,E12.5/1X,F15.5)

Input Data:

```
-1234.5678
-123.456
98.76
12345.6789
```

Output:

```
  -123.4568
.*******
0.98760E-01
12345678.74995
```

Example 2.

Input format: 10 FORMAT(F12.4/E8.3/F15.5/E12.5)

Output format: 20 FORMAT(1X,1PF12.4/1X,3PE12.3/1X,3PF15.5/1X,-3PE12.5)

Input Data:

```
-12.34
123.456
987.654
123.456
```

Output:

```
  -123.4000
123.505E+00
  987653.99167
0.00012E+06
```

A-TYPE CONVERSION

General Form: Aw or nAw			
Examples			
Format Descriptor	Input	Internal	Output
A4	ABCD	ABCDb	ABCD
A6	ABCDEF	BCDEF	bBCDEF

A-type conversion accepts an alphanumeric field of five characters and stores it as an unsigned double integer variable. If a field width greater than five is specified, the excessive characters are dropped from the left. If a field width less than five is specified, trailing blanks are appended.

On output, if field width is greater than five, W-5 leading blanks are output followed by the five internally stored characters. For a field width less than five, the leftmost w characters are output.

R-TYPE CONVERSION

General Form: Rw or nRw			
Examples			
Format Descriptor	Input	Internal	Output
R4	ABCD	bABCD	ABCD
R6	ABCDEF	BCDEF	bBCDEF
R2	AB	bbbAB	AB

R-type conversion accepts an alphanumeric field of five or less characters and right-adjusts them in an unsigned double integer. The descriptor R4, as shown, stores the string bABCD whereas A4 would store ABCDb. For strings equal to or in excess of five characters, R works exactly like A.

H-ALPHANUMERIC FIELD TRANSMISSION

General Form:	nHtext, 'text', $text$, "text"	
	Examples	
Format Descriptor	Output	Format Descriptor after Input of the String LETTERS
3HABC	ABC	3HLET
'TEXT'	TEXT	'LETT'
$MESSAGES$	MESSAGES	$LETTERS$

Character strings described as Hollerith fields are transmitted directly to and from an external device. Since input changes a Hollerith field, a subsequent output operation using the same FORMAT statement will output a text string different from that specified in the source program.

Hollerith field descriptors may be placed among other field descriptors as in:

```
FORMAT(5HbANS=I3)
```

which could be used to output the value of a variable (assume V1=100) in the form:

```
ANS=100
```

Note that the separating comma may be omitted after a Hollerith descriptor if it is of the form nHxxx.

L-TYPE CONVERSION

General Form:	Lw or nLw		
	Examples		
Field Descriptor	Input	Internal	Output
L4	bTbb	777777_8	bbbT
L1	F	0	F

When a value is read in using an L field descriptor, it is assumed to have the value 0 (.FALSE.) unless the letter T appears (and is not preceded by the letter F) in which case it has the value 777777_8. On output, a zero value is output as F and all others as T.

Example 1.

Input format: 10 FORMAT(L1,L2,L5,L8)

Output format: 20 FORMAT(1X,L1,L2,L5,L8)

Input data:
```
TbFbbFbbbb3456
L1L2 L3   L4
```

Internal values: L1=777777
L2=0
L5=0
L8=0

Output:
```
TbFbbbFbbbbbbbF
L1L2 L3    L4
```

Example 2.

Input format: 10 FORMAT(I6)

Output format: 20 FORMAT(1X,L1,L2,L3,L4)

Input data: 000000
777777
1
2345

Internal values: 000000
757061
000001
004451

Output:
```
FbTbbTbbbT
L1L2L3 L4
```

OCTAL FIELDS

General Form:	Ow or nOw		
	Examples		
Field Descriptor	Input	Internal	Output
O7	4000000	4000000	4000000
O12	400000000000	400000000000	400000000000
O6	-1	777777	777777
O12	777777777777	777777777777	777777777777

The programmer may enter values as octal numbers using the descriptor O. For integers and double integers, as shown in the model, the programmer may enter an integer value in octal whose magnitude exceeds the specified maximum (see 1.3.1). This provides a facility for establishing masking variables.

T-TAB

General Form:	Tn (n must be ≥ 2 for output; ≥ 1 for input)	
	Examples	
Format Descriptor	Input	Output
T35, 'ABC'	Characters 35-37 of data	To print positions 34-36

The descriptor T is used to control the emplacement of data on an output record (in terms of print position) and the acquisition of data from an input string (in terms of character position). Print position refers to n-1 since the first character in an output buffer governs carriage control.

Example 1.

Input format: 10 FORMAT(T4,ABC)

Input string: ABCDEFGHI

New format: 10 FORMAT(T4,DEF)

Example 2.

Input format: 10 FORMAT(T10,A3,T1,A2)

Input string: bbABCDbEFGbABC

Values read: A3=GbA
A2=bb

Example 3.

Output format: 10 FORMAT(1X,T2,$1 WON'T$,T4,$/////WILL$)

Output: Print position 1
I ~~WØN'T~~ WILL

Example 4.

Output format: 10 FORMAT(1X,T50,'DATE',T10,$NAME$)

Output: Print position 9 NAME   Print position 49 DATE

X-BLANK

General Form:	nX	
	Examples	
Format Descriptor	Output	Input
3X,A	bbA	4th character of input string

The descriptor X has appeared in previous examples of output formats to introduce a blank as the first character of a record to accommodate the carriage control conventions described below. Any number of blanks may be introduced within an output record using X. On input, X indicates that characters are to be skipped.

Example 1.

Input format: 10 FORMAT(3X,A1,5X,A2)

Input data: ABCDEFGHIJK

Values assigned: A1=D
A2=JK

Example 2.

Output format: 10 FORMAT(1X,A,2X,B)

Output: AbbB

CARRIAGE CONTROL

As stated previously, when a formatted record is output to a printing device, its first character is not printed but governs vertical spacing. The use of blank to initiate a line feed has been illustrated. Other characters may also be used to indicate other than line feed. Carriage control characters and their effects are shown below.

Character	Effect
blank	line feed
0	double space
1	form feed
+	no advance (overprinting)
all others	line feed

Whatever action is specified occurs before the line is printed.

Example

Input format: 10 FORMAT(I3)

Output format: 20 FORMAT(0,I3)

Program variables: V1,V2,V3

Input data: 300400500

V1 V2 V3

Output: [empty line]
300
[empty line]
400
[empty line]
500

6.1.3 Object Time FORMAT Specifications

Format specifications may be entered along with the input data they describe. If the programmer wishes to do this, his formatted input/output statements, instead of referencing a FORMAT statement number, reference the name of an array into which one or more FORMAT specifications will be entered. The array must appear in a DIMENSION statement even if its size is 1.

An object time FORMAT specification has the same general form as the source-program statement with two exceptions - the word FORMAT is omitted and Hollerith fields are not permitted. It can be entered into the appropriate internal array via the DATA statement or by using a formatted input statement which references an A-type FORMAT statement in the source program.

For example, the object time format specification (I7,F10.3) may be entered via a READ statement as follows:

```
   DIMENSION AA(10)
13 FORMAT(10A5)                  enters the FORMAT
   READ(3,13)(AA(I),I=1,10)
      .
      .
      .
   READ(3,AA)JJ,BOB            -- enters the data according to format
                                  stored in array AA
```

6.2 DATA-DIRECTED INPUT-OUTPUT

ASCII data may be entered or written without a FORMAT statement if the data-directed (format-free) input-output option is indicated in an input-output statement. Externally stored data fields in this case are determined not by field width but by the occurrence of a delimiter (multiple carriage returns, spaces, ALT-MODES, or commas). Each data item will be assigned to the variable indicated after conversion to the variable's type. A value will be assigned in all cases even if the value cannot be converted properly. For example, if the value read for an integer variable is too large, the largest legal integer value (377777_8) will be assigned to it.

For a data-directed output statement, both the variable name (subscripted as appropriate) and its value are written in the form: 'NAME'=value. The format in which the value is written is selected according to the variable's type as follows:

Variable Type	Output Format
LOGICAL	L1
INTEGER	I7
REAL	G16.8
DOUBLE PRECISION	D20.11
DOUBLE INTEGER	I12

On input, the data-directed option permits a variety of input items, an item being that portion of the input data delimited by a space, comma, carriage return, or ALT MODE. One item is accepted for each program variable specified in the input-output list. Acceptable input items are:

(1) string constants - delimited by a set of dollar signs ($), single-quotes ('), or double-quotes ("); may include any characters except carriage return and ALT MODE; rules for including the string-delimiter characters within the string are as given in Section 1.3.1 for Hollerith constants. If a Hollerith constant contains more than five characters, only the first five are stored. One of the item delimiters given above must follow the string terminating character. To ensure accuracy, a string constant should be associated with a double-integer variable.

(2) octal constants - preceded by the character # (#D for double integers). Non-integer values may also be given in octal form. Some examples of octal input items are:

```
#123
#D1234567
#1.23
#-1.00E+02
```

An octal integer whose magnitude exceeds 131071 will be considered a double integer even if D is not supplied, except when the value is within the range 400000 and 777777. This exception permits the programmer to perform explicit octal masking.

(3) logical constants - T or F followed by any number of characters up to a legal item delimiter. The values -1(T) and 0(F) are stored.

(4) decimal numbers - such as:

```
123
-12.3
-1.23E+2
+10.234D+15
```

If an illegal character appears in any data item, an error message is printed and input proceeds. For Teletype input, the user may reenter the erroneous item.

6.3 INPUT-OUTPUT STATEMENTS

Input-output statements perform the actual transfer of data. The READ statement specifies input. Output is specified by one of the three synonymous statements WRITE, PRINT, or TYPE.

The general format of these statements is given below (all arguments with the exception of d are optional).

$$\left\{\begin{array}{l}\text{READ}\\ \text{WRITE}\\ \text{PRINT}\\ \text{TYPE}\end{array}\right\} \quad (d \left\{\begin{array}{c}'\\ \#\end{array}\right\} r, \text{ f, END=m, ERR=n) list}$$

where:

d = integer constant or variable giving the logical device number (.DAT slot)

$\left\{\begin{array}{c}'\\ \#\end{array}\right\}$r = integer expression indicating record number (# or ' indicate direct-access I/O)

f = FORMAT statement number or array reference

m,n = statement number

list = an input-output list (must be present unless information is transferred directly to and from a FORMAT statement)

When ,f is absent, unformatted I/O is indicated. If the comma appears alone, data-directed I/O, as discussed in the preceding section, is performed. The END=m and ERR=n options permit the user to specify, respectively, a statement to which control should be transferred when an end-of-file or error condition occurs. If these options are not present, OTS end-of-file or error routines are used.

6.3.1 Input-Output Lists

An input-output list contains the names of variables, arrays, and array elements which are to be assigned data values on input or whose values are to be output.

An input-output list has the form:

$$c_1, c_2, \ldots c_n$$

where each c is a variable, subscripted variable, or an array name. When an array name appears (with no subscripts), the effect is the same as if all elements of the array had been listed. Note that during input the new values of listed variables which appear in a subscript are used. Thus, for:

I, B, C, ARRAY(I)

the array reference will use the value input for I in that statement. For example, if the value 100 is read into I, the fourth data field will be read into ARRAY(100).

If only a portion of an array is to be specified, indexing similar to that of a DO statement may be used to indicate several items in an array. For example, the list element:

(X(K),K=1,4)

specifies X(1), X(2), X(3), and X(4). The indexing may also be compounded by nesting, as in:

((A(I,J),I=1,4),J=1,5)

which specifies:

A(1,1),A(2,1),...,A(4,1),A(1,2),...,A(4,5)

The order of the list must be that in which the input data associated with its elements is stored. For example, if the list is A, B, C, the first three data items (as defined by the associated FORMAT statement or data-directed I/O delimiter) are stored in variables A, B, C. On output, the list specifies variables whose values are to be written either in accordance with a FORMAT statement or, for data-directed I/O, the default formats given previously.

If the data fields in a physical record exceed the number of variables in an input-output list, excessive fields are simply ignored. If the data fields are not sufficient to accommodate all listed variables, successive records are automatically read until all list elements have been satisfied.

6.3.2 Sequential Input-Output Statements

The form and effect of all sequential input-output statements are given in Tables 6-3 and 6-4 below. The END and ERR option, which may be used in any form of these statements, is omitted in the following models for the sake of clarity.

Table 6-3
The READ Statement

Data are transferred from external device (d) to internal storage.

Form	Example	Effect
READ(d,f)list	READ(3,10)A,B,C	ASCII data are read, converted according to FORMAT reference 10 and stored in variables A, B, and C
READ(d)list	READ(3)A,B,C	Binary data are read into variables A, B, and C
READ(d,)list	READ(3,)A,B,C	ASCII data are read, converted to the appropriate type, and stored in A, B, and C
READ(d,f)	READ(3,10)	Data are read into FORMAT reference 10
READ(d)	READ(3)	A binary record is read and ignored

Table 6-4
The WRITE Statement

Data are transferred from internal storage to external device (d).
PRINT or TYPE may be used as synonyms of WRITE.

Form	Example	Effect
WRITE(d,f)list	WRITE(3,10)A,B,C	The values of A, B, and C are converted to ASCII according to FORMAT reference 10 and written
WRITE(d)list	WRITE(3)A,B,C	The values of A, B, and C are written (in binary)
WRITE(d,)list	WRITE(3,)A,B,C	The variable names A, B, and C are written, each followed by its value in the form: 'A' = value
WRITE(d,f)	WRITE(3,10)	FORMAT reference 10 is written

6.3.3 Direct-Access Input-Output*

Direct-access input-output statements permit the user to directly reference any record in a file without indexing from the file's beginning up to the desired record. In addition, data may be changed in a single record without creating a new file.

Before direct-access input-output may be performed, the programmer must initialize the file via a CALL DEFINE statement. (The format of direct-access files is given in Part II, Chapter 2). The form of the statement is:

CALL DEFINE (d, rsiz, fsiz, fnam, v, mode, a dcod)

where d is a logical device number and other arguments are described in Table 6-5.

The purpose of DEFINE is to initialize a file for direct-access operations. If the file exists already, arguments supplied must correspond to the characteristics of the given file. If the file specified in fnam does not exist on the specified logical device, a file of the given name will be created in accordance with the parameters supplied by the other arguments. Or if fnam=0, a file is created and assigned the name TM0ab OTS (where ab are the decimal digits of a logical device number d). If the deletion code is set to one, the file will be deleted when the file is closed. A direct-access file is closed by either of the statements: ENDFILE d or CALL CLOSE (d).

*For use with DOS-15 file structure only.

Table 6-5
Arguments for CALL DEFINE

Argument	Value	Represents	Comments
rsiz	Decimal integer ≤628 (ASCII characters) Decimal integer ≤131071 (Binary words)	Record length in characters Record length in words	For formatted records this is given as the number of ASCII characters and for unformatted as binary words. The maximum for rsiz is based on the fact that a physical block size is 256 words. However, two of these are link words, two are header words, and the first and last characters of the ASCII data (automatically inserted) are, respectively, a forms control character and a carriage return. The maximum number of words rsiz may specify is 251*. The additional five words in the block are allocated as follows: two header words, one I.D. word, and two link words. Only fixed length records may be accessed.
fsiz	Decimal integer ≤131071	The number of records in the file	The I.D. word of a record contains its record number and the 0-th bit is used as a last-record indicator. Fsiz must be less than or equal to the number of records in the existing file unless file size is to be adjusted (argument a, described below). When fsiz is less than the actual number of records, no input-output operations may be performed on records with greater numbers.
fnam	Array name	An array containing a six-character file name and three-character extension (in sixbit ASCII)	Fnam may also be 0 indicating that standard default names are to be used.
v	Integer variable name (referred to as the associated variable)	The number of the record just after the last one accessed	V is assigned a value at the conclusion of an input-output operation.
mode	Integer variable	I/O mode	For unformatted I/O, mode=0; for formatted I/O, mode=non-0
a	Integer code	Indicates file adjustment options	a=0 indicates no adjustment; otherwise, the number of records in the file specified is adjusted to the number indicated by the current fsiz.
dcod	Integer code	Applies only to temporary files	If dcod=0, it means "do not delete." If dcod=1, temporary file .TM0ab OTS is to be deleted on device ab (decimal digits of device number d)

*Logical records greater than 1 physical block are termed <u>long records</u>. These are specified when rsiz is greater than 251.

Two conventions are available for specifying the record size for an unformatted direct-access record. One, for records whose size is less than one physical block, has a maximum of 251 words available for user data. Since the overhead components of an unformatted record comprise three words and since the total record size must be even, the number of user data words must be an odd number. If an even number is specified, DEFINE will treat it as though rsiz +1 were specified.

Another convention is used for "long records" - records greater in size than one physical block. All blocks of a long record except the last will contain unformatted records 251 words long. The last block will contain an unformatted record with an odd number of data words so that the total length is equivalent to fsiz (or fsiz +1 if even).

When a formatted file is created, all data words are filled with 7-bit ASCII spaces (040_8); for unformatted, all data words are set to zero. The I.D. word for all binary records is set to 400000_8 (see Part II for more detail). The associated variable is set to one.

If the user arguments have requested adjustment of an existing file, a temporary file named ..TEMP OTS is created on the device specified and the existing file (temporarily associated with a system device) is copied into it a record at a time. When a file is lengthened, size parameters are used to add null records (spaces or zeros) after the last record is transmitted and the associated variable is set to the old number of records plus one. If file size is being reduced, data from the end of the old file are lost and the associated variable is set to one.

Direct-access input-output transfer is accomplished by READ and WRITE statements of the following form (a single quote) (') may be used instead of # and the END and ERR options may be included):

(1) READ (d#r,f) list
(2) READ (d#r) list
(3) READ (d#r,) list
(4) READ (d#r,f)
(5) READ (d#r)
(6) WRITE (d#r,f) list
(7) WRITE (d#r) list
(8) WRITE (d#r,) list
(9) WRITE (d#r,f)
(10) WRITE (d#r)

These statements have the same effect as their counterparts do on sequential input-output. The major distinction is that record number (r) must be supplied. This indicates the record number relative to the beginning of the file. The same device number may be given for direct-access READ's and WRITE's; if a file is to be read sequentially and direct-access read or written, two logical device numbers must be specified.

6.3.4 The ENCODE/DECODE Statements

General Form:	{ENCODE / DECODE} (c,v,f,ERR=n) list
Where	c = number of ASCII characters per logical record v = name of array containing ASCII record f = optional format reference ERR=n is as for input-output and optional list is a list of internal variables

The ENCODE statement converts binary data stored in the variables listed, converts them to characters, and stores them in the array (v). If more variables are listed than can be stored in that array, the values are ignored. If the values are not sufficient to fill the array, blanks are added. Conversion is performed according to a FORMAT specification (f) or, in its absence, according to the data-directed input-output rules. In the latter case, the data-directed output form 'VAR' = value is stored. Forms-control characters are not analyzed on ENCODE and will be stored explicitly if in the FORMAT used.

The DECODE statement converts character data stored in the array (v) into binary and assigns them to variables in the I/O list.

6.3.5 Auxiliary Input-Output

Two library subroutines, SEEK and ENTER, are available for both sequential and direct-access input-output to disk. ENTER is called as follows:

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

where N is a device number and A is the name of an array containing an ASCII file name and extension. The effect of ENTER is to initialize and open a named output file. For example:

```
DIMENSION FILEN(2)
DATA FILEN(1),FILEN(2)/5HFILNA,4HM EXT/
CALL ENTER (1,FILEN)
```

establishes a file named FILENAM EXT.

SEEK finds and opens a named file for input. It is called as follows:

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

where N and A are as for ENTER. To input the file established by ENTER, the call would be:

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

The device-control statements listed below are applicable to sequential files on magnetic tape or disk with the effects stated.

Statement	Magnetic Tape	Disk*
BACKSPACE d	Repositions unit to the preceding record (if at beginning, does nothing)	Repositions unit one ASCII line or one logical unformatted record (INPUT only)
REWIND d	The tape is positioned to its initial point	The file is closed and reopened for input or output by the next sequential I/O statement to that device. If no file is opened, does nothing
ENDFILE d	Writes an end-of-file record	Closes the named file which is open on device d

The ENDFILE statement may be used to create segmented files with magnetic tape; that is, using the end-of-file indicator to separate the segments. For example, the statements:

```
WRITE (3) list
      .
      .
      .
WRITE (3) list
ENDFILE 3
WRITE (3,10) list
ENDFILE 3
      .
      .
      .
WRITE (3) list
ENDFILE 3
```

creates three segments. A segmented file can be read using the END=n option of the READ statement as follows:

```
   READ (3, END=10) list
         .
         .
         .
10 READ (3, END=20) list
         .
         .
         .
20 READ (3, END=30) list
         .
         .
         .
30 REWIND 3
```

Figure 6-1 shows a program using auxiliary input-output statements for disk. Note that the first series of WRITE statements cause a default file (.TM001 OTS) to be created. This file is read by subsequent READ statements specifying logical device 1.

*Note that the first sequential I/O statement to the disk opens a default file if no file has been explicitly opened.

```
        DIMENSION I(10),II(10)
199     DO 1 K=1,10
1       I(K)=K
        REWIND 1
        REWIND 7
        WRITE (1) I
        DO 2 K=11,20
2       I(K-10)=K
        WRITE (1) I
        DO 3 K=21,30
3       I(K-20)=K
        WRITE (1) I
        ENDFILE 1                     /file .TM001 OTS created
        READ (1) II                   /.TM001 OTS sought automatically
        JT=1
        JV=II(5)
        IF(II(5)-5)90,4,90
4       READ(1)II
        JT=2
        JV=II(5)
        IF(II(5)-15)90,5,90
5       READ(1) II
        JT=3
        JV=II(5)
        IF(II(5)-25)90,6,90
6       BACKSPACE 1                   /backspacing binary data
        BACKSPACE 1
        READ (1) II
        JT=4
        JV=II(7)
        IF(II(7)-17)90,7,90
7       CONTINUE
        PAUSE 1
        JT=5
        WRITE(7,70)I                  /file .TM007 OTS created
70      FORMAT(1X,I6)
        REWIND 7
        DO 8 K=1,7
8       READ (7,71)JV                 /file .TM007 OTS sought
71      FORMAT(I6)
        IF (JV-27)90,9,90
9       JT=6
        DO 11 K=1,3
11      BACKSPACE 7                   /backspacing ASCII data
        READ (7,71) JV
        IF (JV-25)90,12,90
12      PAUSE 2
        GO TO 199
90      WRITE(2,91) JT,JV
91      FORMAT(1X,I6,2X,I6)
        STOP
        END
```

Figure 6-1 Programming Example - Auxiliary I/O to Disk

APPENDIX A
LANGUAGE SUMMARY

Statement	Model	Effect	Text Reference
Arithmetic	var=value array(i)=value	value is assigned to var or array (i)	2.1
ASSIGN	ASSIGN n TO label	Statement n is assigned the symbol name label	2.2
BLOCK DATA	BLOCK DATA	Identifies subprogram which enters data into a labeled COMMON block at run time	4.4
CALL	CALL subr(a_1,a_2,...a_n) CALL subr	Control is transferred to the subroutine; a_1,a_2,... a_n are substituted for dummy variables	5.2.2
COMMON	COMMON/b_1/$vlist_1$/b_2/$vlist_2$/...	vlist items are allocated to b blocks where they are shared by other programs	4.2.2
CONTINUE	CONTINUE	Dummy statement used to prevent illegal termination of DO loops	3.2.3
DATA	DATA $vlist_1$/$clist_1$/, $vlist_2$/$clist_2$/,... $vlist_n$/$clist_n$/	clist is assigned to its corresponding vlist	4.3
DECODE	DECODE(c,v,f,ERR=n)list	Converts character data stored in the array (v) into binary and assigns them to variables in list	6.3.4
DIMENSION	DIMENSION $a_1(I_1)$,$a_2(I_2)$,...$a_n(I_n)$	Storage is allocated for array (a) to the dimensions specified by the subscript list (I)	4.2.1
DO	DO n i=m_1,m_2,m_3 DO n i=m_1,m_2 DO n i=m_1,m_2,-m_3	Statements following the DO are executed repeatedly for values m_1 through m_2 in increments or decrements of m_3	3.2

(continued on next page)

Statement	Model	Effect	Text Reference
ENCODE	ENCODE(c,v,f,ERR=n)list	Converts binary data represented by variables in list into characters according to FORMAT specification (f) or data-directed I/O rules and stores them in the array (v)	6.3.4
EQUIVALENCE	EQUIVALENCE$(l_1),(l_2),\ldots(l_n)$	Elements of each list (l) are assigned to the same storage location	4.2.3
EXTERNAL	EXTERNAL $a_1,a_2,\ldots a_n$	Defines subprograms named a for use as arguments of other subprograms	4.1.3
FORMAT	n FORMAT$(s_1,s_2,\ldots s_n)$	FORMAT statement n established as field-specification references	6.1
FUNCTION	m FUNCTION $f(a_1,a_2,\ldots a_n)$	Defines FUNCTION named f with dummy arguments a and optional mode specification m	5.1.2
GO TO	GO TO n	Control is unconditionally transferred to statement n	3.1.1
	GO TO$(n_1,n_2,\ldots n_k)$,i	Control is transferred to the ith statement in the list of n's	3.1.2
	GO TO label GO TO label,$(n_1,n_2,\ldots n_k)$	Control is transferred to the location specified by label; the list of n's may specify legally ASSIGNable statement numbers	3.1.3
IF	IF(expr)n_1,n_2,n_3	Control is transferred to statement number or ASSIGNed label n_1, n_2, or n_3 if evaluated expr is <0, =0, or >0 respectively	3.3.1
	IF(expr)s	Statement s is executed if expr is .TRUE. (non-zero), ignored if .FALSE. (zero)	3.3.2
IMPLICIT	IMPLICIT $m_1(l_1),m_2(l_2),\ldots m_n(l_n)$	Declares mode (m) for variables beginning with alphabetic characters in list (l)	4.1.2

(continued on next page)

Statement	Model	Effect	Text Reference
PAUSE	PAUSE PAUSE n	Interrupts program execution; if present, integer n is printed on the console to distinguish one PAUSE from another	3.4.1
PRINT	PRINT(d,f)list	The values of variables in list are converted to ASCII according to FORMAT reference (f) and transferred to external device (d)	6.3.2
	PRINT(d)list	The values of variables in list are written in binary on external device (d)	6.3.2
	PRINT(d,)list	The variable names in list are written on external device (d), each followed by its value in the form 'A'=value	6.3.2
	PRINT(d,f)	FORMAT reference (f) is written on external device (d)	6.3.2
READ	READ(d,f)list	The values represented by variables in list are read from external device (d) and converted according to FORMAT reference (f)	6.3.2
	READ(d)list	The binary values represented by variables in list are read from external device (d)	6.3.2
	READ(d,)list	The values represented by variables in list are read from external device (d)	6.3.2
	READ(d,f)	Values are read into FORMAT reference (f)	
	READ(d)	A binary record is read from external device (d) and ignored	6.3.2
STOP	STOP STOP n	Signifies the logical end of a program and returns control to the MONITOR after n is printed; if present, n distinguishes one STOP from another	3.4.2

(continued on next page)

Statement	Model	Effect	Text Reference
SUBROUTINE	SUBROUTINE name $(a_1,a_2,...a_n)$ SUBROUTINE name	Defines an external subroutine named name; a's are dummy arguments representing values supplied by the calling program or returned by the subroutine	5.2.1
TYPE	TYPE(d,f)list	The values of variables in list are converted to ASCII according to FORMAT reference (f) and transferred to external device (d)	6.3.2
	TYPE(d)list	The values of variables in list are written in binary on external device (d)	6.3.2
	TYPE(d,)list	The variable names in list are written on external device (d), each followed by its value in the form 'A'=value	6.3.2
	TYPE(d,f)	FORMAT reference (f) is written on external device (d)	6.3.2
WRITE	WRITE(d,f)list	The values of variables in list are converted to ASCII according to FORMAT reference (f) and transferred to external device (d)	6.3.2
	WRITE(d)list	The values of variables in list are written in binary on external device (d)	6.3.2
	WRITE(d,)list	The variable names in list are written on external device (d), each followed by its value in the form 'A'=value	6.3.2
	WRITE(d,f)	FORMAT reference (f) is written on external device (d)	6.3.2

APPENDIX B
ERROR MESSAGES

B.1 COMPILER ERROR MESSAGES

In the F4X version of FORTRAN, compiler error messages are printed in the form:

>mnA<

where:

mn is the error number
A is the alphabetic mnemonic

characterizing the error class.

In F4I and F4A versions, only the alphabetic character is printed, in the form:

>A<

All error messages and the version(s) of FORTRAN to which they are applicable are given below.

Number	Letter	Meaning
		Common, equivalence, data errors:
01	C	No open parenthesis after variable name in DIMENSION statement
02	C	No slash after common block name
03	C	Common block name previously defined
04	C	Variable appears twice in COMMON
05	C	EQUIVALENCE list does not begin with open parenthesis
06	C	Only one variable in EQUIVALENCE class
07	C	EQUIVALENCE distorts COMMON
08	C	EQUIVALENCE extends COMMON down
09	C	Inconsistent EQUIVALENCing
10	C	EQUIVALENCE extends COMMON down
11	C	Illegal delimiter in EQUIVALENCE list

(continued on next page)

Number	Letter	Meaning
		Common, equivalence, data errors: (cont)
12	C	Non-COMMON variables in BLOCK DATA
15	C	Illegal repeat factor in DATA statement
16	C	DATA statement stores in COMMON in non-BLOCK DATA statement or in non-COMMON in BLOCK DATA statement
		DO errors:
01	D	Statement with unparenthesized = sign and comma not a DO statement
04	D	DO variable not followed by = sign
05	D	DO variable not integer
06	D	Initial value of DO variable not followed by comma
07	D	Improper delimiter in DO statement
09	D	Illegal terminating statement for DO loop
		External symbol and entry-point errors:
01	E	Variable in EXTERNAL statement not simple non-COMMON variable
02	E	ENTRY name non-unique
03	E	ENTRY statement in main program
04	E	No = sign following argument list in arithmetic statement function
05	E	No argument list in FUNCTION subprogram
06	E	Subroutine list in CALL statement already defined as variable
08	E	Function or array name used in expression without open parenthesis
09	E	Function or array name used in expression without open parenthesis
		Format errors:
01	F	Bad delimiter after FORMAT number in I/O statement
02	F	Missing field width, illegal character or unwanted repeat factor
03	F	Field width is 0
04	F	Period expected, not found
05	F	Period found, not expected
06	F	Decimal length missing (no "d" in "Fw.d")
07	F	Unparenthesized comma

(continued on next page)

Number	Letter	Meaning
		Format errors: (cont)
08	F	Minus without number
09	F	No P after negative number
10	F	No number before P
12	F	No number or 0 before H
13	F	No number or 0 before X
15	F	Too many left parentheses
		Hollerith errors:
03	H	Number preceding H not between 1 and 5
04	H	Carriage return inside Hollerith field
05	H	Number preceding H not an integer
06	H	More than five characters inside quotes
07	H	Carriage return inside quotes
		Various illegal errors:
01	I	Unidentifiable statement
02	I	Misspelled statement
03	I	Statement out of order
04	I	Executable statement in BLOCK DATA subroutine
05	I	Illegal character in I/O statement, following unit number
06	I	Illegal delimiter in ASSIGN statement
07	I	Illegal delimiter in ASSIGN statement
08	I	Illegal type in IMPLICIT statement
09	I	Logical IF as target of logical IF
10	I	RETURN statement in main program
11	I	Semicolon in COMMON statement outside of BLOCK DATA
12	I	Illegal delimiter in IMPLICIT statement
13	I	Misspelled REAL or READ statement
14	I	Misspelled END or ENDFILE statement
15	I	Misspelled ENDFILE statement
16	I	Statement function out of order or undimensioned array
17	I	Typed FUNCTION statement out of order
18	I	Illegal character in context
19	I	Illegal logical or relational operator

(continued on next page)

Number	Letter	Meaning
		Various illegal errors: (cont)
20	I	Illegal letter in IMPLICIT statement
21	I	Illegal letter range in IMPLICIT statement
22	I	Illegal delimiter in letter section of IMPLICIT statement
23	I	Illegal character in context
24	I	Illegal comma in GOTO statement
26	I	Illegal variable used in multiple RETURN statement
		Pushdown list errors:
01	L	DO nesting too deep
02	L	Illegal DO nesting
03	L	Subscript/function nesting too deep
04	L	Backwards DO loop (also caused by some illegal I/O lists). Appears after END statement.
		Overflow errors:
01	M	EQUIVALENCE class list full
02	M	Program size exceeds 8K
03	M	Array length larger than 8K
04	M	Element position in array larger than 8K (EQUIVALENCE, DATA)
06	M	Integer negative or larger than 131071
07	M	Exponent of floating point number larger than 76
08	M	Overflow accumulating constant - too many digits
09	M	Overflow accumulating constant - too many digits
10	M	Overflow accumulating constant - too many digits
		Statement number errors:
01	N	Multiply defined statement number or compiler error
02	N	Statement erroneously labeled
03	N	Undefined statement number
04	N	FORMAT statement without statement number
05	N	Statement number expected, not found
07	N	Statement number more than five digits
08	N	Illegal statement number

(continued on next page)

Number	Letter	Meaning
		Partword errors:
01	P	Expected colon, found none
02	P	Expected close bracket, found none
03	P	Last bit number larger than 35
04	P	First bit number larger than last bit number
05	P	First and last bit numbers not simple integer constants
		Subscripting errors:
01	S	Illegal subscript delimiter in specification statements
02	S	More than three subscripts specified
03	S	Illegal delimiter in subroutine argument list
04	S	Non-integer subscript
05	S	Non-scalar subscript
06	S	Integer scalar expected, not found
10	S	Two operators in a row
11	S	Close parenthesis following an operator
12	S	Non-integer subscript
13	S	Non-scalar subscript
14	S	Two arguments in a row
15	S	Digit or letter encountered after argument conversion
16	S	Number of subscripts stated not equal to number declared
		Table overflow errors:
01	T	Arithmetic statement, computed GOTO list, or DATA statement list too large
02	T	Too many dummy variables in arithmetic statement function
03	T	Symbol and constant tables overlap
		Variable errors:
01	V	Two modes specified for same variable name
02	V	Variable expected, not found
03	V	Constant expected, not found
03	V	Array defined twice
05	V	Error: variable is EXTERNAL or argument (EQUIVALENCE, DATA)
07	V	More than one dimension indicated for scalar variable

(continued on next page)

Number	Letter	Meaning
		Variable errors: (cont)
08	V	First character after READ or WRITE not open parenthesis in I/O statement
09	V	Illegal constant in DATA statement
11	V	Variables outnumber constants in DATA statement
12	V	Constants outnumber variables in DATA statement
14	V	Illegal dummy variable (previously used as non-dummy variable)
16	V	Logical operator has non-integer, non-logical arguments
17	V	Illegal mixed mode expression
19	V	Logical operator has non-integer, non-logical arguments
21	V	Signed variable left of equal sign
22	V	Illegal combination for exponentiation
25	V	.NOT. operator has non-integer, non-logical argument
27	V	Function in specification statement
28	V	Two exponents in one constant
29	V	Illegal redefinition of a scalar as a function
30	V	No number after E or D in a constant
32	V	Non-integer record number in random access I/O
35	V	Illegal delimiter in I/O statement
36	V	Illegal syntax in READ, WRITE, ENCODE, or DECODE statement
37	V	END and ERR exists out of order in I/O statement
38	V	Constant and variable modes don't match in DATA statement
39	V	ENCODE or DECODE not followed by open parenthesis
40	V	Illegal delimiter in ENCODE/DECODE statement
41	V	Array expected as first argument of ENCODE/DECODE statement
42	V	Illegal delimiter in ENCODE/DECODE statement
		Expression errors:
01	X	Carriage return expected, not found
02	X	Binary WRITE statement with no I/O list
03	X	Illegal element in I/O list
04	X	Illegal statement number list in computed or assigned GOTO
05	X	Illegal delimiter in computed GOTO
07	X	Illegal computed GOTO statement

(continued on next page)

Number	Letter	Meaning
		Expression errors: (cont)
10	X	Illegal delimiter in DATA statement
11	X	No close parenthesis in IF statement
12	X	Illegal delimiter in arithmetic IF statement
13	X	Illegal delimiter in arithmetic IF statement
14	X	Expression on left of equals sign in arithmetic statement
15	X	Too many right parentheses
16	X	Illegal open parenthesis (in specification statements)
17	X	Illegal open parenthesis
19	X	Too many right parentheses
20	X	Illegal alphabetic in numeric constant
21	X	Symbol contains more than six characters
22	X	.TRUE., .FALSE., or .NOT. preceded by an argument
23	X	Unparenthesized comma in arithmetic expression
24	X	Unary minus in I/O list
26	X	Illegal delimiter in I/O list
27	X	Unterminated implied - DO loop in I/O list
28	X	Illegal equals sign in I/O list
29	X	Illegal partword operator
30	X	Illegal arithmetic expression

B.2 OTS ERROR MESSAGES

Following is a list of OTS error messages. (R) indicates a recoverable error; (T) a terminal error.

Error Number	Error Description	Possible Source
05 (R)	Negative REAL square root argument	SQRT
06 (R)	Negative DOUBLE PRECISION square root argument	DSQRT
07 (R)	Illegal index in computed GO TO	.GO
10 (T)	Illegal I/O device number	.FR, .FW, .FS, .FX, DEFINE, RANCOM
11 (T)	Bad input data - IOPS mode incorrect	.FR, .FA, .FE, .FF, .FS, RANCOM, RBINIO, RBCDIO

(continued on next page)

Error Number	Error Description	Possible Source
12 (T)	Bad FORMAT	.FA, .FE, .FF
13 (T)	Negative or zero REAL logarithmic argument (terminal)	.BC, .BE, ALOG
14 (R)	Negative or zero DOUBLE PRECISION logarithmic argument	.BD, .BF, .BG, .BH, DLOG, DLOG10
15 (R)	Zero raised to a zero or negative power (zero result is passed)	.BB, .BC, .BD, .BE, .BF, .BG, .BH
20 (T)	Fatal I/O error (RSX only)	FIOPS
direct access errors: 21 (T)	Undefined file	RANCOM
22 (T)	Illegal record size	DEFINE
23 (T)	Size discrepancy	RANCOM
24 (T)	Illegal record number	DEFINE, RANCOM
25 (T)	Mode discrepancy	RANCOM
26 (T)	Too many open files	DEFINE
30 (R)	Single integer overflow*	RELEAE, .FPP
**31 (R)	Extended (double) integer overflow****	DBLINT, JFIX, JDFIX, ISNGL
**32 (R)	Single flt. overflow	RELEAE
**33 (R)	Double flt. overflow†	
**34 (R)	Single flt. underflow	RELEAE
**35 (R)	Double flt. underflow†	
**36 (R)	Flt. divide check	RELEAE
***37 (R)	Integer divide check	INTEAE
40 (T)	Illegal number of characters specified [legal: $o<c\leq 625$]	ENCODE
41 (R)	Array exceeded	ENCODE
42 (T)	Bad input data	DD10
**50 (T)	FPP memory protect/non-existent memory	
51 (T)	(READ to WRITE Illegal I/O Direction Change to Disk) without intervening CLOSE or REWIND	BCDIO, BINIO

*Only detected when fixing a floating point number.
**Also prints out PC with FPP system
***If extended integer divide check, prints out PC with FPP system.
****With software F4 system only detected when fixing a floating point number.
†Not detected by software system (only by FPP system).

B.3 OTS ERROR MESSAGES IN FPP SYSTEMS

In software systems, arithmetic errors resulting in the OTS error messages summarized above are detected in the arithmetic package (RELEAE and INTEAE). In the hardware FPP systems, these errors are detected by the hardware (with the exception of single integer divide check) and serviced by a trap routine in the FPP routine .FPP.

Where applicable, on such error conditions, the result is patched for both software and hardware systems as summarized in the following table.

Error	PATCHED VALUE***	
	FPP Hardware System	Software System
Single Floating Overflow (.OTS 32)	± largest single floating value	same
Double Floating Overflow (.OTS 33)	± largest single floating value	not detected
Single Floating Underflow (.OTS 34)	zero	same
Double Floating Underflow (.OTS 35)	zero	not detected
Floating Divide Check (.OTS 36)	± largest single floating value	same
Integer Overflow (.OTS 30)	limited detection*	same
Double Integer Overflow (.OTS 31)	none**	limited detection*
Integer Divide Check (.OTS 37)	none	same

*When fixing a floating point number, integer and extended integer overflow is detected. In these instances, plus or minus the largest integer for the data mode is patched as result.

**With the FPP system all extended integer overflow conditions are detected, but the results are meaningless.

***Where "none" is specified, the result is meaningless unless otherwise indicated.

Further, when converting an extended integer, the magnitude of which is $>2^{17}-1$, to a single integer, no error is indicated and the high order digits are lost.

APPENDIX C
PDP-15 FORTRAN FACILITIES

The extended FORTRAN language described in this manual and in the companion manual (Operating Environment Manual DEC-15-GFZA-D) is available only on the systems described below. The FORTRAN existing on other PDP-15 systems is described in a manual entitled "PDP-15 FORTRAN IV Programmer's Reference Manual" (DEC-15-KFZB-D).

The following tables describe the existing versions of the extended compiler, the extended Object Time System Libraries, and the compiler-library pairs available for different systems. All versions of the compiler are written in PDP-9 code, however, 'PDP-9 mode' versions produce only PDP-9 code as output while 'PDP-15 mode' versions may produce PDP-15 instructions where suitable. Page and Bank Mode libraries differ not only in the use of the PDP-15 versus PDP-9 code, but also in the values of address masking constants used in a few of the routines. Note that the Floating Point Processor (FPP) is supported only on the PDP-15, thus there is no PDP-9 mode version.

The library names used in the following tables are given for designational purposes within this appendix only and do not necessarily reflect the names under which the libraries are distributed.

Table C-1
Versions of the Extended Compiler

Main Version	Features	Version	System	Approx. Size (8)
F4X	All	F4X F4X9 FPF4X	Non-FPP, PDP-15 mode DOS-15 Non-FPP, PDP-9 mode DOS-15 FPP, PDP-15 mode DOS-15	15406 15363 15661
F4B	All except direct-access I/O	F4B F4B9 FPF4B	Non-FPP, PDP-15 mode, ADSS (V5B) Non-FPP, PDP-9 mode ADSS (V5B) FPP, PDP-15 mode ADSS (V5B)	15251 15226 15522
F4RX	All except direct-access I/O	F4RX FPF4RX	Non-FPP, PDP-15 mode RSX FPP, PDP-15 mode RSX	

Table C-2
Versions of the OTS Libraries for the Extended Compiler

System	Contents	Libraries	Subsystem
DOS-15 (BOSS-15)	Contains all routines, assembled for DOS-15 operation.	.LBXP .LBXB .LBXPF .LBXBF	Non-FPP, Page Non-FPP, Bank FPP, Page FPP, Bank
ADSS	Contains all routines except direct-access (DEFINE, RANCOM, RBINIO, RBCDIO) assembled for ADSS operation.	.LBRP .LBRB .LBRPF .LBRBF	Non-FPP, Page Non-FPP, Bank FPP, Page FPP, Bank
RSX	Contains all routines except direct-access (DEFINE, RANCOM, RBINIO, RBCDIO) and magtape subroutines (UNIT, EOF), assembled for RSX operation and includes added routines applicable to RSX only.	.LIBRX .LIBFX	Non-FPP, Page/Bank FPP, Page/Bank

Table C-3
Compilers and Libraries for Extended FORTRAN
Distributed with PDP-9/15 Systems

System		Non-FPP		FPP	
		Page	Bank	Page	Bank
DOS-15	Compiler	F4X	F4X or F4X9	FPF4X	FPF4X
(BOSS-15	Library	.LBXP	.LBXB	.LBXPF	.LBXBF
ADSS V5B	Compiler	F4B	F4B or F4B9	FPF4B	FPF4B
	Library	.LBRP	.LBRB	.LBRPF	.LBRBF
RSX	Compiler	F4RX	F4RX	FPF4RX	FPF4RX
	Library	.LIBRX	.LIBRX	.LIBFX	.LIBFX

INDEX

D (Cont)

E

E (Cont)

F

F (Cont)

G

H

I

I (Cont)

L

READER'S COMMENTS

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

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

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

How can this manual be improved?

Other comments?

Please state your position. ______________________ Date: __________

Name: ______________________ Organization: ______________________

Street: ______________________ Department: ______________________

City: ______________ State: ______________ Zip or Country __________

- - - - - - - - - - - - - - Fold Here -

- - - - - - - - - - - - - - Do Not Tear - Fold Here and Staple - - - - - - - - - - - - - -

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

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

Postage will be paid by:

digital

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

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections are published by Software Information Service in the following newsletters.

Digital Software News for the PDP-8 & PDP-12
Digital Software News for the PDP-11
Digital Software News for the PDP-9/15 Family

These newsletters contain information applicable to software available from Digital's Program Library, Articles in Digital Software News update the cumulative Software Performance Summary which is contained in each basic kit of system software for new computers. To assure that the monthly Digital Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest Digital office.

Questions or problems concerning Digital's Software should be reported to the Software Specialist. In cases where no Software Specialist is available, please send a Software Performance Report form with details of the problem to:

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

These forms which are provided in the software kit should be fully filled out and accompanied by teletype output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software and manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest Digital Field office or representative. U.S.A. customers may order directly from the Program Library in Maynard. When ordering, include the code number and a brief description of the software requested.

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

DECUS
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

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

Postage will be paid by:

digital

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