LOGICON 2+2

FORTRAN MANUAL

LOGICON, INC.
1075 CAMINO DEL RIO, SOUTH
SAN DIEGO, CALIFORNIA

15 December 1970

TABLE OF CONTENTS

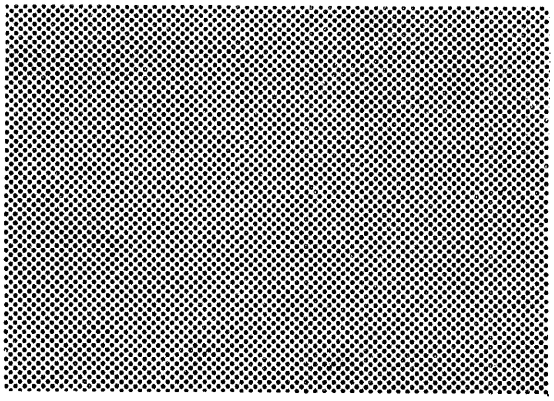
TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES



I...

Introduction

GENERAL

The LOGICON 2+2 FORTRAN is an automatic coding language resembling the language of algebra. It provides the facility for readily expressing problems requiring numerical computation. In particular, it can easily handle problems involving large sets of equations and problems that contain many variables. FORTRAN is especially suited for solving scientific and engineering problems, while by the same merits it is also suitable for many business applications.

The FORTRAN language consists of words and symbols arranged into statements. A set of FORTRAN statements, describing each step in the solution of a problem, is a FORTRAN program (a source language program).

The basic unit of the FORTRAN language is the statement. Statements may be classified according to the following groups:

1. Arithmetic statements specifying numerical or logical computations.

2. Control statements governing the order of execution in the program.

3. Input/Output statements and input/output formats which describe the form of the data.

4. Subprogram statements enabling the programmer to define and use subprograms.

5. Specification statements providing information about variables used in the program, information about storage allocation, and data assigned.

To write FORTRAN programs effectively, it is necessary to understand the usage of the following terms and concepts:

1. Constants, such as 27 or 3.14159
2. Variables, such as X or TEMP3
3. Subscripted variables, such as X(I) or Y(I, J)
4. Mathematical expressions, such as X + Y or 3 * J
5. Arithmetic statements, such as A = B/C
6. Control statements which specify the sequence of control, such as GO TO 23 or IF (Z) 10, 15, 65
7. Input/Output statements used for getting data into the computer and producing hardcopy results, such as READ (5, 37)A, I, J or WRITE (6, 43) W(6)
8. Subroutine and function statements permitting programs to be incorporated into larger programs
9. Specification statements, such as, DIMENSION and COMPLEX

The language of FORTRAN is augmented by the availability of prewritten routines accompanying the system. These routines evaluate the standard arithmetical functions, provide all input/output for the program, and furnish the user with other services to aid in the problem solution. Special purpose routines may be written by the user to be used as subprograms.

The operations of addition and subtraction are indicated in the same was as in mathematical notations; that is, the symbols plus (+) and minus (-) Multiplication is denoted by the asterisk (*) while division is denoted by the slash (/). The double asterisk (**) or the up arrow (↑) is the FORTRAN operation sign for exponentiation. The rule for using this sign is that the quantity to the left of the sign is raised to the power indicated on the right.

FORMAT RULES FOR STATEMENT WRITING

Line Number

From one to five digits, less than 65536, including leading zeros. Line number field is terminated by a non-numeric character or the sixth digit. A program is processed in ascending line number order.

Comment

The character C immediately following the line number indicates a comment statement. The comment appears in the program listing but it does not affect the logic of the FORTRAN program.

Statement Label

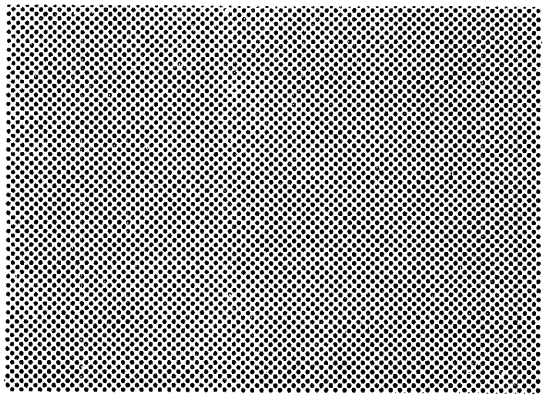
A FORTRAN program statement may optionally contain a statement label. This numeric field (1 thru 99999) follows the line number and precedes the FORTRAN statement text. Blanks and leading zeros are ignored.

Continuation

A statement may be continued on more than one line by placing an ampersand (&) immediately before the carriage return on the first line and continuing the statement on the next line. Control may not be transferred to the line number associated with the continued portion of a statement. Continued statements may not contain a comment statement and comment statements may not be continued.

Statement

The first alphanumeric character on the line identifies the actual FORTRAN statement. Blanks do not have significance except where they appear in a specific string of characters, e.g., in an H field or a quote string.



II...

Constants, Variables, Subscripts & Exprssions

Generally, a constant is a number in a mathematical expression that is known prior to writing the FORTRAN STATEMENT, and whose value does not change during program execution. Conversely, a variable represents a number that is not known when a statement is written, and it is able to take on different values during program execution. When a single variable name is used to represent a list of numbers, it is called a subscripted variable.

In FORTRAN, an expression is a combination of constants, variables, and operation signs which defines a series of related mathematical operations. Thus, using constants and variables, FORTRAN provides a means of expressing quantities specified in an arithmetic formula statement. The constants and variables may be either the fixed-point integer or floating-point mode. The fixed-point mode is used for counting and other operations involving whole (integer) numbers. The floating-point mode is used for nearly all computation. The floating-point number, consisting of an exponent and a fraction, accommodates a greater range of values.

CONSTANTS

Integer

An integer constant requires one memory location and is a number in the range -32767 to +32767. The decimal point of the integer is always omitted; however, it is always assumed to be immediately to the right of the rightmost digit.

Extended Integer

An extended integer requires three contiguous memory locations and is a number in the range -140,737,488,355,327 to +140,737,488,355,327. The decimal point of the extended integer is always omitted; however, it is always assumed to be immediately to the right of the rightmost digit.

Real

A real constant requires three contiguous memory locations and is maintained in floating point mode.

1. One to ten significant decimal digits written with a decimal point, but not followed by a decimal exponent.

2. A sequence of decimal digits written with or without a decimal point, followed by a decimal exponent written as the the letter E followed by a signed or unsigned integer constant. When the decimal point is omitted, it is always assumed to be immediately to the right of the rightmost digit. The exponent value may be explicitly 0, and the field following the E may not be blank.

A real constant has precision to nine digits.

Double Precision

A double-precision constant requires four contiguous memory locations and is maintained in floating-point mode. It is a sequence of decimal digits written with or without a decimal point, followed by a decimal exponent written as the letter D followed by a signed or unsigned integer constant. When the decimal point is omitted, it is always assumed to be immediately to the right of the rightmost digit. The exponent value may be explictly 0, and the field following the D may not be blank.

Examples:

7.0D4 (means 7.0×10^4, 70000.)
7.0D-3 (means 7.0×10^{-3}, .007)

Double-precision constants have precision to 14 digits.

Complex

A complex constant requires six contiguous memory locations and consists of an ordered pair of signed or unsigned real constants separated by a comma and enclosed in parentheses.

Examples:

(10.1, 7.03) is equal to 10.1 + 7.03i
(5.41, 0.0) is equal to 5.41 + 0.0i
(7.0E4, 20.76) is equal to 70000. + 20.76i

where i is the square root of -1.

The first real constant represents the real part of the complex number; the second real constant represents the imaginary part of the complex number. The parentheses are required regardless of the context in which the complex constant appears. Each part of the complex constant may be preceded by a plus sign or a minus sign, or it may be unsigned.

Logical

A logical constant requires one memory location and may take either of two values:

.TRUE.
.FALSE.

and is represented in the machine as

TRUE≠0
FALSE=0

String Constant

A string constant requires a memory location for the character count, followed by two ASCII characters per location. When the character count is odd, a blank fills the last location.

2 forms - (1) string nHxxx where n is character count
(2) string enclosed in quotes "xxx"
Neither string may contain FORTRAN or TTY control characters.

IO NAME Constant

An IO NAME constant requires four contiguous memory locations and consists of one to eight ASCII characters, right justified and blank filled.

VARIABLES

A variable name may have from one to eight alphabetic or numeric characters. The first character must be alphabetic.

Examples:
K3
SUM5
ANSWER
NOTE7

The mode of the variable may be specified in one of two ways: Implicitly by name or explicitly by a Type statement (see "Type Statement" and "Naming Subroutines" in the Index). Implicit type assignment

pertains only to integer and real, variable and function names, and is determined by the first character in the variable name. If the first character is I, J, K, L, M, or N, it is a fixed-point (integer) variable; if it begins with any other letter, it is a floating-point variable. Refer to Implicit statement.

SUBSCRIPTS

When a single variable name is to represent a list of values (array), subscripts provide a means of referring to a specific member of that list. The subscripts are arithmetic expressions whose value determines the member of the array to which reference is made. The array being referenced is of a predetermined length; therefore, the value of a subscript expression cannot be zero, less than zero, or greater than the dimensions (see dimensions in the Index) declared for the array referenced.

A subscripted variable consists of a variable name followed by parentheses enclosing arithmetic subscripts expressions that are separated by commas.

Each variable that appears in subscripted form must have the size of the array specified. (See DIMENSION, COMMON, or Type Statement in Index.)

Arrays are stored in column order in increasing storage locations, with the first of their subscripts varying most rapidly, and the last varying least rapidly. For example, the two-dimensional array A(m, n) is stored as follows, from the lowest core storage location to the highest:

$A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, A_{2,2}, \ldots, A_{m,2}, \ldots, A_{1,n}, A_{2,n}, \ldots, A_{m,n}$

EXPRESSIONS

Arithmetic Expressions

An arithmetic expression consists of certain legal sequences of constants, subscripted and nonsubscripted variables, and arithmetic function references separated by arithmetic operation symbols, commas, and parentheses.

The following arithmetic operation symbols denote addition, subtraction, multiplication, division, and exponentiation (** or ↑), respectively:

+ - * / ** ↑

The rules for constructing arithmetic expressions are:

1. There are no mode restrictions when constructing expressions with respect to +, -, *, and /. Operands for these operators are promoted to the higher mode before the operation is performed. Figure 2-1, shows the mode of +, -, *, and /. Figure 2-2, shows the valid combinations with respect to the ** or ↑ operator.

	I	EI	R	D	C
I	I	EI	R	D	C
EI	EI	EI	R	D	C
R	R	R	R	D	C
D	D	D	D	D	C
C	C	C	C	C	C

Legend

C - Complex
D - Double precision
EI - Extended Integer
I - Integer
R - Real

Figure 2-1. Arithmetic Expressions + - * /

	I	EI	R	D	C
I	I	EI	R	D	C
EI	EI	EI	R	D	C
R	R	R	R	D	C
D	D	D	D	D	C
C	C	C	C	C	C

C - Complex
D - Double precision
EI - Extended Integer
N - Nonvalid
R - Real

Figure 2-2. Arithmetic Expressions – Exponent (** or ↑)

2. Any expression may be enclosed in parentheses.

3. Expressions may be connected by the arithmetic operation symbols to form other expressions, provided that:

   a. No two operators appear in sequence except **, which is a single operator and denotes exponentiation.

b. No operation symbol is assumed to be present. For example, (X)(Y) is not valid.

4. Preceding an expression by a plus or minus sign does not affect the arithmetic type of the expression.

5. In the hierarchy of operations, parentheses may be used in arithmetic expressions to specify the order in which operations are to be computed. Where parentheses are omitted, the order is understood to be as follows (from innermost operations to outermost operations):

   a. Function Reference
   b. ** or ↑ Exponentiation
   c. * and / Multiplication and Division
   d. + and - Addition and Subtraction

   This hierarchy is applied first to the expression within the innermost set of parentheses in the statement; this procedure continues through the outer parentheses until the statement has been evaluated.

6. Defining a term as an unparenthesized sequence of primary operands and exponentiation pairs separated by * and / operators only, the rule for the order of evaluation of arithmetic expressions in the absence of parentheses is: Within a term, the evaluation is left to right; across the expression, terms are evaluated right to left; the sum of the terms is formed from left to right.

The FORTRAN expression

```
A*6.+Z/Y**(W+(A+B)/X**K)
```

represents the mathematical expression

$$6A + \frac{Z}{Y\left(W + \frac{(A+B)}{X^K}\right)}$$

Even if operators are on the same level, parentheses may be used if a particular order of computation is required by the program.

Given I, R, and C as names of integer, real, and complex variables respectively, the expression R+C/I is evaluated by promoting I to complex, performing the division; promoting R to complex and forming the sum.

R+I+I/I is evaluated by finding the integer quotient I/I; converting I to real and computing the sum R+I; promoting the integer quotient to real and adding to the real sum.

Logical Expressions

A logical expression consists of sequences of logical constants, logical variables, or references to logical functions separated by logical operation symbols.

The logical operation symbols (where a and b are logical expressions) are:

Symbol	Definition
. NOT. a	Produces the one's complement of a.
a. AND. b	Produces the logical product a AND b.
a. OR. b	Produces the logical sum a OR b.

The logical operators NOT, AND, and OR must always be preceded and followed by a period. The following are the rules for constructing logical expressions:

1. A logical expression may consist of a single logical constant, a logical variable, or a reference to a logical function.

2. The logical operator . NOT. must be followed by a logical expression, and the logical operators . AND. and . OR. must be preceded and followed by logical expressions to form more complex logical expressions.

3. Any logical expression may be enclosed in parentheses; however, the logical expression to which the . NOT. applies must be enclosed in parentheses if it contains two or more quantities.

Relational Expressions

A relational expression consists of sequences of arithmetic variables, constants, or function references separated by relational operation symbols. Relational expressions can be combined with logical expressions to produce relational expressions. Relational expressions always result in a . TRUE. or . FALSE. evaluation.

The symbols for the six relational operations are:

Symbol	Definition
. GT. or >	Greater than
. GE.	Greater than or equal to
. LT. or <	Less than
. LE.	Less than or equal to
. EQ.	Equal to
. NE.	Not equal to

Example:

A. GT. B has the value . TRUE. if the quantity A is strictly greater than the quantity B, and value . FALSE. otherwise.

The realtional operators must always be preceded and followed by a period. The following are the rules for constructing logical and relational expressions:

1. Figure 2-3 indicates which constants, variables, functions, and arithmetic expressions may be combined by the relational operators to form a relational expression. In Figure 2-3, Y indicates a valid combination and N indicates an invalid combination. The relational expression will have the value . TRUE. if the condition expressed by the relational operator is met; otherwise, the relational expression will have the value . FALSE.

.GT., .GE., .LT., .LE., .EQ., .NE.	I	EI	R	R	C	L	F	S
I	I	EI	R	D	C*	L	N	N
EI	EI	EI	R	D	C*	N	N	N
R	R	R	R	D	C*	N	N	N
D	D	D	D	D	C*	N	N	N
C	C*	C*	C*	C*	C*	N	N	N
L	N	N	N	N	N	N	N	N
F	N	N	N	N	N	N	F	N
S	N	N	N	N	N	N	N	S

*. EQ. or . NE. only

Legend

D - Double Precision
EI - Extended Integer
F - IO NAME
I - Integer
L - Logical
N - Nonvalid
R - Real
S - String

Figure 2-3. Use of Relational Operators

2. The numeric relationships that determine the true or false evaluation of relational expressions are:

   a. For numeric values having unlike signs, the positive value is considered larger than a negative value, regardless of the respective magnitude ; e.g., +3 >-5 and +5 >-5.

   b. For numeric values having like signs, the magnitude of the sign of the values determines the relationship; e.g., +3>+2 and -8<-4.

3. The logical operator . NOT. must be followed by a logical or relational expression, and the logical operators . AND. and . OR. must be preceded and followed by logical ore relational expressions to form more complex logical expressions.

Expression Evaluation

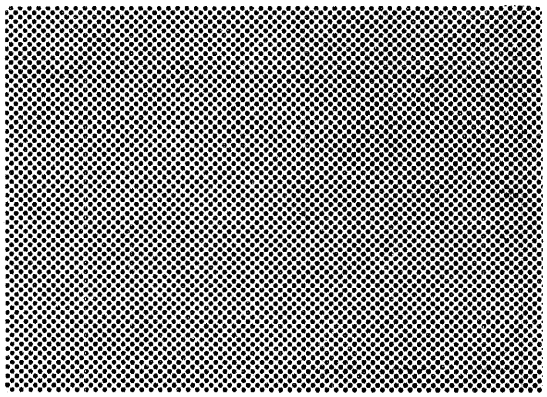
In the hierarchy of operations, parentheses may be used in logical, relational, and arithmetic expressions to specify the order in which operations are to be computed. Where parentheses are omitted, the order is understood to be as follows (from innermost operation to outermost operations):

a. Function Reference

b. ** or ↑ Exponentiation

c.	* and /	Multiplication and Division
d.	+ and -	Addition and Subtraction
e.	< ,.LT.,.LE.,.EQ., .NE., > ,.GT.,.GE.	
f.	.NOT.	
g.	.AND.	
h.	.OR.	

This hierarchy is applied first to the expression within the innermost set of parentheses in the statement; this procedure continues through the outer parentheses until the statement has been evaluated.



III...

Assignment Statement

The assignment statement, consists of a variable name (subscripted or not) followed by an equals sign, followed in turn by any desired expression. The equals sign of the FORTRAN statement implies "is replaced by" and not mathematial equality.

The expression may be a single constant, a single variable, or a complex combination of operations. In essence, the machine computes the complete expression on the right of the equals sign and assigns that computed value to the variable whose name appears on the left of the equals sign.

General Form
v = e where v is a subscripted or nonsubscripted replacement variable e is an expression the equals sign implies "is replaced by"

Examples:

```
      PI   = 3.1416
      W    = N
      E(I) = 1.+EXP(Z)
      T    = .FALSE.
      A    = R.EQ.R1.OR.V
   ALPHA = "8 CHARS"
```

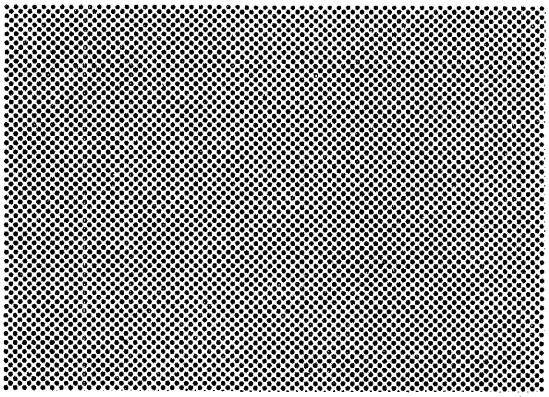
Figure 3-1 indicates the legitimate combinations of expressions and variables in an arithmetic assignment statement. In Figure 3-1, Y indicates a valid statement with the resulting type of expression on the left side of the equals sign. N indicates a nonvalid statement.

V \ E	I	EI	R	D	C	L	S	F
I	Y	Y	Y	Y	Y	N	N	N
EI	Y	Y	Y	Y	Y	N	N	N
R	Y	Y	Y	Y	Y	N	N	N
D	Y	Y	Y	Y	Y	N	N	N
C	Y	Y	Y	Y	Y	N	N	N
L	N	N	N	N	N	Y	N	N
S	N	N	N	N	N	N	Y	Y
F	N	N	N	N	N	N	Y	Y

Legend

C - Complex
D - Double Precision
EI - Extended Integer
F - IO NAME
I - Integer
L - Logical
N - Nonvalid
R - Real
S - String
Y - Valid

Figure 3-1. Arithmetic Statement Combinations



IV...

Control Statements

Control statements enable the programmer to control, terminate, and alter the sequential order in executing statements. To execute a statement which is not in sequence, the programmer assigns a statement label to it for referencing by other statements. Any statement may have an assigned label; however, the numerical value of a statement label has no bearing on the order of execution, and it is not necessary that each statement be labeled.

UNCONDITIONAL GO TO STATEMENT

General Form
GO TO n where n is a statement labeled

This statement causes control to be transferred to n. In the following example, control is transferred to the statement labeled 31.

```
GO TO 31
```

COMPUTED GO TO STATEMENT

General Form
GO TO $(n_1, n_2, \ldots, n_m)$, i where $n_1, n_2, \ldots, n_m$ are statement labels i is an arithmetic expression

This statement causes control to be transferred to the statement labeled $n_1, n_2, \ldots, n_m$ depending on whether the integer value of i is 1, 2, 3, . . . , m, respectively, at the time of execution. The value of i cannot be negative or zero. During execution, if i is greater than m or i is less than 1, an error comment is output and the execution terminates. Thus, in the following example, if K is 3 at the time of execution, a transfer to the third statement in the list (statement 39) occurs.

```
GO TO (17, 21, 39,5), K
```

ASSIGN AND ASSIGNED GO TO STATEMENT

General Form
GO TO i, $(n_1, n_2, \ldots, n_m)$ where i is a variable appearing in a previously executed ASSIGN statement $n_1, n_2, \ldots, n_m$ are statement labels Note: $(n_1, n_2, \ldots n_m)$ is optional

General Form
ASSIGN n TO i where n is a statement label i is a variable that appears in an assigned GO TO statement

ASSIGN sets i to the value of the machine location corresponding to the FORTRAN statement number n, which represents any statement label in the program unit.

Later in the execution of the program, a GO TO i, $(n_1, n_2, \ldots n_m)$ transfers control to the statement label n referenced in the ASSIGN statement.

In the example

```
ASSIGN 24 TO M
    .
    .
    .
GO TO M, (1, 22, 41, 24, 36)
```

the M in GO TO M assumes the machine address of statement 24, transferring control to the fourth statement label in the list, 24.

ARITHMETIC IF STATEMENT

General Form
IF (a) n_1, n_2, n_3 where a is an arithmetic expression (not complex) n_1, n_2, n_3 are statement labels

This statement causes control to be transferred to the statement labeled n_1, n_2, or n_3, if the value of a is less than, equal to, or greater than zero, respectively. Thus in the example:

```
IF (A(J, K)-B)  10, 4, 30

IF (A(J, K)-B) < 0 control goes to statement 10
IF (A(J, K)-B) = 0 control goes to statement  4
IF (A(J, K)-B) > 0 control goes to statement 30
```

LOGICAL IF STATEMENT

General Form
IF(t)s where t is a logical expression s is any executable statement except DO or another logical IF

If the logical expression t is . TRUE. , statement s is executed. Control is then transferred to the next sequential statement unless s causes a transfer in which case control is transferred.

If t is . FALSE. , control is transferred to the next sequential statement.

If t is . TRUE. , and s is a CALL statement that does not take a non-standard return, control is transferred to the next sequential statement upon return from the subprogram.

The following examples illustrate several uses of the logical IF.

1. IF (A. AND. B) F = SIN (R)
2. IF (16. GT. L) GO TO 24
3. IF (D. OR. X. LE. Y) GO TO (18, 20), I
4. IF (Q) CALL SUB

In example 1, if (A. AND. B) is true, compute F and return to the statement following IF.

In example 2, if (16. GT. L), control transfers to statement 24.

In example 3, if (D. OR. X. LE. Y) is true, control transfers to statement 18 or 20 depending upon whether I is 1 or 2.

In example 4, if (Q) is true, control goes to the subprogram SUB.

DO STATEMENT

General Form
DO n i = m_1, m_2, m_3 where n is a statement label i is a nonsubscripted integer variable m_1, m_2, and m_3 must be arithmetic expressions greater than zero; if m_3 is not stated, it is taken to be 1. m_1, m_2 and m_3 are converted to integer type if required before the loop is open.

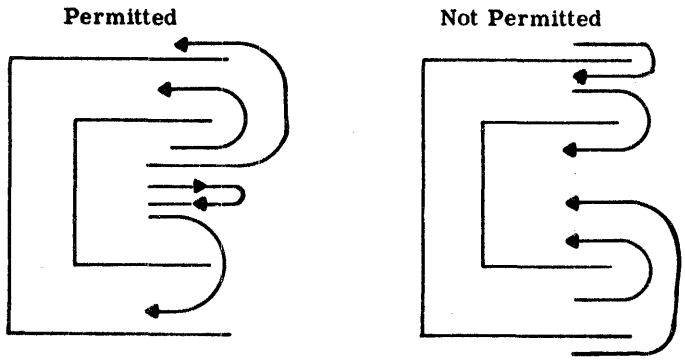
This statement causes repeated execution of the statements that follow, up to and including the statement labeled n. The statements in the range of the DO are executed repeatedly with i equal to m_1, then i equal to $m_1 + m_3$, then i equal to $m_1 + 2m_3$, etc., until i is equal to the highest value in this sequence that does not exceed m_2. The statements in the range of the DO will be executed at least once.

1. The range of a DO is that set of statements that will be executed repeatedly; i.e., it is the sequence of consecutive statements immediately following the DO statement, up to and including the statement labeled n. After the last execution of the range, the DO is said to be satisfied.

2. The index of a DO is the integer variable i. Throughout the range of the DO, the value of the index is available for computation, either as an ordinary integer variable or as a subscript. Upon exiting from a DO by satisfying the DO, index i must be redefined before it is used in computation. Upon exiting from a DO by transferring out of the range of the DO, the index i is available for computation and is equal to the last value it attained.

3. Within the range of a DO statement may be other DO statements; such a configuration is called a DO nest. If the range of a DO includes another DO, all of the statements in the range of the latter must also be in the range of the former.

4. Transfer of control and DO ranges. Control may not be transferred into the range of a DO from outside its range. Thus, the following configurations show permitted and non-permitted transfers.



5. Restrictions on statements in the DO range:

   a. Any statement that redefines the index or any of the indexing parameters (m's) is not permitted in the range of a DO.

   b. The range of a DO cannot end with an arithmetic IF-or GO TO-type statement, with a nonexecutable statement, with a nonexecutable statement, with a RETURN or STOP statement or with another DO statement. The range of a DO may end with a logical IF; in this case, if the logical expression t has the value .FALSE., the DO is reiterated; if the logical expression t has the value .TRUE., statement s is executed and then the DO is reiterated. However, if t has the value .TRUE. and s is an arithmetic IF or transfer type statement, control is transferred as indicated.

6. When a reference to a subprogram is executed in the range of a DO, the called subprogram must not alter the DO index or the indexing parameters.

7. When two or more nested DO statements end in the same CONTINUE statement, a transfer to this DO ending is only allowed from within the innermost DO.

An example of the DO statement follows:

```
      K = 0
      DO 10 I = 1,3
      DO 10 J - 1,2
      K = K + I + J
   10 CONTINUE
```

where the K values are computed as:

	old K	I	J	new K
K =	0			
K =	0 +	1 +	1 =	2
K =	2 +	1 +	2 =	5
K =	5 +	2 +	1 =	8
K =	8 +	2 +	2 =	12
K =	12 +	3 +	1 =	16
K =	16 +	3 +	2 =	21

CONTINUE STATEMENT

General Form
CONTINUE

CONTINUE is a dummy statement that does not generate any instructions in the object program. It is most frequently used as the last statement in the range of a DO. When it is necessary to bypass one or more executable statements at the end of a DO loop, and still continue looping, the nonexecutable CONTINUE statement provides this facility (see DO example).

PAUSE STATEMENT

General Form
PAUSE or PAUSE s where s is a string constant

The message PAUSE "HIT RETURN TO CONTINUE" is output to the terminal, and program execution is suspended until the user types a carriage return.

Examples:

```
PAUSE
PAUSE "AACD"
PAUSE 4HABCD
```

END STATEMENT

General Form
END

The END statement terminates compilation of a program or subprogram and physically it must be the last statement of the program or subprogram.

STOP STATEMENT

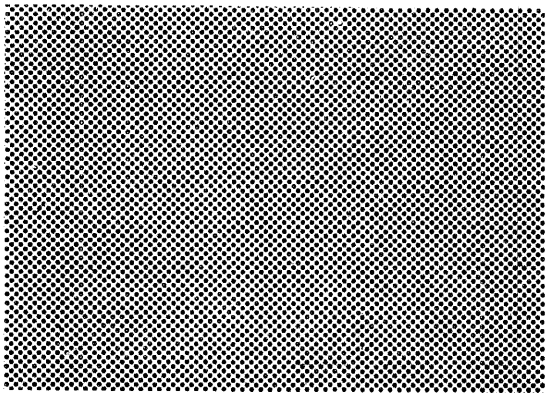
General Form
STOP STOP s where s is a string constant

The STOP statement terminates the execution of any program by returning control to the operating system. The string constant is printed.

CHAIN Statement

General Form
CHAIN IO NAME

The file "IO NAME" will be executed next. If the file is symbolic it will be compiled, loaded, and executed. If the file is relocatable binary, it will be loaded and executed. If the file is absolute binary, it will be executed. Common and open files are saved.



V...

Input/ Ouput Statements

In FORTRAN, input/output statements specify transmission of information to or from input/output files. Rather than be concerned with specific types of input/output devices (for example, card reader, magnetic tape or terminal), the FORTRAN programmer need only concern himself with the manipulation of data files. The statements are grouped as follows:

1. General Input/Output Statements

   The statements READ and WRITE cause the transmission of a specified list of quantities between core storage and an input/output file. The statements READ and PRINT cause transmission of information from or to core storage to the user terminal.

2. FORMAT and NAMELIST Statements

   Either of these two nonexecutable statements (the FORMAT statement or the NAMELIST statement) may be used with the general input/output statements.

   The FORMAT statement, which can be used with any general input/output statement, specifies the arrangement of data in the input/output record. If the FORMAT statement is referred to by a READ statement, the input data must meet the specifications described in "Data Input Referring to a FORMAT Statement".

   The NAMELIST statement specifies an input/output list of variables and arrays. Input/output of the values associated with the list is effected by reference to the list in a READ or WRITE statement. If the NAMELIST statement

is referred to by a READ statement, the input data must meet the specifications described in "Data Input Referring to a NAMELIST Statement".

FILE DESIGNATION

Files may be designated by a file name constant or file name variable. Within the FORTRAN system, the file name (" ") blank is used to designate the user terminal.

THE GENERAL INPUT/OUTPUT STATEMENTS

Input

The READ statement inputs the data to be processed by the computer program. The following table gives the forms of the READ statement, where

f is a file designator which references the input file

n is a FORMAT statement **label**

x is a NAMELIST name

y is a variable format

l is a statement label

Type of Input	General Form
ASCII record	READ, list
ASCII record	READ n, list
ASCII record	READ (f, n) list
ASCII record	READ (f, x)
ASCII record	READ (f, y) list
Binary record	READ (f) list
ASCII record	READ (f, n, END=L)

1. The READ, LIST statement causes ASCII record file to be input from the new user terminal.

2. The READ n, list statement causes records to be read in ASCII mode according to format n.

3. The READ (f, n) list statement causes ASCII information to be read from file f according to format n.

4. The READ (f, x) statement causes ASCII information relating to variables and arrays associated with the NAMELIST name x to be read from file f.

5. The READ (f, y) list statement causes ASCII information to be read from file f via a variable format.

6. The READ (f) list statement causes binary information to be read from file f.

7. If end of file occurs, control is transferred to statement label 1. The END = statement label may appear at the end of any of the parenthesized parameters.

Under the first four forms of the READ statement, successive records are read until the entire input list has been read, converted, and stored in the locations specified by the list.

Binary conversion of input numbers is identical, whether the numbers are compiled into the program, appear in a DATA statement, or are read in at execute time.

Examples:

```
READ 10, (A(I), I=1,5)
READ ("ABC",10)A,B,(D(J),J=1,10)
READ ("DEF",10)K,DC(J)
READ ("ABCDE")(A(J),J-1,10)
READ (ABC)(A(J),J-1,10)
READ (XYZ,NAM1)
READ (XYZ,FMT)A,B,(C(I),I,5)
```

Output

The following table gives the forms of the output statement, where

f (file name designating an unsigned integer constant, or a nonsubscripted integer variable) is a reference to a file

n is a FORMAT statement label

x is a NAMELIST name

y is a variable format

Type of Output	General Form
ASCII record	PRINT, list
ASCII record	PRINT n, list
ASCII record	WRITE (f, n) list
ASCII record	WRITE (f, x)
Binary record	WRITE (f) list
ASCII record	WRITE (f, y) list

1. The PRINT, list statement causes an ASCII record to be output to the user terminal. A list is required.

2. The PRINT n, list statement causes ASCII data to be output to the user terminal according to format n.

3. The WRITE (f, n) list statement causes ASCII information to be written on file f according to the format specified in statement n.

   If f = " " blanks output is to the user terminal

4. The WRITE (f, x) statement causes all variable and array names, as well as their values that belong to the namelist name x, to be written on file f.

5. The WRITE (f) list statement causes binary information to be written on file f.

The PRINT n, WRITE n, and WRITE (f, n) statements cause successive records to be written in accordance with the FORMAT statement until the list has been satisfied. The WRITE (f) list statement causes the writing of one logical record consisting of all the words specified in the list.

When a WRITE statement refers to a NAMELIST name, the values and names of all variables and arrays belonging to the NAMELIST name are written, each according to its type. A complete array is written out by columns. The output data is written such that the fields for the data are large enough to contain all the significant digits.

Examples:

```
PRINT 20,(A(J),J=1,6)
PRINT 2,(A(J),J=1,6)
WRITE ("ABC",10)A,B,(C(J),J=1,10)
WRITE ("DEF",11)K,D(J)
WRITE ("ABCDE")(A(J),J=1,10)
WRITE (ABC)A,B,C
WRITE (XYZ,NAMI)
WRITE (XYZ,FMT)A,B,(C(I),I=1,5)
```

LIST SPECIFICATIONS

If arrays or variables are transmitted by using a read or write binary, or with a FORMAT statement, an ordered list of the quantities to be transmitted must be included in the general input/output statement. The order of the input/output list must be the same as the order in which the information exists in the input/output medium.

The following notes on the information and meaning of an input/output list are most clearly understood by considering the following input/output list:

```
A, B(3), (C(I), D(I,K), I=1, 10),
( (E(I,J), I=1, 10,2), F(J,3), J=1,K)
```

This list implies that the information in the external input/output medium is arranged as follows:

```
A, B(3), C(1), D(1,K), C(2), D(2,K), . . . ,
C(10), D(10,K), E(1,1), E(3,1), . . . ,
E(9,1), F(1,3), E(1,2), E(3,2), . . . ,
E(9,2), F(2,3), . . . , F(K,3)
```

An input/output list is a string of list items separated by commas. A list item may be:

A subscripted or nonsubscripted variable.

An implied DO.

An input/output list reads from left to right with repetition of variables enclosed in parentheses.

A constant may appear in an input list only as a subscript or as an indexing parameter.

Expressions may appear in an output list.

The execution of an input/output list is exactly that of a DO loop, as though each left parenthesis (except subscripting parentheses) were a DO, with indexing given immediately before the matching right parenthesis, and with the DO range extending up to that indexing information. The order of the input/output list **A, B(3), (C(I), D(I, K), I=1, 10), ((E(I, J), I=1, 10, 2), F(J, 3), J=1, K)** may be considered equivalent to the following:

```
      A
      B(3)
      DO 5 I = 1, 10          (C(I), D(I, K), I=1, 10)
      C(I)
5     D(I, K)
      DO 9 J=1, K
      DO 8 I=1, 10, 2         ( (E(I, J), I=1, 10, 2),
8     E(I, J)                 F(J, 3), J=1, K)
9     F(J, 3)
```

An implied DO is best defined by an example. In the input/output list previously shown, the list item (C(I), D(I, K), I=1, 10) is an implied DO; it is evaluated as shown. The range of an implied DO must be clearly defined by parentheses.

On input, if the list has the form

```
K, A(K)
or
K, (A(I), I=1, K)
```

where the definition of an index or an indexing parameter appears earlier in the list than its use, the indexing is carried out with the newly read-in value.

Any number of quantities may appear in a single list. However, each quantity must have the correct format as specified in a corresponding FORMAT statement. Essentially, it is the list that controls the quantity of data read. If there are more quantities to be transmitted than are in the list, only the number of quantities specified in the list are transmitted, and remaining quantities are ignored. Conversely, if a list contains more quantities than are given on one ASCII input record,

more records are read; if a list contains more quantities than are given in one binary record, zero data is placed in the remaining list items.

When an end-of-file occurs and the user requests control (via END=L), the variables in the READ list are unchanged.

INPUT/OUTPUT OF ENTIRE ARRAYS

If input/output of an entire array is desired, an abbreviated notation may be used in the list of the general input/output statement. Only the name of the array need be given and the indexing information may be omitted. The array name used in this manner is called a short-list variable.

1. If A has previously been listed in a statement containing dimension information, the following statement is sufficient to read in all of the elements of the array A (see the "Input" section).

   ```
   READ (5,10)A
   ```

2. The elements read in by this notation are stored in accordance with the description of the arrangement of arrays in storage (see the "Subscripts" section).

3. If A has not been dimensioned, only one element will be read in. (Either double-precision or complex is considered to be one element.)

FORMAT STATEMENT

The ASCII input/output statements require, in addition to a list of quantities to be transmitted, reference to a FORMAT statement that describes the type of conversion to be performed between the internal machine language and the external notation for each quantity in the list.

General Form
n FORMAT ($S_1, S_2, \ldots, S_n/S'_1, S'_2, \ldots, S'_n/\ldots$)
where n is the statement label each subfield, S_1, is a format specification

FORMAT statements are not executed.

The FORMAT statement indicates, among other things, the maximum size of each record to be transmitted. Therefore, it must be remembered that the FORMAT statement is used with the list of some particular input/output statement, except when a FORMAT statement consists entirely of alphanumeric fields. In all other cases, control in the object program switches back and forth between the list, which specifies whether data remains to be transmitted, and the FORMAT statement, which gives the specifications for transmission of that data.

Slashes are used to specify unit records.

Thus, FORMAT (3F9.2,2F10.4/8E14.5) would specify records in which the first, third, fifth, etc., have the format (3F9.2,2F10.4) and the second, fourth, sixth, etc., have the format (8E14.5).

During input/output of data, the object program scans the FORMAT statement to which the relevant input/output statement refers. When a specification for a numerical field is found in the format and list items in the statement remain to be transmitted, input/output takes place according to the specification, and scanning of the FORMAT statement resumes. If no items remain, transmission ceases and execution of that particular input/output statement is terminated. Thus, an ASCII input/output operation is brought to an end when there are no items remaining in the list.

The field descriptors used in the FORMAT statement are listed in Table 5-1.

Numeric Field Descriptors

Numeric field descriptors are specified in the forms Dw.d, Ew.d, Fw.d, Iw, Ow, where:

1. D, E, F, G, I, and O represent the type of conversion.

Table 5-1. FORMAT STATEMENT FIELD DESCRIPTORS

Field Type	Manner Specified	Usage
I	rIw	Integer Field (123)
F	rFw.d	External fixed point decimal (1.23)
E	rEw.d	Floating Point '1.E09)
D	rDw.d	Double precision (1.D09)
G	rGw.d	Generalized (for E formats)
L	rLw	Logical (T or F)
A	rAw	Alphanumeric (JONES)
H	wHs	Hollerith (3HEND)
"	"S"	("END")
$	r$w	$ΔΔXX,XXX.XX where w = 12
X	wX	Spacing - spaces w times
T	Tw	Tab (spaces to column w)
P	fp	Scaling
/	/	Generates a Carriage Return
O	Ow	Octal integers

Symbols

w - field width (entire number of characters required)

d - number of decimal digits

i - number of integer digits

s - string of characters

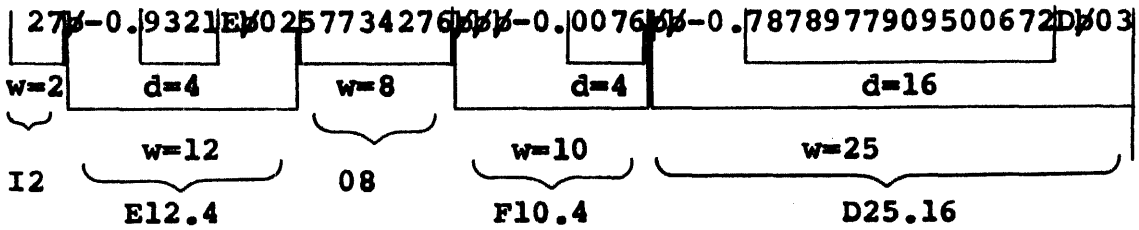
f - power of 10

r - repeat count

2. The w is an unsigned integer constant representing the field width for converted data; this field width may be greater than required to provide spacing between numbers.

3. The d is an unsigned integer or zero representing the number of digits of the field that appear to the right of the decimal point.

   For example, the statement FORMAT (I2, E12.4, 08, F10.4, D25.16) might cause the following line to be printed:



   where b indicates a blank space.

The following are notes on D-, E-, F-, G-, I-, and 0- conversion.

1. No format specification should be given that provides for more characters than permitted for a relevant input/output record. Thus a format for an ASCII record to be printed should not provide for more characters (including blanks) than the capabilities of the printer.

2. Information transmitted with 0-conversion may have real or integer names; information transmitted with G-conversion may have real, integer, or complex names; information transmitted with E-, and F-conversions must have real or complex names; information transmitted with I-conversion must have integer names; information transmitted with D-conversion must have double-precision names.

3. The numeric field descriptor Gw.d indicates that the external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real datum.

   Input processing is the same as for the F-conversion.

The method of representation in the external output string is a function of the magnitude of the real datum being converted. Let N be the magnitude of the internal datum. The following tabulation exhibits a correspondence between N and the equivalent method of conversion that will be effected:

Magnitude of Datum	Equivalent Conversion Effected
$0.1 \leq N < 1$	F(w-4) . d, 4X
$1 \leq N < 10$	F(w-4) . (d-1), 4X
.	.
.	.
.	.
$10^{d-2} \leq N < 10^{d-1}$	F(w-4) . 1, 4X
$10^{d-1} \leq N < 10^{d}$	F(w-4) . 0, 4X
Otherwise	sEw. d

Note that the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of F-conversion.

4. The field width w, for D-, E-, F- and G-conversions, must include a space for the decimal point and a space for the sign. The D-, E-, and G-conversions also require space for the exponent. For example, for D- and E- and G-conversions on output, $w \geq d+6$, and for F-conversion, $w \geq d+2$.

5. The exponent, which may be used with D- and E-conversions, is the power of 10 to which the number must be raised to obtain its true value. The exponent is written with an E (for E-conversion) or D (for D-conversion) followed by a minus sign if the exponent is negative, or a plus sign or a blank if the exponent is positive, and then followed by two numbers that are the exponent. For example, the number .002 is equivalent to the number .2E-02.

6. If a number converted by I-conversion requires more spaces than are allowed by the field width w, the most significant part of the number is truncated to fit the field. If the number requires fewer than w spaces, the leftmost spaces are filled with blanks. If the number is negative, the space preceding the leftmost digit contains a minus sign if sufficient spaces have been reserved.

8. If an output number that is converted by D-, E-, F-, G-, of I-conversions requires more spaces than are allowed by the field width w, the most significant part of the number is truncated to fit the field. If the number requires fewer than w spaces, the leftmost spaces are filled with blanks.

9. Specifications for successive fields are separated by commas and/or slashes. (See the section "Multiple-Record Formats" in this chapter.)

Complex Number Fields

Since a complex quantity consists of two separate and independent real numbers, a complex number is transmitted either by two successive real number specifications or by one real number specification that is repeated; e.g., 2E10.2=E10.2,E10.2.

The following is an example of a FORMAT statement that transmits an array consisting of six complex numbers.

```
FORMAT (2E10.2, E8.3, E9.4, E10.2, F8.4,3(E10.2,F8.2) )
```

Alphanumeric Field Descriptors

FORTRAN provides two ways for transmitting alphanumeric information; both specifications result in storing the alphanumeric information internally in ASCII.

1. The specification Aw causes w characters to be read into, or written from, a variable or array name.

2. The specification nH or a coded string constant allows placing alphanumeric information into a FORMAT statement.

The basic difference between A- and H-descriptor is that information handled by A-descriptor is given a variable name or array name that can be referred to for processing and modification; information handled by string descriptors is not given a name and may not be referred to or manipulated in storage in any way.

A-Descriptor. The variable name to be converted by the A-descriptor may be any type of variable.

1. On input, nAw is interpreted to mean that the next n successive fields of w characters each are to be stored as ASCII information. For a real variable if w is greater than 6, only the 6 rightmost characters will be significant. If w is less than 6, the characters will be left-adjusted and the work filled out with blanks.

2. On output, nAw is interpreted to mean that the next n successive fields of w characters each are to be the result of transmission from storage without conversion.

Logical Field Descriptor

Logical variables may be read or written using the specification Lw, where L represents the logical type of converstion and w is an integer constant that represents the data field width.

1. On input, a value representing either true or false is stored if the first nonblank character in the field of w characters is a T or an F, respectively. If all the w characters are blank, a value representing false is stored.

2. On output, a value of . TRUE. or . FALSE. in storage causes w minus 1 blanks, followed by a T or an F, respectively, to be written out. Output is right justified.

Blank Field Descriptor

The specification nX introduces n blank characters into an input/output record where $0 < n \leq 132$.

1. On input, nX causes n characters in the input record to be skipped, regardless of what they are.

2. On output, nX causes n blanks to be introduced into the output record.

Repetition of Field Format

It may be desired to print or read n successive fields in the same format within one record. This may be specified by using n, an unsigned integer, before D-, E-, F-, G-, I-, L-, O-, or A-descriptor. Thus, the field specification 3E12.4 is the same as writing E12.4, E12.4, E12.4.

Repetition of Groups

A limited parenthetical expression is permitted to enable repetition of data fields according to certain format specifications within a longer FORMAT statement. Thus, FORMAT (2(F10.6, E10.2), I4) is equivalent to FORMAT (F10.6, E10.2, F10.6, E10.2, I4). (See the "Multiple-Record Formats" section.) Two levels of parentheses, in addition to the parentheses required by the FORMAT statement, are permitted. The second level of parentheses facilitates the transmission of complex quantities.

Scale Factors

To permit more general use of D-, E-, F-, and G-descriptors, a scale factor followed by the letter P may precede the specification. The magnitude of the scale factor must be between -8 and +8, inclusive. The scale factor is defined for input as follows:

$$10^{-\text{scale factor}} \times \text{external quantity} = \text{internal quantity}.$$

For output, the scale factor is defined as follows:

$$\text{external quantity} = \text{internal quantity} \times 10^{\text{scale factor}}$$

For input, scale factors have effect only on F-conversion. For example, if input data is in the form xx.xxxx and it is desired to use it internally in the form .xxxxxx, then the FORMAT specification to effect this change is 2PF7.4. For output, scale factors may be used with D-, E-, F-, and G-conversion.

For example, the statement FORMAT (I2, 3F11.3) might output the following printed line:

27␣␣␣-93.209␣␣␣␣-0.008␣␣␣␣␣0.554

But the statement FORMAT (I2. 1P3F11. 3) used with the same data would output the following line:

27␣␣␣␣-932. 094␣␣␣␣␣␣-0. 76␣␣␣␣␣␣␣5. 536

whereas, the statement FORMAT (I2, -1P3F11. 3) would output the following line:

27␣␣␣␣␣␣-9. 321␣␣␣␣␣␣-0. 001␣␣␣␣␣␣␣0. 055

A positive scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it holds for all D-, E-, F-, and G-conversions following the scale factor within the same FORMAT statement. This applies to both single-record formats and multiple-record formats. Once the scale factor has been given, a subsequent scale factor of zero in the same FORMAT statement must be specified by OP. For F-type conversion, output of numbers, whose absolute value is greater than or equal to 2^{35} after scaling, is output in E-conversion. Scale factors have no effect on I- and O-conversion.

Multiple-Record Formats

To deal with a block of more than one line of print, a FORMAT specification may have several different one-line formats separated by a slant to indicate the beginning of a new blank line. Thus, FORMAT (3F9. 2, 2F10. 4/8E14. 5) would specify a multiline block of print in which lines 1, 3, 5, ... have format (3F9. 2, 2F10. 4), and lines 2, 4, 6, ... have format (8E14. 5).

If a multiple-line format is desired in which the first two lines are to be printed according to a special format, and all remaining lines according to another format, the last line-specification should be enclosed in a second pair of parentheses; for example:

FORMAT (I2, 3E12. 4/2F10. 3, 3F9. 4/(10F12. 4))

If data items remain to be output after the format specification has been completely "used", the format repeats from the last previous parenthesis, which is a zero or a first level parenthesis. For example, consider the FORMAT statement:

```
FORMAT  (3E10. 3,  (I2, 2  (F12. 4, F10. 3) ),  D28. 17)
          0          1     2                21          0
```

The parentheses labeled 0 are zero level parentheses; those labeled 1 are first level parentheses; and those labeled 2 are second level parentheses. If more items in the list are to be transmitted after the format statement has been completely used, the FORMAT repeats from the last first-level left parenthesis; that is, the parenthesis preceding I2.

As these examples show, both the slash and the final right parenthesis of the FORMAT statement indicate a termination of a record.

Blank lines may be introduced into a multiline FORMAT statement by inserting consecutive slashes. When n+1 consecutive slashes appear at the end of the FORMAT, they are treated as follows: for input, n+1 records are skipped; for output, n blank lines are written. When n+1 consecutive slashes appear in the middle of the FORMAT, n records are skipped for both input and output.

Carriage Control

The WRITE (f, n) list and PRINT n, list statements prepare ASCII files in edited format for the printer. The first character of each ASCII record is examined to see if it is a control character to regulate the spacing. If the first character is recognized as a control character, it is replaced by a blank in the printed line and the line printed after the proper spacing has been effected. The control characters which will be recognized are:

Character	Effect
Blank	Single space before printing
0	Double space before printing
1	Eject before printing

FORMAT Statement Read in at Object Time

FORTRAN permits specifying a FORMAT for an input/output list at object time.

In the following example, A, B, and the array C are converted and stored according to the FORMAT specifications read into the array FMT at object time.

```
DIMENSION FMT (12)
FORMAT (12A6)
READ ("ABC", 1)(FMT(I(, I=1, 12)
READ ("ABC", FMT)A, B, (C(I), I=1, 5)
```

The format read in at object time must take the same form as a source program FORMAT statement, except that the word FORMAT is omitted; that is, the variable format begins with a left parenthesis and terminates with a right parenthesis.

Data Input Referring to a FORMAT Statement

These specifications must be followed when data is input to the object program.

1. The data must correspond in order, type, and field with the field specifications in the FORMAT statement. Punching begins in card column 1.

2. Plus signs may be omitted or indicated by a +. Minus signs must be indicated.

3. Blanks in numeric fields are regarded as zeros; however, leading zeros are suppressed.

4. Numbers for E- and F-conversion may contain any number of digits, but only the high-order 9 digits of precision are retained. For D-conversion, the high-order 14 digits of precision are retained. In both cases, the number is rounded to 9 or 14 digits of accuracy, as applicable.

To permit economy in typing, certain relaxations in input data format are permitted.

1. Numbers for D-, E-, and F-conversion need not have their decimal point punched; the format specification suffices. For example, the number -09321+2 with the specification E12.4 is treated as though the decimal point had been punched between the 0 and the 9. If the decimal point is punched in the card, its position overrides the position indicated in the FORMAT specification.

2. Field width in an input record can be overruled by using a comma as a delimiter.

NAMELIST STATEMENT

The NAMELIST statement and modified forms of the READ and WRITE statements provide for reading, writing, and converting data without using an input/output list in the input/output statement. Reference is made to a NAMELIST statement instead of a FORMAT statement.

General Form
NAMELIST/X/A, B, . . . , C/Y/D, E, . . . F/Z/G, H, . . . , I where X, Y, Z, . . . are NAMELIST names A, B, C, D, . . . are variable or array names

Each list that is mentioned in the NAMELIST statement is given a NAMELIST name. Thereafter, only the NAMELIST name is needed in an input/output statement to refer to that list. The following rules apply to assigning and using a NAMELIST name:

1. A NAMELIST name consists of one to eight alphanumeric characters; the first character must be alphabetic.

2. A NAMELIST name is enclosed in slashes. The field of entries belonging to a NAMELIST name ends either with a new NAMELIST name enclosed in slashes or with the end of the NAMELIST statement.

3. A variable name of any array name may belong to one or more NAMELIST names.

4. A NAMELIST name must not be the same as any other name in the program.

5. A NAMELIST name may be defined only once by its appearance in a NAMELIST statement. After it has been defined in the NAMELIST statement, the NAMELIST name may appear only in READ, or WRITE, or PRINT statements.

6. A dummy argument of a subprogram cannot be used as a variable in a NAMELIST statement.

In the following examples, the arrays A, I, and L and the variables B and J belong to the NAMELIST name, NAM1; the array A and the variables C, J, and K belong to the NAMELIST name, NAM2.

```
DIMENSION A(10), I(5, 5), L(10)
NAMELIST /NAM1/A, B, I, J, L/NAM2/A, C, J, K
```

Data Input Referring to a NAMELIST Statement

When a READ statement refers to a NAMELIST name, the designated input file is readied and input of data is begun. The first input data record is searched for a $ as the **first** character, immediately followed by the NAMELIST name, immediately followed by a comma or one or more blank characters. If the search fails, additional records are examined consecutively until there is a successful match or end-of-file. When a successful match is made of the NAMELIST name on a data record and the NAMELIST name referred to in a READ statement, data items are converted and placed in storage.

Empty fields (detected as one of the pairs (=,), (b,), or (, ,)) cause a zero to be stored. The data items must be separated by commas. The end of a group of data is signaled by a $ following the last item either in the same data record as the NAMELIST name or anywhere in any succeeding records.

The form that data items may take is:

1. Variable name = constant

   ```
   CON = 17.5
   X(6) = 26.4
   ```

   where the variable name may be an array element name or a simple variable name. Subscripts must be integer constants.

2. Array name = set of constants (separated by commas)

   ```
   X = 1., 2., 3., 5*6.3, 2*3*4
   ```

where k* constant may be included to represent k constants (k must be an unsigned integer). The number of constants must be equal to the number of elements in the array.

3. Subscripted variable = set of constants (separated by commas)

   Y(4) = 9., 6., 19*1.8, 3*7*2*4*0.0

   where k* constant may be included to represent k constants (k must be an unsigned integer). A data item of this form results in the set of constants being placed in array elements, starting with the element designated by the subscripted variable.

   The number of constants given cannot exceed the number of elements in the array that are included between the given element and the last element in the array, inclusive.

4. Variable 1/Variable 2 = constant

   where Variable 1 is a counter which is set after the data has been input, indicating the number of constants that have been stored for Variable 2.

Constants used in the data items may take any of the following forms:

a. Integers, e.g., 1,2,3

b. Real numbers, e.g., 1.,2.,3.3

c. Double-precision numbers, e.g., -2.63D15

d. Complex numbers, which must be written in the usual form, (C1, C2), where C1 and C2 are real numbers.

e. Logical constants, which must be written as T or .TRUE., and F or .FALSE.

f. String constants.

Any selected set of variable or array names belonging to the NAMELIST name, referred to by the READ statement, may be used as specified in the preceding description of data items. Names that are made equivalent to these names may not be used unless they also belong to the NAMELIST name.

```
                        |1    5 6 7
                        |
First Data Card         |  $NAM1   I(2,3)=5,J=4.2,B=4,
                        |
Second Data Card        |  A(3)=7,6.4,L=2,3,8*4.3$
```

If this data is input to be used with the NAMELIST statement previously illustrated and with a READ statement, the following actions take place. The input file designated in the READ statement is prepared and the first record is read. The record is searched for a $ in column 2, immediately followed by the NAMELIST name, NAM1. Since the search is successful, data items are converted and placed in core storage.

The integer constant 5 is placed in I(2, 3), the real constant 4. 2 is converted to an integer and placed in J, and the integer constant 4 is converted to real and placed in B. Since no data items remain in the record, the next input record is read. The integer constant 7 is converted to real and placed in A(3), and the real constant 6. 4 is placed in the next consecutive location of the array, A(4). Since L is an array name not followed by a subscript, the entire array is filled with the succeeding constants. Therefore, the integer constants 2 and 3 are placed in L(1) and L(2), respectively, and the real constant 4. 3 is converted to an integer and placed in L(3), L(4), . . . , L(10). The $ signals termination of the input for the READ operation.

Data Output Referring to a NAMELIST Statement

When data is output via NAMELIST, e.g., WRITE (6, LIST1), all variables associated with LIST1, as specified in the NAMELIST statement, will be output. The output values are labeled with the variable name.

AUXILIARY INPUT/OUTPUT STATEMENTS

The following set of statements enable the user to manipulate magnetic tapes and sequential disk or drum files:

REWIND Statement

General Form
REWIND $e_1, e_2, e_3, \ldots, e_n$ where the e_i are integer, real, or double precision expressions.

Execution of a REWIND statement causes the units whose logical unit numbers are the integer values of the e_i to be rewound, in the order written.

BACKSPACE Statement

General Form
BACKSPACE $e_1, e_2, e_3, \ldots, e_n$ where the e_i are integer, real, or double precision expressions.

When a BACKSPACE statement is executed, the units referenced by the integer values of the e_i are each backspaced one logical record.

REWIND and BACKSPACE statements that are executed for tapes already positioned at "load point" have no effect.

END FILE Statement

General Form
END FILE $e_1, e_2, e_3, \ldots, e_n$ where the e_i are integer, real, or double precision expressions whose values determine the units on which end-of-file marks are to be written.

This statement causes end-of-file marks to be written on the specified units. Sometimes it is desirable to take a program that has been written for output on magnetic tape and assign that logical unit number to some other device (such as a line printer). Since such programs often write an end-of-file and rewind their tapes at the end of the job, it is permissible to specify an ENDFILE or REWIND operation on any device. When the device is not a magnetic tape or sequential disk or drum file, the statements have no effect. It is not permissible to backspace such devices.

MEMORY-TO-MEMORY DATA CONVERSION

The ENCODE and DECODE statements resemble ASCII WRITE and READ statements; however, in an ENCODE/DECODE operation, there is no true input/output, but rather a data conversion between an input/output list and an internal buffer. The buffer is often an integer array and is established by the programmer. Because there are definite physical limits to an external record, the length of the simulated internal record in an ENCODE/DECODE operation can be established by the programmer. If several records are identified by the FORMAT being used, the second and subsequent records are stored in memory in order of increasing memory address.

General Form

{ENCODE / DECODE} (c, f, s, n,)k

where

c is the number of characters per internal record (an arithmetic expression converted to integer mode)

f specifies a FORMAT statement (a FORMAT statement lable or the name of some array that contains a FORMAT statement)

s is the starting location of the buffer area (an array name, an array element, or a scalar variable)

n is an integer variable (optional) into which the number of generated or scanned characters is stored, upon completion of the operation

k is an input/output list.

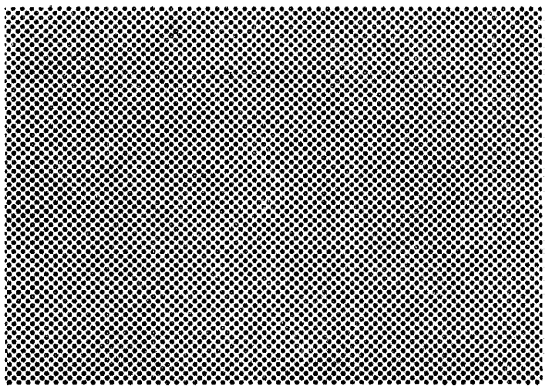
ENCODE Statement

ENCODE converts the list to ASCII character strings, in accordance with the FORMAT established by f, and places those strings in a buffer area that begins at the starting location specified by s. When the number of characters generated by the FORMAT statement exceeds the limits of the record, the overflow characters are lost and are not placed in the next record. When fewer characters are generated than are required to fill the record, it is completed with trailing blanks. As with a WRITE statement, the first operation accomplished on each record is to fill it with blanks, even before any characters have been generated.

DECODE Statement

DECODE initiates decoding of the character string starting at location s, in accordance with the FORMAT indicated by f, and stores the decoded information in the list k. If the FORMAT statement specifies more characters from a record than are indicated by c, it is assumed that the extra characters are to be blanks and, therefore, they are not taken from the following record. A new record is started only when it is requested specifically by the FORMAT statement.

The setting of n is optional; when it is desired, it is set to the number of characters scanned. This can be used to advantage when scanning with widthless formats.



VI...

Indirect File and Data Base Management Statements

In addition to the "standard" FORTRAN input/output statements, described in Section V, the 2+2 FORTRAN subsystems provides an interface to the more general 2+2 file and data-base manager subsystem. This interface consists of a simple set of declarative and operational statements useful in building, manipulating, and accessing data bases used in storing and retrieving information.

With respect to the data-base manager, a file may be considered to be a named space consisting of an ordered sequence of elements. All internal structuring defined for a file, and all access to the contents of a file, are controlled by the data-base manager.

The structure, access, and control of a file is determined by the "schema" definition associated with a file. A "schema" allows for the definition of file to include logical "areas" within the file. "Records" within "areas" of a file are also defined. A "record" is further defined into elements consisting of a name, type, mode, and size. "Schema", "areas", and "records", once defined are referred to by name. Rather than be concerned with the form or structure of the data-base manager, itself, the FORTRAN programmer need only concern himself with the manipulation of files and data bases in terms of the statements described below.

NOTE

For a complete description of the data-base manager, the FORTRAN programmer is advised to obtain the 2+2 system documentation of the data-base manager for a detailed description of "schema", "area", "record", etc.

The statements are grouped as follows:

1. Directory Content Statements

   The statements CREATE, RENAME, ERASE, and DESTROY transmit information to the file manager used to build, modify and update files. CLOSE file indicates deactivation of a previously active file.

2. General Input/Output Statements

   The RETRIEVE statement causes transmission of a set of quantities from a file to core storage.

   The statements, INSERT, APPEND, REPLACE, and REMOVE cause transmission of a set of quantities from core storage to a file in a manner appropriate to the specified statement.

FILE DESIGNATION

Files may be designated by a file name constant or file name variable. Within the FORTRAN system, the file name ("Δ") blank is used to designate the user terminal.

DIRECTORY CONTENT STATEMENTS

1. CREATE a file

The CREATE statement is used to enter a new file name in the user's directory. The file is initially set to contain no information. Optionally, access privileges and additional user access may be specified.

General Form
CREATE file name where: file name is a file designator which may be a file name constant or file name variable.

Examples:

```
CREATE "A"
CREATE "BETA"
CREATE "PAYABLES"
```

The creator of a file is assigned full access privileges to that file. Those privileges include READ, WRITE, and EXECUTE access.

To assign privileges to other users who wish to access a file, the CREATE statement may be expanded in the following form:

CREATE file name/USER ID/ACCESS/USER ID/ACCESS...

Where file name is file designator:

USER ID is a unique user identification string (not currently defined as to form or length) delimited by slashes (division signs).

ACCESS is any combination of the following words separated by commas, -READ, -WRITE, EXECUTE, APPEND, or PASSWORD.

Any number of users may be authorized access to a file. However, access privileges must be specified for each authorized user. Public access is specified by a null user ID(i.e., //). Consider,

CREATE "PAYROLL"/AB234/READ/J. SHMOE/APPEND, EXECUTE

Creates the file named MILLISIN. It also makes the file public with both READ and EXECUTE privileges.

2. RENAME a file

The RENAME statement allows a user to rename any file currently in a user's directory. Optionally, additional users and access privileges may be specified.

General Form
RENAME file name, file name where: file name is a file designator which may be a file name constant or file name variable.

Example:

```
RENAME "NEWTRANS","OLDTRANS"
```

This renames the file currently in the user's directory as NEWTRANS to a file named OLDTRANS. The name NEWTRANS is removed from the user's directory, replaced by the name OLDTRANS.

Additional access privileges may be specified while renaming a file by attaching user ID's and access. For example,

```
RENAME "ABC", "DEF"/USER27/READ,WRITE/USER33/EXECUTE
```

File ABC is renamed as file DEF. In addition, USER27 is authorized to READ and WRITE the file, and USER33 is granted execute only access.

3. ERASE a file

The ERASE statement allows a user to erase or clear the information currently contained in one or more files.

General Form
ERASE file name, filename, ... where: file name is a file designator which may be a file name constant or file name variable.

Examples:

```
ERASE "R"
ERASE "MONTH04","MONTH05","MONTH06"
```

4. DESTROY a file

The DESTROY statement provides the method for destroying both the information contained in a file and removing the entry containing the name of the file in the user's directory. That is, both the contents and the name of the file are destroyed and unrecoverable.

General Form
DESTROY file name, file name, ... where: file name is a file designator which may be a file name constant or file name variable.

Examples:

```
DESTROY "ACCOUNTS"
DESTROY "SMALLEST","AVERAGE","LARGEST"
```

5. CLOSE a file

The CLOSE statement is used to signal the system that processing of an active file has been completed. The specified file is deactivated. Subsequent statements referencing the file will reactivate the file and its contents. This statement need only be executed if a given FORTRAN program is concerned about simultaneously activating more files than the maximum number of allowable active files defined by the 2+2 system.

General Form
CLOSE file name, filename,....   where: file name is a file designator which may be a file constant or file name variable.

Examples:

```
CLOSE "XYZ"
CLOSE "FILE1","FILE3","FILE5"
```

GENERAL INPUT/OUTPUT STATEMENTS

Input

The RETRIEVE statement inputs quantities from a file to be processed by the computer program.

General Form
RETRIEVE file name (schema(area(record))) list where: file name is a file designator. schema is the schema name applied to file name. area is an area name within the schema being accessed. record is a record name within the area of a schema. list is a list specification as defined under LIST SPECIFICATION (Section V, pg. 5-5).

The RETRIEVE statement reads one record from file name using schema/area/record name to isolate the desired record. The contents of the variables named in the accessed record replace the contents of those same variables in the FORTRAN program and may be used for computation.

The list specification is optional and need only be used when it is desirable to access only a few of the items defined in a record definition.

There must be a one-to-one correspondence between the variable names in the FORTRAN program, and the variable names used in a schema/area/record name definition.

The parenthesized arguments - area name and record name are optional so long as no ambiguity exists within the schema.

Examples:

RETRIEVE "MASTFILE" (SCH1(AREA2(REC3)))

RETRIEVE "MASTFILE"(SCH1(AREA2(REC3)))A, B
In this example, assume the definition of REC3 defined the variables A, B, C, D&E. Only variable A, B are accessed via this statement.

RETRIEVE "MASTFIL"(SCH1)
This statement would be equivalent to example 1 if the schema, SCH1, defined only 1 area and 1 record.

Output

The statements, INSERT, APPEND, REPLACE, and REMOVE, output quantities contained in FORTRAN program variables to a file managed by the data-base manager.

General Form
APPEND INSERT NEXT INSERT PRIOR file name (schema(area(record))) list REMOVE REPLACE where: file name is a file designator schema is the schema name applied to file name. area is an area name within the schema being accessed accessed. record is a record name within the area in the schema. list is a list specification as defined under LIST SPECIFICATIONS (Section V).

The output statements write one record from core storage to file name according to the schema/area/record names specified. The contents of variables in core storage (FORTRAN program) defined by the schema are written appropriately into the specified file using the desired schema.

The APPEND statement adds a record at the end of the file according to the specified schema.

The INSERT NEXT statement inserts a record after the current record position of the file according to the specified schema.

The INSERT PRIOR statement inserts a record just before the current record position of the file according to the specified schema.

The REMOVE statement removes or deletes the record at the current record position of the file. In this case, the schema definition is used only to isolate and remove the desired record from the file.

The REPLACE statement replaces the record at the current record position of the file according to the specified schema.

The list specification is optional and need only be used when it is desirable to write out fewer items in a record than specified in the schema definition of a given record.

There must be a one-to-one correspondence between the variables names in the FORTRAN program, and the variable names used in a schema/area/record name definition.

The parenthesized arguments - area name and record name are optional so long as no ambiguity exists within the schema.

Examples:

```
APPEND "ORDERS"(ORDSCH(NEWAREA(NEWREC)))

APPEND "ORDERS"(ORDSCH)
```

This statement is equivalent to the first example if the "ORDERS" file consists of one area and one record type.

```
INSERT NEXT "DATEFILE"(DATSCH(HOLIDAY)))
```

This statement writes one record in the "DATEFILE" according to the schema DATSCH. Area HOLIDAY is written, and the statement assumes only one one record type, hence no record name appears.

INSERT PRIOR "SIGMAS" (STATSCH(SQUARES(SUM))) A,B

This statement inserts a record prior to the current record position of the file SIGMAS. The schema STASCH, with area SQUARES, containing record SUM is written. However, since a list is specified, only variables A & B of record SUM are written - even if the record definition defined more elements than A & B.

REMOVE "CATALOG" (CATSCH((NAMREC)))

The parenthesized quantities in the example show a record deleted from the file "CATALOG". This example assumes the schema CATSCH defines only one area. Within the one area, the current record NAMREC is deleted.

The following examples illustrate possible combinations of optional parameters.

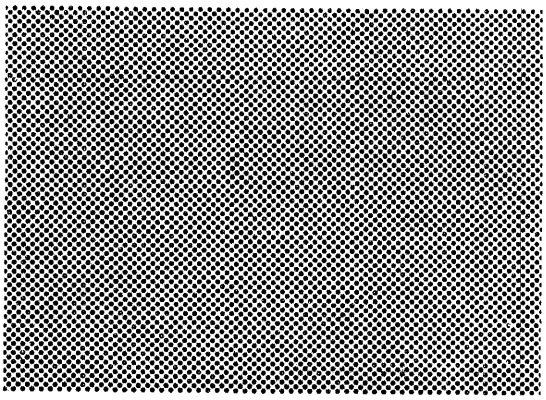
REPLACE "ENGFIL" (SCHEMA1(AREA2(REC3)))

REPLACE "ENGFIL" (SCHEMA1(AREA2(REC))) A,B,C

REPLACE "ENGFIL" (SCHEMA1(AREA2))

REPLACE "ENGFIL" (SCHEMA1((REC3)))

REPLACE "ENGFIL" (SCHEMA1)



VII...

Subroutines, Functions & Subprogram Statements

The three basic elements of scientific programming languages – arithmetic, control, and input/output – are given added flexibility through subroutines. Subroutines are program segments executed under the control of another program and are usually tailored to perform some often-repeated set of operations. A subroutine is written only once, but may be used again and again; it avoids a duplication of effort by eliminating the need for rewriting program segments for use in common operations. There are four classes of subroutines in FORTRAN: statement functions, built-in functions, FUNCTION subprograms, and SUBROUTINE subprograms. The major differences among the four classes of subroutines are as follows:

1. The first three classes may be grouped as functions; they differ from the SUBROUTINE subprogram in the following respects:

   a. Functions return a value that is utilized in evaluating an expression.

   b. A function is referred to by an arithmetic expression containing its name; a SUBROUTINE subprogram is referred to by a CALL statement.

2. The statement function and built-in function are open subroutine; that is, a subroutine that is incorporated into the object program each time it is referred to in the source program. The two other FORTRAN subroutines are closed; that is, they appear only once in the object program.

NAMING SUBROUTINES

All four classes of subroutines are named in the same manner as a FORTRAN variable

1. A subroutine name consists of one to eight alphanumeric characters, the first of which must be alphabetic.

2. The type of the function, which determines the type of the result, may be defined as follows:

   a. The type of statement function may be indicated by the name of the function or by placing the name in a Type statement.

   b. The type of a FUNCTION subprogram may be indicated by the name of the function (if it is real or integer) or by the name of the function (if it is real or integer) or by writing the type (REAL, INTEGER, COMPLEX, DOUBLE PRECISION, LOGICAL) preceding the word FUNCTION. In the latter case, the type implied by name is overridden. The type of the FUNCTION subprograms in the Subroutine Library (the mathematical subroutines) is defined. **Therefore, they need not be typed elsewhere.**

   c. The type of a built-in function is indicated within the FORTRAN compiler and need not appear in a Type statement

3. The name of a SUBROUTINE subprogram has no type and is not defined, since the type of results returned is dependent only on the type of the variable names in the dummy argument list.

DEFINING SUBROUTINES

Statement Functions

Statement functions are defined by a single arithmetic statement and apply only to the source of the program unit containing the definition.

General Form
a = b where a is a function name followed by parentheses enclosing its arguments, which must be distinct, nonsubscripted (dummy) variables, separated by commas b is an expression that does not involve subscripted variables. Any statement function appearing in b must have been previously defined.

1. As many as desired of the variables appearing in b may be stated in a as the arguments of the function. Since the arguments are dummy variables, their names, which indicate the type of the variable, may be the same as names appearing elsewhere in the program of the same type.

2. Variables appearing in the function-defining expression b that are not dummy variables stated in a are considered to be variables defined within the parent program.

3. A statement function definition must precede the first usage in the source program.

4. The type of any statement function name or argument that differs from its implicit type must be defined preceding its use in the statement function definition.

Examples:

```
FIRST(X)      = A*X+B
JOB(X, B)     = C*X+B
THIRD(D)      = FIRST(E)/D
MAX(A, I)     = A**I-B-C
LOGFCT(A, C) = A**2. GE. C/D
```

The arithmetic statement function FIRST(C), in the previous example, might be used as follows:

```
  A = 1
  B = 6.2
  C = 3.3
 AA = 2.0+5.2*FIRST(C)
```

Built-In Functions

Built-in functions are predefined as open subroutines that exist within the FORTRAN compiler. A list of available built-in functions is given in Figure 7-1.

The compiler checks the type or number of arguments for a built-in function. Using the wrong type of argument results in an automatic conversion produced by the compiler. Using the wrong number of arguments results in that function being processed as an external.

Examples:

```
  A = ABS(X)
 AA = FLOAT (II)
  C = AMAX1(C1, C2, C3, C4)
```

FUNCTION Subprogram

FUNCTION subprograms are defined by a special FORTRAN statement, FUNCTION.

General Form
FUNCTION name $(a_1, a_2, \ldots, a_n)$
REAL FUNCTION name $(a_1, a_2, \ldots, a_n)$
INTEGER FUNCTION name $(a_1, a_2, \ldots, a_n)$
EXTENDED INTEGER FUNCTION name
DOUBLE PRECISION FUNCTION name $(a_1, a_2, \ldots, a_n)$
COMPLEX FUNCTION name $(a_1, a_2, \ldots, a_n)$
LOGICAL FUNCTION name $(a_1, a_2, \ldots, a_n)$
STRING FUNCTION name
where
name is the symbolic name of a function
the arguments $a_1, a_2, \ldots, a_n$, of which there must be at least one, are dummy names
the type of the function may be explicitly stated preceding the word FUNCTION, or implicitly indicated by the first letter of the FUNCTION name

Function	Definition	Number of Arguments	Name	Type of Argument	Type of Function
Absolute value	|Arg|	1	ABS IABS	Real Integer	Real Integer
Truncation	Sign of Arg times largest integer ≤ |Arg|	1	AINT INT	Real Real	Real Integer
Remaindering (see note below)	Arg_1 (mod Arg_2)	2	AMOD DMOD MOD	Real Real Integer	Real Real Integer
Choosing largest value	Max(Arg_1, Arg_2,...)	≥2	AMAX0 AMAX1 MAX0 MAX1	Integer Real Integer Real	Real Real Integer Integer
Choosing smallest value	Min(Arg_1, Arg_2,...)	≥2	AMIN0 AMIN1 MIN0 MIN1	Integer Real Integer Real	Real Real Integer Integer
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer with truncation	1	IFIX	Real	Integer
Transfer of sign	Sign of Arg_2 times |Arg_1|	2	SIGN ISIGN	Real Integer	Real Integer
Positive difference	Arg_1 - Min (Arg_1, Arg_2)	2	DIM IDIM	Real Integer	Real Integer
Obtain most significant part of double-precision argument		1	SNGL	Double	Real
Obtain real part of complex argument		1	REAL	Complex	Real
Obtain imaginary part of complex argument		1	AIMAG	Complex	Real
Absolute value Truncation	|Arg| sign of Arg times largest integer ≤ |Arg|	1 1	DABS IDINT	Double Double	Double Integer
Choosing largest value	Max(Arg_1, Arg_2,...)	≥2	DMAX1	Double	Double
Choosing smallest value	Min(Arg_1, Arg_2,...)	≥2	DMIN1	Double	Double
Transfer of sign	Sign of Arg_2 times |Arg_1|	2	DSIGN	Double	Double
Express single-precision argument in double-precision form	D=(Arg, 0)	1	DBLE	Real	Double
Express two real arguments in complex form	$C=Arg_1+iArg_2$	2	CMPLX	Real	Complex
Obtain conjugate of a complex argument	For Arg=X+iY, C=X-iY	1	CONJG	Complex	Complex

NOTE: The function MOD(Arg_1,Arg_2) is defined as $Arg_1 - [Arg_1/Arg_2] Arg_2$, where $[Arg_1/Arg_2]$ is the truncated value of that quotient.

Arg_1 should not exceed $10^{11} * Arg_2$.

Figure 7-1. Built-In Functions

Examples:

```
FUNCTION ARCSIN (RADIAN)
REAL FUNCTION ROOT (A, B, C)
INTEGER FUNCTION CONST (ING, SG)
DOUBLE PRECISION FUNCTION DBLPRE (R, S, T)
COMPLEX FUNCTION CCOT (ABI)
LOGICAL FUNCTION IFTRU (D, E, F)
```

1. The FUNCTION statement must be the first statement of a FUNCTION subprogram. At least one dummy name must be enclosed in parentheses.

2. The name of the function must appear at least once as a variable on the left side of an assignment statement or in an input statement.

Example:

```
FUNCTION CALC (A, B)
.
.
.
CALC=Z+B
.
.
.
RETURN
```

By this method the output value of the function is returned to the calling program. Unlike arithmetic statement functions, those variables not appearing in the FUNCTION statement as dummy variables are considered to be defined within the FUNCTION subprogram.

The calling program is the program in which a subprogram is referred to or called.

The called program is the subprogram that is referred to or called by the calling program.

3. The arguments may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. The actual arguments must correspond in number, order, and type with the dummy arguments.

4. When a dummy argument is an array name, a statement with dimension information must appear in the FUNCTION subprogram; also, the corresponding actual argument must be a dimensioned array name.

5. None of the dummy arguments may appear in an EQUIVALENCE statement in the FUNCTION subprogram nor may they appear in a COMMON, DATA, or NAMELIST statement.

6. The FUNCTION subprogram must contain one path logically terminated by a RETURN statement and physically terminated by an END statement.

7. The FUNCTION subprogram may contain any FORTRAN statements except SUBROUTINE, BLOCK DATA, or another FUNCTION statement.

8. The actual arguments of a FUNCTION subprogram may be any of the following:

   a. A constant.

   b. A subscripted or nonsubscripted variable or an array name.

   c. An arithmetic or a logical expression.

   d. The name of a FUNCTION or SUBROUTINE subprogram.

9. A FUNCTION subprogram is referred to by using its name as an operand in an arithmetic expression and following it with the required actual arguments enclosed in parentheses.

The following example shows the use of a FUNCTION subprogram:

Calling Program	Called Program
.	FUNCTION CALC (A, B)
.	.
.	.
X=Y**2+D*CALC(F, G)	CALC
.	.
.	.
.	.
	RETURN

SUBROUTINE Subprogram

General Form
SUBROUTINE name ($a_1, a_2, \ldots, a_n$) or SUBROUTINE name where name is the symbolic name of the subprogram each argument a, if any, is a dummy name or is the character asterisk and denotes a nonstandard return.

Examples:

```
SUBROUTINE MATMPY (A, N, B, J, *, *)
SUBROUTINE QDRTIC  (B, A, C, ROOT1, ROOT2)
SUBROUTINE OUTPUT
```

1. The SUBROUTINE statement must be the first statement of a SUBROUTINE subprogram.

2. The SUBROUTINE subprogram may use one or more of its arguments to return output. The arguments so used must appear on the left side of an arithmetic statement or in an input list within the subprogram.

3. The arguments may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement, which refers to the SUBROUTINE subprogram. The actual arguments must correspond in number, order, and type with the dummy arguments. Unlike FUNCTIONS, a SUBROUTINE need not have any arguments.

4. When a dummy argument is an array name, a statement containing dimension information must appear in the SUBROUTINE subprogram; also the corresponding actual argument in the CALL statement must be a dimensioned array name.

5. None of the dummy arguments may appear in an EQUIVALENCE, DATA, NAMELIST, or COMMON statement in the SUBROUTINE subprogram.

6. The SUBROUTINE subprogram must contain at least one path that terminates with a RETURN statement and physically terminates with an END statement.

7. The SUBROUTINE subprogram may contain any FORTRAN statements except FUNCTION, another SUBROUTINE statement, or BLOCK DATA.

8. The character *, found as an argument, denotes an alternate EXIT from the subroutine.

Returns from Subprograms

A logical termination of any subprogram is the RETURN statement, which returns control to the calling program. There may be any number of RETURN statements in the program.

General Form
RETURN RETURN i where i is an arithmetic expression whose truncated value, n, denotes the nth dummy statement reference indicated by an * in the argument list, reading from left to right

The normal sequence of execution following the RETURN statement of a SUBROUTINE subprogram is to the next executable statement following the CALL statement in the calling program. It is also possible to return to any labeled executable statement in the calling program by using a special return from the called subprogram. This return must not violate the transfer rules for DO loops. (For an example, see nonstandard return in the CALL STATEMENT section.)

Nonstandard returns may be best understood by considering that a CALL statement using the nonstandard return is logically equivalent to a CALL, a LOGICAL IF, and a computed GO TO statement in that sequence.

FUNCTION subprograms must not have nonstandard returns.

Multiple Entry Points into a Subprogram

The normal entry into a SUBROUTINE subprogram from the calling program is by a CALL statement that refers to the subprogram name. The normal entry into a FUNCTION subprogram is made by a function reference in an expression. Entry is made at the first executable statement following the SUBROUTINE or FUNCTION statement.

It is also possible to enter a subprogram at an alternate entry point by a CALL statement or a function reference that refers to an ENTRY statement in the subprogram. Entry is made at the first executable statement following the ENTRY statement.

ENTRY statements are nonexecutable and, therefore, do not affect control sequencing during normal execution of a subprogram. The order, type, and number of arguments need not agree between the SUBROUTINE or FUNCTION statement and the ENTRY statements, nor do the ENTRY statements have to agree among themselves in these respects. Each CALL or FUNCTION reference, however, must agree in order, type, and number with the SUBROUTINE, FUNCTION, or ENTRY statement that it refers to.

The general form of the ENTRY statement in the called subprogram is:

General Form
ENTRY name $(b_1, b_2, \ldots, b_n)$ where name is the symbolic name of an entry point each b_i is a dummy argument name corresponding to an actual argument in a CALL statement or in a function reference. An ENTRY into a FUNCTION subprogram must have at least one argument an ENTRY into a SUBROUTINE subprogram may have arguments of the form* indicating nonstandard returns (dummy statement references)

Example:

Calling Program

```
        .
        .
        .
1 CALL SUB1(A, B, C, D, E, F)
        .
        .
        .
  2 CALL SUB2(G, H, P)
        .
        .
        .


    3 CALL SUB3
        .
        .
        .
      END
```

Called Program

```
      SUBROUTINE SUB1(U, V, W, X, Y, Z)
                .
                .
                .
10            U = V
                .
                .
                .
             GO TO 60
                .
                .
                .
      ENTRY SUB2(T, U, V)
      GO TO 10
60              .
                .
                .
           ENTRY SUB3
                .
                .
                .
             END
```

In the preceding example, the execution of statement 1 causes entry into SUB1, starting with the first executable statement of the subroutine. Execution of statements 2 and 3 also cause entry into the called program, starting with the first executable statement following the ENTRY SUB2 (T, U, V) and ENTRY SUB3 statements, respectively.

Additional Rules for Entry Points

The following rules also apply to entry points:

1. A dummy argument may not appear in any statement unless it appeared in an argument list of a previously executed FUNCTION, SUBROUTINE, or ENTRY statement.

2. In a FUNCTION subprogram, only the FUNCTION name may be used as the variable to return the function value to the using program. The ENTRY name may not be used for this purpose.

3. An ENTRY name may appear in an EXTERNAL statement in the same manner as a FUNCTION or SUBROUTINE name.

4. Entry into a subprogram initializes all references in the entire subprogram from items in the argument list of the CALL or function reference. (For instance, if, in the example that appeared in the section "Multiple Entry Points into a Subprogram," entry is made at SUB2, the variables in statement 10 will refer to the argument list of SUB2.)

5. The appearance of an ENTRY statement does not alter the rules regarding the placement of statement functions in subroutines. Statement functions may follow an ENTRY statement only if they precede the first usage following the SUBROUTINE or FUNCTION statement.

6. None of the dummy arguments of an ENTRY statement may appear in an EQUIVALENCE, COMMON, NAMELIST, or DATA statement in the same subprogram.

Subprogram Names as Arguments

FUNCTION and SUBROUTINE subprogram names may be the actual arguments of subprograms. To distinguish these subprogram names from ordinary variables or array names when they appear in an argument list, they must appear in an EXTERNAL statement.

Examples:

```
EXTERNAL SIN, COS
CALL SUBR(A, SIN, B)
```

CALL STATEMENT

The CALL statement is used to refer to a SUBROUTINE subprogram.

General Form
CALL subr ($a_1, a_2, \ldots, a_n$) where subr is the name of a SUBROUTINE subprogram $a_1, a_2, \ldots, a_n$ are the n arguments

Examples:

```
CALL MATMPY(X, 5, 10, Y, 7, 2)
CALL QDRTIC(9.732, Q/4.536, R-S**2.0, X1, X2)
CALL OUTPUT
CALL ABC(X, B, C, *5, *200)
```

The CALL statement transfers control to a SUBROUTINE subprogram and presents it with the actual arguments.

The arguments may be any of the following:

1. A constant.
2. A subscripted or nonsubscripted variable or an array name.
3. An arithmetic or logical expression.
4. The name of a FUNCTION or SUBROUTINE subprogram.
5. *n where n is the statement label for a nonstandard return and * differentiates a statement label from an integer constant.

The arguments presented by the CALL statement must agree in number, order, type, and array size (except as explained under the DIMENSION statement) with the corresponding dummy arguments in the SUBROUTINE or ENTRY statement of the called subprogram.

Example of a nonstandard return:

Calling Program

```
                .
                .
                .
10  CALL SUB(A, B, C, *30, *40)
20              ---
                .
                .
                .
39              ---
                .
                .
                .
40              ----
                .
                .
                .
               END
```

Called Program

```
      SUBROUTINE SUB(X, Y, Z, *, *)
                .
                .
                .
100   IF (R) 200, 300, 400
200   RETURN
300   RETURN
400   RETURN 2
             END
```

In the preceding example, execution of statement 10 in the calling program causes entry into subprogram SUB. When statement 100 is executed, the return to the calling program will be via statement 20, 30, or 40, if R is less than, equal to, or greater than zero, respectively.

Mathematical Functions

Many commonly used mathematical functions are provided for use in a FORTRAN program. All the names of these subprograms are automatically typed by the FORTRAN IV Compiler; therefore, they need not appear in Type statements.

Variables used as arguments of mathematical functions are checked for type and converted if required. The mathematical functions are listed in Figure 7-2.

Function	Definition	Number of Arguments	Name	Type of Argument	Type of Function
Exponential	$_{e}^{Arg}$	1	EXP	Real	Real
Natural logarithm	$\log_e(Arg)$	1	ALOG	Real	Real
Common logarithm	$\log_{10}(Arg)$	1	ALOG10	Real	Real
Arctangent	arctan(Arg) in radians	1	ATAN	Real	Real
	$\arctan(Arg_1/Arg_2)$*	2	ATAN2	Real	Real
Trigonometric sine	sin(Arg in radians)	1	SIN	Real	Real
Trigonometric cosine	cos(Arg in radians)	1	COS	Real	Real
Hyperbolic tangent	tanh(Arg)	1	TANH	Real	Real
Square root	$(Arg)^{1/2}$	1	SQRT	Real	Real
Remaindering	$Arg_1 (\text{mod } Arg_2)$	2	DMOD	Double	Double
Exponential	$_{e}^{Arg}$	1	DEXP	Double	Double
Natural logarithm	$\log_e(Arg)$	1	DLOG	Double	Double
Common logarithm	$\log_{10}(Arg)$	1	DLOG10	Double	Double
Arctangent	arctan(Arg) in radians	1	DATAN	Double	Double
	$\arctan(Arg_1/Arg_2)$*	2	DATAN2	Double	Double
Trigonometric sine	sin(Arg in radians)	1	DSIN	Double	Double
Trigonometric cosine	cos(Arg in radians)	1	DCOS	Double	Double
Square root	$(Arg)^{1/2}$	1	DSQRT	Double	Double
Absolute value	For Arg=X+iY $C=(X^2+Y^2)^{1/2}$	1	CABS	Complex	Real
Exponential	$_{e}^{Arg}$	1	CEXP	Complex	Complex
Natural logarithm	$\log_e(Arg)$	1	CLOG	Complex	Complex
Trigonometric sine	sin(Arg in radians)	1	CSIN	Complex	Complex
Trigonometric cosine	cos(Arg in radians)	1	CCOS	Complex	Complex
Square root	$(Arg)^{1/2}$	1	CSQRT	Complex	Complex

* In the source statement, Arg_1 and Arg_2 are separated by a comma.

Figure 7-2. Mathematical FUNCTION Subprogram

BLOCK DATA SUBPROGRAM

A way to enter data into a labeled COMMON block during compilation is by using a BLOCK DATA subprogram. (Data may also be entered into blank COMMON by the use of a DATA statement in any program or subprogram.) This subprogram may contain only the DATA, COMMON, DIMENSION, and Type statements associated with the data being defined.

General Form
BLOCK DATA

1. The BLOCK DATA subprogram may not contain any executable statements.

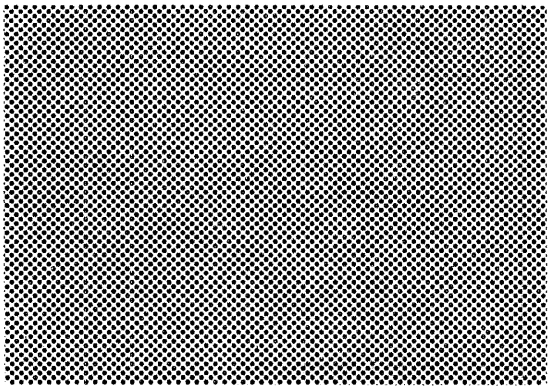
2. The first statement of this subprogram must be the BLOCK DATA statement.

3. All elements of a COMMON block must be listed in the COMMON statement even though they do not all appear in the DATA statement; for example, the variable A in the COMMON statement in Figure 7-3 does not appear in the DATA statement. Therefore, A remains undefined until execution of the program.

4. If two or more BLOCK DATA subprograms occur for the same application, the data specified by each of them is entered into the appropriate COMMON blocks. The data from the last such subprogram is retained for any area of a COMMON block that is referred to more than once.

```
BLOCK DATA
COMMON/ELN/C, A, B/RMC/Z, Y
DIMENSION B(4), Z(3)
DOUBLE PRECISION Z
COMPLEX C
DATA (B(I), I=1, 4)/1. 1. 1. 2, 2*1. 3/, C/(2. 4, 3. 769)/,
   Z(1)/7. 6498085D0/
END
```

Figure 7-3. BLOCK DATA Subprogram



VIII ...

Specification Statements

Specification statements provide information about storage requirements and about the constants and variables used in the program.

DIMENSION STATEMENT

General Form
DIMENSION $v_1(i_1), v_2(i_2), \ldots, v_n(i_n)$ where each v_n is an array variable each i_n is composed of from one to seven unsigned integer constants or integer variables, separated by commas (Integer variables may be a component of i_n only when the DIMENSION statement appears in a subprogram.)

The DIMENSION statement provides the information necessary to allocate storage for arrays in the object program, and it defines the maximum size of the arrays. An array may be declared to have from one to seven dimensions by placing it in a DIMENSION statement with the appropriate number of subscripts appended to the variable.

1. The DIMENSION statement must precede the first appearance of any executable statement, any NAMELIST statement, or any DATA statement in the program, except when NAMELIST is used for DEBUG.

2. A single DIMENSION statement may specify the dimensions of any number of arrays.

3. If a variable is dimensioned in a DIMENSION statement, it must not be dimensioned elsewhere.

4. Dimensions may also be declared in a COMMON or a Type statement. If this is done, these statements are subject to all the rules for the DIMENSION statement.

In the following examples, A, B, and C are declared to be array variables with 4, 1, and 7 dimensions, respectively.

Examples:

```
DIMENSION A(1,2,3,4),B(10)
DIMENSION C(2,2,3,3,4,4,5)
```

Adjustable Dimensions

The name of an array and the constants that are its dimensions may be passed as arguments to a subprogram. In this way, a subprogram may perform calculations on arrays whose sizes are not determined until the subprogram is called. The following example illustrates the use of adjustable dimensions.

```
SUBROUTINE MAYMY(...,R,L,M,...)
                .
                .
                .
DIMENSION...,R(L,M),...
                .
                .
                .
DO 100 I=1,L
```

1. Variables may be used as dimensions of an array only in the DIMENSION statement of a FUNCTION or SUBROUTINE subprogram. For any such array, the array name and all the variables used as dimensions must appear as dummy arguments in the FUNCTION, SUBROUTINE, or ENTRY statement.

2. The adjustable dimensions are not alterable within the subprogram. The values are scanned on entry so the variables may be used as the programmer sees fit.

3. The true dimensions of an actual array must be specified in a DIMENSION statement of the calling program.

4. The calling program passes the specific dimensions to the subprogram. These specific dimensions are those that appear in the DIMENSION statement of the calling program. Variable dimension size may be passed through more than one level of subprogram. The specific dimensions passed to the subprogram as actual arguments cannot exceed the true dimensions of the indicated array.

5. Variables used as dimensions must be integers. If these variables are not implicitly typed by their initial letters, a Type statement must precede the dimension in which they are used as adjustable dimensions.

Example:

```
SUBROUTINE SUB(X, Y, Z)
INTEGER Y, Z
DIMENSION X(Y, Z)
```

COMMON STATEMENT

General Form
COMMON a, b, c, . . . /r/d, e, f, . . . /s/g, h, . . .
where
a, b, . . . are variables that may be dimensioned
/r/, /s/, . . . are labels that are block names

Examples:

```
COMMON A, B, C/X/Q, R/YY/M, P, Q
COMMON /Z/G, H, J/ /D, F
```

There are two types of COMMON storage provided in FORTRAN IV. Blank Common provides an area in which data can be exchanged between various subprograms which may or may not reside in memory at the same time. Labeled Common provides a similar area where data can be exchanged only between those subprograms which currently reside in memory and which make reference to the Labeled Common block.

Variables, including array names, appearing in a COMMON statement are assigned locations relative to the beginning of a particular COMMON block. This COMMON area may be shared by a program and its subprograms.

1. If the variables appearing in a COMMON statement contain dimension information, they must not be dimensioned elsewhere.

2. The order in which storage is assigned in the COMMON area is determined by the sequence in which the variables appear in the COMMON statement, beginning with the first COMMON statement of the program.

3. Elements placed in COMMON may be placed in separate labeled blocks. These separate blocks may share space in core storage at object time. Blocks are given names and those with the same name occupy the same space.

4. COMMON block names. The symbolic name of a block. which is one to eight alphanumeric characters the first of which is alphabetic, precedes the variable names belonging to the block. The block name is always embedded in slashes; for example, /BB/. It must not be the same as the name of any other subprogram that is part of the same job; however, it can be the same as a variable name. In the two types of COMMON blocks:

   a. Blank COMMON is indicated either by omitting the block name if it appears at the beginning of the COMMON statement or by preceding the blank COMMON variable by two consecutive slashes.

   b. Labeled COMMON is indicated by preceding the labeled COMMON variables by the block name embedded in slashes.

5. The field of entries pertaining to a block name ends with a new block name, the end of the COMMON statement, or a blank COMMON designation.

6. Block name entries are cumulative throughout the program. For example, the COMMON statements

```
COMMON A, B, C/R/D, E/S/F
COMMON G, H/R/I/S/P
```

have the same effect as the statement

```
COMMON A, B, C, G, H/R/D, E, I/S/F, P
```

7. Blank COMMON may be any length. Labeled COMMON must conform to the following size requirement: All COMMON blocks of a given name must have the same length in all the programs that are executed together.

8. Variables brought into a COMMON block through EQUIVALENCE statements may increase the size of the block, but they may not reestablish the origin of the block nor reorder the sequence in which variables are stored in the block.

9. Two variables in COMMON may not be made equivalent to each other, directly or indirectly.

EQUIVALENCE STATEMENT

General Form
EQUIVALENCE (a, b, c, . . .), (d, e, f, . . .), . . . where a, b, c, d, e, f, . . . are variables that may be subscripted; these subscripts must be integer constants the number of subscripts appended to a variable must be either equal to the number of dimensions of the variable or must be one

Examples:

```
DIMENSION B(5), C(10, 10), D(5, 10, 15)
EQUIVALENCE (A, B(1), C(5, 4)), (D(1, 4, 3), E)
```

The EQUIVALENCE statement controls the allocation of data storage by causing two or more variables to share the same core storage location.

1. Each pair of parentheses in the statement list encloses the names of two or more variables that are to be assigned the same location during execution of the object program; any number of equivalences (sets of parentheses) may be given.

2. When using the EQUIVALENCE statement with subscripted variables, two methods may be used to specify a single element in the array. For example, D(1,2,1) or D(P) may be used to specify the same element, where p(>0) is the $(p-1)^{th}$ element following the first element of the D array as it will reside in storage. Hence, D(p) references the p^{th} element of the array in storage. (See SUBSCRIPTS for array placement in storage.)

   In the preceding example, the EQUIVALENCE statement indicates that A and the B and C arrays are to be assigned storage locations so that the elements A, B(1), and C(5,4) are to occupy the same location. In addition, it also specifies that D(1,4,3) and E are to share the same location.

3. Quantities or arrays that are not mentioned in an EQUIVALENCE statement will be assigned unique locations.

4. Locations can be shared only among variables, not among constants.

5. The sharing of locations requires a knowledge of which FORTRAN statements will cause a new value to be stored in a location. There are four such statements:

   a. Execution of an arithmetic statement stores a new value in the location assigned to the variable on the left side of the equal sign.

   b. Execution of an ASSIGN i TO n statement stores a new value in the location assigned to n.

c. Execution of a DO statement or an implied DO in an input/output list sometimes stores a new indexing value.

d. Execution of a READ statement stores new values in the location assigned to the variables mentioned in the input list.

6. Variables brought into a COMMON block through EQUIVALENCE statements may increase the size of the block indicated by the COMMON statements, as in the following example:

```
COMMON /X/A, B, C
DIMENSION D(3)
EQUIVALENCE (B, D(1))
```

The layout of core storage indicated by this example (extending from the lowest location of the block to the highest location of the block) is:

```
A
B, D(1)
C, D(2)
   D(3)
```

7. Since arrays must be stored in consecutive forward locations, a variable may not be made equivalent to an element of an array in such a way as to cause the array to extend beyond the beginning of the COMMON block.

8. In non-COMMON, the effect of the EQUIVALENCE statements must be such that the high-order word of any double-word variable is an even number of words away from the start of any other double-word variable linked to it through EQUIVALENCE statements.

9. Two variables in one COMMON block or in two different COMMON blocks must not be made equivalent.

10. The EQUIVALENCE statement does not make two or more elements mathematically equivalent.

11. Equivalenced variables must not appear as dummy arguments in a FUNCTION, SUBROUTINE, or ENTRY statement.

TYPE STATEMENTS

The type of a variable or function may be specified by means of one of the six Type statements.

General Form
INTEGER $a(i_1), b(i_2), c(i_3), \ldots$
EXTENDED INTEGER $(a(i_1), b(i_2), c(i_3)$
REAL $a(i_1), b(i_2), c(i_3), \ldots$
DOUBLE PRECISION $a(i_1), b(i_2), c(i_3), \ldots$
COMPLEX $a(i_1), b(i_2), c(i_3), \ldots$
LOGICAL $a(i_1), b(i_2), c(i_3), \ldots$
STRING $a(i_1):c_1, b(i_2):c_2, c(i_3):c_3, \ldots$
IONAME $a(i_1), b(i_2), c(i_3), \ldots$
EXTERNAL x, y, z, ...
where a, b, c, ... are variable or function names appearing within the program c_1, c_2, c_3 are the ASCII character counts for a, b, c respectively IONAME is defined as 8 ASCII characters long, left justified, and blank filled. x, y, z, ... are function names appearing within the program each i_n represents an optional dimension composed of from one to seven integer constants and/or integer variables

Examples:

```
INTEGER BIXF, X, QF, LSL
EXTENDED INTEGER CJYG, Y, RG, MTM
REAL IMINM, LOG, GRN, KLW
DOUBLE PRECISION Q, J, DSIN
EXTERNAL SIN, MATMPY, INVTRY
INTEGER A(10, 10), B
COMPLEX C(4, 5, 3), D
```

The variable or function names following the type (INTEGER, REAL, etc.) in the Type statement are defined to be of that type and remain so throughout the program; the type may not be changed.

Note in the examples that LSL and GRN need not appear in their respective Type statements since their type is implied by their first characters. Also DSIN (double-precision sine) need not appear in its statement if it is used as a function in the program since mathematical subroutines in the FORTRAN library are automatically typed by the FORTRAN IV Compiler.

1. The appearance of a name in any Type statement overrides the implicit type assignment.

2. Variables that appear in EXTERNAL statements are subprogram names. Subprogram names must appear in an EXTERNAL statement if they are the arguments of other subprograms or if they are the name of a built-in function that is used as the name of a FUNCTION or SUBROUTINE subprogram.

3. A Type statement may also be used to dimension variables. However, any variable that is dimensioned by a Type statement may not be dimensioned elsewhere; that is, it may not appear in a DIMENSION statement or in a COMMON statement that contains dimension information.

DATA STATEMENT

Data may be compiled into the object program by means of the DATA statement.

General Form
DATA list/$d_1, d_2, \ldots, d_n$/, list/$d_1, d_2, k*d_3, \ldots, d_m$/, ...
where
list contains the names of the variables being defined
d is the data literal
$k=k_1*k_2*, \ldots,$ = is an integer constant used as a repeat modifier

Examples:

```
LOGICAL LA, LB, LC, LD
DATA R, Q/14.2, 3HEND/, Z/07
DATA(B(I), C(I), I=2.40, 2)/2.0, 3.0, 38*100.0/
DATA LA, LB, LC, LD/F, .TRUE., .FALSE., T/
DATA(HOL(KY), KY=1, 3)/6HANYbDA, 6HTAbHER, 6HEbbbbbb/
```

1. List. Subscripted variables may appear in the list. When the subscript is a variable, it must be under control of DO-implying parentheses and associated parameters. Subscripts not so controlled must be integer constants. The DO-defining parameters must be integer constants.

2. d. The data literals may take either of the following forms:

   - Any constant previously defined.
   - An octal constant is written as the letter 0, optional six digits.

3. k. The number k may appear before a d-field to indicate that the field is to be repeated k times. An asterisk must follow the letter k to separate it from the field to be repeated. K may be recursed (i.e., $K=K_1*K_2, \ldots$)

4. There must be a one-to-one correspondence between the list items and the data literals. Each data literal (integer, extended integer, real, alphanumeric, complex, logical, double-precision, octal constant or string constant) corresponds to one nondimensioned variable or subscripted array reference on a word basis.

```
DATA G(1)/16HDATAbISbREAD/
```

However, the following would be illegal:

```
DATA G/16HDATA␣TO␣BE␣READ␣/
DATA G/3.0,16HDATA␣TO␣BE␣READ␣,4.0,5.0,6.0/
```

5. The BLOCK DATA subprogram, which includes a DATA statement, compiles data into the labeled COMMON area of the program.

6. The DATA statement may not be used to enter data into blank COMMON.

7. DATA defined variables that are redefined during execution assume their new values regardless of the DATA statement.

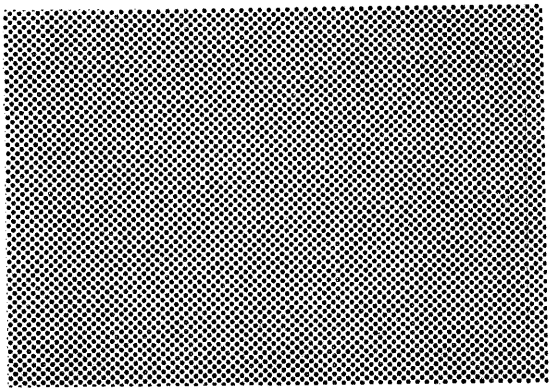
8. Where data is to be compiled into an entire array, the name of the array (with indexing information omitted) can be placed in the list. The number of data literals must exactly equal the size of the array.

   For example, the statements

   ```
   DIMENSION B(25)
   DATA A,B,C/24*4.0,3.0,2.0,1.0/
   ```

   define the values of A, B(1), . . . , B(23) to be 4.0, and the values of B(24), B(25), and C to be 3.0, 2.0, and 1.0, respectively.

9. No check is made between the type of the variable in the variable list and the type of the data in the corresponding data list.



IX ...

Direct Statements & Environment

FORTRAN DIRECT STATEMENTS

1. FORTRAN
2. LIST (arg)
3. EDIT arg
4. DELETE arg
5. EXTRACT arg
6. SOURCE arg
7. OBJECT arg
8. LOAD arg
9. COMPILE (arg)
10. EXECUTE (arg)
11. RUN (arg)
12. QUIT
13. TAPE (arg)
14. D^c (control - D)
15. DEBUG
16. MONITOR arg
17. -MONITOR (arg)
18. BREAK arg
19. -BREAK (arg)
20. TRACE arg
21. -TRACE
22. PRINT arg
23. LET arg
24. GOTO arg
25. STEP
26. CONTINUE
27. ESCAPE key
* 28. CREATE arg
* 29. RENAME arg
* 30. ERASE arg
* 31. DESTROY arg

NOTE: "arg" indicates arguments required.
"(arg)" indicates arguments optional.
* indicates file management statements

SIGNATURE CHARACTERS

- Normal FORTRAN signature character is "#", lb. sign. This character precedes indirect, and some, direct FORTRAN statements.
- Debug mode FORTRAN signature character is "?" question mark.
- Request for input from a terminal during program execution is preceded by a "?", question mark.

DIRECT STATEMENTS

1. FORTRAN - the name of the 2+2 FORTRAN IV subsystem. While in the FORTRAN subsystem, the statement FORTRAN is recognized - but nothing is done. The subsystem responds to the statement with the signature character #.

2. LIST

   or

   LIST LN_1, LN_2-LN_3, LN_4, LN_5-LN_6, ...

   the LIST statement causes the current source program to be listed as follows:

LIST with no arguments lists the whole source program

LIST --- with arguments - lists the designated statements or range of statements. LIST, 30, 10-15, 40 lists lines 10 through 40 (inclusive), line 30, and line 40. No diagnostic or comment is made if designated line numbers are not present.

3. EDIT LN_1, LN_2-LN_3, LN_4, LN_5-LN_6, ...

The EDIT statement prepares the subsystem for editing or modifying one or more source statements in the current source program. For example:

EDIT 10, 20-25

Line 10 is printed at the user terminal. The user modifies or changes the statement. Following a carriage return, line 20 is printed and can be edited. Next, line 21, 22, ... thru line 25. Statements are edited one at a time until the argument list is exhausted. Missing line numbers are ignored.

4. DELETE LN_1, LN_2-LN_3, LN_4, LN_5-LN_6, ...

The DELETE statement causes portions of the current source program to be erased. Note that the execution of DELETE does <u>not</u> affect the permanently saved program unless the statement **SOURCE** file name is executed after the DELETE statement.

DELETE 35, 75-350, 900, 990

causes the deletion from the program of line number 35, line 75 thru 350, line 900, and line 990.

5. EXTRACT LN_1, LN_2-LN_3, LN_4, LN_5-LN_6, ...

The EXTRACT statement is the complement of the DELETE statement. EXTRACT deletes all of the current source program but the referenced line numbers. It is useful in taking portions of one program and preparing them for insertion in another program. Note that execution of EXTRACT does not affect the permanently saved program unless the statement SOURCE file name is executed after the EXTRACT statement.

EXTRACT 100-300, 500-600

causes line numbers 100 thru 300, and line numbers 500 thru 600 to be "pulled out" of the current source program. All other line numbers are deleted.

6. SOURCE file name

The SOURCE statement saves the current source program, as it exists, in a permanent file designated by "file name". The system responds with "NEW FILE" if the file name does not exist in the user's directory. The system responds with "OLD FILE" if this name already appears in the user's directory. The user responds to "NEW FILE" or "OLD FILE" by typing a carriage return (which

creates a new file, or replaces an old file), or, aborting the statement by using the ESCAPE KEY.

7. OBJECT file name

The OBJECT statement is identical to the SOURCE statement but saves the current object program, as it exists, in a permanent file designated by "file name". The system responds with "NEW FILE" if the file name does not exist in the user's directory. The system responds with "OLD FILE" if this name already appears in the user's directory. The user responds to "NEW FILE" or "OLD FILE" by typing a carriage return (which creates a new file, or replaces an old file), or, aborting the statement using the ESCAPE KEY.

8. LOAD file name

The LOAD statement retrieves the permanently saved file designated by file name and places it in working storage for FORTRAN. The file may be in either source or object form. If in source form, the file may be listed, edited, compiled or executed. If the file is in object form, it may only be executed.

9. COMPILE

or

COMPILE file name

The COMPILE statement initiates compilation of either the current source program, or the file designated by file name. If the designated file is not FORTRAN source code, an error diagnostic is initiated.

The whole source program is compiled. Any errors during compilation are listed at the user's terminal.

10. EXECUTE

or

EXECUTE file name

The EXECUTE statement initiates execution of an object (or compiled) program. EXECUTE with no arguments executes the current object program in the user's working area. EXECUTE with file name executes the file designated by file name - if the designated file is in object form.

11. RUN

or

RUN file name

The RUN statement is a combination of the COMPILE and EXECUTE statements. Either the current source program or the file designated by file name, is compiled into object form and then executed. Execution begins only if no compilation errors are present.

12. QUIT

The QUIT statement transfers control from the FORTRAN subsystem to the executive subsystem. The statement is used to terminate processing under FORTRAN. The user may then invoke another subsystem or log off of the system.

13. TAPE

or

TAPE file name

The TAPE statement causes the FORTRAN subsystem to accept source statements prepared on punched paper tape. The statements are accumulated into either the users working storage are, or into the file designated by file name.

Normally, the system acknowledges the receipt of each source statement by sending a line feed and signature character to the user terminal. The Tape statement indicates that the line feed and signature character response are to be deleted since they would interfere with the printout at the user terminal.

14. D^C (control D character)

The D^C character is used to reset system operation to the normal mode after having read in a punched paper tape.

15. DEBUG

The DEBUG statement is used to interrupt an executing program and enter the DEBUG mode.

The DEBUG statement is valid only when typed during program execution. Furthermore, the DEBUG statement only has meaning when the executing program has statements compiled in the debug mode.

When "DEBUG" is typed at the user's terminal, program execution is suspended at the first debuggable statement in the program. The live number of interrupted statement and the debug signature character ("?") are typed at the user's terminal. The user may then execute any of the debug statements; alter MONITOR, BREAK, and TRACE statements; set single or continue step mode; and then continue execution of the program.

16. MONITOR LN_1, LN_2-LN_3, VAR, (VAR, N_1-N_2), ...

Where: LN is a line number.

VAR is a simple variable or array name (i.e., ABC, DEF (3, 4)).

N is an occurrence count.

The MONITOR statement applies only to statements compiled in debug mode. Line numbers and variables can be monitored during program execution. For example,

MONITOR 1000-1010, 257, SDOT, YDOT, ZDOT

causes the following:

- whenever any of the line numbers, 1000 thru 1010, inclusive, are executed, the line numbers are printed at the user's terminal.
- line number 257 is monitored whenever executed by printing the line number at the user's terminal.
- the contents of the variables XDOT, YDOT, and ZDOT are printed at the user's terminal whenever they appear on the left hand side of an assignment statement. The value printed is that resulting after execution of the statement. If XDOT is referred to at line number 2110, the printout would be:

```
2110 XDOT = XXXXXX
```

An additional argument form for the MONITOR, statement can be illustrated with:

```
MONITOR (VELOCITY, 5-8)
```

This statement causes the variable VELOCITY to be monitored only for the 5th thru 8th time it occurs. This allows the user to monitor a variable only during significant portions of execution, thus reducing the amount of information output to the user's terminal. MONITOR may only be used with a program compiled in debug mode.

17. -MONITOR

or

-MONITOR LN_1, LN_2-LN_3, VAR, (VAR, N_1-N_2), ...

Where: LN is a line number.

VAR is a simple variable or array name.

N is an occurrence count.

The -MONITOR statement resets or directs FORTRAN to UN monitor the specified line numbers or variables. -MONITOR with no argument "turns off" all previously referenced monitor functions. -MONITOR with an argument list turns off only those variables and line numbers specified in the argument list.

The interpretation of arguments is described under the MONITOR statement (Item 17).

18. BREAK LN_1, LN_2-LN_3, VAR, (VAR, N_1-N_2), ...

The BREAK statement is similar in execution to the MONITOR statement. The BREAK statement allows the user to set break points within the executable program.

Whenever a line number or variable mentioned as an argument of a BREAK statement is executed, the item is printed at the user's terminal. The debug mode signature character (?) is printed, program execution is temporarily suspended, and

FORTRAN nters debug mode. The user may then execute any of the debug statements or continue execution. Interpretation of the argument list is described under the MONITOR system (Item 17). BREAK may only be used with a program compiled in debug mode.

19. -BREAK
    or
    -BREAK LN_1, LN_2-LN_3, VAR, (VAR, N_1-N_2), ...

The -BREAK statement resets or directs FORTRAN to UN break the specified line numbers or variables. -BREAK with no argurment "turns off" all previously referenced break functions. -BREAK with an argument list turns off only those variables and line numbers specified in the argument list.

The interpretation of arguments is described under the MONITOR statement (Item 17).

20. TRACE { LINE / LABEL / SUBPROG / SOURCE }

The TRACE statement is used to logically trace execution of a program according to one or more of the four available modes.

TRACE LINE — Each line number executed is traced for the user. Rather than tracing long sequences of sequential portions of a program, only line numbers following transfer of control are printed at the user's terminal.

TRACE SUBPROG — Execution of a program is traced through execution of each subprogram. The subprogram name is output at the user's terminal. The main program is indicated by "..MAIN.."

TRACE SOURCE — The entire source program is traced. As each abbreviated source statement is reproduced at the user's terminal. Expressions are printed as the value by type.

Multiple traces may be initiated via the TRACE statement. For example,

```
TRACE LINE, SUBPROG
```

causes a trace by both subprogram name and internal line number.

TRACE may only be used with a program compiled in debug mode.

21. -TRACE

The -TRACE statement clears all references to any traces previously initiated. It is used to delete any traces currently in effect.

22. PRINT VAR, VAR, VAR, ...

Where: VAR is a simple variable or array name.

The PRINT statement may only be invoked during program execution in debug mode. The debug mode is entered by executing a BREAK statement, the DEBUG statement, or via some run-time error diagnostics. The PRINT statement may be used after receiving the debug mode signature character, (?), question mark, at the user's terminal.

Arguments for the PRINT statement may be variable names, array names, or elements of an array. No expressions are evaluated. For example,

```
? PRINT X,B,C(3,5)
```

where X is a variable

B is an array dimensioned as B(5)

C(3,5) is an element of the array C,

results in the following output at the user's terminal:

```
X = XXXX
B(1) = XXXX
B(2) = XXXX
  .
  .
  .
B(5) = XXXX
C(3, 5) = XXXX
```

The PRINT statement may be executed any time the program execution is suspended in debug mode.

23. LET VAR = $\left\{\begin{matrix}\text{LITERAL}\\ \text{VAR}\end{matrix}\right\}$

Where: VAR is a variable or array element name

LITERAL is a numeric or string constant

The LET statement may only be invoked during program execution in debug mode. The debug mode is entered by executing a BREAK statement; the DEBUG statement; or via some run-time error diagnostics. The LET statement may be used after receiving the debug mode signature character, (?), question mark at the user's terminal.

The arguments for the LET statement may be variables and literals. A variable is a variable, name or an array element name. A literal is a numeric constant or a string constant. No expressions are evaluated. Consider

```
? LET A(2, 3, 4) = 27.5
```

when execution of the program is resumed, the array element A(2, 3, 4) will contain the value of the literal, 27.5.

```
? LET X = B(32)
```

when execution is resumed the variable X will contain the value contained in B(32).

The LET statement may be executed any time program execution is suspended in debug mode.

24. GOTO LN

Where: LN is a line number

The GOTO statement may be invoked only during program execution in debug mode. The debug is entered by executing a BREAK statement; the DEBUG statement; or via some run-time error diagnostics. The GOTO statement may be used after receiving the debug mode signature character (?), question mark, at the user's terminal.

The argument of the GOTO statement is the line number of the program at which it is desired that execution of the program resume.

```
? GOTO 1220
```

resumes execution of the program at line number 1220.

The GOTO statement may be executed any time program execution is suspended in debug mode.

25. STEP

The STEP statement may be invoked only during execution of a program compiled in debug mode. Debug mode is entered as described in PRINT, LET, and GOTO (Items 23 - 25).

STEP sets the mode of execution of a program to single step mode. Statements compiled in debug mode are executed in single step. That is after execution of each statement, the program is interrupted; the line number of the statement and the debug signature character (?) are typed at the user's terminal. The user may then execute any of the debug statements; alter MONITOR, BREAK and TRACE statements; reset the continue mode; and resume execution of the program.

The STEP statement may be executed any time program execution is suspended in debug mode.

26. CONTINUE

The CONTINUE statement may be invoked only during execution of program compiled in debug mode. Debug mode is entered as described in PRINT, LET, and GOTO (Items 23 - 25).

CONTINUE sets the mode of execution program to normal or multiple step mode. It should be used to reset the STEP statement when single step execution is no longer desired.

The CONTINUE statement may be executed any time program execution is suspended in debug mode.

27. ESCAPE KEY (ALT MODE)

The ESCAPE (or ALT MODE) key is used to immediately interrupt and abort the current function being performed by the FORTRAN subsystem. The function is terminated. The following interpretation applies to the ESCAPE (ALT MODE) key:

Typing 1 ESCAPE	-	aborts current function and returns control to FORTRAN direct statement processor.
Typing 2 ESCAPES	-	the FORTRAN subsystem; itself, is aborted and contro is returned to the EXEC subsystem.

The ESCAPE (ALT MODE) key may be used any time while under control of the FORTRAN subsystem.

28. CREATE a file

The CREATE statement is used to enter a new file name in the user's directory. The file is initially set to contain no information. Optionally, access privileges and additional user access may be specified.

The simplest form of the statement is:

CREATE file name

where, file name is any legal user file name not exceeding eight characters.

Examples:

```
CREATE Z
CREATE ALPHA
CREATE MASTFILE
```

The creator of a file is assigned full access privileges to that file. Those privileges include (currently) READ, WRITE, EXECUTE, and APPEND access.

To assign privileges to other users who wish to access a file, the CREATE statement may be expanded in the following form:

CREATE file name /user ID/access/user ID/access . . .

where file name is any legal file name of 1 to 8 characters

user ID is a unique identification string delimited by slashes.

access is any combination of the following words separated by commas: READ, WRITE, EXECUTE, APPEND, or PASSWORD.

Any number of users may be authorized access to a file. However, access privileges must be specified for each authorized user. Public access is specified by a null user ID (i.e., 11). Consider,

CREATE PAYROLL /AB234/READ/J. SHMOE/ APPEND, EXECUTE

The file PAYROLL is created. User AB234 is given READ access, and J. SHMOE is authorized to EXECUTE and APPEND to the file. The statement,

CREATE MILLISIN //READ, EXECUTE

creates the file named MILLISIN. It also makes the file public with both READ and EXECUTE privileges.

29. RENAME a file

The RENAME statement allows a user to rename any file currently in a user's directory. Optionally, additional users and access privileges may be specified.

The simplest form of the statement is:

RENAME file name, file name

where file name is any legal file name of 1 to 8 characters.

Example:

RENAME NEWTRANS, OLDTRANS

This renames the file currently in the user's directory as NEWTRANS to a file named OLDTRANS. The name NEWTRANS is removed from the user's directory, and is replaced by the name OLDTRANS.

Additional access privileges may be specified while renaming a file by obtaining user ID's and access.

Example:

RENAME ABC, DEF/USER27/READ, WRITE/USER33/EXECUTE

File ABC is renamed DEF. In addition, USER27 is authorized to READ and WRITE the file, and USER33 is granted execute only access.

30. ERASE a file

The ERASE statement allows a user to erase or clear the information currently contained in one or more files. The form is:

ERASE file name, file name,. . .

where file name is any legal file name of 1 to 8 characters.

Examples:

```
ERASE Q
ERASE WEEK32,WEEK33,WEEK34
```

31. DESTROY a file

The DESTROY statement provides the method for destroying both the information contained in a file and removing the entry containing the name of the file in the user's directory. That is, both the contents and the name of the file are destroyed. The form is:

DESTROY file name, file name,. . .

where file name is any legal file name of 1 to 8 characters.

Examples:

```
DESTROY FIRST
DESTROY SECOND, THIRD, FOURTH
```