File No. S360-25
GC28-2007-4

Systems Reference Library

IBM Time Sharing System
FORTRAN IV

This publication describes and illustrates the use of the IBM FORTRAN IV language for the IBM Time Sharing System (TSS).

The IBM FORTRAN IV language is a symbolic programming language that parallels the symbolism and format of mathematical notation. It provides programming features and facilities that can be used in FORTRAN programs to solve mathematical problems.

The material in IBM FORTRAN IV is arranged to provide a quick definition and syntactical reference to the various elements of FORTRAN IV by means of a box format. Each element is described, with appropriate examples of possible use.

The reader should have a basic knowledge of the FORTRAN language. While some information relating FORTRAN IV to TSS is presented, most of the necessary guidance required by a FORTRAN user to perform a task is given in IBM Time Sharing System: FORTRAN Programmer's Guide, GC28-2025.

PREFACE

This manual describes the IBM Time Sharing System FORTRAN IV language. The material in this publication is arranged to provide a quick definition and syntactical reference to the various elements of FORTRAN IV by means of a box format. Each element is described, with appropriate examples of possible use.

Five appendixes give additional information for writing a FORTRAN Iv source program:

- A: FORTRAN Comparison
- B: Table of Source Program Characters
- C: Other FORTRAN Statements Accepted by IBM FORTRAN IV
- D: FORTRAN-Supplied Subprograms
- E: Examples of FORTRAN-Written Programs

The user should have a basic knowledge of the FORTRAN language before using this publication. While some information relating FORTRAN IV to TSS is presented, most of the necessary guidance required by a FORTRAN user to perform a task is given in IBM Time Sharing System: FORTRAN Programmer's Guide, GC28-2025.

This publication also refers to IBM Time Sharing System: FORTRAN IV Library Subprograms, GC28-2026.

Third Edition (May 1976)

This edition replaces, but does not obsolete, GC28-2007-3.

This edition is current with Release 2.0 of the IBM Time Sharing System (TSS/370), and remains in effect for all subsequent versions or modifications of TSS unless otherwise noted. Significant changes or additions to this publication will be provided in new editions or Technical Newsletters.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to the IBM Corporation, Time Sharing System--Dept 80M, 1133 Westchester Avenue, White Plains, New York 10604.

FIGURES

TABLES

IBM TIME SHARING SYSTEM FORTRAN IV

TSS FORTRAN IV includes a language, a compiler, and a set of system-supplied subprograms.

For more information regarding the compiler and system-supplied programs, see FORTRAN Programmer's Guide and FORTRAN IV Library Subprograms. The FORTRAN compiler operates under the control of TSS, which provides the compiler with input/output and other services. Object programs generated by the compiler operate under TSS control and are dependent on it for similar services.

The TSS FORTRAN IV language is compatible with and encompasses the American National Standards Institute (ANSI) FORTRAN, including its mathematical subroutine provisions. Source programs written in FORTRAN consist of a set of statements constructed from the elements of the language described in this publication.

SPECIAL FEATURES OF TSS FORTRAN IV

TSS FORTRAN IV is a further development of previously implemented FORTRAN systems and contains many of the features of these systems (See "Appendix A: FORTRAN Comparison"). The following features facilitate the writing of source programs and reduce the possibility of coding errors:

1. Variable Attribute Control: The attributes of variables and arrays may now be explicitly specified in the source program. This facility is provided by a single explicit specification statement that allows a programmer to:

   a. Explicitly type a variable as integer, real, complex, double precision, or logical.
   b. Specify the number of storage location bytes to be occupied by each variable or member of an array.
   c. Specify the dimension of an array.
   d. Specify data initialization values for variables.

2. Adjustable Array Dimensions: The dimensions of an array in a subprogram may be specified as variables; when the subprogram is called, the absolute array dimensions are substituted.

3. Additional Format Code: An additional format code - G - can be used to specify the format of numeric and logical data. Previously implemented format codes are also permitted.

4. Mixed Mode: Expressions may consist of constants and variables that are of the same and/or different types and lengths.

5. Namelist I/O: Formatting of input/output data is facilitated by reading and writing operations without reference to a FORMAT statement or list.

6. Spacing Format Code: The T format code allows input/output data to be transferred beginning at any specified position.

7. Literal Format Code: Apostrophes may be used to enclose literal data.

ELEMENTS OF THE LANGUAGE

STATEMENTS

FORTRAN statements are composed of FORTRAN key words used in conjunction with the basic elements of the language (constants, variables, and expressions). The five categories of FORTRAN statements are:

1. Arithmetic and Logical Assignment Statements: Replace the current value of a designated variable after calculations have been performed.

2. Control Statements: Govern the flow and terminate the execution of the object program.

3. Input/Output Statements: Exchange information between a user's program and a named collection of data, called a data set.

4. Specification Statements: Declare the properties of variables, arrays, and subprograms (e.g., type and amount of storage reserved) and describe the format of input or output data.

5. Subprogram Statements: Define and name functions and subroutines.

CODING FORTRAN STATEMENTS

Card Input

The statements of a FORTRAN source program can be written on a standard FORTRAN coding form, Form X28-7327 (see Figure 1). FORTRAN statements are written one to a line from columns 7 through 72. If a statement is too long for one line, it may be continued on as many as 19 successive lines by placing any character, other than a blank or zero, in column 6 of each continuation line. For the first line of a statement, column 6 must be blank or zero.

Columns 1 through 5 of the first line of a statement may contain a statement number consisting of from one through five decimal digits. Leading zeros in a statement number are ignored. Statement numbers may appear anywhere in columns 1 through 5 and may be assigned in any order; the value of statement numbers does not affect the order in which the statements are executed in a FORTRAN program.

Columns 73 through 80 are not significant to the FORTRAN compiler and may, therefore, be used for program identification, sequencing, or any other purpose.

Comments to explain the program may be written in columns 2 through 80, if the letter C is placed in column 1. Comments may appear anywhere within the source program except immediately preceding a continuation line. They are not processed by the FORTRAN compiler, but are printed on the source program listing. Blanks may be inserted where desired to improve readability.

Keyboard Input

The conversational capability of TSS FORTRAN allows statements to be entered via a keyboard at a terminal. The rigid card column rules are relaxed for this means of input. A detailed description of keyboard input is contained in FORTRAN Programmer's Guide.

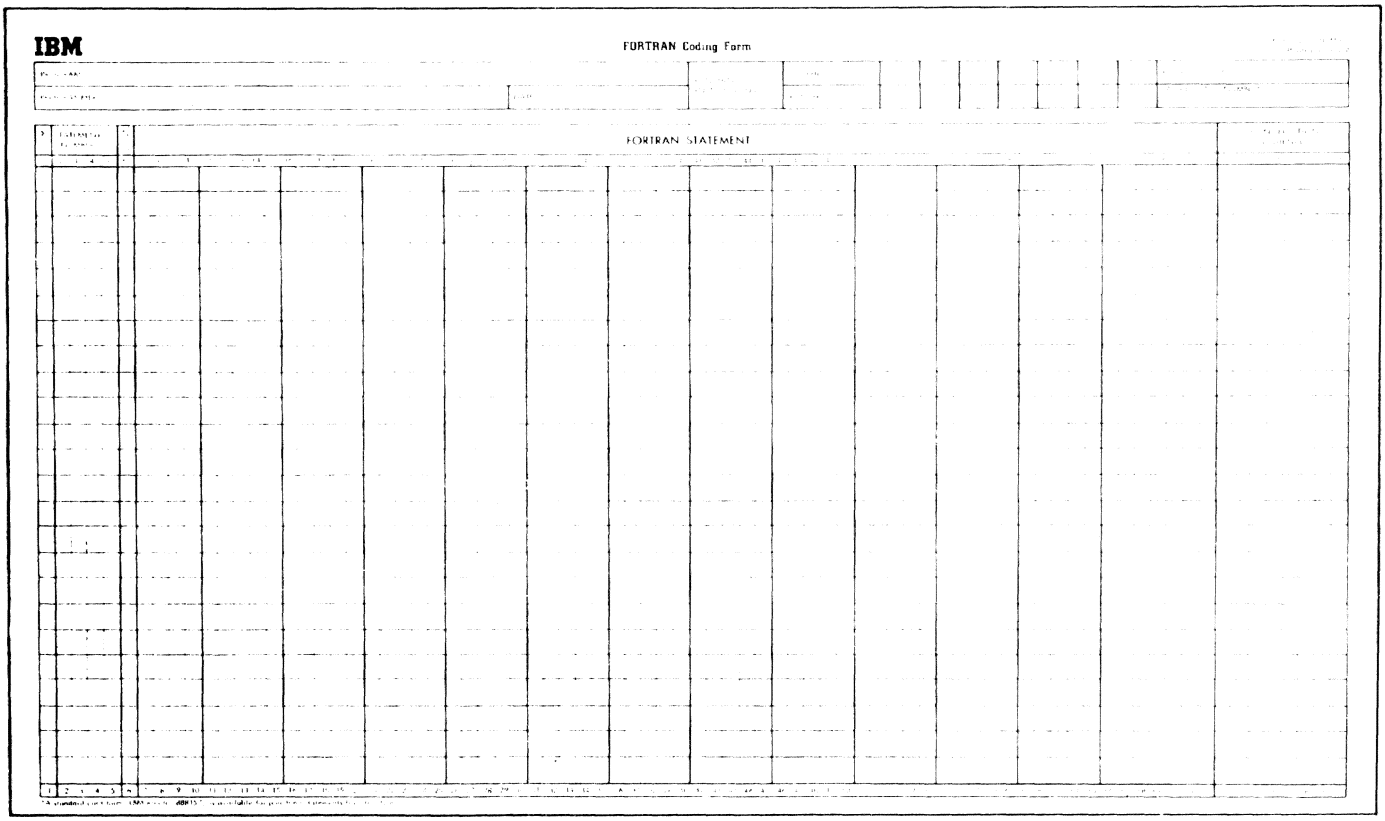


Figure 1. FORTRAN coding form

CONSTANTS

A constant is a fixed, unvarying quantity. The three classes of constants are those that deal with numbers (numerical constants), truth values (logical constants), and literal data (literal constants).

Numerical constants may be integer, real, or complex numbers; logical constants may be .TRUE. or .FALSE.; literal constants may be a string of alphameric and/or special characters enclosed by apostrophes.

INTEGER CONSTANTS

Definition
Integer Constant - a whole number written without a decimal point; it occupies two or four storage locations; four storage locations is the standard length. Maximum Magnitude: 2147483647, i.e., $(2^{31}-1)$.

An integer constant may be positive, zero, or negative; if unsigned, it is assumed to be positive. Its magnitude must not be greater than the maximum and may not contain embedded commas.

Examples: Valid Integer Constants:

```
0
91
173
-2147483647
-12
```

Invalid Integer Constants:

0.0	(contains a decimal point)
27.	(contains a decimal point)
3145903612	(exceeds the allowable range)
5,396	(contains embedded comma)

REAL CONSTANTS

Definition
Real Constant -- a number with a decimal point optionally followed by a decimal exponent, or an integer constant followed by a decimal exponent. The exponent may be written as the letter E or D followed by a signed or unsigned, one- or two-digit integer constant. A real constant may assume one of two forms: 1. From 1 through 7 decimal digits, optionally followed by an E and a decimal exponent. This form occupies 4 storage locations. 2. Either 1 through 7 decimal digits, followed by a D and a decimal exponent, or 8 to 16 decimal digits, optionally followed by a D and a decimal exponent. This form occupies 8 storage locations and is sometimes referred to as a double-precision constant. Magnitude: (either form) 0 or 16^{-65} through 16^{63} (i.e., approximately 10^{75}).

A real constant may be positive, zero, or negative (if unsigned, it is assumed to be positive) and must be of the allowable magnitude. It may not contain embedded commas. The decimal exponent E or D permits the expression of a real constant as the product of a real constant times 10, raised to a desired power. If a decimal exponent is given, the decimal point is not required.

Examples: Valid Real Constants (4 storage locations):

+0.	
-999.9999	
0.0	
5764.1	
7.0E+0	(i.e., $7.0 \times 10^{0} = 7.0$)
19761.25E+1	(i.e., $19761.25 \times 10^{1} = 197612.5$)
7E3	(i.e., $7.0 \times 10^{3} = 7000.0$)
7.E3	
7.0E3	
7.0E03	
7.0E+03	
7.0E-03	(i.e., $7.0 \times 10^{-3} = .007$)

Valid Real Constants (8 storage locations):

21.98753829457168	
1.0000000	
7.9D3	(i.e., $7.9 \times 10^{3} = 7900.0$)
7.9D03	
7.9D+03	
7.9D+3	
7.9D-03	(i.e., $7.9 \times 10^{-3} = .0079$)
7.9D0	(i.e., $7.9 \times 10^{0} = 7.9$)
0.0D0	(i.e., $0.0 \times 10^{0} = 0.0$)
7D3	(i.e., $7 \times 10^{3} = 7000$)

Invalid Real Constants:

0	(missing a decimal point)
3,471.1	(contains embedded comma)
1.E	(missing a one- or two-digit integer constant following the E; note that it is not interpreted as 1.0 x 10^0)
7.9D	(missing a 1- or 2-digit integer constant following the D)
1.2E+113	(E is followed by a 3-digit integer constant)
21.3D90	(value exceeds the magnitude permitted; that is, 21.3 x $10^{90} > 16^{63}$)
23.5E+97	(value exceeds the magnitude permitted; that is, 23.5 x $10^{97} > 16^{63}$)

COMPLEX CONSTANTS

Definition
Complex Constant -- an ordered pair of signed or unsigned real constants separated by a comma and enclosed in parentheses. These real constants may assume one of two forms: 1. From 1 through 7 decimal digits, optionally followed by an E decimal exponent. In this form, each number in the pair occupies 4 storage locations. 2. Either 1 through 7 decimal digits, followed by a D decimal exponent, or 8 through 16 decimal digits, optionally followed by a D decimal exponent. In this form each number in the pair occupies 8 storage locations. Magnitude: (either form) 0 or 16^{-65} through 16^{63} (i.e., approximately 10^{75}) for each real constant in the pair.

The real constants in a complex constant may be positive, zero, or negative (if unsigned they are assumed to be positive), but they must be within the given range. The first real constant in a complex constant represents the real part of the complex number; the second represents the imaginary part of the complex number.

Examples: Valid Complex Constants:

(3.2,- 1.86)	(has value 3.2-1.86i)
(-5.0E+03,.16E+02)	(has value -5000.+16.0i)
(4.0E+03,.16E+02)	(has value 4000.+16.0i)
(2.1,0.0)	(has value 2.1+0.0i)
(4.7D+2,1.9736148)	(has value 470.+1.9736148i)

where $i = \sqrt{-1}$

Invalid Complex Constants:

(292704,1.697)	(real part does not contain decimal point)
(1.2E113,279.3)	(real part contains invalid decimal exponent)
(.0034E4,.005D6)	(parts differ in length)

LOGICAL CONSTANTS

Definition
Logical Constant -- The two logical values are: .TRUE. .FALSE.

A logical constant must be preceded and followed by a period. The logical constants .TRUE. and .FALSE. specify that the value of the logical variable they replace, or the term of the expression they are associated with, is true or false, respectively (see "Logical Expressions").

LITERAL CONSTANTS

Definition
Literal Constant -- a string of alphameric and/or special characters enclosed in apostrophes.

The number of characters in the string, including blanks, may not be greater than 255. Since apostrophes delimit literal data, a single apostrophe within such data is represented by double apostrophes. An alternative form for a literal constant is wH immediately followed by a string, of length w, of alphameric and/or special characters. A single apostrophe within such data is represented as a single apostrophe.

Examples:

```
'DATA'
'INPUT/OUTPUT AREA NO.  2'
'X-COORDINATE     Y-COORDINATE     Z-COORDINATE'
'3.14'
'DON''T'
5HDON'T
```

HEXADECIMAL CONSTANTS

Definition
Hexadecimal Constant -- the character Z followed by a number formed from the set 0 through 9 and A through F.

Hexadecimal constants may be used only as data initialization values.

One storage location contains two hexadecimal digits. If a constant is specified as an odd number of digits, a leading hexadecimal zero is added to fill the storage location. The internal form of each hexadecimal digit is:

0-0000	4-0100	8-1000	C-1100
1-0001	5-0101	9-1001	D-1101
2-0010	6-0110	A-1010	E-1110
3-0011	7-0111	B-1011	F-1111

Examples:

```
Z1C49A2F1
ZBADFAD
```

The maximum number of digits allowed in a hexadecimal constant depends upon the length specification of the variable being initialized (see "Variable Types and Length Specifications"). The following list shows the maximum number of digits for each length specification.

Length Specification of Variable	Maximum Number of Hexadecimal Digits
16	32
8	16
4	8
2	4
1	2

If the number of hexadecimal digits is greater than the maximum, the leftmost digits are truncated; if the number of digits is less than the maximum, leading hexadecimal zeros are added.

SYMBOLIC NAMES

Definition
Symbolic Name -- from 1 through 6 alphameric (i.e., 0 through 9, or alphabetic, A through Z and $) characters, the first of which must be alphabetic.

Symbolic names are used in a program unit (i.e., a main program or a subprogram) to identify elements in the following classes.

- An array and the elements of that array (see "Arrays")
- A variable (see "Variables")
- A statement function (see "Statement Functions")
- An intrinsic function (see Appendix D)
- A FUNCTION subprogram (see "FUNCTION Subprograms")
- A SUBROUTINE subprogram (see "SUBROUTINE Subprograms")
- A block name (see "BLOCK DATA Subprograms")
- An external procedure that cannot be specified as either a SUBROUTINE or FUNCTION subprogram (see "The EXTERNAL Statement")

Symbolic names must be unique within a class in a program unit and can identify elements of only one class with the following exceptions.

A block name can also be an array, variable, or statement function name in a program unit.

A FUNCTION subprogram name must also be a variable name in the FUNCTION subprogram.

Once a symbolic name is used as a FUNCTION subprogram name, a SUBROUTINE subprogram name, a block name, or an external procedure name in any unit of an executable program, no other program unit of that executable

program can use that name to identify an entity of these classes in any other way.

VARIABLES

A FORTRAN variable is a symbolic representation of a quantity that may assume different values. The value of a variable may change either for different executions of a program or at different stages within the program.

For example, in the statement

```
A = B+5.0
```

both A and B are variables. The value of B is determined by some previous statement and may change from time to time; the value of A varies whenever this computation is performed with a new value for B.

VARIABLE NAMES

The use of meaningful variable names can serve as an aid in documenting a program. That is, someone other than the programmer may look at the program and understand its function. For example, to compute the distance a car traveled in a certain length of time at a given rate of speed, the following statement could have been written:

```
X = Y * Z
```

where * designates multiplication. However, it would be more meaningful to someone reading this statement if the programmer had written

```
DIST = RATE * TIME
```

Examples: Valid Variable Names:

```
B292
RATE
SQ704
$VAR
```

Invalid Variable Names:

```
B292704          (contains more than 6 characters)
4ARRAY           (first character is not alphabetic)
SI.X             (contains a special character)
```

VARIABLE TYPES AND LENGTH SPECIFICATIONS

The type of a variable corresponds to the type of data the variable represents. Thus, an integer variable represents integer data, a real variable represents real data, etc.

For every type of variable, a corresponding standard and optional length specification determines the number of storage locations reserved for each variable. The following list shows each variable type with its associated standard and optional length.

Variable Type	Standard	Optional
Integer	4	2
Real	4	8
Complex	8	16
Logical	4	1

The three ways a programmer may declare the type of a variable are by use of the:

1. Predefined specification contained in the FORTRAN language,

2. IMPLICIT specification statement,

3. Explicit specification statements.

The optional-length specification of a variable may be declared only by the IMPLICIT or Explicit specification statements. If, in these statements, no length specification is stated, the standard length is assumed (see "The Type Statements").

TYPE DECLARATION BY THE PREDEFINED SPECIFICATION

The predefined specification is a convention used to specify variables as integer or real:

1. If the first character of the variable name is I, J, K, L, M, or N, the variable is integer of standard length, 4.

2. If the first character of the variable name is any other letter, the variable is real of standard length, 4.

This convention is the traditional FORTRAN method of implicitly specifying the type of a variable as being either integer or real. In all examples that follow, it is presumed that this specification holds, unless otherwise noted.

TYPE DECLARATION BY THE IMPLICIT SPECIFICATION STATEMENT

This statement allows a programmer to specify the type of variables in much the same way as was specified by the predefined convention. That is, in both, the type is determined by the first character of the variable name. However, the programmer, using the IMPLICIT statement, has the option of specifying which initial letters designate a particular variable type. Further, the IMPLICIT statement is applicable to all types of variables -- integer, real, complex, and logical.

The IMPLICIT statement overrides the variable type as determined by the predefined convention. For example, if the IMPLICIT statement specifies that variables beginning with the letters A through M are real variables, and variables beginning with the letters N through Y are integer variables, then the variable ITEM (which would be treated as an integer variable under the predefined convention) is now treated as a real variable. Note that variables beginning with the letters Z and $ are (by the predefined convention) treated as real variables. The IMPLICIT statement is presented in greater detail in the section "Type Statements."

TYPE DECLARATION BY EXPLICIT SPECIFICATION STATEMENTS

Explicit specification statements (INTEGER, REAL, COMPLEX, and LOGICAL) differ from the first two ways of specifying the type of a variable, in that an explicit specification statement declares the type of a particular variable by its name, rather than as a group of variables beginning with a particular character.

For example, assume:

1. That an IMPLICIT specification statement overrode the predefined convention for variables beginning with the letter I, by declaring them to be real.

2. That a subsequent Explicit specification statement declared that the variable named ITEM is complex.

Then, the variable ITEM is complex and all other variables beginning with the character I are real. Note that variables beginning with the letters J through N are specified as integer by the predefined convention.

The Explicit specification statements are discussed in greater detail in the section "Type Statements."

ARRAYS

A FORTRAN array is a set of variables identified by a single variable name. A particular variable in the array may be referred to by its position in the array (e.g., first variable, third variable, seventh variable, etc.). Consider the array named NEXT, which consists of five variables, each currently representing these values:

273, 41, 8976, 59, and 2

```
NEXT(1)        represents 273
NEXT(2)        represents 41
NEXT(3)        represents 8976
NEXT(4)        represents 59
NEXT(5)        represents 2
```

Each variable in this array consists of the name of the array (i.e., NEXT) followed by a number enclosed in parentheses, called a subscript. The variables that constitute the array are called subscripted variables. Therefore, the value of the subscripted variable NEXT(1) is 273; the value of NEXT(2), is 41; etc.

The subscripted variable NEXT(I) refers to the "Ith" subscripted variable in the array, where I is an integer variable that can assume a value of 1, 2, 3, 4, or 5, in this array.

To refer to the first element of an array, the array name must be subscripted; the array name does not represent the first element. The number of subscripts must correspond to the declared dimensionality, except in the EQUIVALENCE statement, which is explained in "EQUIVALENCE Statement."

SUBSCRIPTS

A subscript is an index that specifies one of the coordinates that identify a particular element of an array. From one to seven subscripts are used in accordance with the dimensionality of the array being subscripted. Multidimensional subscripts are separated by commas. The subscripts, enclosed in parentheses, follow the array name.

General Form
Subscripts -- may be one of seven forms: v c' v+c' v-c' c*v c*v+c' c*v-c' where v represents an unsigned, nonsubscripted, integer variable. c and c' represent unsigned integer constants.

Whatever subscript form is used, its evaluated result must always be greater than zero. For example, when reference is made to the subscripted variable V(I-2), the value of I should be greater than 2.

Examples:

```
ARRAY (IHOLD)
NEXT(19)
MATRIX(I-5)
A(5*L)
W(4*M+3)
```

These are valid subscripted variables for their corresponding arrays:

Array Name	Subscripted Variable
A	A(5, 100, J, K+2)
TABLE	TABLE (1, 1, 1, 1, 1, 1, 1)
B	B(I, J, K, L, M, N)
MATRIX	MATRIX(I+2,6*JOB-3,KFRAN)

Consider the following array, named LIST, consisting of two subscript parameters, the first ranging from 1 through 5, the second from 1 through 3.

	Column 1	Column 2	Column 3
Row 1	82	4	7
Row 2	12	13	14
Row 3	91	1	31
Row 4	24	16	10
Row 5	2	8	2

The correct reference for the number in row 2, column 3, would be

```
LIST (2,3)
```

LIST (2,3) has the value 14; LIST (4,1) has the value 24.

Ordinary mathematical notations might use LIST i,j to represent any element of the array LIST. In FORTRAN, this is written as LIST(I,J), where I equals 1, 2, 3, 4, or 5, and J equals 1, 2, or 3.

As a further example, consider the array named COST, consisting of four subscript parameters. This array might be used to store all the premiums for a life insurance applicant, given (1) age, (2) sex, (3) health, and (4) size of life insurance coverage desired. A code number could be developed for each statistic; IAGE represents age; ISEX, sex; IHLTH, health; and ISIZE, policy size desired (see Table 1).

Table 1. Insurance premium codes

AGE	
Age in years	Code
1-5	IAGE=1
6-10	IAGE=2
.	.
.	.
.	.
96-100	IAGE=20

HEALTH	
Health	Code
Poor	IHLTH=1
Fair	IHLTH=2
Good	IHLTH=3
Excellent	IHLTH=4

SEX	
Sex	Code
Male	ISEX=1
Female	ISEX=2

POLICY SIZE	
Dollars	Code
1,000	ISIZE=1
3,000	ISIZE=3
5,000	ISIZE=4
10,000	ISIZE=5
25,000	ISIZE=6
50,000	ISIZE=7
100,000	ISIZE=8

Suppose an applicant is 14 years old, male, in good health, and desires a policy of $25,000. From Table 1, the statistics can be represented by these codes

```
IAGE=3          (11 - 15 years old)
ISEX=1          (male)
IHLTH=3         (good health)
ISIZE=6         ($25,000 policy)
```

Thus, COST (3, 1, 3, 6) represents the premium for a policy, given the statistics above. Note that IAGE can vary from 1 to 20, ISEX from 1 to 2, IHLTH from 1 to 4, and ISIZE from 1 to 8. The number of subscripted variables in the array COST is the number of combinations that can be formed for different ages, sex, health, and policy size available - a total of 20x2x4x8 or 1280. Therefore, up to 1280 different premiums may be stored in the array named COST.

DECLARING THE SIZE OF AN ARRAY

The size of an array is determined by the number of subscript parameters of the array and the maximum value of each subscript. This information must be given for all arrays before using them in a FORTRAN program, so that a storage area of sufficient size may be reserved. Declaration of this information is made by a DIMENSION statement, a COMMON statement, or by one of the Explicit specification statements (INTEGER, REAL, COMPLEX, and LOGICAL); each is discussed in "Specification Statements."

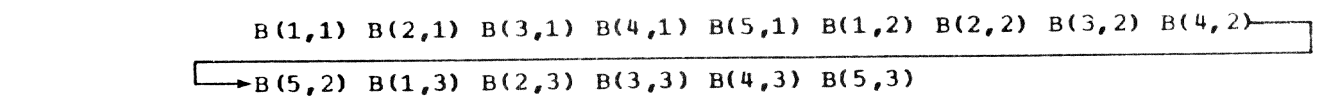
ARRANGEMENT OF ARRAYS IN STORAGE

Arrays are stored in ascending storage locations, with the value of the first of their subscripts increasing most rapidly, and the value of the last increasing least rapidly.

Examples: The array named A, consisting of one subscript parameter, which varies from 1 to 5, appears in storage as

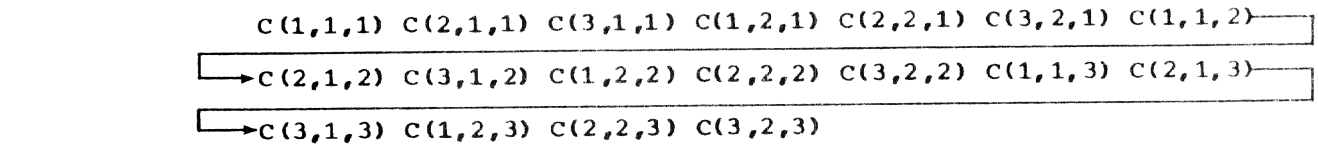
```
A(1) A(2) A(3) A(4) A(5)
```

The array named B consists of two subscript parameters; the first subscript varies over the range from 1 to 5, and the second varies from 1 to 3. The array appears in ascending storage locations in this order



Note that B(1,2) and B(1,3) follow in storage B(5,1) and B(5,2), respectively.

The following list is the order of an array named C that consists of three subscript parameters; the first subscript varies from 1 to 3; the second, from 1 to 2; and the third, from 1 to 3.



Note that C(1,1,2) and C(1,1,3) follow in storage C(3,2,1) and C(3,2,2), respectively.

EXPRESSIONS

Expressions in their simplest form consist of a single constant or variable. They may also designate a computation or show a relationship between two or more constants and/or variables. Expressions may appear in arithmetic and logical assignment statements and in certain control statements.

FORTRAN provides two kinds of expressions: arithmetic and logical. The value of an arithmetic expression is always a number whose type is integer, real, or complex. However, the evaluation of a logical expression always yields a truth value: .TRUE. or .FALSE..

ARITHMETIC EXPRESSIONS

The simplest arithmetic expression consists of a single constant, variable, or subscripted variable of the type integer, real, or complex. If the constant, variable, or subscripted variable is of the type integer, the expression is in the integer mode. If it is of the type real, the expression is in the real mode, etc.

Examples:

Expression	Type of Quantity	Mode of Expression
3	Integer Constant	Integer of length 4
I	Integer Variable	Integer of length 4
3.0	Real Constant	Real of length 4
A	Real Variable	Real of length 4
3.14D3	Real Constant	Real of length 8
B(2*I)	Real Variable	Real of length 4
(2.0,5.7)	Complex Constant	Complex of length 8
C	Complex Variable (Specified as such in a Type statement)	Complex of length 8

In the expression B(2*I), the subscript (2*I), which must always represent an integer, does not affect the mode of the expression. That is, the mode of the expression is determined solely by the type of constant, variable, or subscripted variable appearing in that expression.

More complicated arithmetic expressions containing two or more constants and/or variables may be formed by using arithmetic operators that express the computations to be performed.

Arithmetic Operators

The arithmetic operators are:

Arithmetic Operator	Definition
**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction

RULES FOR CONSTRUCTING ARITHMETIC EXPRESSIONS: These are the rules for constructing arithmetic expressions that contain arithmetic operators:

1. All desired computations must be specified explicitly. That is, if more than one constant, variable, subscripted variable, or function reference (see "SUBPROGRAMS") appears in an arithmetic expression, they must be separated from one another by an arithmetic operator. For example, the two variables A and B will not be multiplied if written as

   AxB or AB or A•B

   If multiplication is desired, then the expression must be written as

   A*B or B*A

2. No two arithmetic operators may appear in sequence in the same expression. For example, these expressions are invalid

   A*/B and A*-B

   However, in the expression, A*-B, if the - is meant to be a minus sign rather than the arithmetic operator designating subtraction, then the expression could be written as

   A*(-B)

   In effect, -B will be evaluated first, and then A will be multiplied with it (for further uses of parentheses, see Rule 6).

3. The mode of an arithmetic expression is determined by the type of the operands (where an operand is a variable, constant, function reference, or another expression) in the expression. Table 2 indicates how the mode of an expression that contains operands of different types may be determined using the operators: +, -, *. /.

   Table 2 shows a hierarchy of type and length specification (see "Type Statements") that determines the mode of an expression. For example, complex data that has a length specification of 16, when combined with any other types of constants and variables, results in complex data of length 16.

Table 2. Determining the mode of an expression containing variables of different types and lengths

+ - * /	INTEGER (2)	INTEGER (4)	REAL (4)	REAL (8)	COMPLEX (8)	COMPLEX (16)
INTEGER (2)	Integer (2)	Integer (4)	Real (4)	Real (8)	Complex (8)	Complex (16)
INTEGER (4)	Integer (4)	Integer (4)	Real (4)	Real (8)	Complex (8)	Complex (16)
REAL (4)	Real (4)	Real (4)	Real (4)	Real (8)	Complex (8)	Complex (16)
REAL (8)	Real (8)	Real (8)	Real (8)	Real (8)	Complex (16)	Complex (16)
COMPLEX (8)	Complex (8)	Complex (8)	Complex (8)	Complex (16)	Complex (8)	Complex (16)
COMPLEX (16)	Complex (16)	Complex (16)	Complex (16)	Complex (16)	Complex (16)	Complex (16)

Assume that the type of the following variables has been specified as

Variable Names	Type	Length Specification
ROOT, E	Real variables	4,8
A, I, F	Integer variables	4,2,2
C,D	Complex variables	16,8

Then, the following examples illustrate how constants and variables of differing types may be combined using the arithmetic operators (+, -, /, *)

Expression	Mode of Expression
ROOT*5	Real of length 4
A+3	Integer of length 4
C+2.9D10	Complex of length 16
E/F+19	Real of length 8
C-18.7E05	Complex of length 16
A/I-D	Complex of length 8

4. The arithmetic operator denoting exponentiation (i.e.,**) may only be used to combine the types of constants, variables, and subscripted variables shown in Table 3.

Table 3. Valid combinations using the arithmetic operator **

Base		Exponent
Integer or Real (either length)	**	Integer or Real (either length)
Complex (either length)	**	Integer (either length)

Assume that the types of the following variables are as specified, and that their length specification is standard.

Variable Names	Type
ROOT,E	Real variables
A, I, F	Integer variables
C	Complex variables

Then the following examples illustrate how constants and variables of differing types may be combined using the arithmetic operator **.

Examples:

Expression	Type	Result
ROOT**(A+2)	(Real**Integer)	(Real)
C**A	(Complex**Integer)	(Complex)
ROOT**I	(Real**Integer)	(Real)
I**F	(Integer**Integer)	(Integer)
7.98E21**ROOT	(Real**Real)	(Real)
ROOT**2.1E5	(Real**Real)	(Real)
A**E	(Integer**Real)	(Real)

5. Order of Computation: Where parentheses are omitted, or where the entire arithmetic expression is enclosed within a single pair of parentheses, the order in which the operations are performed is as follows:

Operation	Hierarchy
Evaluation of Functions (see "Subprograms")	1st (highest)
Exponentiation (**)	2nd
Multiplication and Division (* and /)	3rd
Addition and Subtraction (+ and -)	4th

In addition, if two operators of the same hierarchy (with the exception of exponentiation) are used consecutively, the component operations of the expression are performed from left to right. Thus the arithmetic expression A/B*C is evaluated as if the result of the division of A by B was multiplied by C.

For example, the expression

```
(A*B/C**I+D)
```

is evaluated in this order

a.	C**I	Call the result X	(exponentiation)
b.	A*B	Call the result Y	(multiplication)
c.	Y/X	Call the result Z	(division)
d.	Z+D	Final operation	(addition)

For exponentiation the evaluation is from right to left. Thus, the expression

```
A**B**C
```

is evaluated as

a.	B**C	Call the result Z
b.	A**Z	Final operation

6. Use of Parentheses: Parentheses may be used in arithmetic expressions, as in algebra, to specify the order in which the arithmetic operations are to be performed. Where parentheses are used, the expression within the parentheses is evaluated before the result is used.

For example, the expression

```
(B+((A+B)*C)+A**2)
```

is evaluated in this order

a. (A+B) Call the result X
b. (X*C) Call the result Y
c. A**2 Call the result Z
d. B+Y+Z Final operations

7. Integer Division: When division is performed using two integers, the answer is truncated and an integer answer is given. For example, if I=9 and J=2, then the expression (I/J) would yield an integer answer of 4 after truncation.

LOGICAL EXPRESSIONS

The simplest form of logical expression consists of a single logical constant, logical variable, or logical subscripted variable, the value of which is always a truth value (i.e., either .TRUE. or .FALSE.).

More complicated logical expressions may be formed by using logical and relational operators. These expressions may be in one of three forms.

1. Relational operators combined with arithmetic expressions whose mode is integer or real.

2. Logical operators combined with logical constants (.TRUE. and .FALSE.), logical variables, or subscripted variables.

3. Logical operators combined with either or both forms of the logical expressions described in items 1 and 2.

Item 1 is discussed in the following section, "Relational Operators"; items 2 and 3 are discussed in "Logical Operators."

Relational Operators

The six relational operators, each of which must be preceded and followed by a period, are:

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

The relational operators express an arithmetic condition which can be either true or false. Only arithmetic expressions whose mode is integer or real may be combined by relational operators. For example, assume that the types of these variables have been specified as

Variable Names	Type
ROOT, E	Real variables
A, I, F	Integer variables
L	Logical variable
C	Complex variable

Then, the following examples illustrate valid and invalid logical expressions using the relational operators.

Examples: Valid Logical Expressions Using Relational Operators:

```
(ROOT*A).GT.E
A.LT.I
E**2.7.EQ.(5*ROOT+4)
57.9.LE.(4.7+F)
.5.GE..9*ROOT
E.EQ.27.3D+05
```

Invalid Logical Expressions Using Relational Operators:

C.LT.ROOT	(Complex quantities may never appear in logical expressions)
C.GE.(2.7,5.9E3)	(Complex quantities may never appear in logical expressions)
L.EQ.(A+F)	(Logical quantities may never be joined by relational operators)
E**2.EQ97.1E9	(Missing period immediately after relational operator)
.GT.9	(Missing arithmetic expression before relational operator)

Logical Operators

The three logical operators, each of which must be preceded and followed by a period, are as follows. (A and B represent logical constants or variables, or expressions containing relational operators).

Logical Operator	Definition
.NOT.	.NOT.A - if A is .TRUE., then .NOT.A has the value .FALSE.; if A is .FALSE., then .NOT.A has the value .TRUE.
.AND.	A.AND.B - if A and B are both .TRUE., then A.AND.B has the value .TRUE.; if either A or B or both are .FALSE., then A.AND.B has the value .FALSE.
.OR.	A.OR.B - if either A or B or both are .TRUE., then A.OR.B has the value .TRUE.; if both A and B are .FALSE., then A.OR.B has the value .FALSE.

Two logical operators may appear in sequence only if the second one is the logical operator .NOT..

Only those expressions which, when evaluated, have the value .TRUE. or .FALSE. may be combined with the logical operators to form logical expressions. For example, assume that the types of these variables are as specified.

Variable Names	Type
ROOT, E	Real variables
A, I, F	Integer variables
L, W	Logical variables
C	Complex variable

Then, the following examples illustrate valid and invalid logical expressions using both logical and relational operators.

Examples: Valid Logical Expressions:

```
(ROOT*A.GT.A).AND.W
L.AND..NOT.(I.GT.F)
(E+5.9D2.GT.2*E).OR.L
.NOT.W.AND..NOT.L
L.AND..NOT.W.OR.I.GT.F
(A**F.GT.ROOT).AND..NOT.(I.EQ.E)
```

Invalid Logical Expressions:

A.AND.L	(A is not a logical expression)
.OR.W	(.OR. must be preceded by a logical expression)
NOT.(A.GT.F)	(missing period before the logical operator .NOT.)

(C.EQ.I).AND.L	(a complex variable may never appear in a logical expression)
L.AND..OR.W	(the logical operators .AND. and .OR. must always be separated by a logical expression)
.AND.L	(.AND. must be preceded by a logical expression)

Order of Computations in Logical Expressions: Where parentheses are omitted, or where the entire logical expression is enclosed within a single pair of parentheses, this is the order in which the operations are performed.

Operation	Hierarchy
Evaluation of Functions	1st (highest)
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th
.LT.,.LE.,.EQ.,.NE.,.GT.,.GE.	5th
.NOT.	6th
.AND.	7th
.OR.	8th

For example, the expression

```
(A.GT.D**B.AND..NOT.L.OR.N)
```

is evaluated in this order

1.	D**B	Call the result W	(exponentiation)
2.	A.GT.W	Call the result X	(relational operator)
3.	.NOT.L	Call the result Y	(highest logical operator)
4.	X.AND.Y	Call the result Z	(second highest logical operator)
5.	Z.OR.N	Final operation	

Use of Parentheses in Logical Expressions: Parentheses may be used in logical expressions to specify the order in which the operations are to be performed. Where parentheses are used, the expression contained within the innermost pair of parentheses is evaluated first. For example, the logical expression

```
((I.GT.(B+C)).AND.L)
```

is evaluated in this order

1.	B+C	Call the result X
2.	I.GT.X	Call the result Y
3.	Y.AND.L	Final operation

The logical expression to which the logical operator .NOT. applies must be enclosed in parentheses if it contains two or more quantities. For example, assume that the values of the logical variables A and B are .FALSE. and .TRUE., respectively. Then these two expressions are not equivalent

```
.NOT.(A.OR.B)
.NOT.A.OR.B
```

In the first expression, A.OR.B is evaluated first. The result is .TRUE.; but .NOT.(.TRUE.) implies .FALSE.. Therefore, the value of the first expression is .FALSE..

In the second expression, .NOT.A is evaluated first. The result is .TRUE.; but .TRUE..OR.B implies .TRUE.. Therefore, the value of the second expression is .TRUE..

ARITHMETIC AND LOGICAL ASSIGNMENT STATEMENT

General Form
a = b where a is any subscripted or nonsubscripted variable b is any arithmetic or logical expression Note: a must be a logical variable if, and only if, b is a logical expression.

The FORTRAN arithmetic and logical assignment statement closely resembles a conventional algebraic equation; however, the equal sign of the FORTRAN arithmetic statement specifies replacement rather than equivalence. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable to the left of the equal sign.

Assume that the type of the following variables has been specified as:

Variable Names	Type	Length Specification
I, J, W	Integer variables	4,4,2
A, B, C, D	Real variables	4,4,8,8
E	Complex variable	8
G, H	Logical variables	4,4

Then, the following examples illustrate valid arithmetic statements using constants, variables, and subscripted variables of different types.

Statements	Description
A = B	The value of A is replaced by the current value of B.
W = B	The value of B is converted to an integer value and the least significant part replaces the value of W.
A = I	The value of I is converted to a real value and this result replaces the value of A.
I = I + 1	The value of I is replaced by the value of I + 1.
E = I**J+D	I is raised to the power J and the result is converted to a real value, to which the value of D is added. This result replaces the real part of the complex variable E. The imaginary part of the complex variable is set to zero.
A = C*D	The most significant part of the product of C and D replaces the value of A.
G = .TRUE.	The value of G is replaced by the logical constant .TRUE..
H = .NOT.G	If G is .TRUE., the value of H is replaced by the logical constant .FALSE.. If G is .FALSE., the value of H is replaced by the logical constant .TRUE..

`G = 3..GT.I`	The value of I is converted to a real value; if the real constant 3. is greater than this result, the logical constant .TRUE. replaces the value of G. If 3. is not greater than I, the logical constant .FALSE. replaces the value of G.
`E = (1.0,2.0)`	The value of the complex variable E is replaced by the complex constant (1.0,2.0). Note that the statement E = (A,B), where A and B are real variables, is invalid.
`A = E`	The real part of the complex variable E replaces the value of A.
`E = A`	The value of A replaces the value of the real part of the complex variable E; the imaginary part is set equal to zero.

CONTROL STATEMENTS

Normally, FORTRAN statements are executed sequentially; that is, after one statement has been executed, the statement immediately following it will be executed. This section discusses the statements that may be used to alter and control the normal sequence of statement execution in the program.

GO TO STATEMENTS

The GO TO statements cause control to be transferred to the statement specified by a statement number. The three GO TO statements are: unconditional GO TO, computed GO TO, and assigned GO TO. Every time the same unconditional GO TO statement is executed, a transfer to the same specified statement is made. However, the computed and assigned GO TO statements cause control to be transferred to one of several statements, depending upon the current value of a particular variable.

Unconditional GO TO Statement

General Form
GO TO X where X is an executable statement number

This GO TO statement causes control to be transferred to the statement specified by the statement number. Every subsequent execution of this GO TO statement results in a transfer to that same statement.

Example:

```
    50 GO TO 25
    10 A = B + C
          .
          .
          .
    25 C = E**2
          .
          .
          .
```

Explanation: Every time statement 50 is executed, control is transferred to statement 25.

Computed GO TO Statement

General Form
GO TO (x_1, x_2, x_3, ..., x_n), i where $x_1, x_2 \ldots, x_n$, are executable statement numbers i is a nonsubscripted integer variable in the range: $1 \geq i \geq n$

This statement causes control to be transferred to the statement numbered x_1, x_2, x_3,..., or x_n, depending on whether the current value of i is 1, 2, 3,..., or n, respectively. If the value of i is outside the allowable range, the next statement is executed.

Example:

```
      GO TO (25, 10, 50, 7), ITEM
          .
          .
          .
50    A = B+C
          .
          .
          .
7     C = E**2+A
          .
          .
          .
25    L = C.GT.D.AND.F.LE.G
          .
          .
          .
10    B = 21.3E02
```

Explanation: If the value of the integer variable ITEM is 1, statement 25 will be executed next. If ITEM is equal to 2, statement 10 will be executed next, and so on.

ASSIGN and Assigned GO TO Statements

General Form
ASSIGN i TO m . . . GO TO m, (x_1,x_2,x_3,...,x_n) where i is an executable statement number x_1,x_2,x_3,...,x_n are executable statement numbers m is a nonsubscripted integer variable of length 4, to which is assigned one of these statement numbers: x_1,x_2,x_3,...x_n.

The assigned GO TO statement causes control to be transferred to the statement numbered x_1,x_2,x_3,...,or x_n, depending on whether the current assignment of m is x_1,x_2,x_3,...,or x_n. For example, in the statement

```
GO TO N, (10, 25, 8)
```

if the current assignment of the integer variable N is statement 8, that statement is executed next. If the current assignment of N is statement 10, that statement is executed next. Similarly, if N is assigned statement number 25, that statement is executed next.

The current assignment of the integer variable m is determined by the last ASSIGN statement executed. Only an ASSIGN statement may be used to initialize or change the value of the integer variable m. The

value of the integer variable m is not the integer statement number; ASSIGN 10 TO I is not the same as I=10.

Example 1:

```
          .
          .
          .
       ASSIGN 50 TO NUMBER
    10 GO TO NUMBER, (35, 50, 25, 12, 18)
          .
          .
          .
    50 A = B + C
          .
          .
          .
```

Explanation: Statement 50 is executed immediately after statement 10.

Example 2:

```
          .
          .
       ASSIGN 10 TO ITEM
          .
          .
    13 GO TO ITEM, (8, 12, 25, 50, 10)
          .
          .
     8 A = B + C
          .
          .
    10 B = C + D
       ASSIGN 25 TO ITEM
       GO TO 13
          .
          .
    25 C = E**2
          .
          .
```

Explanation: The first time statement 13 is executed, control is transferred to statement 10. On the second execution of statement 13, control is transferred to statement 25.

Arithmetic IF Statement

General Form
IF (a) x_1, x_2, x_3
where a is an arithmetic expression which is not complex
x_1, x_2, x_3 are statement numbers

This statement causes control to be transferred to the statement numbered x_1, x_2, x_3 when the value of the arithmetic expression a is less than, equal to, or greater than zero, respectively. The first executable statement following the arithmetic IF statement should have a statement number; otherwise, it can never be referred to or executed.

Example:

```
      .
      .
IF (A(J,K)**3-B)10, 4, 30
      .
      .
4 D = B + C
      .
      .
30 C = D**2
      .
      .
10 E = (F*B)/D+1
      .
      .
```

Explanation: If the value of the expression (A(J,K)**3-B) is negative, statement 10 is executed next. If the value of the expression is zero, statement 4 is executed next. If the value of the expression is positive, statement 30 is executed next.

Logical IF Statement

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

The logical IF statement is used to evaluate the logical expression a and to execute or skip statement s, depending on whether the value of the expression is .TRUE. or .FALSE., respectively.

Example 1:

```
      .
      .
      .
5   IF(A.LE.0.0) GO TO 25
10  C = D + E
15  IF(A.EQ.B) ANSWER = 2.0*A/C
20  F = G/H
      .
      .
      .
25  W = X**Z
      .
      .
      .
```

Explanation: In statement 5, if the value of the expression is .TRUE. (i.e., A is less than or equal to 0.0), the statement GO TO 25 is executed next, and control is passed to statement 25. If the value of the expression is .FALSE. (i.e., A is greater than 0.0), the statement GO TO 25 is ignored, and control is passed to statement 10.

In statement 15, if the value of the expression is .TRUE. (i.e., A is equal to B), the value of ANSWER is replaced by the value of the expression (2.0*A/C), and statement 20 is executed. If the value of the expression is .FALSE. (i.e., A is not equal to B), the value of ANSWER remains unchanged, and statement 20 is executed next.

Example 2: Assume that P and Q are logical variables.

```
      .
      .
      .
    5 IF(P.OR..NOT.Q)A=B
   10 C = B**2
      .
      .
      .
```

Explanation: In statement 5, if the value of the expression is .TRUE., the statement A=B is executed next and control continues to statement 10. If the value of the expression is .FALSE., the statement A=B is skipped and statement 10 is executed.

DO Statement

General Form					
	End of range	DO variable	Initial value	Test value	Increment
DO	x	i	= m_1,	m_2,	m_3

where x is the statement number of an executable statement that follows the DO statement

i is a nonsubscripted integer variable

m_1, m_2, m_3, are either unsigned integer constants greater than zero or unsigned nonsubscripted integer variables whose values are greater than zero. The sum m_2+m_3+1 must not exceed the size of virtual storage. (m_3, is optional; if it is omitted, its value is assumed to be 1. In this case, the preceding comma must also be omitted.)

The DO statement is a command to execute repeatedly the statements that follow, up to and including the statement numbered x. 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 statemment. The first time the statements in the range of the DO are executed, i is initialized to the value m_1; each succeeding time i is increased by the value m_3. When, at the end of an iteration, i is equal to the highest value that does not exceed m_2, control passes to the statement following the statement numbered x. Thus, the number of times the statements in the range of the DO is executed is given by the expression:

$$\left[\frac{m_2 - m_1}{m_3}\right] + 1$$

where the brackets represent the largest integral value not exceeding the value of the expression. If m_2 is less than m_1, the statements in the range of the DO are executed once. Upon completion of the DO, the DO variable is undefined.

There are several ways in which looping (repetitively executing the same statements) may be accomplished when using the FORTRAN language. For example, assume that a manufacturer carries 1000 different machine parts in stock. Periodically, he may find it necessary to compute the

amount of each different part that is presently available. This amount may be calculated by subtracting the number of each item used, OUT(I), from the previous stock on hand, STOCK(I).

Example 1:

```
            .
            .
            .
5     I=0
10    I=I+1
25    STOCK(I)=STOCK(I)- OUT(I)
15    IF(I-1000) 10,30,30
30    A=B+C
            .
            .
            .
```

Explanation: The three statements (5, 10, and 15) required to control the loop could be replaced by a single DO statement, as shown in Example 2.

Example 2:

```
            .
            .
            .
      DO 25 I = 1,1000
25    STOCK(I) = STOCK(I)-OUT(I)
30    A=B+C
            .
            .
            .
```

Explanation: The DO variable, I, is set to the initial value of 1. Before the second execution of statement 25, I is increased by the increment 1 and statement 25 is again executed. After 1000 executions of the DO loop, I equals 1000. Since I is now equal to the highest value that does not exceed the test value, 1000, control passes out of the DO loop, and statement 30 is executed next. Note that the DO variable I is now undefined; its value is not necessarily 1000 or 1001.

Example 3:

```
            .
            .
            .
      DO 25 I=1, 10, 2
15    J=I+K
25    ARRAY(J) = BRAY(J)
30    A=B+C
            .
            .
            .
```

Explanation: Statement 25 is the end of the range of the DO loop. The DO variable, I, is set to the initial value of 1. Before the second execution of the DO loop, I is increased by the increment 2, and statements 15 and 25 are executed a second time. After the fifth execution of the DO loop, I equals 9. Since I is now equal to the highest value that does not exceed the test value, 10, control passes out of the DO loop, and statement 30 is executed next. Note that the DO variable I is now undefined; its value is not necessarily 9 or 11.

Programming Considerations in Using a DO Loop

1. The indexing parameters of a DO statment (i, m_1, m_2, m_3) may not be changed by a statement within the range of the DO loop, or by any subprograms that are called within the range of a DO loop.

2. A DO statement may contain other DO statements within its range. All statements in the range of an inner DO must be in the range of the outer DO. A set of DO statements satisfying this rule is called a nest of DOs.

   Example 1:

```
      DO 50 I = 1, 4
      A(I) = B(I)**2
      DO 50 J=1, 5
   50 C(J+1) = A(I)
```

   Range of inner DO; Range of outer DO

   Example 2:

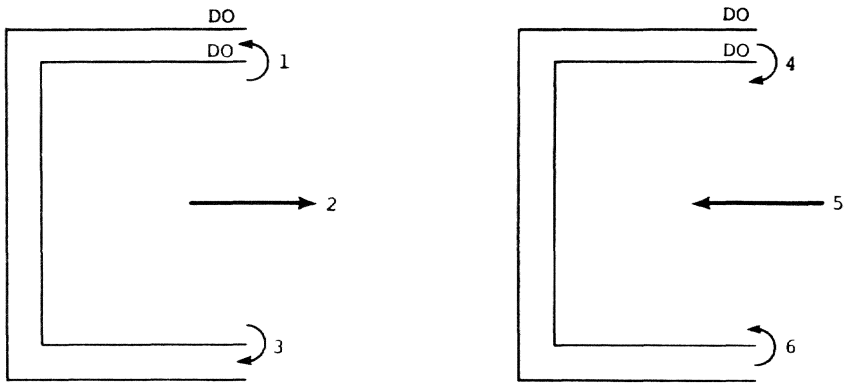
```
      DO 10 INDEX = L, M
      N = INDEX + K
      DO 15 J = 1, 100, 2
   15 TABLE(J) = SUM(J,N)-1
   10 B(N) = A(N)
```

   Range of inner DO; Range of outer DO

3. A transfer out of the range of any DO loop is permissible at any time.

4. If, and only if, a transfer is made from the range of an innermost DO loop, transfer back into that loop is allowed, provided none of the indexing parameters (i, m_1, m_2, m_3) are changed outside the range. A transfer back into the range of any other DO within a nest of DOs is not permitted.

   Example:



   Explanation: The transfers specified in the example by the numbers 1, 2, and 3 are permissible; those specified by 4, 5, and 6 are not.

5. The indexing parameters (i,m_1,m_2,m_3) may be changed by statements outside the range of the DO statement only if no transfer is made back into the range of the DO statement using those parameters.

6. The last statement in the range of a DO loop (statement x) may not be a GO TO, arithmetic IF, PAUSE, STOP, RETURN, or another DO statement. Also, the last statement may not be a logical IF statement containing any of those statements.

CONTINUE Statement

General Form
CONTINUE

CONTINUE is a dummy statement which may be placed anywhere in the source program without affecting the sequence of execution. It may be used as the last statement in the range of a DO statement to avoid ending the DO loop with any of the statements that are not permitted as the last statement in the range of a DO.

Example 1:

```
          .
          .
          .
       DO 30 I = 1, 20
    7  IF  (A(I)-B(I)) 5,30,30
    5  A(I) =A(I) +1.0
       B(I) = B(I) -2.0
       GO TO 7
   30 CONTINUE
   40 C = A(3) + B(7)
          .
          .
          .
```

Explanation: The CONTINUE statement is used as the last statement in the range of the DO statement, to avoid ending the DO loop with the statement GO TO 7.

Example 2:

```
          .
          .
          .
       DO 30 I=1,20
       IF(A(I)-B(I))5,40,40
    5  A(I) = C(I)
       GO TO 30
   40  A(I) = 0.0
   30 CONTINUE
          .
          .
          .
```

Explanation: The CONTINUE statement provides a branch point that enables the programmer to bypass the execution of statement 40.

PAUSE Statement

General Form
PAUSE PAUSE n PAUSE 'message' where n is an unsigned 1-through 5-digit integer constant message is any literal constant

A PAUSE statement executed in a program results in a message being written as follows:

PAUSE Statement	Resulting Message
PAUSE	PAUSE 00000
PAUSE n	PAUSE 1-5 digit integer
PAUSE 'message'	PAUSE text of message

In nonconversational mode, the message is written on the standard system output data set and the pause is ignored, so the program continues execution at the next executable statement. In converstaional mode of execution, the pause message is written at the user's terminal and the program waits until the user resumes execution via the TSS command system.

STOP Statement

General Form
STOP STOP n where n is an unsigned 1-through 5-digit integer constant

This statement terminates the execution of the object program; message will be displayed as follows:

STOP Statement	Message
STOP	STOP
STOP n	STOP 1-5 digit integer

END Statement

General Form
END

The END statement is a nonexecutable statement that defines the end of a source program or source subprogram for the compiler. Physically, it must be the last statement of each program or subprogram.

The END statement must be contained on a single line.

The input/output statements enable a user to transfer data, belonging to a named collection of data, between input/output devices and internal storage. The named collection of data, called a data set, is a continuous string of data that may be divided into FORTRAN records.

A data set is referred to by an integer constant or integer variable, called the data set reference number.

There are five I/O statements: READ, WRITE, END FILE, REWIND, and BACKSPACE. The READ and WRITE statements cause transfer of records from and to data sets and internal storage. The END FILE statement defines the end of a data set; the REWIND and BACKSPACE statements control the positioning of data sets.

In addition to these five statements, the FORMAT and NAMELIST statements, although they are not I/O statements, are used with certain forms of the READ and WRITE statements. The FORMAT statement specifies the form in which the data is to be transmitted; the NAMELIST statement specifies a list of variables or array names to be used in an input/output operation. Also, both statements allow the user to divide a data set into FORTRAN records.

Even though the I/O statements are device independent, the source or the destination of the data being transferred influences the specification of the records and data formats. Therefore, subsequent examples are in terms of card input and print-line output, unless otherwise noted.

READ STATEMENT

General Form
READ (a, b, END=c, ERR=d) list where a is an unsigned integer constant or an integer variable of length 4 that represents a data set reference number. b is either the statement number or array name of the FORMAT statement describing the data being read, or a NAMELIST name. c is the statement number to which transfer is made upon encountering the end of the data set. d is the statement number of the statement to which transfer is made upon encountering an error condition in data transfer. list is a series of variable or array names, separated by commas, which may be indexed and incremented; they specify the number of items to be read and the storage locations into which the data is placed.

The READ statement may take many different forms. For example, the parameters END=c and ERR=d are optional and, therefore, may or may not

appear in a READ statement. Furthermore, the parameter list, the parameter b, or both, may be omitted.

When one or more of the parameters END=c or ERR=d is used after the a and b portion of a READ statement, they may appear in any order within the parentheses. For example, these are valid READ statements

```
READ(5,50,ERR=10)A,B,C
READ(5,25, END=15) D,E,F,F,H
READ(N,30, ERR=100, END=8) X,Y,Z
```

If a transfer is made to a statement specified by the END parameter, no indication is given the program as to the number of items in the list read (if any) before encountering the end of the data set. If an END parameter is not specified in a READ statement, the end of the data set terminates execution of the object program.

If a transfer is made to a statement specified by the ERR parameter, no data is read into the list items associated with the record in error. No indication is given to the program as to which input record or records are in error; only that an error occurred during transmission of data to fill the READ list. If an ERR parameter is not specified in a READ statement, an error terminates execution of the object program.

The three basic forms of the READ statement are

```
READ (a,x)
READ(a,b)list
READ(a)list
```

The parameters END=c and ERR=d may be used, in the combination described above, in each of these three forms.

READ (a,x)

This form is used to read data from the data set associated with a into the locations in storage specified by the NAMELIST name x. The NAMELIST name x is a single variable name that refers to a specific list of variables or array names into which the data is placed. A specific list of variable or array names receives a NAMELIST name by use of a NAMELIST statement. The programmer need only use the NAMELIST name in the READ (a,x) statement to reference that list thereafter in the program.

The format and rules for constructing and using the NAMELIST statement are described below.

General Form
NAMELIST/x/a,b,...,c/y/d,e,...,f/z/g,h,...,i
where x,y, and z,... are NAMELIST names
a,b,c,d,... are variable or array names

The following rules apply to defining and using a NAMELIST name:

1. A NAMELIST name consists of from 1 through 6 alphameric characters, the first of which is alphabetic.

2. A NAMELIST name is enclosed in slashes. The list of variable or array names belonging to a NAMELIST name ends with a new NAMELIST name enclosed in slashes or with the end of the NAMELIST statement.

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

4. A NAMELIST name may be defined only once by its appearance in a NAMELIST statement and must be so defined before its use. the NAMELIST name may appear only in READ or WRITE statements in the program.

5. A NAMELIST statement may appear anywhere in a FORTRAN program prior to its use in a READ/WRITE statement.

6. Variable or array names appearing anywhere in a NAMELIST statement or NAMELIST name may not appear in a FUNCTION, SUBROUTINE, or ENTRY statement.

Example: Assume that A, I, and L are array names.

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

Explanation: The READ statement causes the record that contains the input data for the variables and arrays that belong to the NAMELIST name, NAM1, to be read from the data set associated with the data set reference number 5.

Input Data

When a READ statement refers to a NAMELIST name, input data in the form described below is read from the designated input data set.

The first character in the record must always be blank. The second character of the first record of a group of data records to be read must be & (ampersand), immediately followed by the NAMELIST name. The NAMELIST name must be followed by a blank and must not contain embedded blanks. This name is followed by any combination of data items 1 and 2 below, separated by commas. (A comma after the last item is optional.) The end of a data group is signaled by &END.

The form the data items may take is

1. Variable name = constant

   The variable name may be a subscripted variable name or a single variable name, subscripts must be integer constants.

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

   The set of constants may be in the form

   k* constant

   where k is an unsigned integer called the repeat constant. It represents the number of successive elements in the array to be initialized by the specified constant. The number of constants must not be greater than the number of elements in the array.

Input constants may also be Hollerith (H format) or hexadecimal (Z format) data. The H format is used as in FORMAT statements. The repeat constant may not be used with the H-format option. The size of the character string should not exceed the size of an element. To use the Z format, prefix the hexadecimal characters to be read with a "Z".

Constants used in the data items may be integer, real, literal, complex, or logical data. If the constants are logical data, they may be in the form T or .TRUE. and F or .FALSE..

Any selected set of variable or array names belonging to the NAMELIST name appearing on the first record may be used as specified by items 1 and 2 in the preceding text. Names that are made equivalent to these names may not be used unless they also belong to the NAMELIST name.

The end of a group of data is signaled by the character string &END, with no embedded blanks and all appearing in the same record.

Blanks must not be embedded in a constant or repeat constant, but may be used freely elsewhere in a data record. The last item on each record that contains data items must be a constant followed by a comma. (The comma is optional on the record that precedes the record containing &END.) Trailing blanks after integers and exponents are treated as zeros.

Example: Assume that L is an array consisting of one subscript parameter ranging from 1 to 10.

```
                          Column 2
                          .
First data card:          &NAM1 I(2,3)=5, J=4,
      .                     .
      .                     .
      .                     .
Last data card:           A(3)=4.0, L=2,3,8*4,&END
```

Explanation: If this data is input to be used with the NAMELIST and READ statements previously illustrated, the following actions take place: The first data card is read and examined to verify that the name is consistent with the NAMELIST name in the READ statement. If the name does not match the NAMELIST name, the next NAMELIST group is read. When the data card is read, the integer constants 5 and 4 are placed in I(2,3) and J, respectively; the real constant 4.0 is placed in A(3). Since L is an array not followed by a subscript, the entire array is filled with the succeeding constants. Therefore, the integer constants 2 and 3 is placed in L(1) and L(2), respectively, and the integer constant 4 is placed in L(3), L(4),...,L(10).

READ (a,b) List

This form is used to read data from the data set associated with a into the storage locations specified by the variable names in the list. The list, used in conjunction with the specified FORMAT statement b (see "FORMAT statement"), determines the number of items (data) to be read, the locations, and the form the data will take in storage.

Example 1: Assume that the variables A, B, and C have been declared as integer variables.

```
          .
          .
          .
    75 FORMAT (G10, G8, G9)
          .
          .
          .
```

```
      READ (J, 75) A, B, C
         .
         .
         .
```

Explanation: The READ statement above causes input data from the data set associated with data set reference number J to be read into locations A, B, and C, according to the FORMAT statement referenced (statement 75). That is, the first 10 characters of the record are read, converted to internal form, and stored into A, the next 8 characters into B, and the next 9 characters into C.

The list can be omitted from the READ (a,b)list statement. In this case, a record is skipped or data is read from the data set associated with a, into the storage locations occupied by FORMAT statement b.

Example 2:

```
         .
         .
         .
   98 FORMAT ('HEADING')
         .
         .
         .
      READ (5,98)
         .
         .
         .
```

Explanation: The statements above would cause the characters H, E, A, D, I, N, and G, in storage, to be replaced by the next 7 characters in the data set associated with data set reference number 5.

Example 3:

```
         .
         .
         .
   98 FORMAT (G10,'HEADING')
         .
         .
         .
      READ (5,98)
         .
         .
         .
```

Explanation: The statements above would cause the next record in the data set associated with data set reference number 5 to be skipped. No data is transferred into internal storage because there is no list item that corresponds with format code G10.

READ (a) List

The form READ (a) list of the READ statement causes binary data (internal form) to be read from the data set associated with a into the storage locations specified by the variable names in the list. Since the input data is always in internal form, a FORMAT statement is not required. This statement is used to retrieve the data written by a WRITE (a) list statement.

Example:

```
   READ (5) A, B, C
```

Explanation: This statement causes the binary data from the data set associated with data set reference number 5 to be read into the storage locations specified by the variable names A, P, and C.

The list may be omitted from the READ (a) list statement; in this case, a record is skipped.

Example:

```
READ (5)
```

Explanation: The statements above would cause the next record in the data set associated with data set reference number 5 to be skipped. No data is transferred into internal storage.

Indexing I/O Lists

Variables within an I/O list may be indexed and incremented in the same manner as those within a DO statement. These variables and their indexes must be included in parentheses. For example, suppose it is desired to read data into the first five positions of array A. This may be accomplished by using an indexed list:

```
15 FORMAT (G10.3)
      .
      .
      .
   READ (2,15) (A(I),I=1,5)
```

This is equivalent to

```
15 FORMAT (G10.3)
      .
      .
      .
   DO 12 I = 1,5
12 READ (2,15) A(I)
```

As with DO statements, a third indexing parameter may be used to specify the amount by which the index is to be incremented at each iteration. Thus,

```
READ (2,15) (A(I), I=1,10,2)
```

causes transmission of values for A(1), A(3), A(5), A(7), and A(9).

Furthermore, this notation may be nested. For example, the statement

```
READ (2,15) ((C(I,J),D(I,J),J=1,3),I=1,4)
```

would transmit data in this order:

```
C(1,1), D(1,1), C(1,2), D(1,2), C(1,3), D(1,3)
C(2,1), D(2,1), C(2,2), D(2,2), C(2,3), D(2,3)
C(3,1), D(3,1), C(3,2), D(3,2), C(3,3), D(3,3)
C(4,1), D(4,1), C(4,2), D(4,2), C(4,3), D(4,3)
```

Since J is the innermost index, it varies more rapidly than I.

As another example, consider

```
READ (2,25) I,(C(J),J=1,I)
```

The variable I is read first and its value then serves as an index to specify the number of data items to be read into array C.

If it is desired to read data into an entire array, it is not necessary to index that array in the I/O list. For example, assume that the array A consists of one subscript parameter, varying in the range of 1 to 10. Then this READ statement, referring to FORMAT statement numbered 5,

```
READ (2,5) A
```

would cause data to be read into A(1), A(2),...,A(10).

The indexing of I/O lists applies to WRITE lists, as well as READ lists.

Reading Format Statements

FORTRAN provides the facility for variable FORMAT statements by allowing a FORMAT statement to be read into an array in storage and using the data in the array as the FORMAT specifications for subsequent I/O statements.

For example, the statements below result in A, B, and array C being read, converted, and stored according to the format specifications read into the array FMT at object time.

```
      DIMENSION FMT (18)
1     FORMAT (18A4)
      READ (5,1) FMT
      READ (5,FMT) A,B,(C(I),I=1,5)
```

1. The name of the variable format specification must appear in a DIMENSION statement, even if the array size is only 1.

2. The form of the format codes read into the FMT array at object time must take the same form as a source program FORMAT statement, except that the word FORMAT is omitted (see "The FORMAT Statement").

3. If a format code read in at object time contains double apostrophes within a literal field that is defined by apostrophes, it should be used for output only. If an object time format code is to be used for input and if it must contain a literal field with an internal apostrophe, the H format code must be used for the literal field definition.

WRITE STATEMENT

General Form
WRITE (a, b) list where a is an unsigned integer constant or an integer variable of length 4 that represents a data set reference number. b is either the statement number or array name of the FORMAT statement describing the data being written, or a NAMELIST name. list is a series of variable or array names, separated by commas, which may be indexed and incremented; they specify the number of items to be written and the storage locations from which the data is taken.

The WRITE statement may take many different forms. For example, the list or the parameter b may be omitted.

The three basic forms of the WRITE statement are

```
WRITE(a,x)
WRITE(a,b)list
WRITE(a)list
```

WRITE (a,x)

This form is used to write data from the storage locations specified by the NAMELIST name x into the data set associated with a (see "READ(a,x)").

Example:

```
WRITE(6,NAM1)
```

Explanation:

This statement causes all variable and array names (as well as their values) that belong to the NAMELIST name, NAM1, to be written on the data set associated with data set reference number 6.

When a WRITE statement references a NAMELIST name

1. All variables and arrays and their values belonging to the NAMELIST name will be written out, each according to its type. The complete array is written out by columns.

2. The output data will be written such that

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

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

Example: Assume that A is a 3-by-3 array.

```
      .
      .
      .
NAMELIST/NAM1/A,B,I,D
WRITE (8,NAM1)
      .
      .
      .
```

Assuming that the output is punched on cards, the format would be:

Output Card	Column 2
	▪
First	&NAM1
Second	A=3.4, 4.5, 6.2, 25.1,
Third	9.0, -15.2,-7.6, 0.576Eb12,
Fourth	2.717,B=3.14,I=10,D=0.378E-15,
Fifth	&END

WRITE (a,b) List

This form is used to write data in the data set associated with a from the locations in storage specified by the variable names in the list. The list, used in conjunction with the specified FORMAT statement

b, determines the number of items (data) to be written, the locations, and the form the data will take in the data set.

Example 1:

```
75  FORMAT (G10, G8, G9)
     .
     .
     .
WRITE (J, 75) A, B, C
```

Explanation: The WRITE statement above causes output data to be written in the data set associated with the data set reference number J, from locations A, B, C, according to the FORMAT statement referred to (statement 75). (Format statements are described in a later section.)

The list may be omitted from the WRITE (a,b) list statement. In this case, a blank record is inserted, or data is written in the data set associated with a from the locations in storage occupied by FORMAT statement b.

Example 2:

```
98   FORMAT (' HEADING')
       .
       .
       .
     WRITE (5,98)
```

The statements above would cause a blank and the characters H, E, A, D, I, N, and G in storage to be written in the data set associated with data set reference number 5.

Example 3:

```
98   FORMAT (G10, 'HEADING')
       .
       .
       .
     WRITE (5,98)
```

Explanation: The statements above would cause a blank record to be placed in the data set associated with data set reference number 5. No data is transferred into the data set because there is no list item that corresponds with format code G10.

WRITE (a) List

The WRITE (a) list form of the WRITE statement causes binary data (internal form) from the storage locations specified by the variable names in the list to be written in the data set associated with a. Since the output data is always in internal form, a FORMAT statement is not required. The READ (a) list statement is used to retrieve the data written by a WRITE (a) list statement.

Example:

```
WRITE (5)A, B, C
```

Explanation: The statement causes the binary data from the locations specified by variable names A, B, and C to be written in the data set associated with data set reference number 5.

FORMAT STATEMENT

General Form
x FORMAT (c_1,c_2,...,c_n/c_1',c_2',...,c_n'/...)
where x is a statement number (1 through 5 digits)
c_1,c_2,...,c_n and c_1',c_2',...,c_n' are format codes which may be delimited by one of the separators: comma, slash, or parenthesis; these codes specify the length, decimal point (if any), and position of the data in the data set.
The character / is used to separate FORTRAN records.

The FORMAT statement is used in conjunction with the READ and WRITE statements to specify the desired form of the data to be transmitted; the form is varied by the use of different format codes.

The format codes are

G -- to transfer integer, real, complex, or logical data

I -- to transfer integer data

F -- to transfer real data that does not contain a decimal exponent

D -- to transfer real data that contains a D decimal exponent

E -- to transfer real data that contains an E decimal exponent

L -- to transfer logical data

Z -- to transfer hexadecimal data

A -- to transfer alphameric data

Literal -- to transfer a string of alphameric and special characters

H -- to transfer literal data

X -- to either skip data when reading or insert blanks when writing

T -- to specify the position in a FORTRAN record where transfer of data is to start

P -- to specify a scale factor

Any number used in a FORMAT statement, except the statement number or a literal, must be less than or equal to 255.

USE OF THE FORMAT STATEMENT: This section contains general information on the FORMAT statement. The points discussed below are illustrated by the examples that follow.

1. FORMAT statements are nonexecutable and may be placed anywhere in the source program.

2. A FORMAT statement may be used to define a FORTRAN record, as follows:

   a. If no slashes or additional parentheses appear within a FORMAT statement, a FORTRAN record is defined by the beginning of the FORMAT statement (left parenthesis) to the end of the FORMAT statement (right parenthesis). Thus, a new record is read when

the format control is initiated (left parenthesis); a new record is written when the format control is terminated (right parenthesis).

Example:

```
x FORMAT (----, ----, ----)

          <-------------->
                 .
                 |
                 |
                 L---corresponds to 1
                     FORTRAN record
```

b. If slashes appear within a FORMAT statement, FORTRAN records are defined as starting from the beginning of the FORMAT statement and proceeding to the first slash in the FORMAT statement, from one slash to the next succeeding slash, or from the last slash to the end of the FORMAT statement. Thus, a new record is read when the format control is initiated, and thereafter a record is read upon encountering a slash; a new record is written upon encountering a slash or when format control is terminated.

Example:

```
x FORMAT (----/ ----/ ----)

          <---> <---> <--->
            |     |     |
            |     |     |
              each corresponds to 1 FORTRAN record
```

c. If more than one level of parentheses appears with a FORMAT statement, a record is defined by the beginning of the FORMAT statement and the end of the FORMAT statement. At this point, the definition of the FORTRAN record continues at the first-level left parenthesis that is closest to the right of the FORMAT statement and finishes at the end of the FORMAT statement.

Example 1:

```
                0    1    2   21    0
x FORMAT (--- (--- (---)) ---)

          <----------------->
           |   <---------->
           |           |
           |           |
           |           |
           |           |
            each corresponds to 1 FORTRAN record
```

Example 2:

```
                0    1   1     1   1    0
x FORMAT (--- (---) --- (---) ---)

          <--------------------->
              |
              |         <------->
              |             |
              |             |
              |             |
               each corresponds to 1 FORTRAN record
```

When defining a FORTRAN record by a FORMAT statement, it is important to consider the original source (input) or ultimate destination (output) of the record. For example, if a FORTRAN record is to be punched for output, the record should not be greater than 80 characters. For input, the FORMAT statement should not define a FORTRAN record longer than the actual record in the data set.

3. Blank output records may be introduced or input records may be skipped by using consecutive slashes (/) in a FORMAT statement. If there are n consecutive slashes at the beginning or end of a FORMAT statement, n input records are skipped or n blank records are inserted between output records, respectively. If n consecutive slashes appear anywhere else in a FORMAT statement, the number of records skipped or blank records inserted is n-1.

4. Successive items in an I/O list are transmitted according to successive format codes in the FORMAT statement, until all items in the list are transmitted. If there are more items in the list than there are codes in the FORMAT statement, control transfers to the preceding left parenthesis of the FORMAT statement, and the same format codes are used again with the next record. If there are fewer items in the list, the remaining format codes are not used.

5. A format code may be repeated as many times as desired by preceding the format code with an unsigned integer constant.

6. A limited parenthetical expression is permitted to enable repetition of data fields according to certain format codes within a longer FORMAT statement. 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.

7. When transferring data on input or output, the type of format code used, type of data, and type of variables in the I/O list should correspond.

8. In the examples below, the output is shown as a printed line. A carriage control character 'x' (see "Carriage Control") is specified in the FORMAT statement but does not appear in the first print position of the print line. This carriage control character appears as the first character of the output record on any I/O medium except the printed line.

G Format Code

General Form
aGw.s
where a is optional and is an unsigned integer constant, less than or equal to 255, used to denote the number of times the same format code is repetitively referenced
w is an unsigned integer constant, less than or equal to 255, specifying the total field length
s is an unsigned integer constant specifying the number of significant digits

The G format code is a generalized code, in that it may be used to determine the desired form of data, whether integer, real, complex, or logical.

The .s portion may be omitted when transferring integer or logical data; if present, it is ignored. When real data is transferred, the w portion of the G format code includes four positions for a decimal exponent field.

If the real data, say n, is in the range 0.1≤n<10**s (where s is the s portion of the format code Gw.s), this exponent field is blank. Otherwise, the real data is transferred with an E or D decimal exponent, depending on the length specification (either four or eight storage locations, respectively) of the real data.

If insufficient positions are reserved by s, the number is rounded to s digits for output, and truncated to s digits for input. If excessive positions are reserved by s, zeros are filled in on the right.

For simplification, the following examples deal with the printed line; however, the concepts developed apply to all input/output media.

Example 1: Assume that the variables A, B, C, and D are real, with values of 292.7041, 82.43441, 136.7632, and .8081945, respectively.

```
1   FORMAT  ('x',G12.4,G12.5,G12.4,G12.7)
2   FORMAT  ('x',G13.4,G13.5,G13.4)
3   FORMAT  ('x',G13.4)
        .
        .
        .
    WRITE (5, n) A, B, C, D
        .
        .
        .
```

Explanation:

a. If n had been specified as 1, the printed output would be (b represents a blank)

```
Print position 1                    Print position 48
■                                                   ■
bbb292.7bbbbbb82.434bbbbbbb136.8bbbb.8081945bbbb
```

b. If n had been specified as 2, the printed output would be

```
Print position 1          Print position 39
■                                    ■
bbbb292.7bbbbbbb82.434bbbbbbbb136.8bbbb          Line 1
bbb0.8082bbbb                                    Line 2
```

It can be seen that by increasing the field width reserved (w), blanks are inserted.

c. If n had been specified as 3, the printed output would be

```
Print position 1
■
bbbb292.7bbbb                                    Line 1
bbbb82.43bbbb                                    Line 2
bbbb136.8bbbb                                    Line 3
bbb0.8082bbbb                                    Line 4
```

The same format code was used for each variable in the list. Each repetition of the same format code caused a new line to be printed.

Example 2: Assume that the variables I, J, K, and L are integer, with values of 292, 443428, 4908081, and 40018, respectively.

```
1   FORMAT  ('x',G10,2G7,G5)
2   FORMAT  ('x',G6)
3   FORMAT  ('x',4G10)
        .
        .
        .
    WRITE (5, n) I, J, K, L
        .
        .
        .
```

Explanation:

a. If n had been specified as 1, the printed output would be

```
Print position 1              Print position 29
■                             ■
bbbbbbb292b443428490808140018                    Line 1
```

The same results would be achieved, if FORMAT statement 1 had been written as

```
FORMAT ('x',G10, G7, G7, G5)
```

Note that the .s portion of the G format may be omitted when transmitting integer data.

b. If n had been specified as 2, the printed output would be

```
Print position 1
■
bbb292                                           Line 1
443428                                           Line 2
******                                           Line 3
b40018                                           Line 4
```

Note that the second format code G6 is an incorrect specification for the third variable K, i.e., 4908081. The field will be filled with asterisks.

c. If n had been specified as 3, the printed output would be

```
Print position 1                Print position 40
■                                               ■
bbbbbbb292bbbb443428bbb490808 1bbbbb40018        Line 1
```

From the above example, it can be seen that increasing the field width w improves readability.

Example 3: Assume

Variable	Type	Length	Value
I	Integer	2	292
A	Real	4	471.93
B	Real	4	81.91
D	Real	8	6.9310072
C	Complex	8	(2.1,3.7)
L	Logical	1	.TRUE.

```
1   FORMAT  ('x',G3,2G9.2,G13.7,2G8.2,G3)
2   FORMAT  ('x',G3/'x',2G10.2/'x',G9.1/'x',2G8.2,G3)
3   FORMAT  (//'x',G3,2G9.2//'x',G13.7,2G8.2,G3///)
```

```
      .
      .
      .
   WRITE (5,n) I,A,B,D,C,L
      .
      .
      .
```

Explanation:

a. If n has been specified as 1, the printed output would be

```
Print position 1                              Print position 53
■                                                             ■
292b0.47Eb03bb82.bbbbb6.931007bbbbb2.1bbbbb3.7bbbbbbT
```

When complex data is being transmitted, two format codes are required. The real and imaginary parts are each treated as separate real numbers, and the parentheses and comma are not printed as part of the output.

b. If n has been specified as 2, the printed output would be

```
Print position 1
■
292                                                    Line 1
bb0.47Eb03bbb82.bbbb                                   Line 2
bbb7.bbbb                                              Line 3
b2.1bbbbb3.7bbbbbbT                                    Line 4
```

The use of the slash (/) to separate two format codes causes the data not yet printed to be printed on a new line. If the output data is to be punched on cards, the slash specifies that the following data will be punched on another card.

c. If n has been specified as 3, the printed output would be

```
Print position 1
■
(blank line)                                           Line 1
(blank line)                                           Line 2
292b0.47Eb03bb82.bbbb                                  Line 3
(blank line)                                           Line 4
b6.931007bbbbb2.1bbbbb3.7bbbbbbT                       Line 5
(blank line)                                           Line 6
(blank line)                                           Line 7
(blank line)                                           Line 8
```

Note that the two consecutive slashes appearing at the beginning and the three at the end of the series of format codes cause blank lines to be inserted as shown. However, n consecutive slashes appearing elsewhere in a FORMAT statement cause the insertion of n-1 blank lines, as shown in line 4.

The principles illustrated in the previous output examples also apply when using the READ statement on input. Also, there are further considerations when using the FORMAT statement on input or output.

1. When reading real input data with a G format code, a decimal point must be included.

2. The use of additional parentheses (up to two levels) within a FORMAT statement is permitted to enable the user to repeat the same format code when tr nsmitting data. For example,

```
10  FORMAT (2(G10.6,G7.1),G4)
```

is equivalent to

```
10  FORMAT (G10.6, G7.1, G10.6, G7.1, G4)
```

3. If a multiline listing is desired, with the first two lines to be printed according to a special format and all remaining lines according to another format, the last format code in the statement should be enclosed in a second pair of parentheses. For example,

```
FORMAT ('x',G2,2G3.1/'x',G10.8/('x',3G5.1))
```

If more data items are to be transmitted after the format codes have been completely used, the format repeats from the last left parenthesis. Thus, the printed output would take the form

```
G2,G3.1,G3.1
G10.8
G5.1,G5.1,G5.1
G5.1,G5.1,G5.1
  .    .    .
  .    .    .
  .    .    .
```

As another example, consider the statement

```
FORMAT ('x',G2/2('x',G3,G6.1),G9.7)
```

If there are 13 data items to be transmitted, the printed output on a WRITE statement would take the form

```
G2
G3,G6.1,'x',G3,G6.1,G9.7
G3,G6.1,'x',G3,G6.1,G9.7
G3,G6.1
```

Numeric Format Codes (I,F,E,D, and Z)

Five types of format codes are available for the transfer of numeric data. These are specified in this form

General Form
aIw aFw.d aEw.d aDw.d aZw
where a is optional and is an unsigned integer constant, less than or equal to 255, used to denote the number of times the same format code is repetitively referenced I, F, E, D, and Z are format codes w is an unsigned integer constant less than or equal to 255, specifying the total field length of the data d is an unsigned integer constant specifying the number of decimal places to the right of the decimal point, i.e., the fractional portion

For purposes of simplification, the following description of format codes deals with the printed line. The concepts developed apply to all input/output media.

I Format Code

The I format code is used to transmit integer data.

If the number of characters to be transmitted is greater than w, on input, the excess rightmost characters are lost; on output, the entire field, w characters, will be filled with asterisks.

If the number of characters is less than w, on input, leading blanks are not significant; embedded and trailing blanks are treated as zeros. On output, the leftmost positions are filled with blanks.

If the quantity is negative, the position preceding the leftmost digit contains a minus sign. In this case, an additional position should be specified in w for the minus sign. If w is such that no space exists for the minus sign, the entire field, w characters, will be filled with asterisks.

The following examples show how each of the quantities on the left is printed according to the format code I3 (b represents a blank).

Internal Value	Printed Value	
721	721	
-721	***	(incorrect because of insufficient specification)
-12	-12	
568114	***	(incorrect because of insufficient specification)
0	bb0	
-5	b-5	
9	bb9	

F Format Code

For F format codes which are used in conjunction with the transfer of real data, w is the total field length reserved, and d is the number of places to the right of the decimal point (the fractional portion). This differs from the G format code, where the number of significant digits is specified. The total field length reserved must include sufficient positions for a minus sign (if any) and a decimal point. The sign, if negative, is printed.

If insufficient positions are reserved by d, the fractional portion is rounded to the dth position. If excessive positions are reserved by d, zeros are filled in on the right. The integer portion of the number is handled in the same manner as numbers transmitted by the I format code.

The following examples show how each of the quantities on the left is printed according to the format code F5.2.

Internal Value	Printed Value	
12.17	12.17	
-41.16	*****	(incorrect, insufficient specification)
-.2	-0.20	
7.3542	b7.35	(last two digits of accuracy lost; insufficient specification)
-1.	-1.00	
9.03	b9.03	
187.64	*****	(incorrect; insufficient specification)

D and E Format Codes

The D and E format codes are used in conjunction with the transferral of real data that contains a D or E decimal exponent, respectively. A D format code indicates a field length of 8; an E code indicates a length of 4. For D and E format codes, the fractional portion is again indicated by d. The w includes field d, spaces for a sign, the decimal point, plus four spaces for the exponent.

For output, space for at least one digit preceding the decimal point should be reserved. In general, w should be at least equal to d+7. If insufficient positions for d are supplied, the fraction is rounded to the dth position. If excessive positions are supplied, zeros are added.

The exponent is the power of 10 by which the number must be multiplied to obtain its true value. The exponent is written with a D or an E, followed by a space for the sign and two spaces for the exponent (maximum is 75).

The following examples show how each of the quantities on the left is printed according to the format codes (D10.3/E10.3).

Internal Value	Printed Value	
238.	b0.238Db03	
-.002	-0.200E-02	
.00000000004	b0.400D-10	
-21.0057	-0.210Eb02	(last three digits of accuracy lost; insufficient field width)

When reading input data, the start of the exponent field must be marked by an E or, if that is omitted, by a + or - sign (not a blank). Thus, E2, E+2, +2, +02, E02, and E+02 all have the same effect and are permissible decimal exponents for input.

Numbers for E, D, and F format codes need not have their decimal point punched. If it is not present, the decimal point is supplied by the d portion of the format code. If it is present in the card, its position overrides the position indicated by the d portion of the format code.

Z Format Code

General Form
aZw where a is optional and is an unsigned integer constant, less than or equal to 255, used to denote the number of times the same format code is repetitively referenced w is an unsigned integer constant, less than or equal to 255, specifying the number of characters of data

The Z format code is used in conjunction with the transfer of hexadecimal numbers.

One storage location contains two hexadecimal digits. In read and write operations, padding and truncation are on the left. However, in a read operation, the padding character is a hexadecimal zero; in a write operation, it is a blank.

If an eight-byte internal field with the hexadecimal pattern '0123456789ABCDEF' is required, the external record could contain the

characters 123456789ABCDEF and would be read by a Z15 format code. The high-order zero is automatically provided as the padding character.

L Format Code

General Form	
aLw	
where	a is optional and is an unsigned integer constant, less than or equal to 255, used to denote the number of times the same format code is repetitively referenced
	w is an unsigned integer constant less than or equal to 255, specifying the number of characters of data

Logical variables may be read or written by means of the format code Lw.

On input, the first T or F encountered in the next w characters of the input record causes a value of .TRUE. or .FALSE., respectively, to be assigned to the corresponding logical variable. If the field w consists entirely of blanks, a value of .FALSE. is assumed.

On output, a T or an F is inserted in the output record corresponding to the value of the logical variable in the I/O list. The single character is preceded by w - 1 blanks.

A Format Code

General Form	
aAw	
where	a is optional and is an unsigned integer constant, less than or equal to 255, used to denote the number of times the same format code is repetitively referenced
	w is an unsigned integer constant less than or equal to 255, specifying the number of characters of data

The format code Aw is used to read or write alphameric data. If w is equal to the number of characters corresponding to the length specification of each item in the I/O list, w characters are read or written.

On input, if w is less than the length specification of each item in the I/O list, w characters are read and the remaining rightmost characters in the item are replaced with blanks. If w is greater than the length specification, the number of characters equal to the difference between w and the length specification are skipped, beginning with the leftmost character, and the remaining characters are read.

On output, if w is less than the length specification of the item in the I/O list, the printed line will consist of the leftmost w characters of the item. If w is greater than the length specification, the printed line will consist of the characters right-justified in the field and will be preceded by blanks. Therefore, it is important to always allocate enough storage area to handle the characters being written (see "The Type Statements").

Example 1: Assume that the array ALPHA consists of one subscript parameter ranging from 1 through 20. The following statements could be written to copy a record from one data set to another.

```
      .
      .
      .
10    FORMAT (20A4)
      .
      .
      .
      READ (5,10) (ALPHA(I),I=1,20)
      .
      .
      .
      WRITE (6,10) (ALPHA(I),I=1,20)
      .
      .
      .
```

Explanation: The READ statement would cause 20 groups of characters to be read from the data set associated with data set reference number 5. Each group of four characters would be placed into the 20 storage locations starting with ALPHA(1) and ending with ALPHA(20). The WRITE statement would cause the 20 groups of four characters to be written on the data set associated with data set reference number 6.

Example 2: As another example, consider all the variable names in the list of the READ statement, below, to have been explicitly specified as REAL, and the array CONST to have been specified as having one subscript parameter ranging from 1 through 10. Then assume this input data is associated with data set reference number 5

```
ABCDE...XYZ$1234567890b
```

where ... represents the alphabetic characters F through W, and b means a blank. These statements could be written

```
      .
      .
10   FORMAT  (27A1,10A1,A1)
20   FORMAT  ('x',6(7A1,5X))
      .
      .
     READ  (5,10)A,B,C,D,E,F,G,H,I,
    1            J,K,L,M,N,O,P,Q,R,
    2            S,T,U,V,W,X,Y,Z,$,
    3            (CONST (IND),IND=1, 10), BLANK
      .
      .
     DO 50 INDEX = 1,5
      .
      .
     WRITE (6,20)G,R,O,U,P,BLANK,CONST(INDEX),
    1            B,L,O,C,K,BLANK,CONST(INDEX),
    2            F,I,E,L,D,BLANK,CONST(INDEX),
    3            G,R,O,U,P,BLANK,CONST(INDEX+5),
    4            B,L,O,C,K,BLANK,CONST(INDEX+5),
    5            F,I,E,L,D,BLANK,CONST(INDEX+5)
      .
      .
50   CONTINUE
      .
      .
```

Explanation: The READ statement would cause the 37 alphameric characters and the blank in the data set associated with data set reference number 5 to be placed in the storage locations specified by the variable names in the READ list. Thus, the variables A through Z receive the values A through Z, respectively; the variable $ receives the value $; the numbers 1 through 9, and 0, are placed in the 10 fields in storage starting with CONST(1) and ending with CONST(10); and the variable BLANK receives a blank. The WRITE statement within the DO loop would cause the following heading to be printed. A subsequent WRITE statement within the DO loop could then be written to print the corresponding output data.

```
Print Position 1                                        Print Position 67
•                                                                      •
|                                                                      |
GROUP 1     BLOCK 1    FIELD 1    GROUP 6    BLOCK 6    FIELD 6
    -           -          -          -          -          -
    -           -    (output data)    -          -          -
    -           -          -          -          -          -
GROUP 2     BLOCK 2    FIELD 2    GROUP 7    BLOCK 7    FIELD 7
    -           -          -          -          -          -
    -           -    (output data)    -          -          -
    -           -          -          -          -          -
    .           .          .          .          .          .
    .           .          .          .          .          .
    .           .          .          .          .          .
GROUP 5     BLOCK 5    FIELD 5    GROUP 0    BLOCK 0    FIELD 0
    -           -          -          -          -          -
    -           -    (output data)    -          -          -
    -           -          -          -          -          -
```

Literal Data in a Format Statement

Literal data consists of a string of alphameric and special characters written within the FORMAT statement and enclosed in apostrophes. The string of characters must be less than or equal to 255. For example:

```
25  FORMAT  (' 1970 INVENTORY REPORT')
```

An apostrophe within the string is represented by two successive apostrophes; for example, the characters DON'T are represented as:

```
DON''T
```

The effect of the literal format code depends on whether it is used with an input or output statement.

INPUT

A number of characters, equal to the number of characters between the apostrophes, are read from the designated data set. These characters replace, in the FORMAT statement, the characters within the apostrophes. For example,

```
          .
          .
          .
5     FORMAT (' HEADINGS')
          .
          .
          .
      READ (3,5)
          .
          .
          .
```

would cause the next nine characters to be read from the data set associated with data set reference number 3; these characters would replace the blank and the eight characters in H E A D I N G S in the FORMAT statement.

OUTPUT

All characters (including blanks) within the apostrophes are written as part of the output data; thus,

```
          .
          .
          .
    5     FORMAT (' THIS IS ALPHAMERIC DATA')
          .
          .
          .
          WRITE (2,5)
          .
          .
          .
```

would cause the following record to be written on the data set associated with the data set reference number 2

bTHIS IS ALPHAMERIC DATA

where b indicates a blank.

H Format Code

General Form	
wH	
where	w is an unsigned integer constant less than or equal to 255, specifying the number of characters following H

The H format code is used in conjunction with the transfer of literal data.

The format code wH is followed in the FORMAT statement by w ($w \leq 255$) characters; for example,

```
    5   FORMAT (31H THIS IS ALPHAMERIC INFORMATION)
```

Blanks are significant and must be included as part of the count w. The effect of wH depends on whether it is used with input or output.

1. On input, w characters are extracted from the input record and replace the w characters of the literal data in the FORMAT statement.

2. On output, the w characters following the format code are written as part of the output record.

X Format Code

General Form	
wX	
where	w is an unsigned integer constant less than or equal to 255, specifying the number of blanks to be inserted on output or the number of characters to be skipped on input

When the wX (w≤255) format code is used with a READ statement (i.e., on input), w characters are skipped before the next data item is read in. For example, if a card has six 10-column fields of integer quantities, and the second quantity is not to be read, then

```
5   FORMAT   (I10,10X,4I10)
```

may be used with the appropriate READ statement.

When the wX format code is used with a WRITE statement (i.e., on output), w characters are left blank. Thus, the facility for spacing within a printed line is available. For example,

```
10 FORMAT ('x',3(F6.2,5X))
```

may be used with an appropriate WRITE statement to print this line

```
123.45bbbbb817.32bbbbb524.67bbbbb
```

T Format Code

General Form	
Tw	
where	w is an unsigned integer constant less than or equal to 255, specifying the position in a FORTRAN record where the transfer of data is to begin

Input and output may begin at any position by using the format code Tw (w≤255). Only when the output is printed does the correspondence between w and the actual print position differ. In this case, because of the carriage control character, the print position corresponds to w-1, as in

```
5  FORMAT (T40, '1970 INVENTORY REPORT' T80, 'DECEMBER' T1, ' PART
   NO. 10095')
```

The FORMAT statement above would result in this printed line

```
Print              Print                      Print
Position 1         Position 39                Position 79
•                  •                          •
PART NO. 10095     1964 INVENTORY REPORT      DECEMBER
```

These statements

```
5  FORMAT  (T40, ' HEADINGS')
      .
      .
      .
   READ (3,5)
```

would cause the first 39 characters of the input data to be skipped, and the next 9 characters would then replace the blank and the characters H E A D I N G S in the FORMAT statement.

The T format code may be used in a FORMAT statement with any type of format code. For example, this statement is valid

```
5 FORMAT (T100, F10.3, T50, E9.3, T1, ' ANSWER IS')
```

Scale Factor - P

The representation of the data, internally or externally, may be modified by the use of a scale factor followed by the letter P preceding a format code.

The scale factor is defined for input and output as

external quantity = internal quantity x 10**scale factor

For input, when scale factors are used in a FORMAT statement, they have effect only on real data which does not contain an E or D decimal exponent. For example, if input data is in the form xx.xxxx and, it is to be used internally in the form .xxxxxx, the format code used to effect this change is 2PF7.4.

INPUT

As another example, consider this input data

```
27bbb-93.2094bb-175.8041bbbb55.3647
```

where b represents a blank.

These statements

```
5  FORMAT (I2,3F11.4)
        .
        .
        .
   READ (6,5) K,A,B,C
```

would cause these variables in the list to assume these values

```
K : 27            B : -175.8041
A : -93.2094      C : 55.3647
```

These statements

```
5  FORMAT (I2,1P3F11.4)
        .
        .
        .
   READ (6,5) K,A,B,C
```

would cause these variables in the list to assume these values

```
K : 27            B : -17.5804
A : -9.3209       C : 5.5364
```

These statements

```
5  FORMAT (I2,-1P3F11.4)
        .
        .
        .
   READ (6,5) K,A,B,C
```

would cause the variables in the list to assume these values

```
K : 27            B : -1758.041x
A : -932.094x     C : 553.647x
```

where x represents an extraneous digit.

OUTPUT

Assume that the variables K, A, B, and C have these values

```
K : 27            B : -175.8041
A : -93.2094      C : 55.3647
```

these statements

```
5  FORMAT (I2,1P3F11.4)
        .
        .
        .
   WRITE (4,5) K,A,B,C
```

would cause the variables in the list to output these values

```
K : 27            B : -1758.041x
A : -932.094x     C : 553.647x
```

where x represents an extraneous digit.

These statements

```
5  FORMAT (I2,-1P3F11.4)
        .
        .
        .
   WRITE (4,5) K,A,B,C
```

would cause the variables in the list to output these values

```
K : 27            B : -17.5804
A : -9.3209       C : 5.5364
```

For output, when scale factors are used, they have effect only on real data. However, this real data may contain an E or D decimal exponent. A positive scale factor used with real data that contains an E or D decimal exponent increases the number and decreases the exponent. Thus, if the real data was in a form using an E decimal exponent, and the statement FORMAT (1X,I2,3E13.3) used with an appropriate WRITE statement resulted in this printed line

```
27bbb-0.932Eb02bbb-0.175Eb03bbbb0.553Eb02
```

then the statement FORMAT (1X,I2,1P3E13.3) used with the same WRITE statement would result in this printed output

```
27bbb-9.320Eb01bbb-1.758Eb02bbbb5.536Eb01
```

The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it will hold for all format codes (i.e., those that correspond to real data) following the scale factor within the same FORMAT statement. This also applies to format codes enclosed within an additional pair of parentheses. Once the scale factor has been given, a subsequent scale factor of zero in the same FORMAT statement must be specified by 0P.

Carriage Control

When records written under format control are prepared for printing, the following convention for carriage control applies:

First Character	Carriage Advance Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

The first character of the output record may be used for carriage control and does not appear in the first print position of the print line. However, it appears in all other media as data.

Carriage control can be specified in either of two forms of literal data. These statements would both cause two lines to be skipped before printing

```
10 FORMAT ('0', 5(F7.3))
10 FORMAT (1H0, 5(F7.3))
```

ADDITIONAL INPUT/OUTPUT STATEMENTS

The statements END FILE, REWIND, and BACKSPACE are used to control the data sets, as described in the following text.

END FILE Statement

General Form
END FILE a where a is an unsigned integer constant or integer variable of length 4 that represents a data set reference number

The END FILE statement defines the end of the data set associated with a. A subsequent WRITE statement defines the beginning of a new data set.

REWIND Statement

General Form
REWIND a where a is an unsigned integer constant or integer variable of length 4 that represents a data set reference number

The REWIND statement causes a subsequent READ or WRITE statement referring to a to read data from or write data into the first data set associated with a. REWIND causes a logical rewinding to the beginning of the first data set associated with the specified data set reference number; it does not always cause a physical rewinding of the tape. If, however, the specified data set is the first on the tape, physical, as well as logical, rewinding occurs.

BACKSPACE Statement

General Form
BACKSPACE a where a is an unsigned integer constant or integer variable of length 4 that represents a data set reference number

The BACKSPACE statement causes the data set associated with a to backspace one record. If the data set associated with a is already at its beginning, execution of this statement has no effect.

The specification statements provide the compiler with information about the nature of the data used in the source program. In addition, they supply the information required to allocate storage locations for this data. Specification statements describing data may appear anywhere in the source program, but must precede any statements which refer to that data. The specification statements are the type statements IMPLICIT, INTEGER, REAL, COMPLEX, and LOGICAL, and the DIMENSION, COMMON, and EQUIVALENCE statements.

THE TYPE STATEMENTS

There are two kinds of type statements: the IMPLICIT specification statement and the Explicit specification statements (INTEGER, REAL, COMPLEX, and LOGICAL).

The IMPLICIT specification statement enables the user to

1. Specify the type of a group of variables or arrays according to the initial character of their names.

2. Specify the amount of storage to be allocated for each variable according to the associated type.

The Explicit specification statements enable the user to

1. Specify the type of a variable or array according to their particular name.

2. Specify the amount of storage to be allocated for each variable according to the associated type.

3. Specify the dimensions of an array.

4. Assign initial data values for variables and arrays.

IMPLICIT Statement

General Form
IMPLICIT *type*s(a1,a2,...),...,type*s(a1,a2,...)*
where *type* represents one of the following: INTEGER, REAL, COMPLEX, or LOGICAL
**s* is optional and represents one of the permissible length specifications for its associated type
a1, a2,... represent single alphabetic characters each separated by commas, or a range of characters (in alphabetic sequence) denoted by the first and last characters of the range separated by a minus sign (e.g., (A-D))

The IMPLICIT statement, if specified, should be the first statement in a main program, and the second statement in a FUNCTION, SUBROUTINE, or BLOCK DATA subprogram.

The IMPLICIT type statement enables the user to declare the type of the variables appearing in his program (i.e., integer, real, complex, or

logical) by specifying that variables beginning with certain designated letters are of a certain type. Furthermore, the IMPLICIT statement allows the programmer to declare the number of locations to be allocated for each specified variable in a group. The types that a variable can assume, and the permissible lengths are

Type	Length Specification
INTEGER	2 or 4 (standard length is 4)
REAL	4 or 8 (standard length is 4)
COMPLEX	8 or 16 (standard length is 8)
LOGICAL	1 or 4 (standard length is 4)

For each type there is a corresponding standard length specification. If this standard length specification (for its associated type) is desired, the *s may be omitted in the IMPLICIT statement. That is, the variables will assume the standard length specification. For each type there is also a corresponding optional length specification. If this optional length specification is desired, the *s must be included within the IMPLICIT statement.

Example 1:

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

Explanation: All variables beginning with the characters I through N are declared as INTEGER. Since no length specification was explicitly given (i.e., the *s was omitted), four storage locations (the standard length for INTEGER) are allocated for each variable.

All other variables (those beginning with the characters A through H, O through Z, and $) are declared as REAL with four storage locations allocated for each.

Note that the statement in Example 1 performs the same function of typing variables as the predefined convention (see "Type Declaration by the Predefined Specification").

Example 2:

```
IMPLICIT INTEGER*2(A-H), REAL*8(I-K), LOGICAL(L,M,N)
```

Explanation: All variables beginning with the characters A through H are declared as integer, with two storage locations allocated for each. All variables beginning with the characters I through K are declared as real, with eight storage locations allocated for each. All variables beginning with the characters L, M, and N are declared as logical, with four locations allocated for each.

Since the remaining letters of the alphabet (O through Z and $) were left undefined by the IMPLICIT statement, the predefined convention will take effect. Thus, all variables beginning with the characters O through Z and $ are declared as real, each with a standard length of four locations each.

Example 3:

```
IMPLICIT COMPLEX*16(C-F)
```

Explanation: All variables beginning with the characters C through F are declared as complex, each with eight storage locations reserved for the real part of the complex data and eight storage locations reserved for the imaginary part. The types of the variables beginning with the characters A, B, G through Z, and $ are determined by the predefined convention.

Explicit Specification Statements

General Form

type*s a*s$_1$(k_1)/x_1/,b*s$_2$(k_2)/x_2/,...,z*s$_n$(k_n)/x_n/

where Type is INTEGER, REAL, LOGICAL, or COMPLEX

*s,*s$_1$,*s$_2$,...,*s$_n$ are optional; each s represents one of the permissible length specifications for its associated type

a,b,...,z represent variable, array, or function names (see "SUBPROGRAMS")

(k_1),(k_2),...,(k_n) are optional; each k is composed of 1 through 7 unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array; each k may be an unsigned integer variable only when it appears in a Type statement in a subprogram

/x_1/,/x_2/,.../x_n/ are optional and represent initial data values

The Explicit specification statements declare the type (INTEGER, REAL, COMPLEX, or LOGICAL) of a variable or array by its name, rather than by its initial character. This differs from the other ways of specifying the type of a variable or array (i.e., the predefined convention and the IMPLICIT statement). Also, the information necessary to allocate storage for arrays (dimension information) may be included within the statement. However, if this information does not appear in an Explicit specification statement, it must appear in a DIMENSION or COMMON statement (see "DIMENSION Statement" or "COMMON Statement").

Initial data values may be assigned to variables or arrays by use of /x_n/, where x_n is a constant or list of constants separated by commas. This set of constants may be in the form "r* constant", where r is an unsigned integer, called the repeat constant.

No element may have more than one initial value given in the same program. A function name may not have an initial value assigned to it. An initially defined variable or a variable of an array may not be in blank common; in a labeled common block, they may be initially defined only in a BLOCK DATA subprogram. Initial data values may not be assigned to dummy segments appearing in a FUNCTION, SUBROUTINE, or ENTRY statement.

In the same manner in which the IMPLICIT statement overrides the predefined convention, the Explicit specification statements override the IMPLICIT and predefined convention. If the length specification is omitted (i.e.,*s), the standard length per type is assumed.

Example 1:

```
INTEGER*2 ITEM/76/, VALUE
```

Explanation: This statement declares that the variables ITEM and VALUE are of type integer, with two storage locations reserved for each. Also, the variable ITEM is initialized to the value 76.

Example 2:

```
COMPLEX C,D/(2.1,4.7)/,E*16
```

Explanation: This statement declares that the variables C, D, and E are of type complex. Since no length specification was explicitly given for

C and D, the standard length is assumed. Thus, C and D have eight storage locations reserved for each (four for the real part, four for the imaginary part) and D is initialized to the value (2.1,4.7). In addition, 16 storage locations are reserved for the variable E. Thus, if a length specification is explicitly written, it overrides the assumed standard length.

Example 3:

```
REAL*8 ARRAY, HOLD, VALUE*4, ITEM(5,5)
```

Explanation: This statement declares that the variables ARRAY, HOLD, VALUE, and the array named ITEM are of type real. In addition, it declares the size of the array ITEM. ARRAY and HOLD have eight storage locations reserved for each; VALUE has four locations reserved; and ITEM has 200 storage locations reserved (eight for each variable in the array). Note that when the length is associated with the type (e.g., REAL*8), the length applies to each variable in the statement unless explicitly overridden (as in the case of VALUE*4).

Example 4:

```
REAL A(5,5)/20*6.9E2,5*1.0/, B(100)/100*0.0/,TOAD*8(5)/5*0.0/
```

Explanation: This statement declares the size of each array, A and B, and their type (real). The array A has 100 storage locations reserved (four for each variable in the array); the array B has 400 storage locations reserved (four for each variable). Also, the first 20 variables in the array A are initialized to the value 6.9E2 and the last five variables are initialized to the value 1.0. All 100 variables in the array B are initialized to the value 0.0. The array TOAD has 40 storage locations reserved (eight for each variable). Also, each variable is initialized to the value 0.0.

Example 5:

```
REAL A/Z1234CAF9/,B
```

Explanation: This statement declares that the variables A and B are of type real, each with four storage locations reserved. Also, variable A is initialized to 1234CAF9 by using the hexadecimal constant. Note that the maximum number of digits allowed is dependent upon the length specification of the variable being initialized. If the number of digits is greater than the maximum allowed, the leftmost hexadecimal digits are truncated; if less than the maximum, hexadecimal zeros are supplied on the left (see "Hexadecimal Constants").

ADDITIONAL SPECIFICATION STATEMENTS

DIMENSION Statement

General Form
DIMENSION $a_1(k_1), a_2(k_2), a_3(k_3), \ldots, a_n(k_n)$ where $a_1, a_2, a_3, \ldots, a_n$ are array names $k_1, k_2, k_3, \ldots, k_n$ are each composed of 1 through 7 unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array; k_1 through k_n may be integer variables of length 4 only when they appear in a DIMENSION statement within a subprogram.

The information necessary to allocate storage for arrays used in the source program may be provided by the DIMENSION statement. The following examples illustrate how this information may be declared.

Examples:

```
DIMENSION A(10), ARRAY (5,5,5,5,5), LIST(10,100)
DIMENSION B(25,50),TABLE(25,25,25)
```

Adjustable Dimensions

The previous examples showed that the maximum value of each subscript in an array was specified by a numeric value. These numeric values (maximum value of each subscript) are known as the absolute dimensions of an array and may never be changed. However, if an array is used in a subprogram (see "Subprograms") and is not in Common, the size of this array does not have to be explicitly declared in the subprogram by a numeric value. That is, the Explicit specification statement, appearing in a subprogram, may contain integer variables that specify the size of the array. When the subprogram is called, these integer variables receive their values from the calling program. Thus, the dimensions (size) of a dummy array appearing in a subprogram are adjustable and may change each time the subprogram is called.

The absolute dimensions of an array must be declared in a calling program. The adjustable dimensions of an array, appearing in a subprogram, should be less than or equal to the absolute dimensions of that array, as declared in the calling program.

The following example illustrates the use of adjustable dimensions:

Example:

```
Calling Program                 Subprogram
      .
      .
      .
REAL*8 A(5,5)                   SUBROUTINE MAPMY(...,R,L,M,...)
      .                               .
      .                               .
      .                               .
CALL MAPMY(...,A,2,3,...)       REAL*8...,R(L,M),...
      .                               .
      .                               .
      .                               .
                                DO 100 I=1,L
                                      .
                                      .
                                      .
```

Explanation: The statement REAL*8 A(5,5) appearing in the calling program declares the absolute dimensions of the array A. When the subroutine MAPMY is called, the dummy argument R assumes the array name A, and the dummy arguments L and M assume the values 2 and 3, respectively.

The correspondence of the subscripted variables of the arrays A and R is shown in the following example.

```
R(1,1) R(2,1) R(1,2) R(2,2) R(1,3) R(2,3)
A(1,1) A(2,1) A(3,1) A(4,1) A(5,1) A(1,2) A(2,2) ...
A(1,2) A(2,2) A(3,2) A(4,2) A(5,2)
A(1,3) A(2,3) A(3,3) A(4,3) A(5,3)
A(1,4) A(2,4) A(3,4) A(4,4) A(5,4)
A(1,5) A(2,5) A(3,5) A(4,5) A(5,5)
```

Thus, in the calling program, the subscripted variable A(1,2) refers to the sixth subscripted variable in the array A. However, in the subprogram MAPMY the subscripted variable R(1,2) refers to the third subscripted variable in the array A, which is A(3,1). This is so because the dimensions of the array R, as declared in the subprogram, are not the same as those in the calling program.

If the absolute dimensions in the calling program were the same as the adjusted dimensions in the subprogram, the subscripted variables R(1,1) through R(5,5) in the subprogram would always refer to the same storage locations as specified by the subscripted variables A(1,1) through A(5,5) in the calling program, respectively.

The numbers 2 and 3, which became the adjusted dimension of the dummy array R, could also have been variables in the argument list of the calling program. For example, assume that this statement was in the calling program:

```
CALL MAPMY (...,A,I,J,...)
```

Then, as long as the values of I and J were previously determined, the arguments may be variables. Also, the variable dimension size may be passed through more than one level of subprograms; for example, within the subprogram MAPMY could have been a call statement to another subprogram in which dimension information about A could have been passed.

If an array has a variable dimension, that array name must be a dummy variable (i.e., must appear in a FUNCTION, SUBROUTINE, or ENTRY statement). The variable dimension itself can be a dummy variable or can appear in a COMMON statement.

COMMON Statement

General Form
COMMON /r/a (k_1),b(k_2),.../r/c(k_3),d(k),...
where a,b,...,c,d... are variable or array names
k_1,k_2,...k_3,k ... are optional and are each composed of one through seven unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array
/r/... represent optional common block names consisting of one through six alphameric characters, the first of which is alphabetic. These names must always be embedded in slashes

Variables or arrays that appear in a calling program or a subprogram may be made to share the same storage locations with variables or arrays in other subprograms by use of the COMMON statement. For example, if one program contains the statement

```
COMMON TABLE
```

and a second program contains the statement

```
COMMON LIST
```

the variable names TABLE and LIST refer to the same storage locations.

If the main program contains the statements:

```
REAL A,B,C
COMMON A,B,C
```

and a subprogram contains the statements:

```
REAL X,Y,Z
COMMON X,Y,Z
```

A shares the same storage location as X; B shares the same storage location as Y; and C shares the same storage location as Z.

Consider the following examples:

Example 1:

```
Calling Program              Subprogram
      .                      SUBROUTINE MAPMY (...)
      .                            .
      .                            .
COMMON A, B, C, R(100)             .
REAL A,B,C                   COMMON X, Y, Z, S(100)
INTEGER R                    REAL X,Y,Z
      .                      INTEGER S
      .                            .
      .                            .
CALL MAPMY (...)                   .
```

Explanation: In the calling program, the statement COMMON A,B,C,R(100) would cause 412 storage locations (four locations per variable) to be reserved in this manner

Beginning of common area				Layout of storage
	A 4 locations	B 4 locations	C 4 locations	
	R(1) 4 locations	. . .	R(100) 4 locations	

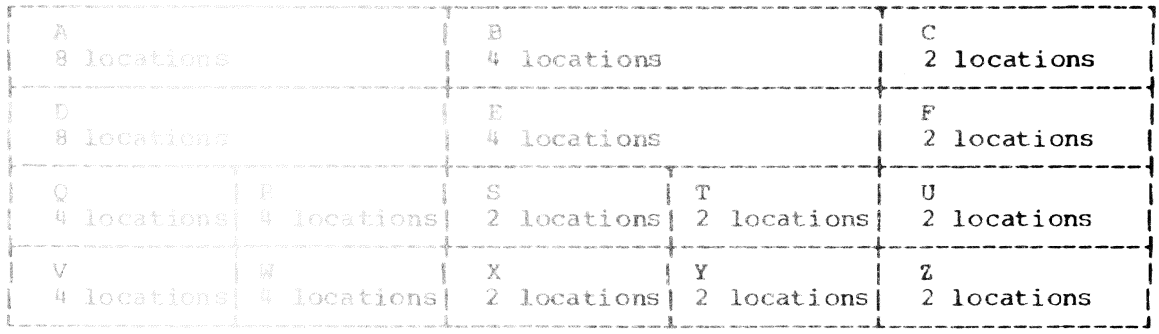
The statement COMMON X, Y, Z, S(100) would then cause the variables X, Y, Z, and S(1)...S(100) to share the same storage space as A, B, C, and R(1)...R(100).

The example above shows that COMMON statements may be used to function as a medium to implicitly transmit data from the calling program to the subprogram. That is, values for X, Y, Z, and S(1)...S(100), because they occupy the same storage locations as A, B, C, and R(1)...R(100), do not have to be transmitted in the argument list of a CALL statement. Arguments passed through COMMON must follow the same rules of presentation with regard to type, length, etc., as arguments passed in a list (see "SUBPROGRAMS").

Example 2: Assume COMMON is defined in a main program and three subprograms as

```
Main program --     COMMON    A,B,C
Subprogram 1 --     COMMON    D,E,F
Subprogram 2 --     COMMON    Q,R,S,T,U
Subprogram 3 --     COMMON    V,W,X,Y,Z
```

Also, assume the length specifications of these variables are so defined that the common area is shared as follows



In this case, the variables A,B,C and D,E,F may be validly referred to in their respective programs, as may Q,R,S,T,U and V,W,X,Y,Z. In addition, all programs may validly refer to C,F,U, and Z. It is also possible to cross-reference D in Subprogram 1 and Q and R in Subprogram 2. Such correspondences are highly data-dependent and in certain cases may be useful. For instance, if D is a complex variable, and Q and R are real variables, Q and R correspond to the real and imaginary parts of D, respectively. However, each such cross reference by the programmer must be considered on its own merits.

Blank and Labeled Common

In each of the preceding two examples, the common storage area (common block) established is called a blank common area. That is, no particular name was given to that area of storage. The variables that appeared in the COMMON statements were assigned locations relative to the beginning of this blank common area. However, variables and arrays may be placed in separate common areas. Each of these separate areas (or blocks) is given a name consisting of one through six alphameric characters (the first of which is alphabetic); those blocks that have the same name occupy the same storage space.

Variables that are to be placed in labeled (or named) common are preceded by a common block name enclosed in slashes. For example, the variables A,B, and C will be placed in the labeled common area HOLD by this statement

```
COMMON/HOLD/A,B,C
```

In a COMMON statement, blank common may be distinguished from labeled common by preceding the variables in blank common by two consecutive slashes or, if the variables appear at the beginning of the common statement, by omitting any block name. For example, in

```
COMMON A, B, C /ITEMS/ X, Y, Z / / D, E, F
```

the variables A, B, C, D, E, and F will be placed in blank common in that order; the variables X, Y, and Z will be placed in the common area labeled ITEMS.

Blank and labeled common entries appearing in COMMON statements are cumulative throughout the program. For example, consider

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

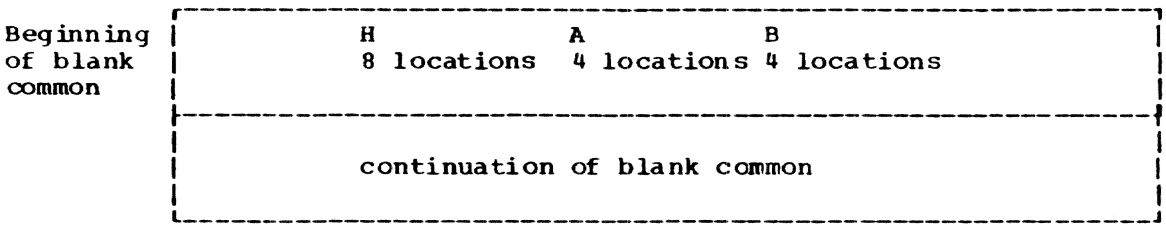
These two statements have the same effect as the single statement

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

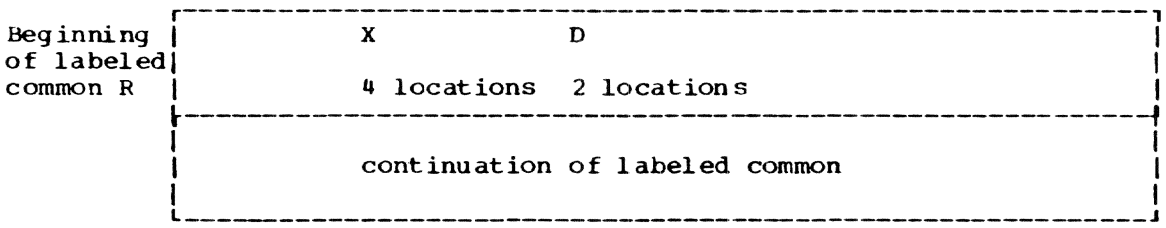
Example 3: Assume that A, B, C, K, X, and Y each occupy four locations of storage, H and G each occupy eight locations, and D and E each occupy two locations.

```
Calling Program                 Subprogram
                                SUBROUTINE MAPMY(...)
      .                               .
      .                               .
      .                               .
COMMON H, A /R/ X, D // B             .
      .                         COMMON G, Y, C /R/ K, E
      .                               .
      .                               .
      .                               .
CALL MAPMY(...)
      .
      .
      .
```

Explanation: In the calling program, the statement COMMON H,A/R/X,D//B causes 16 locations (four locations each for A and B, and eight for H) to be reserved in blank common in this order



and also causes six locations (four for X and two for D) to be reserved in this labeled common area R in this order



The statement COMMON G,Y,C/R/K,E appearing in the subprogram MAPMY would cause the variables G, Y, and C to share the same storage space (in blank common) as H, A, and B, respectively. It would also cause the variables K and E to share the same storage space (in labeled common area R) as X and D, respectively. The length of a COMMON area may be increased by using an EQUIVALENCE statement (see "EQUIVALENCE Statements")

Programming Considerations

Variables in a COMMON block may be in any order. However, considerable object-time efficiency is lost unless the programmer ensures that all the variables have proper boundary alignment.

Proper alignment is achieved either by arranging the variables in a fixed descending order according to length, or by constructing the block so that dummy variables force proper alignment. If the fixed order is used, the variables must appear in this order

```
length of 16 (complex)
length of 8  (complex or real)
length of 4  (real or integer or logical)
length of 2  (integer)
length of 1  (logical)
```

If the fixed order is not used, proper alignment can be ensured by constructing the block so that the displacement of each variable can be evenly divided by the reference number associated with the variable. (Displacement is the number of storage locations from the beginning of the block to the first storage location of the variable.) The following list shows the reference number for each type of variable.

Type of Variable	Length Specification	Reference Number
Logical	1	1
	4	4
Integer	2	2
	4	4
Real	4	4
	8	8
Complex	8	8
	16	8

The first variable in every COMMON block is positioned as if its length specification were eight. Therefore, a variable of any length may be the first assigned within a block. To obtain the proper alignment for other variables in the same block, it may be necessary to add a dummy variable to the block. For example, the variables A, I, and CMPLX are REAL*4, INTEGER*4, and COMPLEX*8, respectively, and form a COMMON block that is defined as

```
COMMON A, I, CMPLX
```

Then, the displacement of these variables within the block is illustrated by

A	I	CMPLX
4 storage locations	4 storage locations	8 storage locations
displacement 0 storage locations	displacement 4 storage locations	displacement 8 storage locations

The displacements of I and CMPLX are evenly divisible by their reference numbers. However, if I were an integer with a length specification of 2, then CMPLX is not properly aligned (its displacement of 6 is not evenly divisible by its reference number of 8). In this case, proper alignment is ensured by inserting a dummy variable with a length specification of 2 either between A and I or between I and CMPLX.

EQUIVALENCE Statement

General Form

```
EQUIVALENCE (a, b, c, ...), (d, e, f,...)
```

where a, b, c, d, e, f,... are variables that may be subscripted. The subscripts may have two forms: If the variable is singly subscripted it refers to the position of the variable in the array (i.e., first variable, 25th variable, etc.); if the variable is multisubscripted, it refers to the position in the array in the same manner as the position is referred to in an arithmetic statement

The EQUIVALENCE statement provides the option for controlling the allocation of data storage within a single program or subprogram. It is analogous to the option of using the COMMON statement to control the allocation of data storage among several programs. When the logic of the program permits, the number of storage locations used can be reduced by causing locations to be shared by two or more variables of the same or differing types and lengths. The EQUIVALENCE statement cannot be used to obtain mathematical equality of two variables.

Example 1:

```
DIMENSION B(5), C(10, 10), D(5, 10, 15)
EQUIVALENCE (A, B(1), C(5,3)), (D(5,10,2), E)
```

Explanation: This EQUIVALENCE statement indicates that the variables A,B(1), and C(5,3) are assigned to the same storage locations; also that D(5,10,2) and E are assigned to the same storage locations. In this case, the subscripted variables refer to the position in an array in the same manner as the position is referred to in an arithmetic statement. Note: Variables or arrays that are not mentioned in an EQUIVALENCE statement are assigned to unique storage locations. The EQUIVALENCE statement must not contradict itself or any previously established equivalences. For example, the further equivalence specification of B(2) with any other element of the array C, other than C(6,3), is invalid.

Example 2:

```
DIMENSION B(5), C(10, 10), D(5, 10, 15)
EQUIVALENCE (A, B(1), C(25)), (D(100), E)
```

Explanation: This EQUIVALENCE statement indicates that the variable A, the first variable in the array B, namely B(1), and the 25th variable in the array C, namely C(5,3), are to be assigned the same storage locations. Also, it also specifies that D(100), i.e., D(5,10,2), and E are to share the same storage locations. Note: The effects of the EQUIVALENCE statements in examples 1 and 2 are the same.

Variables that are brought into COMMON through EQUIVALENCE statements may increase the size of the block, as indicated by these statements:

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

This would cause a common area to be established containing the variables A, B, and C. The EQUIVALENCE statement would then cause the variable D(1) to share the storage location with B, D(2) would share with C, and D(3) would extend the size of the common area, in this manner

```
A        (lowest location of the common area)
B, D(1)
C, D(2)
   D(3)  (highest location of the common area)
```

Since arrays must be stored in consecutive forward locations, a variable may not be made equivalent to another variable of an array in such a way as to cause the array to extend before the beginning of the common area. For example, this EQUIVALENCE statement is invalid

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

because it would force D(1) to precede A, as follows:

```
   D(1)
A, D(2)   (lowest location of the common area)
B, D(3)
C         (highest location of the common area)
```

Programming Considerations

Two variables in one COMMON block or in two different COMMON blocks may not be made equivalent. Variables in an equivalence group may be in any order. However, considerable object-time efficiency is lost unless the programmer ensures that all the variables have proper boundary alignment.

Proper alignment is achieved either by arranging the variables in a fixed descending order according to length, or by constructing the group so that dummy variables force proper alignment. If the fixed order is used, the variables must appear in this order:

```
length of 16 (complex)
length of 8  (complex or real)
length of 4  (real or integer or logical)
length of 2  (integer)
length of 1  (logical)
```

If the fixed order is not used, proper alignment can be ensured by constructing the group so that the displacement of each variable in the group can be evenly divided by the reference number associated with the variable. (Displacement is the number of storage locations from the beginning of the group to the first storage location of the variable.) (The reference numbers for each type of variable are given under "COMMON Statement.") The first variable in each group is positioned as if its length specification were eight.

For example, the variables A, I, and CMPLX are REAL*4, INTEGER*4, and COMPLEX*8, respectively, and are defined as

```
DIMENSION A(10), I(16), CMPLX(5)
EQUIVALENCE (A(1), I(7), CMPLX(1))
```

Then, the displacement of these variables within the group is illustrated by

```
I(1)                          64 storage locations          I(16)

              A(1)            40 storage locations          A(10)

              CMPLX(1)        40 storage locations        CMPLX(5)

displacement  displacement
0 storage     24 storage
locations     locations
```

The displacements of A and CMPLX are evenly divisible by their reference numbers. However, if the EQUIVALENCE statement were written as

```
EQUIVALENCE (A(1), I(6), CMPLX(1))
```

then CMPLX is not properly aligned (its displacement of 20 is not evenly divisible by its reference number, 8).

It is sometimes desirable to write a program that, at various points, requires the same computation to be performed with different data for each calculation. Writing that program would be simplified if the statements required to perform the computation could be written only once and then could be referred to freely, with each subsequent reference having the same effect as though these instructions were written at the point in the program where the reference was made.

For example, to take the cube root of a number, a program must be written. If a general program were written to take the cube root of any number, it would be desirable to be able to combine that program (or subprogram) with other programs where cube root calculations are required.

The FORTRAN language provides for the above situation through the use of subprograms. There are three classes of subprograms: Statement Functions, FUNCTION subprograms, and SUBROUTINE subprograms. Also, there is a group of FORTRAN-supplied subprograms (see Appendix D).

The first two classes of subprograms are called functions, which differ from SUBROUTINE subprograms, in that functions return at least one value to the calling program; SUBROUTINE subprograms need not return any.

NAMING SUBPROGRAMS

A subprogram name consists of from one through six alphameric characters, the first of which must be alphabetic. A subprogram name may not contain special characters (see Appendix B). Certain subprogram names may be typed (as variables are) according to the following rules.

1. Type declaration of a statement function - Such declaration may be accomplished in one of three ways: by the predefined convention, by the IMPLICIT statement, or by the Explicit specification statements. Thus, the same rules for declaring the type of variables apply to Statement Functions.

2. Type declaration of FUNCTION subprograms - Such declaration may be made in one of three ways: by the predefined convention, by the IMPLICIT statement, or by an explicit specification (see "Type Specification of the FUNCTION Subprogram").

3. Type declaration of a subroutine subprogram - The type of a SUBROUTINE subprogram cannot be defined, because the results that are returned to the calling program are dependent only on the type of the variable names appearing in the argument list of the calling program and/or the implicit arguments in COMMON.

For the FUNCTION and SUBROUTINE type of subprograms, a reference or call by subprogram name gives the standard entry point. However, references or calls can also be made on the basis of other entry points defined in ENTRY statements.

FUNCTIONS

A function is a statement of the relationship between a number of variables. To use a function in FORTRAN, it is necessary to

1. Define the function (i.e., specify what calculations are to be performed),

2. Refer to the function by name, where required in the program.

Function Definition

The three steps in the definition of a function are

1. The function must be assigned a unique name by which it may be called (see "Naming Subprograms").

2. The arguments of the function must be stated.

3. The procedure for evaluating the function must be stated.

Items 2 and 3 are discussed in detail in the sections dealing with the specific subprogram ("Statement Functions," "FUNCTION Subprograms," etc.).

Function Reference

The name of a function, appearing in any FORTRAN arithmetic expression, refers to the function. Thus, the appearance of a function, with its arguments in parentheses, causes the computations to be performed as indicated by the function definition. The resulting quantity replaces the function reference in the expression and assumes the type of the function. The type and length of the name used for the reference must agree with the type and length of the name used in the definition.

STATEMENT FUNCTIONS

Statement functions are defined by a single arithmetic or logical assignment statement within the program in which they appear. For example,

```
FUNC(A,B) = 3.*A+B**2.+X+Y+Z
```

defines the statement function FUNC, where FUNC is the function name and A and B are the function arguments.

The expression on the right defines the computations that are to be performed when the function is used in an arithmetic statement. This function might be used in this way

```
C = FUNC(D,E)
```

which is equivalent to

```
C = 3.*D+E**2.+X+Y+Z
```

Note the correspondence between A and B in the function definition statement and D and E in the arithmetic statement. The quantities A and B, enclosed in parentheses following the function name, are the arguments of the function. They are dummy variables for which the quantities D and E, respectively are substituted when the function is used in an arithmetic statement.

General Form
name (a,b,...,n) = expression where name is any subprogram name (see "Naming Subprograms"). a,b,...,n are distinct (within the same statement) nonsubscripted variables expression is any arithmetic or logical expression that does not contain subscripted variables; any statement functions appearing in this expression must be defined previously

The actual arguments must correspond in order, number, and type to the dummy arguments. At least one argument must be used. A maximum of 15 variables appearing in the expression may be used as arguments of the function.

Note: All Statement Function definitions to be used in a program must precede the first executable statement of the program.

Examples: Valid statement function definitions:

```
SUM(A,B,C,D) = A+B+C+D
FUNC(Z) = A+X*Y*Z
AVG(A,B,C,D) = (A+B+C+D)/4
ROOT(A,B,C) = SQRT(A**2+B**2+C**2)
VALID(A,B) = .NOT.A.OR.B
```

Note: The same dummy arguments may be used in more than one Statement Function definition and, as variables, outside Statement Function definitions.

Invalid statement function definitions

```
SUBPRG(3,J,K)=3*I+J**3          (arguments must be variables)
SOMEF(A(I),B)=A(I)/B+3.         (arguments must be nonsub-
                                 scripted)
SUBPROGRAM(A,B)=A**2+B**2       (function name exceeds limit
                                 of six characters)
3FUNC(D)=3.14*E                 (function name must begin with
                                 an alphabetic character)
ASF(A)=A+B(I)                   (subscripted variable in the
                                 expression)
```

Valid statement function references

```
NET = GROS - SUM(TAX, FICA, HOSP, MISC)
ANS = FUNC(RESULT)
GRADE = AVG(LAB, LECTUR, SUM(TEST1, TEST2, TEST3, TEST4), FACTOR)
```

Invalid statement function references

```
WRONG = SUM(TAX,FICA)           (number of arguments
                                 does not agree with
                                 above definition)
MIX = FUNC(I)                   (mode of argument
                                 does not agree with
                                 above definition)
```

FUNCTION SUBPROGRAMS

The FUNCTION subprogram is a FORTRAN subprogram consisting of any number of statements. It is an independently written program that is executed wherever its name appears in another program.

General Form

```
FUNCTION name (a1,a2,a3,...,an)
    .
    .
    .
RETURN
    .
    .
    .
END
```

where *name* is subprogram name (see "Naming Subprograms")

$a_1, a_2, a_3, \ldots, a_n$ are nonsubscripted variable, array, or dummy names of SUBROUTINE or other FUNCTION subprograms. There must be at least one argument in the argument list. (Arguments in a FUNCTION or SUBROUTINE subprogram may be enclosed in slashes within the commas. This form is equivalent to the normal format without the slashes.)

Since the FUNCTION is a separate subprogram, the variables and statement numbers within it do not relate to any other program.

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

The arguments of the FUNCTION subprogram (i.e., $a_1, a_2, a_3, \ldots, a_n$) may be considered as dummy variable names. These are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. The actual arguments may be: any type of constant, any type of subscripted or nonsubscripted variable, an array name, an arithmetic or logical expression, or the name of another subprogram. The actual arguments must correspond in number, order, type, and length to the dummy arguments. The array size must also be the same, except when adjustable dimensions are used. If the actual argument corresponds to a dummy argument that is defined or redefined in the subprogram, the argument must be a variable name, subscripted variable name, or array name. All arguments in a subprogram refer to the storage area assigned to the arguments by the calling program.

The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the FUNCTION subprogram is illustrated by

Example 1:

Calling Program

```
      .
      .
      .
A = SOMEF(B,C)
      .
      .
      .
```

FUNCTION Subprogram

```
FUNCTION SOMEF(X,Y)
SOMEF = X/Y
RETURN
END
```

Explanation: The value of the variable B of the calling program is used in the subprogram as the value of the dummy variable X; the value of C is used in place of the dummy variable Y. Thus if B = 10.0 and C = 5.0, then A = B/C, which is equal to 2.0.

The name of the function must be assigned a value at least once in the subprogram as the argument of a CALL statement, as a DO variable, as the variable name on the left side of an arithmetic statement, or in an input list (READ statement) within the subprogram.

Example 2:

Calling Program

```
      .
      .
      .
ANS = ROOT1*CALC(X,Y,I)
      .
      .
      .
```

FUNCTION Subprogram

```
FUNCTION CALC (A,B,J)
        .
        .
        .
I = J*2
        .
        .
        .
CALC = A**I/B
        .
        .
        .
RETURN
END
```

Explanation: The values of X, Y, and I are used in the FUNCTION subprogram as the values of A, B, and J, respectively. The value of CALC is computed, and this value is returned to the calling program, where the value of ANS is computed. The variable I in the argument list of CALC in the calling program is not the same as the variable I appearing in the subprogram.

When a dummy argument is an array name, an appropriate DIMENSION or Explicit specification statement must appear in the FUNCTION subprogram. None of the dummy arguments may appear in an EQUIVALENCE or COMMON statement within the subprogram nor may they be given initial data values.

Type Specification of the FUNCTION Subprogram

In addition to the three ways of declaring the type of a FUNCTION name (i.e., predefined convention, IMPLICIT statement, Explicit specification statement), there is the option of explicitly specifying the type of a FUNCTION name within the FUNCTION statement.

General Form
type FUNCTION name*s (a_1,a_2,a_3,...,a_n)
where type is INTEGER, REAL, COMPLEX, or LOGICAL
name is the name of the FUNCTION subprogram
*s is optional and represents one of the permissible length specifications for its associated type
a_1,a_2,a_3,...,a_n are nonsubscripted variable, array, or dummy names of SUBROUTINE or other FUNCTION subprograms. (There must be at least one argument in the argument list)

Example 1:

```
REAL FUNCTION SOMEF (A,B)
   .
   .
   .
SOMEF = A**2 + B**2
   .
   .
   .
RETURN
END
```

Example 2:

```
INTEGER FUNCTION CALC*2 (X,Y,Z)
   .
   .
   .
CALC = X+Y+Z**2
   .
   .
   .
RETURN
END
```

Explanation: The FUNCTION subprograms SOMEF and CALC in examples 1 and 2 are declared as type REAL (length 4) and INTEGER (length 2), respectively.

RETURN and END Statements in a FUNCTION Subprogram

All FUNCTION subprograms must contain both an END and at least one RETURN statement. The END statement specifies, for the compiler, the end of the subprogram; the RETURN statement signifies a logical conclusion of the computation and returns any computed value and control to the calling program. More than one RETURN statement may be used in a FORTRAN subprogram.

Example:

```
      FUNCTION DAV (D,E,F)
      IF (D-E) 10, 20, 30
10    A = D+2.0*E
          .
          .
          .
5     A = F+2.0*E
          .
          .
          .
20    DAV = A+B**2
          .
          .
          .
      RETURN
30    DAV = B**2
          .
          .
          .
      RETURN
      END
```

Multiple Entry into a FUNCTION Subprogram

The standard entry into a FUNCTION subprogram is made by a function reference in an arithmetic expression, when the function reference uses the name defined in the FUNCTION statement. Entry is made at the first executable statement following the FUNCTION statement.

It is also possible to enter a FUNCTION subprogram by a function reference to a name defined in an ENTRY statement in the FUNCTION subprogram. Entry is made at the first executable statement following the ENTRY statement. The name given in the FUNCTION statement is used to return the value of the function to the point of reference, rather than the name of the ENTRY statement.

SUBROUTINE SUBPROGRAMS

The SUBROUTINE subprogram is similar to the FUNCTION subprogram in many respects: the rules for naming FUNCTION and SUBROUTINE subprograms are the same, they both require an END statement, and they both contain the same sort of dummy arguments. Like the FUNCTION subprogram, the SUBROUTINE subprogram is a set of commonly used computations, but it does not need to return any results to the calling program, as does the FUNCTION subprogram.

The CALL statement (discussed later in this section) is used in a main program or another subprogram to invoke a SUBROUTINE subprogram.

Since the SUBROUTINE is a separate subprogram, the variables and statement numbers within it do not relate to any other program.

The SUBROUTINE statement must be the first statement in the subprogram. The SUBROUTINE subprogram may contain any FORTRAN statement except a FUNCTION statement, another SUBROUTINE statement, or a BLOCK DATA statement. If an IMPLICIT statement is used in a SUBROUTINE subprogram, it must immediately follow the SUBROUTINE statement.

General Form

```
SUBROUTINE name (a1,a2,a3,...,an)
    .
    .
    .
RETURN

END
```

where *name* is the subprogram name (see "Naming Subprograms")

$a_1, a_2, a_3, \ldots, a_n$ are arguments. (There need not be any.) Each argument used must be a nonsubscripted variable or array name, the dummy name of another SUBROUTINE or FUNCTION subprogram, or of the form * where the character "*" denotes a return point specified by a statement number in the calling program

The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. Any arguments so used must appear on the left side of an arithmetic statement or in an input list within the subprogram, as arguments of a CALL statement, as DO variables, or as arguments in a function reference. The SUBROUTINE name must not appear in any other statement in the SUBROUTINE subprogram.

The arguments (a_1, a_2, a_3,...,a_n) may be considered as dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. The actual arguments must correspond in number, order, type, and length to the dummy arguments. The array size must also be the same except when adjustable dimensions are used. Dummy arguments may not appear in an EQUIVALENCE or COMMON statement within the subprogram nor may they be given initial data values.

Example: The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the SUBROUTINE subprogram is illustrated in this example. The object of the subprogram is to copy one array directly into another.

```
Main Program                    SUBROUTINE Subprogram

DIMENSION X(100),Y(100)
   .                            SUBROUTINE COPY(A,B,N)
   .                            DIMENSION A (100),B(100)
   .                            DO 10 I = 1, N
CALL COPY (X,Y,K)            10 B(I) = A (I)
   .                            RETURN
   .                            END
   .
```

CALL Statement

The CALL statement is used only to call a subroutine subprogram.

General Form
CALL name (a_1,a_2,a_3,...,a_n)
where name is the subroutine's subprogram name, or a name defined in an ENTRY statement in the SUBROUTINE subprogram
a_1,a_2,a_3,...,a_n are the actual arguments that are being supplied to the subroutine subprogram; each may be of the form &n where n is a statement number (see "RETURN Statements in a SUBROUTINE Subprogram")

The standard entry into a SUBROUTINE subprogram is made by a CALL statement that refers to that subroutine's subprogram name. Entry is made at the first executable statement following the SUBROUTINE statement. Also, it is possible to enter a SUBROUTINE subprogram by a CALL statement that refers to a name defined in an ENTRY statement in the SUBROUTINE subprogram. The ENTRY statement is described below.

Examples:

```
CALL OUT
CALL MATMPY (X,5,40,Y,7,2)
CALL QDRTIC (X,Y,Z,ROOT1,ROOT2)
CALL SUB1 (X+Y*5,'ABDF',SINE)
```

The CALL statement transfers control to the subroutine subprogram and replaces the dummy variables with the value of the actual arguments that appear in the CALL statement. The arguments in a CALL statement may be: any type of constant, any type of subscripted or nonsubscripted variable, an arithmetic expression, the name of a subprogram, or a statement number (see "RETURN Statements in a SUBROUTINE Subprogram").

The arguments in a CALL statement must agree in number, order, and type with the corresponding arguments in the subroutine subprogram. The

array sizes must also be the same in the subroutine and the calling programs, except when adjustable dimensions are used (see "Adjustable Dimensions"). If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, subscripted variable name, or array name. All arguments in a subprogram refer to the storage area assigned to the arguments by the calling program.

RETURN Statement in a SUBROUTINE Subprogram

General Form
RETURN RETURN i where i is an integer constant or variable of length 4 whose value, say n, denotes the nth statement number in the argument list of a SUBROUTINE statement

The normal sequence of execution following the RETURN statement of a SUBROUTINE subprogram is to the next statement following the CALL in the calling program. It is also possible to return to any numbered statement in the calling program by using a return of the type where i is an integer constant or variable. Returns of the type RETURN may be made in either a SUBROUTINE or FUNCTION subprogram (see, "RETURN and END Statements in a FUNCTION Subprogram"). Returns of the type RETURN i may only be made in a SUBROUTINE subprogram. In a main program, a RETURN statement performs the same function as a STOP statement.

Example:

```
      Calling Program                      Subprogram

          .                                SUBROUTINE SUB (X,Y,Z,*,*)
          .                                    .
          .                                    .
   10 CALL SUB (A,B,C,&30,&40)                 .
   20 Y = A + B                       100   IF (R) 200,300,400
          .                           200   RETURN
          .                           300   RETURN 1
          .                           400   RETURN 2
   30 Y = A + C                             END
          .
          .
          .
   40 Y = B + C
          .
          .
          .
      END
```

Explanation: 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 to statement 20, 30, or 40, if R is less than, equal to, or greater than zero, respectively.

A CALL statement that uses a RETURN i form may be best understood by comparing it to a CALL and computed GO TO statement in sequence. For example,

```
CALL SUB (P,&20,Q,&35,R,&22)
```

is equivalent to:

```
CALL SUB (P,Q,R,I)
GO TO (20,35,22),I
```

where the index I is assigned a value of 1, 2, or 3 in the called subprogram.

ENTRY Statement

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

A subprogram may also be entered (either SUBROUTINE or FUNCTION) by a CALL statement or a function reference that references an ENTRY statement in the subprogram. Entry is made at the first executable statement following the ENTRY statement.

General Form
ENTRY name (a_1,a_2,a_3,...,a_n)
where name is the name of an entry point containing from one to six alphabetic and/or numeric characters, the first of which is alphabetic
a_1,a_2,a_3,...,a_n are the dummy arguments corresponding to an actual argument in a CALL statement or in a function referrence

ENTRY statements 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 must the ENTRY statements 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 references. Entry may not be made into the range of a DO; further, a subprogram may not reference itself directly or through any of its entry points. This statement is regarded as nonexecutable within its subprogram, . If it appears in a function subprogram the name given in the FUNCTION statement is still used to return the value of the function to the point of reference, rather than the name of the ENTRY statement.

Example 1:

Calling Program

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

Subprogram

```
SUBROUTINE SUB1 (U,V,W,X,Y,Z)
    .
    .
    .
U = V
    .
    .
    .
ENTRY SUB2 (T,U,V)
    .
```

```
3 CALL SUB3                          .
    .                                .
    .                            ENTRY SUB3
    .                                .
                                     .
                                     .
                                 END
```

Explanation: 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 causes entry into the called program, starting with the first executable statement following the ENTRY SUB2(T, U,V) and ENTRY SUB3 statements, respectively.

Entry into a subprogram initializes all references in the entire subprogram to items in the argument list. Return from a subprogram is made by way of the entry point referenced. ENTRY statements may only appear in FUNCTION or SUBROUTINE subprograms. The dummy arguments in a subprogram may appear in any statement if they first appear as dummy arguments in a FUNCTION, SUBROUTINE, or ENTRY statement. This is a valid example

```
SUBROUTINE SUB (X,Y,Z,I)
     .
     .
     .
ENTRY SUB1 (A,B)
     .
     .
     .
     .
C = A+B
     .
     .
     .
```

Example 2:

Calling Program

```
     .
     .
     .
CALL SUB1 (A,B,C,D,E,F)
     .
     .
     .
CALL SUB2(G,&10,&20)
     .
     .
     .
CALL SUB3(&10,&20)
5    Y =A+B
     .
     .
     .
10   Y = C+D
20   Y = E+F
     .
     .
     .
```

Subprogram

```
     SUBROUTINE SUB1 (U,V,W,X,Y,Z)
     RETURN
     ENTRY SUB2 (T,*,*)
     T=V*W+U
     ENTRY SUB3 (*,*)
     T=T+X*Y**Z
50   IF (U-X) 100, 200, 300
100  RETURN 1
200  RETURN 2
300  RETURN
     END
```

Explanation: A call to SUB1 merely performs initialization. Subsequent calls to SUB2 and SUB3 result in execution of different sections of the subroutine SUB1. Then, depending upon the result of the arithmetic IF

at statement 50, return is made to the calling program at statement 10, 20, or the next executable statement following the CALL.

Additional Rules for using ENTRY

1. A CALL may only change the value of explicit arguments (or implicit arguments in COMMON). It cannot affect the value of those that were initialized by some previous CALL.

2. If a name is identified as a dummy argument only by its appearance in a given ENTRY statement, no use of that dummy argument may appear in statements preceding (physically) the ENTRY statement.

3. The appearance of an ENTRY statement does not alter the rules regarding the placement of Statement Functions in subprograms.

4. If new dimensions for an adjustable dimension array are to be passed to a subprogram with an ENTRY, the array name must appear in the argument list of the ENTRY.

The EXTERNAL Statement

General Form
EXTERNAL a,b,c,...
where a,b,c,... are names of subprograms that are used as arguments in other subprograms

If a FORTRAN-supplied in-line function is used in an EXTERNAL statememt, it is not expanded in-line; the function is assumed to be part of a library. (The FORTRAN-supplied in-line and out-of-line functions are given in Appendix D.)

The name of any subprogram that is used as an argument in another subprogram must appear in an EXTERNAL statement. For example, assume that SUB and MULT are subprogram names in the following statements.

Example 1

```
Calling Program                    Subprogram

     .                             SUBROUTINE SUB(X,Y,Z)
     .                             IF (X) 4,6,6
     .                        4    D = Y (X,Z**2)
EXTERNAL MULT
     .                                  .
     .                                  .
     .                                  .
CALL SUB (A, MULT,C)             6    RETURN
     .                             END
     .
     .
```

Explanation: The subprogram name MULT is used as an argument in the subprogram SUB. The subprogram name MULT is passed to the dummy variable Y; the variables A and C are passed to the dummy variables X and Z. The subprogram MULT will be called and executed only if the value of A is negative.

Example 2

```
      .
      .
      .
CALL SUB (A,B,MULT (C,D),37)
      .
      .
      .
```

Explanation: An EXTERNAL statement is not required because the subprogram named MULT is not an argument; it is executed first and the result becomes the argument.

BLOCK DATA SUBPROGRAM

To enter data into a COMMON block, a separate subprogram must be written. This separate subprogram contains only the DATA, COMMON, DIMENSION, EQUIVALENCE, and Type statements associated with the data being defined. Data may be entered into labeled (named), but not unlabeled, COMMON by the BLOCK DATA subprogram.

General Form
BLOCK DATA . . . END

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 below does not appear in the DATA statement:

```
BLOCK DATA
COMMON/ELN/C,A,B/RMG/Z,Y
REAL B(4)/1.0,1.2,2*1.3/,Z*8(3)/3*7.64980825D0/
COMPLEX C/(2.4,3.769)/
END
```

4. Data may be entered into more than one COMMON block in a single BLOCK DATA subprogram.

5. No element may have more than one initial value assigned in the same program.

APPENDIX A: FORTRAN COMPARISON

This appendix contains a description of the differences in the FORTRAN language supported by IBM OS and OS/VS, and by the IBM Time Sharing System. The FORTRAN language for IBM OS and OS/VS is described in IBM FORTRAN IV Language, GC28-6515.

1. Extensions

   TSS -- Does not allow generalized subscripts and direct access I/O statements; no list-directed I/O; no free format input.
   OS and OS/VS -- Allows the above.

2. Call by Value

   TSS -- Treats all arguments as call-by-name whether or not they are enclosed in slashes.
   OS and OS/VS -- Treats arguments not enclosed in slashes, and not declared as an array, as call-by-value.

3. Dummy Arguments

   TSS -- Dummy arguments may not appear in any statement until defined as such in an ENTRY, SUBROUTINE, or FUNCTION statement.
   OS and OS/VS -- Restriction holds only for executable statements.

4. ENTRY in FUNCTION Subprograms

   TSS -- The name of a FUNCTION subprogram must be used to return the value of the function, even though entry was made through an ENTRY statement.
   OS and OS/VS -- The ENTRY name may be used to return the value of the function.

Alphabetic Characters	EBCDIC or BCD Card Punches	Numeric Characters	EBCDIC or BCD Card Punches
A	12-1	0	0
B	12-2	1	1
C	12-3	2	2
D	12-4	3	3
E	12-5	4	4
F	12-6	5	5
G	12-7	6	6
H	12-8	7	7
I	12-9	8	8
J	11-1	9	9
K	11-2		
L	11-3		
M	11-4		
N	11-5	Special Characters	EBCDIC Card Punches / BCDIC Card Punches
O	11-6		
P	11-7		
Q	11-8	+	12-6-8 / 12
R	11-9	-	11 / 11
S	0-2	/	0-1 / 0-1
T	0-3	=	6-8 / 3-8
U	0-4	.	12-3-8 / 12-3-8
V	0-5	)	11-5-8 / 12-4-8
W	0-6	*	11-4-8 / 11-4-8
X	0-7	,(comma)	0-3-8 / 0-3-8
Y	0-8	(**	12-5-8 / 0-4-8
Z	0-9	'(apostrophe)	5-8 / 4-8
$*	11-3-8	blank	(no punch) / (no punch)

Source programs are coded in either BCD or EBCDIC character codes; mixing the two, however, is not allowed.
*Considered an alphabetic character in EBCDIC only.
**Considered a special character in EBCDIC only.

APPENDIX C: OTHER FORTRAN STATEMENTS ACCEPTED BY TSS FORTRAN IV

This appendix describes features of previously implemented FORTRAN IV languages that are incorporated into the IBM Time Sharing System FORTRAN IV language. The inclusion of these language facilities allows existing FORTRAN programs to be recompiled for use in IBM Time Sharing System with little or no reprogramming.

READ Statement

General Form
READ b,list where b, is the statement number or array name of the FORMAT statement describing the data list is a series of variable or array names, separated by commas, which may be indexed and incremented; they specify the number of items to be read and the storage locations into which the data is placed

This statement causes data to be read from the data set associated with the system input.

PUNCH Statement

General Form
PUNCH b, list where b is the statement number or array name of the FORMAT statement describing the data list is a series of variable or array names, separated by commas, which may be indexed and incremented; they specify the number of items to be written and the storage locations from which the data is taken

The PUNCH statement causes data to be written in the data set associated with the system output.

PRINT Statement

General Form
PRINT b, list where b is the statement number or array name of the FORMAT statement describing the data list is a series of variable or array names, separated by commas, which may be indexed and incremented; they specify the number of items to be written and the locations in storage from which the data is taken

The PRINT statement causes data to be written in the data set associated with the system output.

DATA Initialization Statement

General Form
DATA $v_1,\ldots,v_n/i_1*d_1,\ldots,i_n*d_n/,v_{n+1},\ldots,v /i_{n+1}*d_{n+1},\ldots,i *d /,\ldots$
where $v_1,\ldots,v$ are variables, subscripted variables (in which case, the subscripts must be integer constants), or array names $d_1,\ldots,d$ are values representing integer, real, complex, logical, or literal hexadecimal data constants $i_1,\ldots,i$ represent unsigned integer constants indicating the number of consecutive variables that are to be assigned the value of $d_1,\ldots,d$

A data initialization statement is used to define initial values of variables and arrays. There must be a one-for-one correspondence between these variables (i.e., $v_1,\ldots,v$) and the data constants (i.e., $d_1,\ldots,d$).

Example 1:

```
DIMENSION D(5,10)
DATA A, B, C/5.0,6.1,7.3/,D/25*1.0,25*2.0/
```

Explanation: The DATA statement indicates that the variables A, B, and C are to be initialized to the values 5.0, 6,1, and 7.3, respectively. Also, the statement specifies that the first 25 variables in the array D are to be initialized to the value 1.0, and the second 25 to the value 2.0.

Example 2:

```
DIMENSION A(5), B(3,3), L(4)
DATA A/5*1.0/, B/9*2.0/, L/4*.TRUE./, C/'FOUR'/
```

Explanation: The DATA statement specifies that all the variables in the arrays A and B are to be initialized to the values 1.0 and 2.0, respectively. All the logical variables in the array L are initialized to the value .TRUE.. The letters T and F may be used as an abbreviation for .TRUE. and .FALSE., respectively. Also, the variable C is initialized with the literal data constant FOUR.

An initially defined variable, or variable of an array, may not be in blank common; however, in a labeled common block, they may be initially defined only in a block data subprogram (see "SUBPROGRAMS").

DOUBLE PRECISION Statement

General Form
DOUBLE PRECISION a,b,c,...
where a,b,c,... are variable names that may be dimensioned in the statement, or function names

The DOUBLE PRECISION statement explicitly specifies that the variables a,b,c,... are of type double precision. This statement overrides any specification of a variable made by either the predefined convention or the IMPLICIT statement. This specification is identical to that of type REAL*8.

Also, FUNCTION subprograms may be typed double precision, in this way

DOUBLE PRECISION FUNCTION name (a_1,a_2,a_3,...,a_n)

The FORTRAN supplied subprograms are of either of two types: mathematical subprograms and service subprograms. The mathematical subprograms correspond to a FUNCTION subprogram; the service subprograms correspond to a SUBROUTINE subprogram. Appendix D lists the in-line and out-of-line mathematical FUNCTION subprograms. An in-line subprogram is inserted by the FORTRAN compiler at any point in the program where the function is referenced. An out-of-line subprogram is located on a library. A detailed description of out-of-line mathematical subprograms and service subprograms is given in FORTRAN IV Library Subprograms.

MATHEMATICAL SUBPROGRAMS

All functions are used as described in the section "FUNCTION Subprograms" -- i.e., A = AMOD(X_1,X_2), where A is the value and X_1 and X_2 are the arguments.

Table 4. Mathematical function subprograms (part 1 of 3)

Function	Name	Definition/Usage	In-Line (I) Out-of-Line (O)	No. of Arg.	Type of Arguments	Function
Exponential	EXP	earg	O	1	Real *4	Real *4
	DEXP	earg	O	1	Real *8	Real *8
	CEXP	earg	O	1	Complex *8	Complex *8
	CDEXP	earg	O	1	Complex *16	Complex *16
		A=EXP(X_1)				
Natural Logarithm	ALOG	ln (Arg)	O	1	Real *4	Real *4
	DLOG	ln (Arg)	O	1	Real *8	Real *8
	CLOG	ln (Arg)	O	1	Complex *8	Complex *8
	CDLOG	ln (Arg)	O	1	Complex *16	Complex *16
		A=ALOG(X_1)				
Common Logarithm	ALOG10	$\log_{10}$ (Arg)	O	1	Real *4	Real *4
	DLOG10	$\log_{10}$ (Arg)	O	1	Real *8	Real *8
		A=ALOG10(X_1)				
Arcsine	ARSIN	arcsin (Arg)	O	1	Real *4	Real *4
	DARSIN	A=ARSIN(X_1)			Real *8	Real *8
Arccosine	ARCOS	arcos (Arg)	O	1	Real *4	Real *4
	DARCOS	A=ARCOS(X_1)			Real *8	Real *8
Arctangent	ATAN	arctan (Arg)	O	1	Real *4	Real *4
	ATAN2	arctan (Arg_1/Arg_2)	O	2	Real *4	Real *4
	DATAN	arctan (Arg)	O	1	Real *8	Real *8
	DATAN2	arctan (Arg_1/Arg_2)	O	2	Real *8	Real *8
		A=ATAN(X_1)				
Trigonometric Sine (Argument in radians)	SIN	sin(Arg)	O	1	Real *4	Real *4
	DSIN	sin(Arg)	O	1	Real *8	Real *8
	CSIN	sin(Arg)	O	1	Complex *8	Complex *8
	CDSIN	sin(Arg)	O	1	Complex *16	Complex *16
		A=SIN(X_1)				
Trigonometric Cosine (Argument in radians)	COS	cos(Arg)	O	1	Real *4	Real *4
	DCOS	cos(Arg)	O	1	Real *8	Real *8
	CCOS	cos(Arg)	O	1	Complex *8	Complex *8
	CDCOS	cos(Arg)	O	1	Complex *16	Complex *16
		A=COS(X_1)				
Trigonometric Tangent	TAN	tan (Arg)	O	1	Real *4	Real *4
	DTAN	A=TAN(X_1)			Real *8	Real *8
Trigonometric Cotangent	COTAN	cotan (Arg)	O	1	Real *4	Real *4
	DCOTAN	A=COTAN(X_1)			Real *8	Real *8
Square Root	SQRT	(Arg)	O	1	Real *4	Real *4
	DSQRT	(Arg)	O	1	Real *8	Real *8
	CSQRT	(Arg)	O	1	Complex *8	Complex *8
	CDSQRT	(Arg)	O	1	Complex *16	Complex *16
		A=SQRT(X_1)				
Hyperbolic Sine	SINH	sinh (Arg)	O	1	Real *4	Real *4
	DSINH	A=SINH(X_1)			Real *8	Real *8

Table 4. Mathematical function subprograms (part 2 of 3)

Function	Name	Definition/Usage	In-Line (I) Out-of-Line (O)	No. of Arg.	Type of Arguments	Function
Hyperbolic Cosine	COSH DCOSH	cosh (Arg) A=COSH(X_1)	O	1	Real *4 Real *8	Real *4 Real *8
Hyperbolic Tangent	TANH DTANH	tanh(Arg) tanh (Arg) A=TANH(X_1)	O O	1 1	Real *4 Real *8	Real *4 Real *8
Error Function	ERF DERF	$\frac{2}{\pi}\int_0^x e^{-u^2} du$ A=ERF(X_1)	O O	1 1	Real *4 Real *8	Real *4 Real *8
Complemented Error Function	ERFC DERFC	1-erf (x) A=ERFC(x_1)	O O	1 1	Real *4 Real *8	Real *4 Real *8
Gamma	GAMMA DGAMMA	$\int_0^\infty x-1\ -u\ u\ e\ du$ ANS=GAMMA(X_1)	O O	1 1	Real *4 Real *8	Real *4 Real *8
Log-gamma	ALGAMA DLGAMA	$\log_e \Gamma(x)$ A=ALGAMA(x_1)	O O	1 1	Real *4 Real *8	Real *4 Real *8
Remaindering	MOD AMOD DMOD	Arg_1 (mod Arg_2) A=MOD(X_1,X_2)	I I I	2 2 2	Integer *4 Real *4 Real *8	Integer *4 Real *4 Real *8
Absolute value	IABS ABS DABS	|Arg| A=ABS(X_1)	I I I	1 1 1	Integer *4 Real *4 Real *8	Integer *4 Real *4 Real *8
	CABS CDABS	$\|\sqrt{x^1+x^2}\|$ for x_1+x_2 A=CABS(X_1)	O O	1 1	Complex *8 Complex *16	Real *4 Real *8
Truncation	INT AINT IDINT	Sign of Arg times largest integer $\leq$|Arg| I=INT(X_1)	I I I	1 1 1	Real *4 Real *4 Real *8	Integer *4 Real *4 Integer *4
Largest value	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Max (Arg_1,Arg_2,...) A=AMAX0(X_1,X_2,..X_n)	I I I I I	≥2 ≥2 ≥2 ≥2 ≥2	Integer *4 Real *4 Integer *4 Real *4 Real *8	Real *4 Real *4 Integer *4 Integer *4 Real *8
Smallest value	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Min (Arg_1,Arg_2,...) A=AMIN0(X_1,X_2,..X_n)	I I I I I	≥2 ≥2 ≥2 ≥2 ≥2	Integer *4 Real *4 Integer *4 Real *4 Real *8	Real *4 Real *4 Integer *4 Integer *4 Real *8
Float	FLOAT DFLOAT	Convert from integer to real A=FLOAT(X_1)	I I	1 1	Integer *4 Integer *4	Real *4 Real *8
Fix	IFIX HFIX	Convert from real to integer I=IFIX(X_1)	I I	1 1	Real *4 Real *4	Integer *4 Integer *2
Transfer of sign	SIGN ISIGN DSIGN	Sign of Arg_2 times |Arg_1| A=SIGN(X_1,X_2)	I I I	2 2 2	Real *4 Integer *4 Real *8	Real *4 Integer *4 Real *8
Positive difference	DIM IDIM	Arg_1-Min(Arg_1,Arg_2) A=DIM(X_1,X_2)	I	2	Real *4 Integer *4	Real *4 Integer *4
Obtaining most significant part of a Real *8 argument	SNGL A=SNGL(X_1)		I	1	Real *8	Real *4
Obtain real part of complex argument	REAL A=SNGL(X_1)		I	1	Complex *8	Real *4

Table 4. Mathematical function subprograms (part 3 of 3)

Function	Name	Definition	In-Line (I) Out-of-Line (O)	No. of Arg.	Type of Arguments	Function
Obtain imaginary part of complex argument	AIMAG	A=AIMAG(X_1)	I	1	Complex *8	Real *4
Express a Real *4 argument in Real *8 form	DBLE	A=DBLE(X_1)	I	1	Real *4	Real *8
Express two real arguments in complex form	CMPLX DCMPLX	$C=Arg_1+iArg_2$ A=CMPLX(X_1,X_2)	I I	2 2	Real *4 Real *8	Complex *8 Complex *16
Obtain conjugate of a complex argument	CONJG DCONJG	C=X-iY For Arg=X+iY A=CONJG(X_1)	I I	1 1	Complex *8 Complex *16	Complex *8 Complex *16

SERVICE SUBPROGRAMS

MACHINE INDICATOR TESTS SUBPROGRAMS

In the list of pseudo machine-indicator test subroutines below, assume that i is an integer expression and that j is an integer variable. These subroutines are referred to by CALL statements.

SLITE (i): If i = 0, all sense lights will be turned off. If i = 1, 2, 3, or 4, the corresponding sense light will be turned on.

SLITET (i,j): Sense light i (equal to 1, 2, 3, or 4) will be tested and turned off. The variable j will be set to 1 if i was on, or j will be set to 2 if i was off.

Example: Assume that the program is to continue if sense light i is on and the results are to be written if sense light i is off. This can be done by using the logical IF statement or a computed GO TO statement:

```
      .
      .
      .
      CALL SLITET (3,KEN)
      GO TO (6, 17) ,KEN
17    WRITE (3, 26) (ANS (K) , K=1, 10)
6     CONTINUE
      .
      .
```

Explanation: When the statement CALL SLITET(3,KEN) is executed, the variable KEN is assigned the value 1 or 2 depending on whether sense light 3 is on or off, (and the sense light is turned off). If KEN is 1, statement 6 is executed next; if KEN is 2, statement 17 is executed.

OVERFL (j): j is set to 1 if a floating-point overflow condition exists, i.e., if the result of an arithmetic operation is greater than 16^{63}; j is set to 2 if neither an overflow condition or underflow condition exists; j is set to 3 if floating-point underflow condition exists, i.e., if the result of an arithmetic operation is less than 16^{-63}. The machine is left in a no-overflow condition. If a sequence of operations caused both overflow and underflow to occur, the value of j returned represents whichever of these two conditions occurred last.

DVCHK (j): If the divide check indicator is on, j is set to 1 and the divide check indicator is turned off; if the divide check indicator is off, j is set to 2.

THE EXIT, DUMP, AND PDUMP SUBPROGRAMS

EXIT Subprogram

A CALL to the EXIT subprogram terminates the execution of the object program.

DUMP Subprogram

A CALL to the DUMP subprogram

CALL DUMP (A_1,B_1,F_1,...,A_n,B_n,F_n)

causes the indicated limits of storage to be dumped and execution to be terminated.

1. A and B are variable data names that indicate the limits of storage to be dumped; either A or B may represent upper or lower limits.

2. F_n is an integer indicating the dump format desired

F_n =	
0	hexadecimal
1	logical *1
2	logical *4
3	integer *2
4	integer *4
5	real *4
6	real *8
7	complex *8
8	complex*16
9	literal character

3. If the argument F_n is omitted, it is assumed to be equal to 0, and the dump will be hexadecimal.

4. The arguments A and B should be in the same program (main program or subprogram) or same common block.

PDUMP Subprogram

A CALL to the PDUMP subprogram

CALL PDUMP (A_1,B_1,F_1,...,A_n,B_n,F_n)

causes the indicated limits of storage to be dumped and control to be returned to the calling program.

EXAMPLE PROGRAM 1

Example program 1 (Figure 2) is designed to find all the prime numbers between 1 and 1000. A prime number is an integer that cannot be evenly divided by any integer except itself and 1. Thus 1, 2, 3, 5, 7, 11,... are prime numbers. The number 9, for example, is not a prime number, since it can be divided evenly by 3.

IBM FORTRAN Coding Form

SAMPLE PROGRAM 1 6/66

FORTRAN STATEMENT

```
C       PRIME NUMBER PROBLEM
  100   WRITE (6,8)
    8   FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
       1 19X,1H1/19X,1H2/19X,1H3)
  101   I=5
    3   A=I
  102   A=SQRT(A)
  103   J=A
  104   DO 1 K=3,J,2
  105   L=I/K
  106   IF(L*K-I)1,2,4
    1   CONTINUE
  107   WRITE (6,5)I
    5   FORMAT (I20)
    2   I=I+2
  108   IF(1000-I)7,4,3
    4   WRITE (6,9)
    9   FORMAT (14H PROGRAM ERROR)
    7   WRITE (6,6)
    6   FORMAT (31H THIS IS THE END OF THE PROGRAM)
  109   STOP
        END
```

Figure 2. Example Program 1

EXAMPLE PROGRAM 2

The n points (x , y) are to be used to fit an m degree polynomial by the least-squares method.

$$y = a_0 + a_1x + a_2x^2 + \ldots + a\ xm$$

To obtain the coefficients a , a+,..., a , it is necessary to solve these normal equations

$$(1) \quad W_0a_0 + W_1a_1 + \ldots + Wmam = Z_0$$
$$(2) \quad W_1a_0 + W_2a_1 + \ldots + Wm+_1am = Z_1$$
$$\vdots$$
$$(m+1) \quad Wma + Wm+_1a_1 + \ldots + W_2mam = Zm$$

where

$$W_0 = n \qquad Z = \sum_{i=1}^{n} yi$$

$$W_1 = \sum_{i=1}^{n} xi \qquad Z_1 = \sum_{i=1}^{n} yixi$$

$$W_2 = \sum_{i=1}^{n} xi^2 \qquad Z_2 = \sum_{i=1}^{n} yixi^2$$

$$\vdots \qquad\qquad \vdots$$

$$W_2m = \sum_{i=1}^{n} xi^2m \qquad Zm = \sum_{i=1}^{n} yixim$$

After the Ws and Zs have been computed, the normal equations are solved by the method of elimination, which is illustrated by the following solution of the normal equations for a second degree polynomial (m = 2).

(1) $W_0 a_0 + W_1 a_1 + W_2 a_2 = Z_0$

(2) $W_1 a_0 + W_2 a_1 + W_3 a_2 = Z_1$

(3) $W_2 a_0 + W_3 a_1 + W_0 a_2 = Z_2$

The forward solution is

1. Divide equation (1) by W_0.
2. Multiply the equation resulting from step 1 by W_1, and subtract from equation (2).
3. Multiply the equation resulting from step 1 by W_2, and subtract from equation (3).

The resulting equations are

(4) $a_0 + b_{12} a_1 + b_{13} a_2 = b_{14}$

(5) $b_{22} a_1 + b_{23} a_2 = b_{24}$

(6) $b_{32} a_1 + b_{33} a_2 = b_{34}$

where

$b_{12} = W_1/W_0$, $b_{13} = W_2/W_0$, $b_1 = Z_1/W_0$

$b_{22} = W_2 - b_{12} W_1$, $b_{23} = W_3 - b_{13} W_1$, $b_{24} = Z_1 - b_1 W_{14}$

$b_{32} = W_3 - b_{12} W_2$, $b_{33} = W_4 - b_{13} W_{24}$, $b_{34} = Z_2 - b_1 W_{24}$

Steps 1 and 2 are repeated, using equations (5) and (6), with b_{22} and b_{32} instead of W_0 and W_1. The resulting equations are

(7) $a_1 + c_{23} a_2 = c_{24}$

(8) $c_{33} a_2 = c_{34}$

where:

$$c_{23} = b_{23}/b_{22} \quad , \quad c_{24} = b_2/b_{22}$$

$$c_{33} = b_{33} - c_{23} b_{32} \quad , \quad c_{34} = b_3 - c_{24} b_{32}$$

The backward solution is

(9) $a_2 = c_{34}/c_{33}$ from equation (8)

(10) $a_1 = c_{24} - c_{23} a_2$ from equation (7)

(11) $a_0 = b_{14} - b_{12} a_1 - b_{13} a_2$ from equation (4)

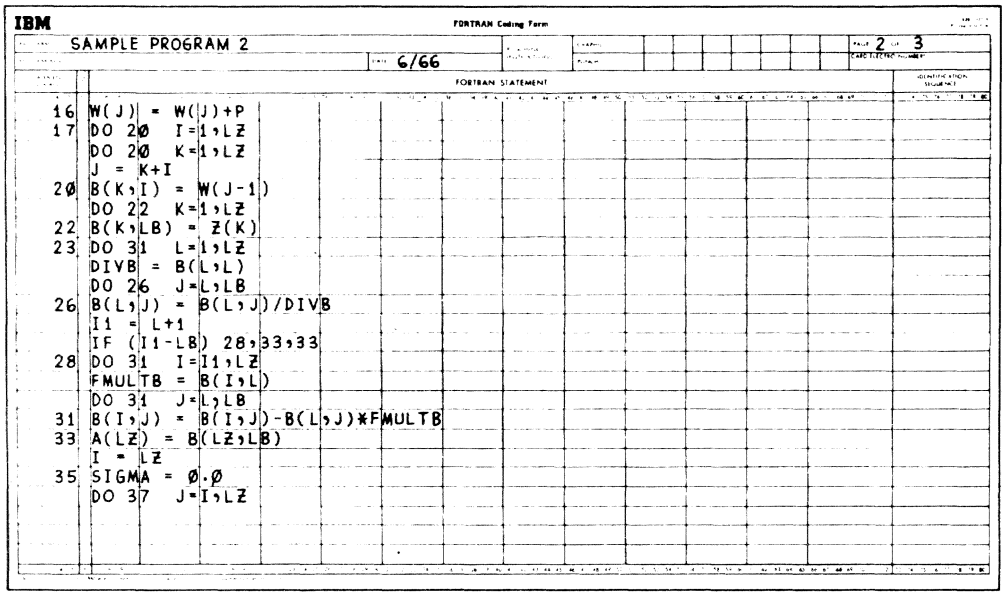
Figure 3 is a possible FORTRAN program for carrying out the calculations for the case: $n = 100$, $m \leq 10$. W_0, W_1, W_2, ..., W_2m are stored in W(1), W(2), W(3), ..., W(2M+1), respectively. Z_0, Z_1, Z_2, ..., Zm are stored in Z (1), Z (2), Z (3), ..., Z(M+1), respectively.

IBM FORTRAN Coding Form

SAMPLE PROGRAM 2 6/66

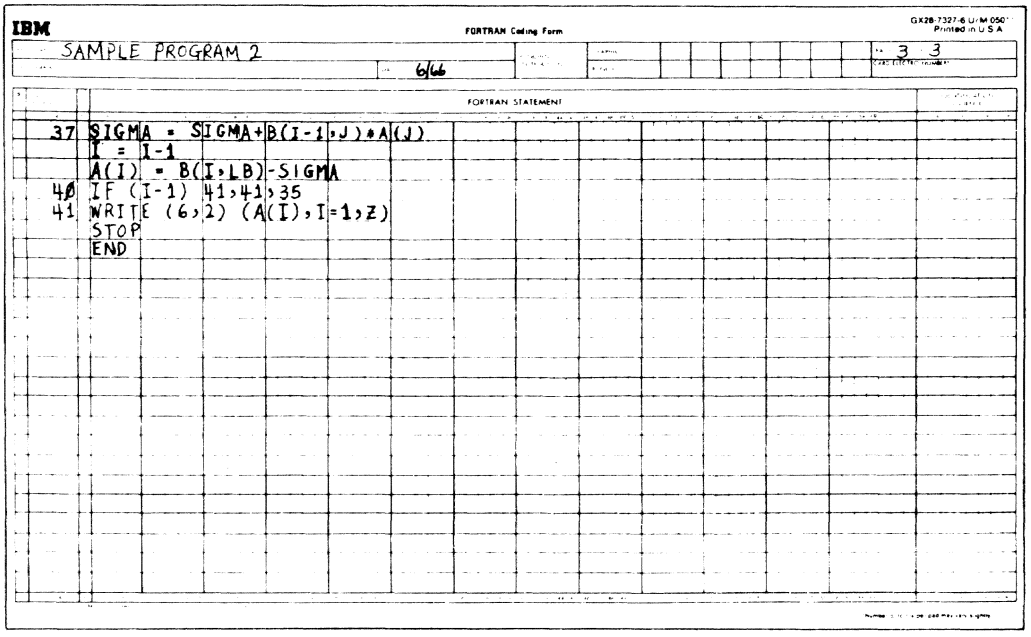
```
      REAL X(100),Y(100),W(21),Z(11),A(11),B(11,12)
    1 FORMAT (I2,I3/(4F14.7))
    2 FORMAT (5E15.6)
      READ (5,1) M,N,(X(I),Y(I),I=1,N)
      LW = 2*M+1
      LB = M+2
      LZ = M+1
      DO 5  J=2,LW
    5 W(J) = 0.0
      W(1) = N
      DO 6  J=1,LZ
    6 Z(J) = 0.0
      DO 16  I=1,N
      P = 1.0
      Z(1) = Z(1)+Y(I)
      DO 13  J=2,LZ
      P = X(I)*P
      W(J) = W(J)+P
   13 Z(J) = Z(J)+Y(I)*P
      DO 16  J=LB,LW
      P = X(I)*P
```

Figure 3. Example Program 2 (part 1 of 3)



```
IBM                          FORTRAN Coding Form
SAMPLE PROGRAM 2                    6/66                    2 3
FORTRAN STATEMENT
   16 W(J) = W(J)+P
   17 DO 20  I=1,LZ
      DO 20  K=1,LZ
      J = K+I
   20 B(K,I) = W(J-1)
      DO 22  K=1,LZ
   22 B(K,LB) = Z(K)
   23 DO 31  L=1,LZ
      DIVB = B(L,L)
      DO 26  J=L,LB
   26 B(L,J) = B(L,J)/DIVB
      I1 = L+1
      IF (I1-LB) 28,33,33
   28 DO 31  I=I1,LZ
      FMULTB = B(I,L)
      DO 31  J=L,LB
   31 B(I,J) = B(I,J)-B(L,J)*FMULTB
   33 A(LZ) = B(LZ,LB)
      I = LZ
   35 SIGMA = 0.0
      DO 37  J=I,LZ
```

Figure 3. Example Program 2 (part 2 of 3)



```
IBM                          FORTRAN Coding Form
SAMPLE PROGRAM 2                    6/66                    3 3
FORTRAN STATEMENT
   37 SIGMA = SIGMA+B(I-1,J)*A(J)
      I = I-1
      A(I) = B(I,LB)-SIGMA
   40 IF (I-1) 41,41,35
   41 WRITE (6,2) (A(I),I=1,Z)
      STOP
      END
```

Figure 3. Example Program 2 (part 3 of 3)

The elements of the W array, except W(1), are set equal to zero; W(1) is set equal to N. For each value of I, X and Y are selected. The powers of X are computed and accumulated in the correct W counters. The powers of X are multiplied by Y, and the products are accumulated in the correct Z counters. To save machine time when the object program is being run, the previously computed power of X is used when computing the next power of X. Note the use of variables as index parameters. By the time control has passed to statement 17, the counters have been set to

$$W(1) = N \qquad Z(1) = \sum_{I=1}^{N} Y_i$$

$$W(2) = \sum_{I=1}^{N} X_i \qquad Z(2) = \sum_{I=1}^{N} Y_i X_i$$

$$W(3) = \sum_{I=1}^{N} X^2 \qquad Z(3) = \sum_{I=1}^{N} Y_i X_i^2$$

$$\vdots \qquad\qquad \vdots$$

$$Z(M+1) = \sum_{I=1}^{N} Y_i X_i M$$

$$W(2M+1) = \sum_{I=1}^{N} X_i^{2M}$$

By the time control has passed to statement 23, the values of W_0, W_1, ..., W_{2m+1} have been placed in the storage locations corresponding to columns 1 through M + 1, rows 1 through M + 1, of the B array, and the values of Z_0, Z_1, ..., Zm have been stored in the locations corresponding to the column of the B array. For example, for the illustrative problem (M = 2), columns 1 through 4, rows 1 through 3, of the B array would be set to these computed values

W_0	W_1	W_2	Z_0
W_1	W_2	W_3	Z_1
W_2	W_3	W_4	Z_2

This matrix represents equations (1), (2), and (3), the normal equations for M = 2.

The forward solution, which results in equations (4), (7), and (8) in the illustrative problem, is carried out by statements 23 through 31. By the time control has passed to statement 33, the coefficients of the AI terms in the M + 1 equations, which would be obtained in hand calculations, have replaced the contents of the locations corresponding to columns 1 through M + 1, rows 1 through M + 1, of the B array, and the constants on the right-hand side of the equations have replaced the contents of the locations corresponding to column M + 2, rows 1 through M + 1, of the B array. For the illustrative problem, columns 1 through 4, rows 1 through 3, of the B array would be set to these computed values

1	b_{12}	b_{13}	b_1
0	1	c_{23}	c_{24}
0	0	c_{33}	c_{34}

This matrix represents equations (4), (7), and (8).

The backward solution, which results in equations (9), (10), and (11) in the illustrative problem, is carried out by statements 33 through 40. By the time control has passed to statement 41, which prints the values of the A9 terms, the values of the (M + 1) *A_i terms have been stored in the M + 1 locations for the A array. For the illustrative problem, the A array would contain these computed values for a_2, a_1, and a_0

Location	Contents
A(3)	c_{34}/c_{33}
A(2)	$c_{24}-c_{23}\,a_2$
A(1)	$b_{14}-b_{12}\,a_1-b_{13}\,a_2$

The resulting values of the AI terms are then printed according to the FORMAT specification in statement 2.

Where more than reference is given, the major reference is first.

GC28-2007-4

IBM Time Sharing System FORTRAN IV (File No. S360-25) Printed in U.S.A. GC28-2007-4

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601