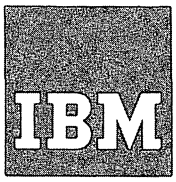
File No. S360-25
Form C28-6515-6

OS
44PS
DOS



Systems Reference Library

IBM System/360

FORTRAN IV Language

This publication describes and illustrates the use of the FORTRAN IV language for the IBM System/360 Operating System, the IBM System/360 Model 44 Programming System, and the IBM System/360 Disk Operating System.

PREFACE

This publication describes the IBM System/360 FORTRAN IV language for the IBM System/360 Operating System, the IBM System/360 Model 44 Programming System, and the IBM System/360 Disk Operating System. A reader should have some knowledge of an existing FORTRAN language before using this publication. A useful source for this information is the set of programmed instruction texts, FORTRAN IV for IBM System/360, Forms R29-0080 through R29-0087.

The material in this publication is arranged to provide a quick definition and syntactical reference to the various elements of the language by means of a box format. In addition, sufficient text describing each element, with appropriate examples as to possible use, is given.

Appendixes contain additional information useful in writing a FORTRAN IV program. This information consists of a table of source program characters, a list of other FORTRAN statements accepted by FORTRAN IV, a list of FORTRAN-supplied mathematical subprograms and service subprograms, lists of differences between FORTRAN IV and Basic FORTRAN IV and USA FORTRAN IV, and sample programs. Out-of-line mathematical subprograms and service subprograms are described in the publication IBM System/360: FORTRAN IV Library Subprograms, Form C28-6596. Compiler restrictions and programming considerations are contained in the programmer's guide for the respective system. The programmer's guides are as follows:

IBM System/360 Operating System: FORTRAN IV (G) Programmer's Guide, Form C28-6639

IBM System/360 Operating System: FORTRAN IV (H) Programmer's Guide, Form C28-6602

IBM System/360 Model 44 Programming System: Guide to System Use for FORTRAN Programmers, Form C28-6813

No programmer's guide is currently available for the Disk Operating System FORTRAN IV compiler.

A comparison of FORTRAN IV compilers is in the publication IBM FORTRAN IV Reference Data, Form X28-6383.

Seventh Edition

This is a major revision of, and makes obsolete, the previous edition, Form C28-6515-5.

This revision corrects errors that appeared in the previous edition. It also makes this publication applicable to FORTRAN IV for use under the Disk Operating System. Technical changes are indicated by a vertical line to the left of the text.

Significant changes or additions to the specifications contained in this publication are continually being made. When using this publication in connection with the operation of IBM equipment, check the latest SRL Newsletter for revisions or contact the local IBM branch office.

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

Address comments concerning the contents of the publication to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, N. Y. 10020. Comments should mention the compiler and level being used.

FIGURES

TABLES

IBM System/360 FORTRAN IV for the Operating System, the Model 44 Programming System, and the Disk Operating System comprise a language, a library of subprograms, and a compiler.

The FORTRAN IV language is especially useful in writing programs for applications that involve mathematical computations and other manipulation of numerical data. The name FORTRAN is derived from FORmula TRANslator.

Source programs written in the FORTRAN IV language consist of a set of statements constructed by the programmer from the language elements described in this publication.

In a process called compilation, a program called the FORTRAN compiler analyzes the source program statements and translates them into a machine language program called the object program, which will be suitable for execution on IBM System/360. In addition, when the FORTRAN compiler detects errors in the source program, it produces appropriate diagnostic error messages. The FORTRAN programmers' guides, listed in the Preface, contain information about compiling and executing FORTRAN programs.

The FORTRAN compiler operates under control of an operating system which provides the FORTRAN compiler with input/output and other services. Object programs generated by the FORTRAN compiler also operate under operating system control and depend on it for similar services.

The IBM System/360 FORTRAN IV language is compatible with and encompasses the United States of America (USA) FORTRAN, including its mathematical subroutine provisions. It also contains, as a proper subset, Basic FORTRAN IV. Appendixes F and G contain lists of differences between FORTRAN IV and Basic FORTRAN IV and USA FORTRAN IV.

STATEMENTS

Source programs consist of a set of statements from which the compiler generates machine instructions, constants, and storage areas. A given FORTRAN statement effectively performs one of three functions:

1. Causes certain operations to be performed (e.g., addition, multiplication, branching)
2. Specifies the nature of the data being handled
3. Specifies the characteristics of the source program

FORTRAN statements usually are composed of certain FORTRAN key words used in conjunction with the basic elements of the language: constants, variables, and expressions. The categories of FORTRAN statements are as follows:

1. Arithmetic and Logical Assignment Statements: These statements cause calculations to be performed or conditions to be tested. The result replaces the current value of a designated variable or subscripted variable.
2. Control Statements: These statements enable the user to govern the flow of and to terminate the execution of the object program.
3. Input/Output Statements: These statements, in addition to controlling input/output devices, enable the user to transfer data between internal storage and an input/output medium.
4. FORMAT Statement: This statement is used in conjunction with certain input/output statements to specify the form of a FORTRAN record.
5. NAMELIST Statement: This statement is used in conjunction with certain input/output statements to specify the form of a special kind of record.
6. DATA Initialization Statement: This statement is used to assign initial values to variables.
7. Specification Statements: These statements are used to declare the properties of variables, arrays, and functions (such as type and amount of storage reserved) and, in addition, can be used to assign initial values to variables and arrays.
8. Statement Function Definition Statement: This statement specifies operations to be performed whenever the statement function name appears in the program.
9. Subprogram Statements: These statements enable the user to name and define functions and subroutines, which can be compiled separately or with the main program.

The basic elements of the language are discussed in this section. The actual FORTRAN statements in which these elements are used are discussed in following sections. The term program unit refers to a main

program or a subprogram; the term executable statements refers to those statements in groups 1, 2, and 3.

The order of a FORTRAN program unit (other than a BLOCK DATA subprogram) is as follows:

1. Subprogram statement, if any.
2. IMPLICIT statement, if any.
3. Other specification statements, if any. (Explicit specification statements that initialize variables or arrays must follow other specification statements that contain the same variable or array names.)
4. Statement function definitions, if any.
5. Executable statements, at least one of which must be present.
6. END statement.

FORMAT, NAMELIST, and DATA statements may appear anywhere after the IMPLICIT statement, if present, and before the END statement. DATA statements, however, must follow any specification statements that contain the same variable or array names.

The order of statements in BLOCK DATA subprograms is discussed in the section "BLOCK DATA Subprogram."

CODING FORTRAN STATEMENTS

The statements of a FORTRAN source program can be written on a standard FORTRAN coding form, Form X28-7327. Each line on the coding form represents one 80-column card. FORTRAN statements are written one to a card within columns 7 through 72. If a statement is too long for one card, it may be continued on as many as 19 successive cards by placing any character, other than a blank or zero, in column 6 of each continuation card. For the first card of a statement, column 6 must be blank or zero.

As many blanks as desired may be written in a statement to improve its readability. They are ignored by the compiler. Blanks that are inserted in literal data are retained and treated as blanks within the data.

Columns 1 through 5 of the first card of a statement may contain a statement number consisting of from 1 through 5 decimal digits. Blanks and 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 of a card if the letter C is placed in column 1. Comments may appear between FORTRAN statements; a comments card may not immediately precede a continuation card. Comments are not processed by the FORTRAN compiler, but are printed on the source program listing. Blanks may be inserted where desired to improve readability.

CONSTANTS

A constant is a fixed, unvarying quantity. There are four classes of constants - those that specify numbers (numerical constants), those that specify truth values (logical constants), those that specify literal data (literal constants), and those that specify hexadecimal data (hexadecimal 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; and hexadecimal constants must be hexadecimal (base 16) numbers.

INTEGER CONSTANTS

Definition
Integer Constant - a whole number written without a decimal point. It occupies four locations of storage (i.e., four bytes).
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 it may not contain embedded commas.

Examples:

Valid Integer Constants:

```
0
91
173
-2147483647
```

Invalid Integer Constants:

```
27.                 (Contains a decimal point)
3145903612          (Exceeds the allowable range)
5,396               (Contains an embedded comma)
```

Definition
Real Constant -- has one of three forms: a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent. A basic real constant is a string of decimal digits with a decimal point. If the string contains fewer than eight digits, the basic real constant occupies four storage locations (bytes); if the string contains eight or more digits, the basic real constant occupies eight storage locations (bytes). The storage requirement (length) of a real constant can also be explicitly specified by appending an exponent to a basic real constant or an integer constant. An exponent consists of the letter E or the letter D followed by a signed or unsigned 1- or 2-digit integer constant. The letter E specifies a constant of length four; the letter D specifies a constant of length eight. Magnitude: (either four or eight locations) 0 or 16^{-65} (approximately 10^{-78}) through 16^{63} (approximately 10^{75}) Precision: (four locations) 6 hexadecimal digits (approximately 7.2 decimal digits) (eight locations) 14 hexadecimal digits (approximately 16.8 decimal digits)

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 permits the expression of a real constant as the product of a basic real constant or integer constant times 10 raised to a desired power.

Examples:

Valid Real Constants (four storage locations):

```
+0.
-999.9999
7.0E+0              (i.e., 7.0 x 10^0 = 7.0)
19761.25E+1         (i.e., 19761.25 x 10^1 = 197612.5)
7.E3
7.0E3               (i.e., 7.0 x 10^3 = 7000.0)
7.0E+03
7E-03               (i.e., 7.0 x 10^-3 = 0.007)
```

Valid Real Constants (eight storage locations):

```
1234567890123456.D-94    (Equivalent to .1234567890123456x10^-75)
21.98753829457168
1.0000000
7.9D03
7.9D+03                  (i.e., 7.9 x 10^3 = 7900.0)
7.9D+3
7.9D0                    (i.e., 7.9 x 10^0 = 7.9)
7D03                     (i.e., 7.0 x 10^3 = 7000.0)
```

Invalid Real Constants:

1	(Missing a decimal point or a decimal exponent)
3,471.1	(Embedded comma)
1.E	(Missing a 1- or 2-digit integer constant following the E. Note that it is not interpreted as 1.0×10^0)
1.2E+113	(E is followed by a 3-digit integer constant)
23.5E+97	(Magnitude outside the allowable range; that is, $23.5 \times 10^{97} > 16^{63}$)
21.3E-90	(Magnitude outside the allowable range; that is, $21.3 \times 10^{-90} < 16^{-65}$)

COMPLEX CONSTANTS

Definition
Complex Constant - an ordered pair of signed or unsigned real constants separated by a comma and enclosed in parentheses. 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. Both parts must occupy the same number of storage locations (either four or eight).

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 in the given range.

Examples:

Valid Complex Constants

(3.2,-1.86)	(Has the value 3.2 - 1.86i)
(-5.0E+03,.16E+02)	(Has the value -5000. + 16.0i)
(4.7D+2,1.9736148)	(Has the value 470. + 1.9736148i)
(47D+2,38D+3)	(Has the value 4700. + 38000.i)

Where $i = \sqrt{-1}$

Invalid Complex Constants:

(292704,1.697)	(The real part is not a valid real constant)
(.003E4,.005D6)	(The parts differ in length)

LOGICAL CONSTANTS

Definition
Logical Constant - a constant that specifies a logical value. There are two logical values: .TRUE. .FALSE. Each occupies four storage locations. The words TRUE and FALSE must be preceded and followed by periods as shown above.

The logical constant .TRUE. or .FALSE. when assigned to a logical variable specifies that the value of the logical variable is true or false, respectively. (See the section "Logical Expressions.")

LITERAL CONSTANTS

Definition
Literal Constant - a string of alphameric and/or special characters, delimited as follows: 1. The string can be enclosed in apostrophes. 2. The string can be preceded by wH where w is the number of characters in the string.

The string may contain any characters (see Appendix A). The number of characters in the string, including blanks, may not be greater than 255. If apostrophes delimit the literal, a single apostrophe within the literal is represented by two apostrophes. If wH precedes the literal, a single apostrophe within the literal is represented as a single apostrophe.

Literals can be used only in CALL statement or function reference argument lists, as data initialization values, or in FORMAT statements. The first form, a string enclosed in apostrophes, may be used in PAUSE statements.

Examples:

```
24H INPUT/OUTPUT AREA NO.2
'DATA'
'X-COORDINATE     Y-COORDINATE     Z-COORDINATE'
'3.14'
'DON''T'
5HDON'T
```

HEXADECIMAL CONSTANTS

Definition
Hexadecimal Constant - the character Z followed by a hexadecimal 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 (byte) contains two hexadecimal digits. If a constant is specified as an odd number of digits, a leading hexadecimal zero is added on the left to fill the storage location. The internal form of each hexadecimal digit is as follows:

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 represents the bit string: 00011100010010011010001011110001

ZBADFADE represents the bit string: 00001011101011011111101011011110
where the first four zero bits are implied because an odd number of hexadecimal digits is written.

The maximum number of digits allowed in a hexadecimal constant depends upon the length specification of the variable being initialized (see "Variable Types and Lengths"). 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 digits is greater than the maximum, the leftmost hexadecimal digits are truncated; if the number of digits is less than the maximum, hexadecimal zeros are supplied on the left.

SYMBOLIC NAMES

Definition
Symbolic Name - from 1 through 6 alphameric (i.e., numeric, 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 C)
- A FUNCTION subprogram (see "FUNCTION Subprograms")

- A SUBROUTINE subprogram (see "SUBROUTINE Subprograms")

- A block name (see "BLOCK DATA Subprogram")

- An external procedure that cannot be classified as either a SUBROUTINE or FUNCTION subprogram (see "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 occupies a storage area. The value specified by the name is always the current value stored in the area.

For example, in the statement:

```
A = 5.0+B
```

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 is calculated whenever this statement is executed and changes as the value of B changes.

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 amount 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:

```
B292S
RATE
$VAR
```

Invalid Variable Names:

```
B292704          (Contains more than six characters)
4ARRAY           (First character is not alphabetic)
SI.X             (Contains a special character)
```

VARIABLE TYPES AND LENGTHS

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. There is no variable type associated with literal or hexadecimal data. These types of data are identified by a name of one of the other types.

For every type of variable, there is a corresponding standard and optional length specification which determines the number of storage locations (bytes) that are 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 ways a programmer may declare the type of a variable are by use of the:

- Predefined specification contained in the FORTRAN language
- Explicit specification statements
- IMPLICIT statement

An Explicit specification statement overrides an IMPLICIT statement, which, in turn, overrides the predefined specification. 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 section, "Type Statements").

Type Declaration by the Predefined Specification

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

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

2. If the first character of the variable name is any other alphabetic character, the variable is real of a 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 in this publication it is presumed that this specification applies unless otherwise noted. Variables defined with this convention are of standard length.

Type Declaration by the IMPLICIT Statement

The IMPLICIT 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. The IMPLICIT statement can be used to specify all types of variables -- integer, real, complex, and logical -- and to indicate standard or optional length.

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 "Specification Statements."

Type Declaration by Explicit Specification Statements

Explicit specification statements 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 that an IMPLICIT statement overrode the predefined convention for variables beginning with the letter I by declaring them to be real and 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 "Specification 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 the following values: 273, 41, 8976, 59, and 2.

```
NEXT(1)  is the location containing 273
NEXT(2)  is the location containing 41
NEXT(3)  is the location containing 8976
NEXT(4)  is the location containing 59
NEXT(5)  is the location containing 2
```

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

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

To refer to any element in an array, the array name must be subscripted. In particular, array name alone does not represent the first element.

Consider the following array named LIST described by two subscript quantities, 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

Suppose it is desired to refer to the number in row 2, column 3; this would be:

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

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

Ordinary mathematical notation might use $LIST_{ij}$ 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.

DECLARING THE SIZE AND TYPE OF AN ARRAY

The size (number of elements) of an array is specified by the number of subscript quantities of the array and the maximum value of each subscript quantity. This information must be given for all arrays before using them in a FORTRAN program so that an appropriate amount of storage may be reserved. Declaration of this information is made by a DIMENSION statement, a COMMON statement, or by one of the Explicit specification statements; these statements are discussed in detail in the section "Specification Statements." The type of an array name is determined by the conventions for specifying the type of a variable name. Each element of an array is of the type specified for the array name.

ARRANGEMENT OF ARRAYS IN STORAGE

Arrays are stored in ascending storage locations, with the value of the first of their subscript quantities increasing most rapidly and the value of the last increasing least rapidly. ie Columns stored 'sideways' Rows stored 'vertically'

For example, the array LIST, whose values are given in the previous example, is arranged in storage as follows:

```
82 12 91 24 2 4 13 1 16 8 7 14 31 10 2
```

The array named A, described by one subscript quantity which varies from 1 to 5, appears in storage as follows:

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

The array named B, described by two subscript quantities with the first subscript quantity varying over the range from 1 to 5, and the second varying from 1 to 3, appears in ascending storage locations in the following order:

```
   B(1,1) B(2,1) B(3,1) B(4,1) B(5,1)--┐
┌--------------------------------------┘
└->B(1,2) B(2,2) B(3,2) B(4,2) B(5,2)--┐
┌--------------------------------------┘
└->B(1,3) B(2,3) B(3,3) B(4,3) B(5,3)
```

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, described by three subscript quantities with the first varying from 1 to 3, the second varying from 1 to 2, and the third varying from 1 to 3:

```
   C(1,1,1) C(2,1,1) C(3,1,1) C(1,2,1) C(2,2,1) C(3,2,1)--┐
┌-----------------------------------------------------------┘
└->C(1,1,2) C(2,1,2) C(3,1,2) C(1,2,2) C(2,2,2) C(3,2,2)--┐
┌-----------------------------------------------------------┘
└->C(1,1,3) C(2,1,3) C(3,1,3) C(1,2,3) C(2,2,3) C(3,2,3)
```

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

SUBSCRIPTS

A subscript is an integer subscript quantity or a set of integer subscript quantities separated by commas, which is used to identify a particular element of an array. The number of subscript quantities in any subscript must be the same as the number of dimensions of the array with which the subscript is associated. A subscript is enclosed in parentheses and is written immediately after the array name. A maximum of seven subscript quantities can appear in a subscript.

The following rules apply to the construction of subscript quantities (see the section "Expressions" for additional information about the terms used below).

1. Subscript quantities may contain arithmetic expressions that use any of the arithmetic operators: +, -, *, /, **.

2. Subscript quantities may contain function references.

3. Subscript quantities may contain subscripted names.

4. Mixed mode expressions (integer and real only) within subscript quantities are evaluated according to normal FORTRAN rules. If the evaluated expression is real, it is converted to integer.

5. The evaluated result of a subscript quantity should always be greater than zero and less than or equal to the size of the corresponding dimension.

Examples:

Valid Subscripted Variables:

```
ARRAY (IHOLD)
NEXT (19)
MATRIX (I-5)
BAK (I,J(K+1*L,.3*A(M,N)))
ARRAY (I,J/4*K**2)
```

Invalid Subscripted Variables

```
ARRAY (-5)          (A subscript quantity may not be negative)
LOT (0)             (A subscript quantity may never be nor assume a
                      value of zero)
ALL(1.GE.I)         (A  subscript quantity may not assume a true or
                      false value)
NXT (1+(1.3,2.0))   (A subscript quantity may not assume a  complex
                      value)
```

EXPRESSIONS

FORTRAN IV 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. The value of a logical expression is always a truth value: .TRUE. or .FALSE.. Expressions may appear in assignment statements and in certain control statements.

ARITHMETIC EXPRESSIONS

The simplest arithmetic expression consists of a primary which may be a single constant, variable, subscripted variable, function reference, or another expression enclosed in parentheses. The primary may be either integer, real, or complex.

In an expression consisting of a single primary, the type of the primary is the type of the expression.

Examples:

Primary	Type of Primary	Type of Expression
3	Integer constant	Integer of length 4
A	Real variable	Real of length 4
3.14D3	Real constant	Real of length 8
(2.0,5.7)	Complex constant	Complex of length 8
SIN(X)	Real function reference	Real of length 4
(A*B+C)	Parenthesized real expression	Real of length 4

More complicated arithmetic expressions containing two or more primaries may be formed by using arithmetic operators that express the computation(s) to be performed.

Arithmetic Operators

The arithmetic operators are as follows:

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

RULES FOR CONSTRUCTING ARITHMETIC EXPRESSIONS: The following 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 primary 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:

   AB

   If multiplication is desired, the expression must be written as follows:

   A*B or B*A

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

   A*/B and A*-B

   The expression A*-B could be written correctly as follows:

   A*(-B)

   In effect, -B will be evaluated first and then A will be multiplied with it. (For further uses of parentheses, see rule 3.)

3. Order of Computation: Computation is performed from left to right according to the hierarchy of operations shown in the following list.

Operation	Hierarchy
Evaluation of functions	1st
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th

This hierarchy is used to determine which of two consecutive operations is performed first. If the first operator is higher than or equal to the second, the first operation is performed. If not, the second operator is compared to the third, etc. When the end of the expression is encountered, all of the remaining operations are performed in reverse order.

For example, in the expression A*B+C*D**I, the operations are performed in the following order:

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

If there are consecutive exponentiation operators, the evaluation is from right to left. Thus, the expression:

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

is evaluated as follows:

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

Parentheses may be used in arithmetic expressions, as in algebra, to specify the order in which the arithmetic operations are to be computed. Where parentheses are used, the expression within the parentheses is evaluated before the result is used. This is equivalent to the definition above since a parenthesized expression is a primary.

For example, the following expression:

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

is effectively evaluated in the following order:

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

Table 1. Determining the Type and Length of the Result of +, -, *, / Operations

+ - * /	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)

Note: 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.

4. The type and length of the result of an operation depends upon the type and length of the two operands (primaries) involved in the operation. Table 1 shows the type and length of the result of the operations +, -, *, and /.

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

Variable Names	Type	Length Specification
C	Real variable	4
I, J, K	Integer variable	4, 2, 2
D	Complex variable	16

   Then the expression I*J/C**K+D is evaluated as follows:

Subexpression	Type and Length
I*J (Call the result X)	Integer of length 4
C**K (Call the result Y)	Real of length 4
X/Y (Call the result Z)	Real of length 4
Z+D	Complex of length 16

   Thus the final type of the entire expression is complex of length 16, but the type changed at different stages in the evaluation. Note that, depending on the values of the variables involved, the result of the expression I*J*C might be different from I*C*J.

5. The arithmetic operator denoting exponentiation (i.e.,**) may only be used to combine the types of operands shown in Table 2.

   The type of the result depends upon the type of the two operands involved, as shown in Table 1. For example, if an integer is raised to a real power, the type of the result is real.

Table 2. Valid Combinations with the Arithmetic Operator **

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

LOGICAL EXPRESSIONS

The simplest form of logical expression consists of a single logical primary, which can be a logical constant, logical variable, logical subscripted variable, logical function reference, or logical expression enclosed in parentheses, which always has the value .TRUE. or .FALSE..

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

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

2. Logical operators combined with logical primary.

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 the section "Logical Operators."

Relational Operators

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

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 type is integer or real may be combined by relational operators. For example, assume that the type of the following variables has been specified as follows:

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:

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

Invalid Logical Expressions Using Relational Operators:

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 .EQ 97.1E9	(Missing period immediately after the relational operator)
.GT. 9	(Missing arithmetic expression before the relational operator)

Logical Operators

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

Logical Operator	Use	Meaning
.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 type of the following variables has been specified as follows:

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 quantity may never be an operand of a relational operator)
`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: The order in which the operations are performed is:

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 effectively evaluated in the following 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	

Note: Logical expressions may not require that all parts be evaluated. Functions within logical expressions may or may not be called. For example, in the expression A.OR.LGF(.TRUE.), it should not be assumed that the LGF function is always invoked.

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 most deeply nested parentheses (that is, the innermost pair of parentheses) is effectively evaluated first. For example, the logical expression:

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

is effectively evaluated in the following 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 the following 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 a subscripted or nonsubscripted variable. b is an arithmetic expression or logical expression.

This FORTRAN statement closely resembles a conventional algebraic equation; however, the equal sign 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.

If b is a logical expression, a must be a logical variable. If b is an arithmetic expression, a must be an integer, real, or complex variable. Table 3 gives the conversion rules used for placing the evaluated result of arithmetic expression b into variable a.

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 truncated to an integer value, and this value 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.
A = E	The real part of the complex variable E replaces the value of A.

Statements	Description
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.
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.

Table 3. Conversion Rules for the Arithmetic Assignment Statement a = b

Type of a \ Type of b	INTEGER*2 INTEGER*4	REAL*4	REAL*8	COMPLEX*8	COMPLEX*16
INTEGER*2 INTEGER*4	Assign	Fix and Assign		Fix and Assign real part; imaginary part not used	
REAL*4	Float and Assign	Assign	Real Assign	Assign real part; imaginary part not used	Real Assign real part; imaginary part not used
REAL*8	DP Float and Assign	Assign		Assign real part; imaginary part not used	
COMPLEX*8	Float and Assign to real part; imaginary part set to zero	Assign to real part; imaginary part set to zero	Real Assign real part; imaginary part set to zero	Assign	Real Assign real and imaginary parts
COMPLEX*16	DP Float and Assign to real part; imaginary part set to zero	Assign to real part; imaginary part set to zero		Assign	

Notes:

1. Assign means transmit the resulting value, without change. If the significant digits of the resulting value exceed the specified length, results are unpredictable.
2. Real Assign means transmit to a as much precision of the most significant part of the resulting value as REAL*4 data can contain.
3. Fix means transform the resulting value to the form of a basic real constant and truncate the fractional portion.
4. Float means transform the resulting value to the form of a REAL*4 number, retaining in the process as much precision of the value as a REAL*4 number can contain.
5. DP Float means transform the resulting value to the form of a REAL*8 number.
6. An expression of the form E=(A,B), where E is a complex variable and A and B are real variables, is invalid. The mathematical function subprogram CMPLX can be used for this purpose. See Appendix C.

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

GO TO STATEMENTS

GO TO statements permit transfer of control to an executable statement specified by number in the GO TO statement. Control may be transferred either unconditionally or conditionally. The GO TO statements are:

1. Unconditional GO TO statement
2. Computed GO TO statement
3. Assigned GO TO statement

UNCONDITIONAL GO TO STATEMENT

General Form
GO TO xxxxx Where: xxxxx 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. Any executable statement immediately following this statement should have a statement number; otherwise it can never be referred to or executed.

Example:

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

Explanation:

In this example, each time the GO TO statement 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 whose current value is in the range: $1 \leq i \leq n$

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, 7), ITEM
          .
          .
          .
7     C = E**2+A
          .
          .
          .
25    L = C
          .
          .
          .
10    B = 21.3E02
```

Explanation:

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

ASSIGN AND ASSIGNED GO TO STATEMENTS

General Form
ASSIGN i TO m . . . GO TO m, ($x_1, x_2, x_3, \ldots, x_n$) Where: i is an executable statement number. It must be one of the numbers $x_1, x_2, x_3, \ldots, x_n$. $x_1, x_2, x_3, \ldots, x_n$ are executable statement numbers in the program unit containing the GO TO statement. m is a nonsubscripted integer variable of length 4 which is assigned one of the statement numbers: $x_1, x_2, x_3, \ldots, x_n$.

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

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

If the current assignment of the integer variable N is statement number 8, then the statement numbered 8 is executed next. If the current assignment of N is statement number 10, the statement numbered 10 is executed next. If N is assigned statement number 25, statement 25 is executed next.

At the time of execution of an assigned GO TO statement, the current value of m must have been defined to be one of the values $x, x_2 \ldots x_n$ by the previous execution of an ASSIGN statement. 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:

In example 1, 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:

In example 2, 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.

ADDITIONAL CONTROL STATEMENTS

ARITHMETIC IF STATEMENT

General Form
IF (a) x_1, x_2, x_3
Where: a is any arithmetic expression except complex.
x_1, x_2, x_3 are executable statement numbers.

The arithmetic IF statement causes control to be transferred to the statement numbered $x_1, x_2,$ or 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:

In this example, if the value of the expression (A(J,K)**3-B) is negative, the statement numbered 10 is executed next. If the value of the expression is zero, the statement numbered 4 is executed next. If the value of the expression is positive, the statement numbered 30 is executed next.

General Form
IF(a)s Where: a is any logical expression. s is any executable statement except a 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:

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

Explanation:

In the first statement, 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 the statement numbered 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 the second statement.

In the third statement, 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 then the fourth statement 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 the fourth statement is executed next.

Example 2:

Assume that P and Q are logical variables.

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

Explanation:

In the first statement, if the value of the expression is .TRUE., the value of A is replaced by the value of B and the second statement is executed next. If the value of the expression is .FALSE., the statement A = B is skipped and the second statement 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 an executable statement number appearing after the DO statement.

i is a nonsubscripted integer variable.

m_1, m_2, and m_3, are either unsigned integer constants greater than zero or unsigned nonsubscripted integer variables whose value is greater than zero. m_2 may not exceed $2^{31}-2$ in value. 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, at least once, the statements that physically follow the DO statement, up to and including the statement numbered x. These statements are called the range of the DO. 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 the 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 are 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 within the brackets. 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 and may not be used until assigned a value (e.g., in a READ list).

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 1,000 different machine parts in stock. Periodically, he may find it necessary to compute the amount of each different part 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:

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

Explanation:

The first, second, and fourth statements required to control the previously shown 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)
      A = B+C
          .
          .
          .
```

Explanation:

In example 2, 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 the third statement 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
      J = I+K
25    ARRAY(J) = BRAY(J)
      A = B + C
          .
          .
          .
```

Explanation:

In example 3, 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 the second and third statements 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 the fourth statement 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 statement (i, m_1, m_2, m_3) should not be changed by a statement within the range of the DO loop.

2. There may be other DO statements within the range of a DO statement. All statements in the range of the inner DO must be in the range of the outer DO. A set of DO statements satisfying this rule is called a nest of DO's.

Example 1:

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

Range of Outer DO

Example 2:

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

Range of Outer DO

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

4. The extended range of a DO is defined as those statements in the program unit containing the DO statement that are executed between the transfer out of the innermost DO of a nest of DO's and the transfer back into the range of this innermost DO. The following restrictions apply:

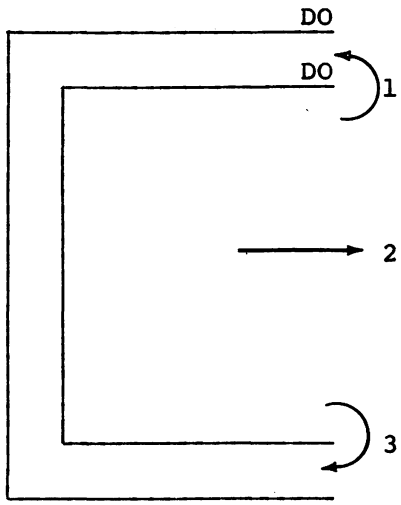
   - Transfer into the range of a DO is permitted only if such a transfer is from the extended range of the DO.

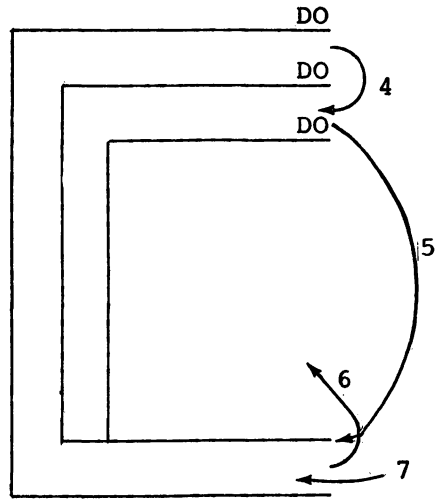
   - The extended range of a DO statement must not contain another DO statement that has an extended range if the second DO is within the same program unit as the first.

   - The indexing parameters (i, m_1, m_2, m_3) cannot be changed in the extended range of the DO.

Note that a statement that is the end of the range of more than one DO statement is within the innermost DO.

Example:





Explanation:

In the preceding example, the transfers specified by the numbers 1, 2, and 3 are permissible, whereas those specified by 4, 5, 6, and 7 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 that uses those parameters.

6. The last statement in the range of a DO loop (statement x) must be an executable statement. It cannot be a GO TO statement of any form, or a PAUSE, STOP, RETURN, arithmetic IF statement, another DO statement, or a logical IF statement containing any of these forms.

7. The use of, and return from, a subprogram from within any DO loop in a nest of DO's is permitted.

CONTINUE STATEMENT

General Form
CONTINUE

CONTINUE is a dummy statement that 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 in order to avoid ending the DO loop with a GO TO, PAUSE, STOP, RETURN, arithmetic IF, another DO statement, or a logical IF statement containing any of these forms.

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
       C = A(3) + B(7)
          .
          .
          .
```

Explanation:

In example 1, the CONTINUE statement is used as the last statement in the range of the DO in order 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:

In example 2, the CONTINUE statement provides a branch point enabling the programmer to bypass the execution of statement 40.

PAUSE STATEMENT

General Form
PAUSE PAUSE n PAUSE 'message' Where: n is a string of 1 through 5 decimal digits. 'message' is a literal constant of up to 255 characters enclosed in apostrophes.

PAUSE n, PAUSE message, or PAUSE 00000 is displayed, depending upon whether n, 'message' or no parameter was specified, and the program waits until operator intervention causes it to resume execution, starting with the next statement after the PAUSE statement. For further information, see the FORTRAN programmers' guides listed in the Preface.

STOP STATEMENT

General Form
STOP STOP n Where: n is a string of 1 through 5 decimal digits.

The STOP statement terminates the execution of the object program and displays n if specified. For further information, see the FORTRAN programmers' guides listed in the Preface.

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, and it may not be continued. The END statement does not terminate program execution. To terminate execution, a STOP statement or a RETURN statement in the main program is required.

INPUT/OUTPUT STATEMENTS

Input/output statements are used to transfer and control the flow of data between internal storage and an input/output device, such as a card reader, printer, punch, magnetic tape unit, or disk storage unit. The data that is to be transferred belongs to a data set. Data sets are composed of one or more records. Typical records are punched cards, printed lines, or the images of either on magnetic tape or disk.

Operation: In order for the input or output operation to take place, the programmer must specify the kind of operation he desires; READ, WRITE, or BACKSPACE, for example.

Data Set Reference Number: A FORTRAN programmer refers to a data set by its data set reference number. (The FORTRAN programmers' guides, listed in the preface, explain how data set reference numbers are associated with data sets.) In the statement specifying the type of input/output operation, the programmer must give the data set reference number corresponding to the data set he wishes to operate on.

I/O List: Input/output statements in FORTRAN are primarily concerned with the transfer of data between storage locations defined in a FORTRAN program and records which are external to the program. On input, data is taken from a record and placed into storage locations that are not necessarily contiguous. On output, data is gathered from diverse storage locations and placed into a record. An I/O list is used to specify which storage locations are used. The I/O list can contain variable names, subscripted array names, unsubscripted array names, or array names accompanied by indexing specifications in a form called an implied DO. No function references or arithmetic expressions are permitted in an I/O list.

If a variable name or subscripted array name appears in the I/O list, one item is transmitted between a storage location and a record.

If an unsubscripted array name appears in the list, the entire array is transmitted in the order in which it is stored. (If the array has more than one dimension, it is stored in ascending storage locations, with the value of the first subscript quantity increasing most rapidly and the value of the last increasing least rapidly. An example is given in the section "Arrangement of Arrays in Storage.")

If an implied DO appears in the I/O list, the elements of the array(s) specified by the implied DO are transmitted. The implied DO specification is enclosed in parentheses. Within the parentheses are one or more subscripted array names, separated by commas with a comma following the last name, followed by indexing parameters $i=m_1$, m_2, m_3 like those in the DO statement. The indexing parameters specify an initial value, test value, and increment. If the increment is omitted, 1 is assumed.

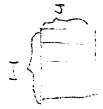
For example, assume that A is a variable and that B, C, and D are one-dimensional arrays each containing 20 elements. Then the statement:

```
WRITE (6) A, B, (C(I), I=1,4), D(4)
```

writes the current value of variable A, the entire array B, the first four elements of the array C, and the fourth element of D. (The 6 following the WRITE is the data set reference number.)

Implied DO's can be nested if required. For example, to read an element into array B after values are read into each row of a 10 x 20 array A, the following would be written:

```
READ (5) ((A(I,J), J=1,10), B(I), I=1,20)
```



The order of the names in the list specifies the order in which the data is transferred between the record and the storage locations.

A special kind of I/O list called a NAMELIST list is explained in the section "READ and WRITE Using NAMELIST."

Formatted and Unformatted Records: Data can be transmitted either under control of a FORMAT statement or without the use of a FORMAT statement.

When data is transmitted with format control, the data in the record is coded in a form that can be read by the programmer or satisfies the needs of machine representation. The transformation for input takes the character codes and constructs a machine representation of an item. The output transformation takes the machine representation of an item and constructs character codes suitable for printing. Most transformations involve numeric representations that require base conversion. To obtain format control, the programmer must include a FORMAT statement in the program and must give the statement number of the FORMAT statement in the READ or WRITE statement specifying the input/output operation.

When data is transmitted without format control, no FORMAT statement is used. In this case, there is a one-to-one correspondence between internal storage locations (bytes) and external record positions. A typical use of unformatted data is for information that is written out during a program, not examined by the programmer, and then read back in later in the program or in another program for additional processing.

For unformatted data, the I/O list determines the length of the record. For example, an output record is complete when the current values of all the items in the I/O list have been placed in it, plus any control words supplied by the input/output routines or Data Management. For further information, see the FORTRAN IV programmers' guides listed in the Preface.

For formatted data, the I/O list and the FORMAT statement determine the form of the record. For further information see the section "FORMAT Statement" and the FORTRAN IV programmers' guides.

There are two types of input/output statements: sequential and direct access. Sequential input/output statements are used for storing and retrieving data sequentially. These statements are device independent and can be used for data sets on either sequential or direct access devices.

The direct access input/output statements are used to store and retrieve data in an order specified by the user. These statements can be used only for a data set on a direct access storage device.

SEQUENTIAL INPUT/OUTPUT STATEMENTS

There are five sequential input/output statements: READ, WRITE, END FILE, REWIND, and BACKSPACE. The READ and WRITE statements cause transfer of records of sequential data sets. 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 not input/output statements, are used with certain forms of the READ and WRITE statements. The FORMAT and NAMELIST statements are not executable statements and can appear anywhere in the program.

READ STATEMENT

General Form
READ(a,b,END=c,ERR=d) list
Where: a is an unsigned integer constant or an integer variable that is of length 4 and represents a data set reference number.
b is optional and is either the statement number or array name of the FORMAT statement describing the record(s) being read, or a NAMELIST name.
END=c is optional and c is the number of the statement to which transfer is made upon encountering the end of the data set.
ERR=d is optional and d is the number of the statement to which transfer is made upon encountering an error condition in data transfer.
list is optional and is an I/O list.

The READ statement may take many forms. The value of a must always be specified, but under appropriate conditions b, c, d, and list can be omitted. The order of the parameters END=c and ERR=d can be reversed within the parentheses.

Transfer is made to the statement specified by the END parameter when the end of the data set is encountered; i.e., when a READ statement is executed after the last record on the data set has already been read. (No indication is given of the number of list items read into before the end of the data set was encountered.) If the END parameter is omitted, object program execution is terminated upon encountering the end of the data set.

Transfer is made to the statement specified by the ERR parameter if an input/output device error occurs. No data is read into the list items and no indication is given of which record or records could not be read, only that an error occurred during transmission of data. If the ERR parameter is omitted, object program execution is terminated when an input/output device error occurs.

The basic forms of the READ statements are:

Form	Purpose
READ(a,b)list	Formatted READ
READ(a)list	Unformatted READ
READ(a,x)	READ using NAMELIST

The discussion of READ using NAMELIST is in the section "READ and WRITE Using NAMELIST."

Formatted READ

The form READ (a,b)list is used to read data from the data set associated with data set reference number a into the variables whose names are given in the list. The data is transmitted from the data set to storage according to the specifications in the FORMAT statement, which is statement number b.

Example:

```
READ (5,98) A,B,(C(I,K),I=1,10)
```

Explanation: The above statement causes input data to be read from the data set associated with data set reference number 5 into the variables A, B, C(1,K), C(2,K),..., C(10,K) in the format specified by the FORMAT statement whose statement number is 98.

Unformatted READ

The form READ(a) list is used to read a single record from the data set associated with data set reference number a into the variables whose names are given in the list. Since the data is unformatted, no FORMAT statement number is given. This statement is used to read unformatted data written by a WRITE(a)list statement. If the list is omitted, a record is passed over without being processed.

Example:

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

Explanation: The above statement causes data to be read from the data set associated with data set reference number J into the variables A, B, and C.

WRITE STATEMENT

General Form
WRITE(a,b)list
Where: a is an unsigned integer constant or an integer variable that is of length 4 and represents a data set reference number.
b is optional and is either the statement number or array name of the FORMAT statement describing the record(s) being written, or a NAMELIST name.
list is optional and is an I/O list.

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:

Form	Purpose
WRITE(a,b)list	Formatted WRITE
WRITE(a)list	Unformatted WRITE
WRITE(a,x)	WRITE using NAMELIST

The discussion of WRITE using NAMELIST is in the section "READ and WRITE Using NAMELIST."

Formatted WRITE

The form WRITE(a,b)list is used to write data into the data set whose reference number is a from the variables whose names are given in the list. The data is transmitted from storage to the data set according to the specifications in the FORMAT statement, whose statement number is b.

Example:

```
WRITE(7,75)A,(B(I,3),I=1,10,2),C
```

Explanation: The above statement causes data to be written from the variables A, B(1,3), B(3,3), B(5,3), B(7,3), B(9,3), C into the data set associated with data set reference number 7 in the format specified by the FORMAT statement whose statement number is 75.

Unformatted WRITE

The form WRITE(a)list is used to write a single record from the variables whose names are given in the list into the data set whose data set reference number is a. This data can be read back into storage with the unformatted form of the READ statement, READ(a)list. The list cannot be omitted.

Example:

```
WRITE (L) ((A(I,J,I=1,10,2), B(J,3), J=1,K)
```

Explanation: The above statement causes data to be written from the variables A(1,1), A(3,1),..., A(9,1), B(1,3), A(1,2), A(3,2),..., A(9,2), B(2,3),..., B(K,3) into the data set associated with the data set reference number L. Since the record is unformatted, no FORMAT statement number is given. Therefore, no FORMAT statement number should be given in the READ statement used to read the data back into storage.

READ AND WRITE USING NAMELIST

The NAMELIST statement is used in conjunction with the READ(a,x) and WRITE(a,x) statements to provide for reading and writing data without including the list specification in the READ and WRITE statements. The NAMELIST statement declares a name x to refer to a particular list of variables or array names. Neither a dummy variable name nor a dummy array name may appear in the list. Thereafter, the forms READ(a,x) and WRITE(a,x) are used to transmit data between the data set associated with the reference number a and the variables specified by the NAMELIST name x.

The format and rules for constructing and using the NAMELIST statements are described in the following text.

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 declaring and using a NAMELIST name:

1. A NAMELIST name is a symbolic name.

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 lists.

4. A NAMELIST name must be declared in a NAMELIST statement before it is used in an input/output statement, and it may be declared only once. After it is declared, it may appear only in input/output statements.

5. The rules for input/output conversion of NAMELIST data are the same as the rules for data conversion described in the section "FORMAT Statement." The NAMELIST data must be in a special form, described in the following sections.

6. A NAMELIST name may not be used as an argument.

NAMELIST Input Data

Input data must be in a special form in order to be read using a NAMELIST list. The first character in each record to be read must be blank. The second character in the first record of a group of data records must be an &, immediately followed by the NAMELIST name. The NAMELIST name must be followed by a blank and must not contain any embedded blanks. This name is followed by data items separated by commas. (A comma after the last item is optional.) The end of a data group is signaled by &END.

The form of the data items in an input record may be:

- variable name = constant

  The variable name may be a subscripted array name or a single variable name. Subscripts must be integer constants. The constant may be integer, real, literal, complex, or logical. (If the constants are logical, they may be in the form T or .TRUE. and F or .FALSE.)

- array name = set of constants (separated by commas)

  The array name is not subscripted. The set of constants consists of constants of the type integer, real, literal, complex, or logical. The number of constants must be less than or equal to the number of elements in the array. Successive occurrences of the same constant can be represented in the form k*constant.

The variable names and array names specified in the input data set must appear in the NAMELIST list, but the order is not significant. A name that has been made equivalent to a name in the input data cannot be substituted for that name in the NAMELIST list. The list can contain names of items in COMMON but must not contain dummy argument names.

Each data record must begin with a complete variable or array name or constant. Embedded blanks are not permitted in names or constants.

NAMELIST Output Data

When output data is written using a NAMELIST list, it is written in a form that can be read using a NAMELIST list. All variable and array names specified in the NAMELIST list and their values are written out, each according to its type. The fields for the data are made large enough to contain all the significant digits. The values of a complete array are written out in columns.

Example: Assume that A is a 3 by 1 array, I and L are 3 by 3 arrays, and that the following statements are given:

```
NAMELIST /NAM1/A,B,I,J,L/NAM2/C,J,I,L

READ (5,NAM1)

WRITE (6,NAM2)
```

Explanation: The NAMELIST statement defines two NAMELIST lists, NAM1 and NAM2. The READ statement causes input data to be read from the data set associated with data set reference number 5 into the variables and arrays specified by NAM1. Assume that the data cards have the form:

```
                        Column 2
                        |
                        v
First card              &NAM1 I(2,3)=5,J=4,B=3.2
   .                          .
   .                          .
Last card               A(3)=4.0,L=2,3,7*4,&END
```

The first data card is read and examined to verify that its name is consistent with the NAMELIST name in the READ statement. (If that NAMELIST name is not found, then it reads to the next NAMELIST group.) When the data is read, the integer constants 5 and 4 are placed in I(2,3) and J, respectively; and the real constants 3.2 and 4.0 are placed in B and A(3), respectively. Since L is an array name not followed by a subscript, the entire array is filled with the succeeding constants. Therefore, the integer constants 2 and 3 are placed in L(1,1) and L(2,1), respectively, and the integer constant 4 is placed in L(3,1), L(1,2),..., L(3,3).

The WRITE statement causes data to be written from the variables and arrays specified by NAM2 into the data set associated with data set reference number 6. Assume that the values of J, L, and I(2,3) were not altered since the previous READ statement, that C was given the value 428.0E+03, that I(1,3) was given the value 6, and that the rest of the elements of I were set to zero. Then, if the output is punched on cards, the form is:

```
                        Column 2
                        |
                        v
First card              &NAM2

Second card             C=428000.00,J=4,I=0,0,0,0,0,0,6,5,

Third card                0,L=2,3,4,4,4,4,4,4,4,

Fourth card             &END
```

General Form

xxxxx FORMAT ($c_1, c_2, \ldots, c_n$)

Where: xxxxx is a statement number (1 through 5 digits).

$c_1, c_2, \ldots, c_n$ are format codes.

The format codes are:

aIw	(Describes integer data fields.)
aFw.d	(Describes basic real constant data fields.)
aEw.d	(Describes fields for real data with an E decimal exponent.)
aDw.d	(Describes fields for real data with a D decimal exponent.)
aZw	(Describes hexadecimal data fields.)
aGw.s	(Describes integer, real, complex, or logical data fields.)
pP	(Specifies a scale factor for real numbers.)
aLw	(Describes logical data fields.)
aAw	(Describes alphameric data fields.)
'Literal'	(Transmits literal data.)
wH	(Transmits literal data.)
wX	(Indicates that a field is to be skipped on input or filled with blanks on output.)
Tw	(Indicates the position in a FORTRAN record where transfer of data is to start.)
a(...)	(Indicates a group format specification.)

Where: a is optional and is an unsigned integer constant used to denote the number of times the format code is to be used. If a is omitted, the code is used only once.

w is an unsigned nonzero integer constant less than or equal to 255 that specifies the number of characters of data in the field.

d is an unsigned integer constant specifying the number of decimal places to the right of the decimal point; i.e., the fractional portion.

s is an unsigned integer constant specifying the number of significant digits.

p is an unsigned or negatively signed integer constant specifying the scale factor.

(...) is a group format specification. Within the parentheses are format codes separated by commas or slashes. Group format specifications can be nested to a level of two. The a preceding this form is called a group repeat count.

Note: Complex number fields in records require two successive D, E, or F format codes. These codes may be grouped within parentheses.

Both commas and slashes can be used as separators between format codes (see the section "Various Forms of a FORMAT Statement").

The FORMAT statement is used in conjunction with the I/O list in the READ and WRITE statements to specify the structure of FORTRAN records and the form of the data fields within the records. In the FORMAT statement, the data fields are described with format codes, and the order in which these format codes are specified gives the structure of the FORTRAN records. The I/O list gives the names of the data items to make up the record. The length of the list in conjunction with the FORMAT statement specifies the length of the record (see the section "Various Forms of a FORMAT Statement"). Throughout this section, the examples show punched card input and printed line output. The concepts apply to all input/output media. In the examples, the character b represents a blank.

The following list gives general rules for using the FORMAT statement:

1. FORMAT statements are not executed; their function is to supply information to the object program. They may be placed anywhere in the source program.

2. When defining a FORTRAN record by a FORMAT statement, it is important to consider the maximum size record allowed on the input/output medium. For example, if a FORTRAN record is to be punched for output, the record should not be longer than 80 characters. If it is to be printed, it should not be longer than the printer's line length. For input, the FORMAT statement should not define a FORTRAN record longer than the record referred to in the data set.

3. All FORMAT specifications describing records to be printed must begin with a carriage control character. This character can be specified in one of two forms of literal data: either '*x*' or 1H*x*, where *x* is one of the following:

x	Meaning
blank	Advance one line before printing
0	Advance two lines before printing
1	Advance to first line of next page
+	No advance

   The carriage control character is not printed. It is treated as data for all input/output media except the printer.

4. If the I/O list is omitted from the READ or WRITE statement, a record is skipped on input, or a blank record is inserted on output, unless the record was transmitted between the data set and the FORMAT statement (see "H Format Code and Literal Data").

Various Forms of a FORMAT Statement

All of the format codes in a FORMAT statement are enclosed in a pair of parentheses. Within these parentheses, the format codes are delimited by the separators: comma and slash.

Each time a READ or WRITE statement is executed, successive items in the I/O list are transmitted according to successive format codes in the FORMAT statement until all the items in the list are transmitted. If there are more items in the I/O list than there are format codes in the

FORMAT statement, control is transferred to the group repeat count of the group format specification terminated by the last right parenthesis that precedes the right parenthesis ending the FORMAT statement; the same format codes are used again with the next record. (If there are no group format specifications, control is transferred to the left parenthesis beginning the FORMAT statement.) If there are fewer items in the I/O list than there are format codes in the FORMAT statement, the remaining FORMAT codes are ignored. For an example, see "Group Format Specifications."

Comma: The simplest form of a FORMAT statement is the one shown in the box at the beginning of this section with the format codes, separated by commas, enclosed in a pair of parentheses. One FORTRAN record is defined by the beginning of the FORMAT statement (left parenthesis) to the end of the FORMAT statement (right parenthesis). For an example, see the section "Examples of Numeric Format Codes."

Slash: A slash is used to indicate the end of a FORTRAN record format. For example, the statement:

```
25    FORMAT    (I3,F6.2/D10.3,F6.2)
```

describes two FORTRAN record formats. The first, third, etc., records are transmitted according to the format I3, F6.2 and the second, fourth, etc., records are transmitted according to the format D10.3, F6.2.

Consecutive slashes can be used to introduce blank output records or to skip input records. 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. If n consecutive slashes appear anywhere else in a FORMAT statement, the number of records skipped or blank records inserted is n-1. For example, the statement:

```
25    FORMAT    (1X,10I5//1X,8E14.5)
```

describes three FORTRAN record formats. On output, it causes double spacing between the line written with format 1X,10I5 and the line written with the format 1X,8E14.5.

I Format Code

The I format code is used in transmitting integer data. For example, if a READ statement refers to a FORMAT statement containing I format codes, the input data is stored in internal storage in integer format. The magnitude of the data to be transmitted must not exceed the maximum magnitude of an integer constant.

Input: Leading, embedded, and trailing blanks in a field of the input card are interpreted as zeros.

Output: If the number of significant digits and sign required to represent the quantity in the storage location is less than w, the leftmost print positions are filled with blanks. If it is greater than w, asterisks are printed instead of the number.

F Format Code

FORMAT(aFw.d)

The F format code is used in transmitting basic real constants. The magnitude of the data to be transmitted must not exceed the maximum magnitude for a real or double-precision constant.

Input: Leading, embedded, and trailing blanks in a field of the input card are interpreted as zeros. The decimal point of the number need not be punched in the card. If it is, its position overrides the position indicated by the d portion of the format code and the positions reserved by w must include a place for the decimal point.

Output: On output, only d digits are printed for the fractional portion. If the fractional portion is greater than d, it is rounded off. The positions reserved by w must include a position for the decimal point. If the integer portion, including the decimal point and sign, if any, is too large, asterisks are printed instead of the number. If the number is too small, it is preceded by leading blanks.

E and D Format Codes

FORMAT(aEw.d , aDw.d)

The E and D format codes are used in transmitting real or double-precision data that contains an E or D decimal exponent, respectively. The w specification should include four places for the exponent portion, space for d (the fractional portion), and places for the decimal point, a zero preceding it, and a sign if any. In general w should be at least equal to d+7.

Input: Since four positions are reserved for the exponent, the rest of the number must not exceed w-4 even if not all four positions are used for the exponent. Leading, embedded, and trailing blanks are treated as zeros. Therefore, if the number is not right justified in the field, significant zeros are appended to the exponent. The decimal point need not be punched. If it is, its position overrides the position indicated by d and the position reserved by w must include a place for the decimal point.

Output: The first significant digit appears just to the right of the decimal point. Therefore, the d specification controls the number of significant digits that are printed. Digits in excess of d are dropped after rounding from the right. Places should be reserved in w for one digit, and a sign, if necessary, to the left of the decimal point.

Z Format Code

aZw

The Z format code is used in transmitting hexadecimal data.

Input: Leading, embedded, and trailing blanks in an input field are treated as zeros. One storage location (byte) in internal storage contains two hexadecimal digits; thus, if a number punched in a field of an input card contains an odd number of digits, the number will be padded on the left with a hexadecimal zero when it is stored.

Output: If the number of characters in the storage location is less than w, the leftmost print positions are filled with blanks. If the number of characters in the storage location is greater than w, the leftmost digits are truncated and the rest of the number is printed.

G Format Code

The G format code is a generalized code used to transmit integer, real, complex, or logical data according to the type specification of the corresponding variable in the I/O list.

Input: The rules for input for G format code depend upon the type of the variable in the I/O list and the form of the number punched on the card. For example, if the variable is real and the number punched in the card has an E decimal exponent, the rules are the same as for the E format code. If the variable in the I/O list is integer or logical, the s portion of the format code can be omitted; if it is given, it is ignored. For complex and real data, the s portion gives the location of the implied decimal point for input -- just like the d specification for D, E, and F format codes.

Output: The .s portion of the format code can be omitted for integer and logical data and the numbers are printed according to the rules for I and L format codes. For complex and real data, the s is used to determine the number of digits to be printed and whether the number should be printed with or without a decimal exponent. If the number, say n, is in the range $0.1 \leq n < 10^{**}s$, the number is printed without a decimal exponent. Otherwise, it is printed with an E or D decimal exponent depending on the length specification (either four or eight storage locations, respectively) of the variable in the I/O list. The w specification for complex and real data should include positions for a decimal point and a sign and four positions for a decimal exponent field, in case one is necessary. All other rules for output are the same as those for the individual format codes.

Examples of Numeric Format Codes

The following examples illustrate the use of the format codes I, F, D, E, Z, and G.

Example 1:

```
75 FORMAT (I3,F5.2,E10.3,G10.3)

READ (5,75) N,A,B,C
```

Explanation:

1. Four input fields are described in the FORMAT statement and four variables are in the I/O list. Therefore, each time the READ statement is executed, one input card is read from the data set associated with data set reference number 5.

2. When an input card is read, the number in the first field of the card (3 columns) is stored in integer format in location N. The number in the second field of the input card (5 columns) is stored in real format, with no decimal exponent, in location A, etc.

3. If there were one more variable in the I/O list, say M, another card would be read and the information in the first three columns in that card would be stored in integer format in location M. The rest of the data on the card would be ignored.

4. If there were one fewer variable in the list (say C is omitted), no number would be stored according to the format G10.3.

5. This format statement defines only one record format. The section "Various Forms of a FORMAT Statement" explains how to define more than one record format in a FORMAT statement.

Example 2: Assume that the following statements are given:

```
75 FORMAT (Z4,D10.3,2G10.3)

   READ (5,75) A,B,C,D
```

where A, C, and D are REAL*4 and B is REAL*8 and that the following input cards are read:

```
Column      1    5          15         25         35
            |    |          |          |          |
            v    v          v          v          v
Input      (b3F1 156432D+02 276.38E+15 bbbbbbbbbb
Cards       2AF3 155381+02b 382506E+28 276.38E+15
            3ACb 346.18D-03 485.322836 276.38E+15

Format      Z4    D10.3      G10.3      G10.3
```

Then the variables A, B, C, and D receive values as if the following had been punched:

A	B	C	D
03F1	156.432D+02	276.38E+15	000000.000
2AF3	155.381D+020	382.506E+28	276.38E+15
3AC0	346.18D-03	485.322836	276.38E+15

Explanation:

1. Leading, trailing, and embedded blanks in an input field are treated as zeros. Therefore, since the value for B on the second input card was not right-justified in the field, the exponent is 20 not 2.

2. Values read into the variables C and D with a G format code are converted according to the type of the corresponding variable in the I/O list.

Example 3: Assume that the following statements are given:

```
76 FORMAT   ('1',I3,F6.2,E10.3,G12.6)

   WRITE    (6,76) N,A,B,C
```

and that the variables N, A, B, and C have the following values:

N	A	B	C
031	034.40	123.380E+02	123.380E+02
130	031.1	1156.1E+02	123456789.
428	-354.32	834.621E-03	1234.56789
000	01.132	83.121E+06	123380D+02

Then, the following lines are printed:

```
Column   1  4     10        20
         |  |     |         |
         v  v     v         v
          31 34.40 0.123E 05 12338.0

         130 31.10 0.116E 060.123457E 09

         428****** 0.835E 00 1234.57

           0  1.13 0.831E 08 12338.0
```

Explanation:

1. The integer portion of the third value of A exceeds the format specification, so asterisks are printed instead of a value. The fractional portion of the fourth value of A exceeds the format specification, so the fractional portion is rounded.

2. Note that for the variable B the decimal point is printed to the left of the first significant digit and that only three significant digits are printed because of the format specification E10.3. Excess digits are rounded off from the right.

3. The values of variable C are printed according to the format code G12.6. The s specification, which in this case is 6, determines the number of digits to be printed and whether the number should be printed with or without a decimal exponent. The numbers whose values are between 0.1 and 1000000 are printed without a decimal exponent. Thus the first, third, and fourth values have no decimal exponent. The second value is greater than or equal to 1000000 so it is printed with a decimal exponent.

Scale Factor - P

The P scale factor is used to change the location of the decimal point in real numbers. The effect of the scale factor is:

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

Input: A scale factor can be specified only for basic real numbers. For example, if the input data is in the form xx.xxxx and is to be used internally in the form .xxxxxx, then the format code used to effect this change is 2PF7.4. Or, if the input data is in the form xx.xxxx and is to be used internally in the form xxxx.xx, then the format code used to effect this change is -2PF7.4.

Output: A scale factor can be specified for real numbers with or without E or D decimal exponents. For numbers without an E or D decimal exponent, the effect is the same as for input data except that the decimal point is moved in the opposite direction. For example, if the number has the internal form xx.xxxx and is to be written out in the form xxxx.xx, the format code used to effect this change is 2PF7.4.

For numbers with an E or D decimal exponent, when the decimal point is moved, the exponent is adjusted to account for it, i.e., the value is not changed. For example, if the internal number 238. were printed according to the format E10.3, it would appear as 0.238Eb03. If it were printed according to the format 1PE10.3, it would appear as 2.380Eb02.

A repetition code can precede the D, E, or F format code. For example, 2P3F7.4 is valid.

Warning: Once a scale factor has been given, it holds for all format codes following the scale factor within the same FORMAT statement. This also applies to format codes enclosed in an additional pair of parentheses. A scale factor of 0P must be specified to remove the effect of a previous scale factor.

L Format Code

The L format code is used in transmitting logical variables.

Input: The first T or F encountered in the w characters of the input field causes a value of .TRUE. or .FALSE., respectively, to be assigned to the corresponding logical variable in the I/O list. If the field w consists entirely of blanks, a value of .FALSE. is assumed.

Output: A T or F is inserted in the output record depending upon whether the value of the logical variable in the I/O list was .TRUE. or .FALSE., respectively. The single character is right-justified in the output field and preceded by w-1 blanks.

aAw

A Format Code

The A format code is used in transmitting data that is stored internally in character format. The number of characters transmitted under A format code depends on the length of the corresponding variable in the I/O list. Each alphabetic or special character is given a unique internal code. Numeric data is converted digit by digit into internal format, rather than the entire numeric field being converted into a single binary number. Thus, the A format code can be used for numeric fields, but not for numeric fields requiring arithmetic.

Input: The maximum number of characters stored in internal storage depends on the length of the variable in the I/O list. If w is greater than the variable length, say v, then the leftmost w-v characters in the field of the input card are skipped and the remaining v characters are read and stored in the variable. If w is less than v, then w characters from the field in the input card are read and the remaining rightmost characters in the variable are filled with blanks.

Output: If w is greater than the length of the variable in the I/O list, say the length is v, then the printed field will contain v characters right-justified in the field, preceded by leading blanks. If w is less than v, the leftmost w characters from the variable will be printed and the rest of the data will be truncated.

Example 1: Assume that B has been specified as real of length 8, that N and M are integers of standard length 4, and that the following statements are given.

REAL*8 B
INTEGER*4 N,M

```
25    FORMAT (3A7)

      READ    (5,25) B, N, M
```

When the READ statement is executed, one input card is read from the data set associated with data set reference number 5 into the variables B, N, and M in the format specified by FORMAT statement number 25. The following list shows the values stored for the given input cards (b represents a blank).

Input Card	B	N	M
ABCDEFG46bATb11234567	ABCDEFGb	ATb1	4567
HIJKLMN765432133344445	HIJKLMNb	4321	4445

Example 2: Assume that A and B are real variables of length 4, that C is a real variable of length 8, and that the following statements are given:

```
26    FORMAT    (A6,A5,A6)

      WRITE     (6,26) A,B,C
```

When the WRITE statement is executed, one line is written on the data set associated with data set reference number 6 from the variables A, B, and C in the format specified by FORMAT statement 26. The following list shows the printed output for values of A, B, and C (b represents a blank).

A	B	C	Printed Line
A1B2	C3D4	E5F6G7H8	bbA1B2bC3D4E5F6G7

H Format Code and Literal Data

Literal data can appear in a FORMAT statement in one of two ways: it can be enclosed in apostrophes or it can follow the H format code. For example, the following FORMAT statements are equivalent.

```
25    FORMAT (' 1968 INVENTORY REPORT')

25    FORMAT (22H 1968 INVENTORY REPORT)
```

No item in the I/O list corresponds to the literal data. The data is read or written directly into or from the FORMAT statement. (The FORMAT statement can contain other types of format codes with corresponding variables in the I/O list.)

Input: Information is read from the input card and replaces the literal data in the FORMAT statement. (If the H format code is used, w characters are read. If apostrophes are used, as many characters as there are spaces between the apostrophes are read.) For example, the following statements:

```
8     FORMAT (' HEADINGS')

      READ    (5,8)
```

cause the first 9 characters of the next record to be read from the data set associated with data set reference number 5 into the FORMAT statement 8, replacing the blank and the 8 characters H, E, A, D, I, N, G, and S.

Output: The literal data from the FORMAT statement is written on the output data set. (If the H format code is used, the w characters following the H are written. If apostrophes are used, the characters enclosed in apostrophes are written.) For example, the following statements:

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

     WRITE  (6,8)
```

cause the following record to be written on the data set associated with data set reference number 6:

```
THIS IS ALPHAMERIC INFORMATION
```

Note: If the literal data is enclosed in apostrophes, an apostrophe character in the data is represented by two successive apostrophes. For example, DON'T is represented as DON''T.

X Format Code

The X format code specifies a field of w characters to be skipped on input or filled with blanks on output. For example, the following statements:

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

     READ   (5,5) I,J,K,L,M
```

cause the first 10 characters of the input card to be read into variable I, the next 10 characters to be skipped over without transmission, and the next four fields of 10 characters each to be read into the variables J, K, L, and M.

T Format Code

The T format code specifies the position in the FORTRAN record where the transfer of data is to begin. (If the output is printed, the actual print position is one less than w because of the carriage control character; i.e., the print position corresponds to w-1. For example, the following statements:

```
5    FORMAT (T40,'1968 INVENTORY REPORT',T80,

     'DECEMBER',T1,'0PART NO. 10095')

     WRITE  (6,5)
```

cause the following line to be printed:

```
Print                Print                        Print
Position 1           Position 39                  Position 79
|                    |                            |
v                    v                            v
PART NO. 10095       1968 INVENTORY REPORT        DECEMBER
```

The T format code can be used in a FORMAT statement with any type of format code.

Group Format Specification

The group format specification is used to repeat a set of format codes and to control the order in which the format codes are used.

The group repeat count a is the same as the repeat indicator a which can be placed in front of other format codes. For example, the following statements are equivalent:

```
10    FORMAT  (I3,2(I4,I5),I6)

10    FORMAT  (I3,(I4,I5,I4,I5),I6)
```

Group repeat specifications control the order in which format codes are used since control returns to the last group repeat specification when there are more items in the I/O list than there are format codes in the FORMAT statement (see "Various Forms of a FORMAT Statement"). Thus, in the previous example, if there were more than six items in the I/O list, control would return to the group repeat count 2 which precedes the specification (I4,I5).

If the group repeat count is omitted, a count of 1 is assumed. For example, the statements:

```
15    FORMAT  (I3,(F6.2,D10.3))

      READ    (5,15)  N,A,B,C,D,E
```

cause values to be read from the first record for N, A, and B, according to the format codes I3,F6.2, and D10.3, respectively. Then, because the I/O list is not exhausted, control returns to the last group repeat specification, the next record is read, and values are transmitted to C and D according to the format codes F6.2 and D10.3, respectively. Since the I/O list is still not exhausted, another record is read and a value is transmitted to E according to the format code F6.2 -- the format code D10.3 is not used.

The format codes within the group repeat specification can be separated by commas and slashes. For example, the following statement is valid:

```
40    FORMAT  (2I3/(3F6.2,F6.3/D10.3,3D10.2))
```

The first record is transmitted according to the specification 2I3, the second, fourth, etc., records are transmitted according to the specification 3F6.2,F6.3, and the third, fifth, etc., records are transmitted according to the specification D10.3,3D10.2, until the I/O list is exhausted.

Reading FORMAT Specifications at Object Time

FORTRAN provides for variable FORMAT statements by allowing a FORMAT specification to be read into an array in storage and using the data in the array as the FORMAT specification for subsequent input/output statements.

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

2. The format codes read into the array at object time must have the same form as a source program FORMAT statement, except that the word FORMAT is omitted.

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.

Example: Assume that the following statements are given:

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

and that the first input card associated with data set reference number 5 contains (2E10.3, 5F10.8).

The data on the rest of the input cards is read, converted, and stored in A, B, and the array C, according to the format codes 2E10.3, 5F10.8.

END FILE STATEMENT

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

The END FILE statement defines the end of the data set associated with a.

REWIND STATEMENT

General Form
REWIND a Where: a is an unsigned integer constant or integer variable that is of length 4 and 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 record of the data set associated with a.

BACKSPACE STATEMENT

General Form
BACKSPACE a Where: a is an unsigned integer constant or integer variable that is of length 4 and 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. For further information, see the FORTRAN IV programmers' guides listed in the Preface.

DIRECT ACCESS INPUT/OUTPUT STATEMENTS

There are four direct access input/output statements: READ, WRITE, DEFINE FILE, and FIND. The READ and WRITE statements cause transfer of data into or out of internal storage. These statements allow the user to specify the location within a data set from which data is to be read or into which data is to be written.

The DEFINE FILE statement specifies the characteristics of the data set(s) to be used during a direct access operation. The FIND statement overlaps record retrieval from a direct access device with computation in the program. In addition to these four statements, the FORMAT statement (described previously) specifies the form in which data is to be transmitted. The direct access READ and WRITE statements and the FIND statement are the only input/output statements that may refer to a data set reference number defined by a DEFINE FILE statement.

Each record in a direct access data set has a unique record number associated with it. The programmer must specify in the READ, WRITE, and FIND statements not only the data set reference number, as for sequential input/output statements, but also the number of the record to be read, written, or found. Specifying the record number permits operations to be performed on selected records of the data set, instead of on records in their sequential order.

The number of the record physically following the one just processed is made available to the program in an integer variable known as the associated variable. Thus, if the associated variable is used in a READ or WRITE statement to specify the record number, sequential processing is automatically secured. The associated variable is specified in the DEFINE FILE statement, which also gives the number, size, and type of the records in the direct access data set.

DEFINE FILE STATEMENT

The DEFINE FILE statement describes the characteristics of any data set to be used during a direct access input/output operation. To use the direct access READ, WRITE, and FIND statements in a program, the data set(s) must be described with a DEFINE FILE statement. Each data set must be described once, and this description may appear once in each program or subprogram. Subsequent descriptions have no effect.

The DEFINE FILE statement must logically precede any input/output statement referring to the data set described in the DEFINE FILE statement.

General Form

DEFINE FILE $a_1(m_1,r_1,f_1,v_1),a_2(m_2,r_2,f_2,v_2),\ldots,a_n(m_n,r_n,f_n,v_n)$

Where: *a* represents an integer constant that is the data set reference number.

m represents an integer constant that specifies the number of records in the data set associated with *a*.

r represents an integer constant that specifies the maximum size of each record associated with *a*. The record size is measured in characters (bytes), storage locations (bytes), or storage units (words). (A storage unit is the number of storage locations divided by four and rounded to the next highest integer.) The method used to measure the record size depends upon the specification for *f*.

f specifies that the data set is to be read or written either with or without format control; *f* may be one of the following letters:

L indicates that the data set is to be read or written either with or without format control. The maximum record size is measured in number of storage locations (bytes).

E indicates that the data set is to be read or written under format control (as specified by a format statement). The maximum record size is measured in number of characters (bytes).

U indicates that the data set is to be read or written without format control. The maximum record size is measured in number of storage units (words).

v represents a nonsubscripted integer variable called an associated variable. At the conclusion of each read or write operation, *v* is set to a value that points to the record that immediately follows the last record transmitted. At the conclusion of a find operation, *v* is set to a value that points to the record found.

The associated variable cannot appear in the I/O list of a READ or WRITE statement for a data set associated with the DEFINE FILE statement.

Example:

```
DEFINE FILE 8(50,100,L,I2),9(100,50,L,J3)
```

This DEFINE FILE statement describes two data sets, referred to by data set reference numbers 8 and 9. The data in the first data set consists of 50 records, each with a maximum length of 100 storage locations. The L specifies that the data is to be transmitted either with or without format control. I2 is the associated variable that serves as a pointer to the next record.

The data in the second data set consists of 100 records, each with a maximum length of 50 storage locations. The L specifies that the data is to be transmitted either with or without format control. J3 is the associated variable that serves as a pointer to the next record.

If an E is substituted for the L in the preceding DEFINE FILE statement, a FORMAT statement is required and the data is transmitted under format control. If the data is to be transmitted without format control, the DEFINE FILE statement can be written as:

```
DEFINE FILE 8(50,25,U,I2),9(100,13,U,J3)
```

DIRECT ACCESS PROGRAMMING CONSIDERATIONS

When programming for direct access input/output operations, the user must establish a correspondence between FORTRAN records and the records described by the DEFINE FILE statement. All conventions of FORMAT control discussed in the section "FORMAT Statement" are applicable.

For example, to process the data set described by the statement:

```
DEFINE FILE 8(10,48,L,K8)
```

the FORMAT statement used to control the reading or writing could not specify a record longer than 48 characters. The statements:

```
FORMAT(4F12.1)    or
FORMAT(I12,9F4.2)
```

define a FORTRAN record that corresponds to those records described by the DEFINE FILE statement. The records can also be transmitted under FORMAT control by substituting an E for the L and rewriting the DEFINE FILE statement as:

```
DEFINE FILE 8(10,48,E,K8)
```

To process a direct access data set without format control, the number of storage locations specified for each record must be greater than or equal to the maximum number of storage locations in a record to be written by any WRITE statement referencing the data set. For example, if the I/O list of the WRITE statement specifies transmission of the contents of 100 storage locations, the DEFINE FILE statement can be either:

```
DEFINE FILE 8(50,100,L,K8) or
DEFINE FILE 8(50,25,U,K8)
```

Programs may share an associated variable as a COMMON variable. The following example shows how this can be accomplished.

```
    COMMON IUAR                              SUBROUTINE SUBI(A,B)
    DEFINE FILE 8(100,10,L,IUAR)             COMMON IUAR
         .                                        .
         .                                        .
         .                                        .
    ITEMP=IUAR
    CALL SUBI(ANS,ARG)
8   IF (IUAR-ITEMP) 20,16,20
         .
         .
         .
```

In this example, the program and the subprogram share the associated variable IUAR. An input/output operation that references data set 8 and is performed in the subroutine causes the value of the associated variable to be changed. The associated variable is then tested in the main program in statement 8.

READ STATEMENT

The READ statement causes data to be transferred from a direct access device into internal storage. The data set being read must be defined with a DEFINE FILE statement.

General Form
READ (a'r, b, ERR=d) list Where: a is an integer constant or unsigned integer variable that is of length 4 and represents a data set reference number; a must be followed by an apostrophe ('). r is an integer expression that represents the relative position of a record within the data set associated with a. b is optional and, if given, is either the statement number of the FORMAT statement that describes the data being read or the name of an array that contains an object time format. ERR=d is optional and d is the statement number to which control is given when a device error condition is encountered during data transfer from device to storage. list is optional and is an I/O list.

The I/O list must not contain the associated variable defined in the DEFINE FILE statement for data set a.

Example:

```
      DEFINE FILE 8(500,100,L,ID1),9(100,28,L,ID2)
      DIMENSION M(10)
              .
              .
              .
      ID2 = 21
              .
              .
              .
   10 FORMAT (5I20)
    9 READ (8'16,10) (M(K),K=1,10)
              .
              .
              .
   13 READ (9'ID2+5) A,B,C,D,E,F,G
```

READ statement 9 transmits data from the data set associated with data set reference number 8, under control of FORMAT statement 10; transmission begins with record 16. Ten data items of 20 characters each are read as specified by the I/O list and FORMAT statement 10. Two records are read to satisfy the I/O list, because each record contains

only five data items (100 characters). The associated variable ID1 is set to a value of 18 at the conclusion of the operation.

READ statement 13 transmits data from the data set associated with data set reference number 9, without format control; transmission begins with record 26. Data is read until the I/O list for statement 13 is satisfied. Because the DEFINE FILE statement for data set 9 specified the record length as 28 storage locations, the I/O list of statement 13 calls for the same amount of data (the seven variables are type real and each occupies four storage locations). The associated variable ID2 is set to a value of 27 at the conclusion of the operation. If the value of ID2 is unchanged, the next execution of statement 13 reads record 32.

The DEFINE FILE statement in the previous example can also be written as:

```
DEFINE FILE 8(500,100,E,ID1),9(100,7,U,ID2)
```

The FORMAT statement may also control the point at which reading starts. For example, if statement 10 in the example is

```
10 FORMAT (//5I20)
```

records 16 and 17 are skipped, record 18 is read, records 19 and 20 are skipped, record 21 is read, and ID1 is set to a value of 22 at the conclusion of the READ operation in statement 9.

WRITE STATEMENT

The WRITE statement causes data to be transferred from internal storage to a direct access device. The data set being written must be defined with a DEFINE FILE statement.

General Form
WRITE (a'r,b) list Where: a is an integer constant or unsigned integer variable that is of length 4 and represents a data set reference number; a must be followed by an apostrophe ('). r is an integer expression that represents the relative position of a record within the data set associated with a. b is optional and, if given, is either the statement number of the FORMAT statement that describes the data being written or the name of an array that contains an object time format. list is optional and is an I/O list.

Example:

```
      DEFINE FILE 8(500,100,L,ID1),9(100,28,L,ID2)
      DIMENSION M(10)
              .
              .
              .
      ID2=21
              .
              .
              .
   10 FORMAT (5I20)
    8 WRITE (8'16,10) (M(K),K=1,10)
              .
              .
              .
   11 WRITE (9'ID2+5) A,B,C,D,E,F,G
```

WRITE statement 8 transmits data into the data set associated with the data set reference number 8, under control of FORMAT statement 10; transmission begins with record 16. Ten data items of 20 characters each are written as specified by the I/O list and FORMAT statement 10. Two records are written to satisfy the I/O list because each record contains 5 data items (100 characters). The associated variable ID1 is set to a value of 18 at the conclusion of the operation.

WRITE statement 11 transmits data into the data set associated with data set reference number 9, without format control; transmission begins with record 26. The contents of 28 storage locations are written as specified by the I/O list for statement 11. The associated variable ID2 is set to a value of 27 at the conclusion of the operation. Note the correspondence between the records described (28 storage locations per record) and the number of items called for by the I/O list (7 variables, type real, each occupying four storage locations).

The DEFINE FILE statement in the example can also be written as:

```
DEFINE FILE 8(500,100,E,ID1), 9(100,7,U,ID2)
```

As with the READ statement, a FORMAT statement may also be used to control the point at which writing begins.

FIND STATEMENT

The FIND statement causes the next input record to be found while the present record is being processed, thereby increasing the execution speed of the object program. The program has no access to the record that was found until a READ statement for that record is executed. (There is no advantage to having a FIND statement precede a WRITE statement.)

General Form
FIND (a'r)
Where: a is an integer constant or unsigned integer variable that is of length 4 and represents a data set reference number; a must be followed by an apostrophe (').
r is an integer expression that represents the relative position of a record within the data set associated with a.

The data set on which the record is being found must be defined with a DEFINE FILE statement.

Example:

```
    10  FIND (8'50)
          .
          .
          .
    15  READ (8'50) A,B
```

While the statements between statements 10 and 15 are executed, record 50, in the data set associated with data set reference number 8, is found.

General Example -- Direct Access Operations

```
        DEFINE FILE 8(1000,72,L,ID8)
        DIMENSION A(100),B(100),C(100),D(100),E(100),F(100)
           .
           .
     15 FORMAT (6F12.4)
        FIND (8'5)
           .
           .
        ID8=1
        DO 100 I=1,100
        READ (8'ID8+4,15)A(I),B(I),C(I),D(I),E(I),F(I)
    100 CONTINUE
           .
           .
        DO 200 I=1,100
        WRITE (8'ID8+4,15)A(I),B(I),C(I),D(I),E(I),F(I)
    200 CONTINUE
           .
           .
        END
```

The general example illustrates the ability of direct access statements to gather and disperse data in an order designated by the user. The first DO loop in the example fills arrays A through F with data from the 5th, 10th, 15th,..., and 500th record associated with data set reference number 8. Array A receives the first value in every fifth record, B the second value and so on, as specified by FORMAT statement 15 and the I/O list of the READ statement. At the end of the READ operation, each record has been dispersed into arrays A through F. At the conclusion of the first DO loop, ID8 has a value of 501.

The second DO loop in the example groups the data items from each array, as specified by the I/O list of the WRITE statement and FORMAT statement 15. Each group of data items is placed in the data set associated with data set reference number 8. Writing begins at the 505th record and continues at intervals of five, until record 1000 is written, if ID8 is not changed between the last READ and the first WRITE.

General Form

DATA $v_1,\ldots,v_n/i_1*d_1,\ldots,i_n*d_n/,v_{n+1},\ldots,v_m/i_{n+1}*d_{n+1},\ldots,i_m*d_m/,\ldots$

Where: $v_1,\ldots,v_m$ are variables, subscripted variables (in which case the subscripts must be integer constants), or array names. Dummy arguments are not permitted.

$d_1,\ldots,d_m$ are values representing integer, real, complex, hexadecimal, logical, or literal data constants.

$i_1,\ldots,i_m$ represent unsigned integer constants indicating the number of consecutive variables that are to be assigned the value of $d_1,\ldots,d_m$.

A DATA initialization statement is used to define initial values of variables, array elements, and arrays. There must be a one-to-one correspondence between the total number of variables, subscripted variables, and array elements specified or implied by $v_1,\ldots,v_n$ and the total number of constants specified by $d_1,\ldots,d_n$ after application of any replication factors, $i_1,\ldots,i_n$. The DATA initialization statement can appear anywhere in the program as long as it does not assign a value to a variable that has not yet been defined.

Example 1:

```
DIMENSION D(5,10)
DATA A, B, C/5.0,6.1,7.3/,D,E/25*1.0,25*2.0,5.1/
```

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. In addition, the statement specifies that the first 25 variables in the array D are to be initialized to the value 1.0, the remaining 25 variables in D to the value 2.0, and the variable E to the value 5.1.

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. In addition, 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. In a labeled common block, they may be initially defined only in a BLOCK DATA subprogram. (See the section "Subprograms.")

SPECIFICATION STATEMENTS

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 locations in storage for this data.

Specification statements must precede statement function definitions, which must precede the program part containing at least one executable statement. Within the specification statements, any statement describing data must precede references to that data. In particular, the IMPLICIT statement, if used, must be the first specification statement.

The specification statement EXTERNAL is described in the section "Subprograms."

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 the DIMENSION statement in which they appear is in 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), LIST (10,100)
DIMENSION B(25,50),TABLE(5,8,4)
```

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 statement enables the user to:

- Specify the type of a group of variables or arrays according to the initial character of their names.
- Specify the amount of storage to be allocated for each variable according to the associated type.

The Explicit specification statements enable the user to:

- Specify the type of a variable or array according to its particular name.
- Specify the amount of storage to be allocated for each variable according to the associated type.
- Specify the dimensions of an array.
- Assign initial data values for variables and arrays.

The Explicit specification statement overrides the IMPLICIT statement, which, in turn, overrides the predefined convention for specifying type.

IMPLICIT STATEMENT

General Form
IMPLICIT type*s(a_1,a_2,...),...,type*s(a_1,a_2,...) Where: type is one of the following: INTEGER, REAL, COMPLEX, or LOGICAL. *s is optional and represents one of the permissible length specifications for its associated type. a_1, a_2,... are 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 specification statement must be the first statement in a main program and the second statement in a subprogram. There can be only one IMPLICIT statement per program or subprogram. The IMPLICIT specification 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 (bytes) to be allocated for each in the group of specified variables. The types that a variable may assume, along with the permissible length specifications, are as follows:

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, then 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, namely, O through Z and $, are 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.

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.

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 are variable, array, or function names (see the section "Subprograms")

(k$_1$),(k$_2$),...,(k$_n$) are optional and give dimension information for arrays. 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 of length four only when the Type statement in which it appears is 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 particular 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., predefined convention and the IMPLICIT statement). In addition, the information necessary to allocate storage for arrays (dimension information) may be included within the 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. Lists of constants are used only to assign initial values to array elements. r successive occurrences of the same constant can be represented by the form r*constant. If initial data values are assigned to an array in an Explicit specification statement, the dimension information for the array must be in the Explicit specification statement or in a preceding DIMENSION or COMMON statement. An initial data value may not be assigned to a function name.

Initial data values cannot be assigned to variables or arrays in blank common. The BLOCK DATA subprogram must be used to assign initial values to variables and arrays in labeled common.

In the same manner in which the IMPLICIT statement overrides the predefined convention, the Explicit specification statements override the IMPLICIT statement 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, each with two storage locations reserved. In addition, 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, the standard length is assumed. Thus, C and D each have eight storage locations reserved (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. The variables ARRAY and HOLD have eight storage locations reserved for each; the variable VALUE has four storage locations reserved; and the array named 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/,TEST*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) and the array B has 400 storage locations reserved (four for each variable). In addition, 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 TEST has 40 storage locations reserved (eight for each variable). In addition, each variable is initialized to the value 0.0.

DOUBLE PRECISION STATEMENT

General Form
DOUBLE PRECISION $a(k_1),b(k_2),\ldots,z(k_n)$
Where: $a,b,\ldots,z$ represent variable, array, or function names (see the section "Subprograms")
$(k_1),(k_2),\ldots,(k_n)$ are optional. Each k is composed of 1 through 7 unsigned integer constants, separated by commas, that represent the maximum value of each subscript in the array.

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. This statement cannot be used to define initial data values.

In addition, FUNCTION subprograms may be typed double-precision as follows:

DOUBLE PRECISION FUNCTION name (a_1,a_2,a_3,...,a_n)

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 names or array names that cannot be dummy arguments.✓
k_1,k_2,...k_3,k ... are optional and are each composed of 1 through 7 unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.✓
/r/... represent optional common block names consisting of 1 through 6 alphameric characters, the first of which is alphabetic. These names must always be embedded in slashes.✓

The COMMON statement is used to define a storage area that can be referred to by a calling program and one or more subprograms and to specify the names of variables and arrays to be placed in this area. Therefore, variables or arrays that appear in a calling program or subprogram can be made to share the same storage locations with variables or arrays in other subprograms. Also, a common area can be used to implicitly transfer arguments between a calling program and a subprogram. Arguments passed in common are subject to the same rules with regard to type, length, etc., as arguments passed in an argument list (see the section "Subprograms").

If more than one COMMON statement appears in a calling program or subprogram, the entries in the statements are cumulative. Redundant entries are not permitted.

Although the entries in a COMMON statement can contain dimension information, object-time dimensions may never be used.

The length of a common area can be increased by using an EQUIVALENCE statement.

Since the entries in a common area share storage locations, the order in which they are entered is significant. Consider the following example:

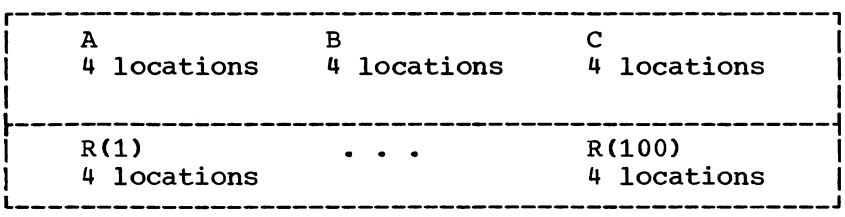
Example:

```
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 the following order:



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), respectively. Note that 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.

BLANK AND LABELED COMMON

In the preceding example, the common storage area (common block) 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 1 through 6 alphameric characters (the first of which is alphabetic); those blocks which have the same name occupy the same storage space. This permits a calling program to share one common block with one subprogram and another common block with another subprogram and also facilitates program documentation.

Those 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 the following 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 the following statement:

```
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 the following two COMMON statements:

```
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:

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

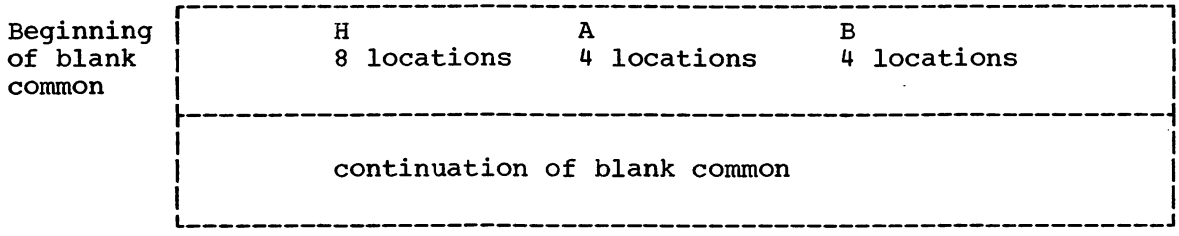
```
      .
      .
      .
COMMON H, A /R/ X, D // B
      .
      .
      .
CALL MAPMY(...)
      .
      .
      .
```

Subprogram

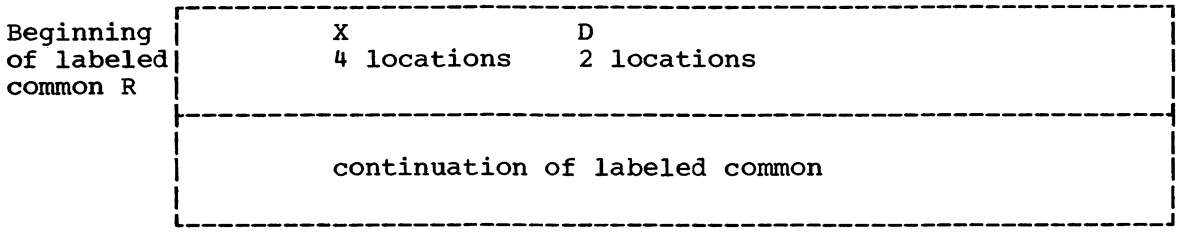
```
SUBROUTINE MAPMY(...)
      .
      .
      .
COMMON G, Y, C /R/ K, E
      .
      .
      .
```

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 the following order:



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



The statement COMMON G,Y,C/R/K,E appearing in the subprogram MAPMY would then 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.

ARRANGEMENT OF VARIABLES IN COMMON

Variables in a common block need not be aligned properly. However, considerable object-time efficiency is lost unless the programmer ensures that all of 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 the following 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)
```

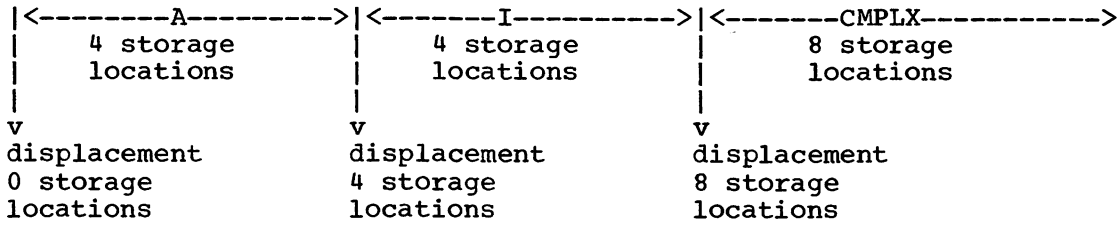
If the fixed order is not used, proper alignment can be ensured by constructing the block so that the displacement of each variable can be evenly divided by the reference number associated with the variable. (Displacement is the number of storage locations (bytes) from the beginning of the block to the 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 though 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 as follows:



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 (not dummy arguments) 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 multi-subscripted it refers to the position in the array in the same fashion 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 unit. In particular, when the logic of the program permits it, the number of storage locations used can be reduced by causing locations to be shared by two or more variables of the same or different types. Equivalence between variables implies storage sharing only, not mathematical equivalence.

Since arrays are stored in a predetermined order (see "Arrangement of Arrays in Storage"), equivalencing two elements of two different arrays may implicitly equivalence other elements of the two arrays. The EQUIVALENCE statement must not contradict itself or any previously established equivalences.

Two variables in one common block or in two different common blocks cannot be made equivalent. However, a variable in a program or a subprogram can be made equivalent to a variable in a common block. If the variable that is equivalenced to a variable in the common block is an element of an array, the implicit equivalencing of the rest of the elements of the array may extend the size of the common block (see example 2). The size of the common block must not be extended so that elements are added before the beginning of the established common block.

Example 1:

Assume that in the initial part of a program, an array C of size 100x100 is needed; in the final stages of the program C is no longer used, but arrays A and B of sizes 50x50 and 100, respectively, are used. The elements of all three arrays are of the type REAL*4. Storage space can then be saved by using the statements:

```
DIMENSION C(100,100), A(50,50), B(100)
EQUIVALENCE (C(1), A(1)), (C(2501), B(1))
```

The array A, which has 2500 elements, can occupy the same storage as the first 2500 elements of array C since the arrays are not both needed at the same time. Similarly, the array B can be made to share storage with elements 2501 to 2600 of array C.

Example 2:

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

This EQUIVALENCE statement specifies that the variables A, B(1), and C(5,3) are assigned the same storage locations and that variables D(5,10,2) and E are assigned the same storage locations. It also implies that B(2) and C(6,3), etc., are assigned the same storage locations. Note that further equivalence specification of B(2) with any element of array C other than C(6,3) is invalid.

The designations C(5,3) and D(5,10,2) could have been replaced with the designations C(25) and D(100) and the effect would have been the same.

Example 3:

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

Explanation:

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 same storage location as B, D(2) to share the same storage location as C, and D(3) would extend the size of the common area, in the following manner:

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

The following 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)
```

ARRANGEMENT OF VARIABLES IN EQUIVALENCE GROUPS

Variables in an equivalence group may be in any order. However, considerable object-time efficiency is lost unless the programmer ensures that all of 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 the following order:

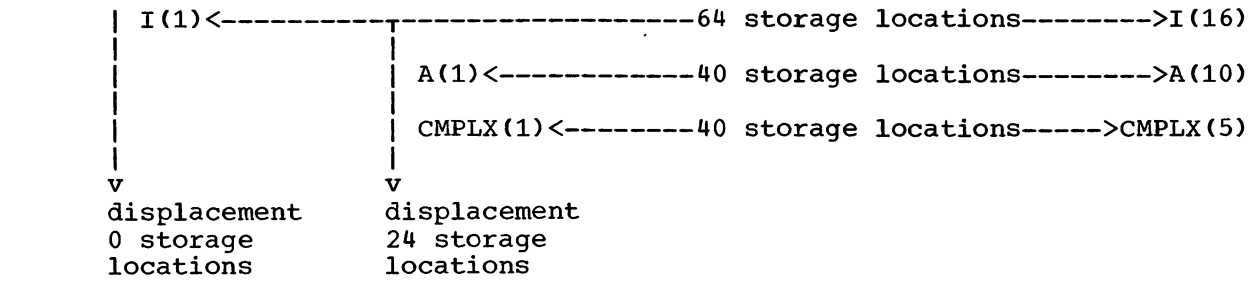
```
length of 16 (complex)
length of 8  (complex or real)
length of 4  (real or integer or logical)
length of 2  (integer)
length of 1  (logical)
```

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 (bytes) from the beginning of the group to the first storage location of the variable.) The reference numbers for each type of variable are given in the section "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 as follows:



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 of 8).

It is sometimes desirable to write a program which, at various points, requires the same computation to be performed with different data for each calculation. It would simplify the writing of that program if the statements required to perform the desired 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 with this object in mind. 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 two classes of subprograms: FUNCTION subprograms and SUBROUTINE subprograms. In addition, there is a group of FORTRAN supplied subprograms (see Appendix C). Functions differ from SUBROUTINE subprograms in that they return at least one value to the calling program, whereas SUBROUTINE subprograms need not return any.

Statement functions are also discussed in this section since they are similar to FUNCTION subprograms. The difference is that subprograms are a separate program unit from the program unit referring to them while statement functions definitions and references are in the same program unit.

NAMING SUBPROGRAMS

A subprogram name consists of from 1 through 6 alphameric characters, the first of which must be alphabetic. A subprogram name may not contain special characters (see Appendix A). The type of a function determines the type of the result that can be returned from it.

- 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 rules for declaring the type of variables apply to statement functions.

- Type Declaration of FUNCTION Subprograms: The declaration may be made by the predefined convention, by the IMPLICIT statement, by an Explicit specification in the FUNCTION statement, or by an Explicit specification statement within the FUNCTION 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.

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

There are three steps in the definition of a function in FORTRAN:

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

2. The dummy 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 (e.g., "Statement Functions," "FUNCTION Subprograms," etc.).

Function Reference

When the name of a function, followed by a list of its arguments, appears in any FORTRAN expression, it references the function and 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 of the name used for the reference must agree with the type of the name used in the definition.

STATEMENT FUNCTIONS

A statement function definition specifies operations to be performed whenever that statement function name appears as a function reference in another statement in the same program unit.

General Form
name(a_1,a_2,a_3,...,a_n) = expression
Where: name is the statement function name (see the section "Naming Subprograms")
a_1,a_2,a_3,...,a_n are dummy arguments. They must be unique (within the statement) nonsubscripted variables.
expression is any arithmetic or logical expression that does not contain subscripted variables. Any statement function appearing in this expression must have been defined previously.

The expression to the right of the equal sign defines the operations to be performed when a reference to this function appears in an assignment statement. The expression defining the function must not contain a reference to the function.

The dummy arguments enclosed in parentheses following the function name are dummy variables for which the arguments given in the function reference are substituted when the function reference is encountered. The same dummy arguments may be used in more than one statement function definition and may be used as variables outside the statement function definitions.

The actual arguments in the function reference must correspond in order, number, and type to the dummy arguments. There must be at least one argument. The arguments can be any of the following: any type of constant except hexadecimal, any type of subscripted or nonsubscripted variable, an array name, an arithmetic or logical expression, or the name of another subprogram.

All statement function definitions to be used in a program must precede the first executable statement of the program.

Example: The statement:

```
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 dummy arguments. The expression to the right of the equal sign defines the operations to be performed when the function reference appears in an arithmetic statement.

The function reference might appear in a statement as follows:

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

This is equivalent to:

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

Note the correspondence between the dummy arguments A and B in the function definition and the actual arguments D and E in the function reference.

Examples:

Valid statement function definitions and statement function references:

Definition	Reference
SUM(A,B,C,D) = A+B+C+D	NET = GROS-SUM(TAX,FICA,HOSP,STOCK)
FUNC(Z) = A+X*Y*Z	ANS = FUNC(RESULT)
VALID(A,B) = .NOT. A .OR. B	VAL = TEST .OR. VALID(D,E)
	BIG SUM = SUM(A,B,SUM(C,D,E,F),G)

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 nonsubscripted)
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)
BAD(A,B)=A+B+BAD(C,D)	(recursive definition not permitted)

Invalid statement function references (the functions are defined as above):

```
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 a FUNCTION statement followed by other statements including at least one RETURN statement. It is an independently written program that is executed wherever its name is referenced in another program.

General Form
Type FUNCTION name*s (a_1,a_2,a_3,...,a_n)
Where: Type is INTEGER, REAL, COMPLEX, or LOGICAL. It can be omitted if the predefined convention is used to type the function.
name is the name of the FUNCTION.
*s is optional and represents one of the permissible length specifications for its associated type.
a_1,a_2,a_3,...,a_n are dummy arguments. They must be nonsubscripted variable, array, or dummy names of SUBROUTINE or other FUNCTION subprograms. (There must be at least one argument in the argument list.)

The function must also be typed in the calling program, if the predefined convention is not used.

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

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

The name of the function must be assigned a value at least once in the subprogram -- either as the variable name to the left of the equal sign in an assignment statement, as an argument of a CALL statement, or in an input list (READ statement) within the subprogram.

The dummy arguments of the FUNCTION subprogram (i.e., $\underline{a}_1, \underline{a}_2, \underline{a}_3, \ldots, \underline{a}_n$) may be considered to be dummy variable names. These are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. Additional information about arguments is in the section "Arguments in a FUNCTION or SUBROUTINE Subprogram."

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 in the following example:

Example 1:

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:

In this example, 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.

Example 2:

Calling Program

```
INTEGER*2 CALC
      .
      .
      .
ANS=ROOT1*CALC(N,M,P)
      .
      .
      .
```

FUNCTION Subprogram

```
INTEGER FUNCTION CALC*2(I,J,K)
        .
        .
        .
CALC = I+J+K**2
        .
        .
        .
RETURN
END
```

Explanation:

The FUNCTION subprogram CALC is declared as type INTEGER of length 2.

RETURN and END Statements in a FUNCTION Subprogram

All FUNCTION subprograms must contain an END statement 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 the computed value and control to the calling program. There may be more than one RETURN statement 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
```

SUBROUTINE SUBPROGRAMS

The SUBROUTINE subprogram is similar to the FUNCTION subprogram in many respects. The rules for naming FUNCTION and SUBROUTINE subprograms are similar. 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 need not return any results to the calling program, as does the FUNCTION subprogram.

The SUBROUTINE subprogram is referenced by the CALL statement, which consists of the word CALL followed by the name of the subprogram and its parenthesized arguments.

General Form

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

Where: name is the SUBROUTINE name (see the section "Naming Subprograms").

$a_1, a_2, a_3, \ldots, a_n$ are dummy 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.

Since the SUBROUTINE is a separate program, 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.

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

The dummy arguments ($a_1, a_2, a_3, \ldots, a_n$) may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. Additional information about dummy arguments is in the section "Arguments in a FUNCTION or SUBROUTINE Subprogram."

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

```
Calling Program                 SUBROUTINE Subprogram

DIMENSION X(100),Y(100)
   .                            SUBROUTINE COPY(A,B,N)
   .                            DIMENSION A (100),B(100)
K = 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 to call a SUBROUTINE subprogram.

General Form
CALL name (a_1,a_2,a_3,...,a_n) Where: name is the name of a 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").

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.

RETURN Statements 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; i may be specified only in a SUBROUTINE subprogram.

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 RETURN i. 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

```
      .
      .
      .
10 CALL SUB (A,B,C,&30,&40)
20 Y = A + B
      .
      .
      .
30 Y = A + C
      .
      .
      .
40 Y = B + C
      .
      .
      .
   END
```

Subprogram

```
      SUBROUTINE SUB (X,Y,Z,*,*)
          .
          .
          .
  100 IF (M) 200,300,400
  200 RETURN
  300 RETURN 1
  400 RETURN 2
      END
```

Explanation:

In the preceding example, execution of statement 10 in the calling program causes entry into subprogram SUB. When statement 100 is executed, the return to the calling program will be to statement 20, 30, or 40, if M 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, the following CALL statement:

```
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.

ARGUMENTS IN A FUNCTION OR SUBROUTINE SUBPROGRAM

The dummy arguments of a subprogram appear after the FUNCTION or SUBROUTINE name and are enclosed in parentheses. They are replaced at the time of execution by the actual arguments supplied in the CALL statement or function reference in the calling program. The dummy arguments must correspond in number, order, type, and length to the actual arguments. For example, if an actual argument is an integer constant, then the corresponding dummy argument must be an integer of length 4. The array sizes must also be the same except when the arrays are one-dimensional, in which case, the actual argument array size can be less than or equal to the dummy argument array size.

The actual arguments can be:

- Any type of constant except hexadecimal
- Any type of subscripted or nonsubscripted variable (except one last defined by an ASSIGN statement)
- An array name
- An arithmetic or logical expression
- The name of a FUNCTION or SUBROUTINE subprogram
- A statement number (for a SUBROUTINE subprogram only, see the section "RETURN Statements in a SUBROUTINE Subprogram")

If a literal constant is passed as an argument, the actual argument passed is the literal as defined, without delimiting apostrophes or the preceding wH specification. An actual argument which is the name of a subprogram must be identified by an EXTERNAL statement containing that name.

When the dummy argument is an array name, an appropriate DIMENSION or Explicit specification statement must appear in the subprogram. None of the dummy arguments may appear in an EQUIVALENCE or COMMON statement.

If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a subscripted or unsubscripted variable name, or an array name. A constant should not be specified as an actual argument unless the programmer is certain that the corresponding dummy argument is not assigned a value in the subprogram.

A referenced subprogram cannot define dummy arguments such that the subprogram reference causes those arguments to be associated with other dummy arguments within the subprogram or with variables in COMMON. For example, if the function DERIV is defined as

```
FUNCTION DERIV (X,Y,Z)
COMMON W
```

and if the following statements are included in the calling program

```
COMMON B
    .
    .
    .
C = DERIV (A,B,A)
```

then X, Y, Z, and W cannot be defined (e.g., cannot appear to the left of an equal sign in an arithmetic statement) in the function DERIV.

Arguments may be referred to in a subprogram in one of two ways: by value or by location.

In reference by value, the dummy argument is assigned a storage location in the subprogram to which the value of the actual argument is brought from the calling program at execution time. During execution, all intermediate values are also stored in this location. Upon return to the calling program, the final value is transmitted from the dummy argument to the actual argument.

An argument is referenced by value when the corresponding dummy argument is enclosed only in commas and is not an array name or subprogram name.

In reference by location, no storage is assigned to the dummy argument and during execution of the subprogram, all intermediate values and the final value are referenced using the location of the actual argument.

An argument is referenced by location when the corresponding dummy argument is enclosed in slashes, or declared to be an array name or a subprogram name.

Example:

```
Calling Program                 Subprogram
      .
      .
      .                         SUBROUTINE SUB(X,Y,Z)
CALL SUB (A,B(1),C)                   .
      .                               .
      .                               .
      .
```

Explanation:

The actual arguments A, B(1), and C are associated with X, Y, and Z, respectively. The arguments A, B(1), and C are referred to by value.

Example:

```
Calling Program                 Subprogram
      .
      .
      .                         SUBROUTINE SUB(/X/,/Y/,Z)
CALL SUB (A,B(1),C)                   .
      .                               .
      .                               .
      .
```

Explanation:

The actual arguments A,B(1), and C are associated with X, Y, and Z, respectively. The arguments A and B(1) are referred to by location, C is referred to by value.

MULTIPLE ENTRY INTO A SUBPROGRAM

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.

It is also possible to enter a subprogram (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 (see the section "Naming Subprograms"). 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 reference.

ENTRY statements are non-executable and do not affect control sequencing during execution of a subprogram. A subprogram must not reference itself directly or through any of its entry points. Entry cannot be made into the range of a DO. The appearance of an ENTRY statement does not alter the rule that statement functions in subprograms must precede the first executable statement of the subprogram.

The dummy arguments in the ENTRY statement need not agree in order, type, or number with the dummy arguments in the SUBROUTINE or FUNCTION statement or any other ENTRY statement in the subprogram. However, the arguments for each CALL or function reference must agree in order, type, and number with the dummy arguments in the SUBROUTINE, FUNCTION, or ENTRY statement that it references.

Entry into a subprogram initializes the dummy arguments of the referenced ENTRY statement. Thus, all appearances of these arguments in the whole subprogram are initialized. Arguments that were referenced by value at some previous use of the subprogram need not appear in the argument list of the ENTRY statement. In this case, the reference will not transmit new values for the arguments not listed. A function reference, and hence the corresponding ENTRY statement, must have at least one argument.

If a dummy argument is listed at more than one entry point, it must be consistently referenced either by name or by value. A dummy argument must not be used in any executable statement in the subprogram unless it has been previously defined as a dummy argument in an ENTRY, SUBROUTINE, or FUNCTION statement.

If information for an object-time dimension array is passed in a reference to an ENTRY statement, the array name and all of its dimension parameters (except any that are in a common area) must appear in the argument list of the ENTRY statement.

In a FUNCTION subprogram, the types of the function name and entry name are determined by the FUNCTION and ENTRY statements. The types of these variables (i.e., the function name and entry names) can be different; the variables are treated as if they were equivalenced. After one of these variables is assigned a value in the subprogram, the others become indeterminate in value.

Upon exit from a FUNCTION subprogram, the value returned is the value last assigned to the function name or any entry name. It is returned as though it were assigned to the name in the current function reference. If the last value is assigned to a different entry name, and that entry name differs in type from the name in the current function reference, the value of the function is undefined.

Example 1:

Calling Program

```
      .
      .
      .
TABLE(1) = FUNC(W,X,Y,Z)
DO 5 I=2,100
TABLE(I) = ENT(U)
      .
      .
      .
5 CONTINUE
      .
      .
      .
```

Subprogram

```
FUNCTION FUNC(T,A,B,C)
        .
        .
        .
ENTRY ENT(T)
        .
        .
        .
FUNC = A * B + C ** T
RETURN
        .
        .
        .
END
```

Explanation: The FUNCTION subprogram is entered once at entry point FUNC and initial values are assigned to the dummy arguments T, A, B, and C. Thereafter, the FUNCTION subprogram is entered at entry point ENT, and only one value is transmitted. No new values are passed for A, B, or C, so their values are changed only by operations in the subprogram. (Note that the original reference to A, B, and C must be by value -- not a reference by location.)

Each time, the result of the FUNCTION subprogram is returned to the main program function reference by the variable FUNC. If FUNC and ENT had been of different types, it would have been necessary to have returned the result by FUNC the first time and by ENT the rest of the times.

Example 2:

Calling Program

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

Subprogram

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

Explanation:

In this example, a call to SUB1 merely performs initialization. A subsequent call to SUB2 or SUB3 causes execution of a different section of the SUB1 subroutine. Then, depending upon the result of the arithmetic IF statement at statement 50, control returns to the calling program at statement 10, 20, or the statement following the call.

EXTERNAL STATEMENT

General Form
EXTERNAL a,b,c,...
Where: a,b,c,... are names of subprograms that are passed as arguments to other subprograms.

The EXTERNAL statement is a specification statement and must precede statement function definitions and the executable statements.

If the name of a FORTRAN supplied in-line function is used in an EXTERNAL statement, the function is not expanded in-line when it appears as a function reference. Instead, it is assumed that the function is supplied by the user or is part of the FORTRAN-supplied library. (The FORTRAN supplied in-line and out-of-line functions are given in Appendix C.)

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

Example 1:

```
Calling Program                    Subprogram

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

Explanation:

In this example, the subprogram name MULT is used as an argument in the subprogram SUB. The subprogram name MULT is passed to the dummy variable Y as are the variables J and C passed to the dummy variables K and Z, respectively. The subprogram MULT is called and executed only if the value of K is negative.

Example 2:

```
                .                  SUBROUTINE SUB (W,X,Y,Z)
                .                     .
                .                     .
CALL SUB (A,B,MULT (C,D),37)          .
                .                  RETURN
                .                  END
                .
```

Explanation:

In this example, 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.

OBJECT-TIME DIMENSIONS

If an array is used in a FUNCTION or SUBROUTINE subprogram and its name is not in a COMMON statement within the subprogram, the absolute dimensions of the array do not have to be explicitly declared in the subprogram by constants. Instead, an Explicit specification statement or DIMENSION statement appearing in the subprogram may contain integer variables of length 4 to specify the size of the array. When the subprogram is called, these integer variables receive their values from the actual arguments in the calling program reference or from common. Thus, the dimensions of a dummy array appearing in a subprogram may change each time the subprogram is called.

The absolute dimensions of an array must be declared in the calling program or in a higher level calling program and the array name must be passed to the subprogram in the argument list of the calling program. The dimensions passed to the subprogram must be less than or equal to the absolute dimensions of the array declared in the calling program. (Note that if the arrays have more than one dimension, the corresponding elements must agree, so the dimensions must be the same.) The variable dimension size can be passed through more than one level of subprogram (i.e., to a subprogram that calls another subprogram, passing it dimension information).

Integer variables in the Explicit specification or DIMENSION statement that provide dimension information must not be redefined within the subprogram; i.e., they must not appear to the left of an equal sign.

The name of an array with object-time dimensions cannot appear in a COMMON statement.

Example 1:

```
        .                              SUBROUTINE SUBR1(...R,L,M...)
        .                                 .
        .                                 .
    DIMENSION A(5,10)...                  .
        .                              REAL...R(L,M)...
        .                                 .
        .                                 .
    CALL SUBR1(...A,5,10...)              .
        .                              DO 10 I=1,L
        .                              DO 10 J=1,M
        .                           10 R(I,J)=0.
    END                                   .
                                          .
                                          .
                                       RETURN
                                          .
                                          .
                                          .
                                       END
```

Explanation:

This example shows the use of object-time dimensions to supply dimension information to a subroutine that will perform some operation on an array of any specified size. In this case, the dimensions passed are those specified for the array in the calling program, i.e., the full size of the array.

Example 2:

```
      .                                  SUBROUTINE SUBR1(...R,L,M...)
      .                                     .
      .                                     .
      .                                     .
DIMENSION A(5,10)                        REAL...R(L,M)...
      .                                     .
      .                                     .
      .                                     .
I = 4                                    DO 10 I=1,L
      .                                  DO 10 J=1,M
      .                                10 R(I,J)=0.
      .                                     .
J = 7                                       .
      .                                     .
      .                                  RETURN
      .                                     .
CALL SUBR1(...A,I,J...)                     .
      .                                     .
      .                                  END
      .
END
```

Explanation:

This example shows the use of object-time dimensions to specify a subset of the extent of an array (i.e., a partitioning of the matrix if two-dimensional) to a subprogram. The effect of this coding is the same as if another array, B, of dimensions (4,7) had been defined in the calling program and had been made equivalent to array A; the array B and its dimensions would then have been passed to SUBR1 as follows:

```
      .
      .
      .
DIMENSION...A(5,10),B(4,7)...
      .
      .
      .
EQUIVALENCE (A(1,1),B(1,1))
      .
      .
      .
I = 4
      .
      .
      .
J = 7
      .
      .
      .
CALL SUBR1 (...B,I,J...)
      .
      .
      .
END
```

To initialize variables in a labeled (named) 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 not be initialized in unlabeled common.

General Form
BLOCK DATA . . . END

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

2. The BLOCK DATA statement must be the first statement in the subprogram. If an IMPLICIT statement is used in a BLOCK DATA subprogram, it must immediately follow the BLOCK DATA statement. The COMMON statement must precede the data initialization statements.

3. Any main program or subprogram using a common block must contain a COMMON statement defining that block. If initial values are to be assigned, a BLOCK DATA subprogram is necessary.

4. All elements of a common block must be listed in the COMMON statement, even though they are not all initialized; for example, the variable A in the COMMON statement in the following example does not appear in the data initialization 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
```

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

APPENDIX A: SOURCE PROGRAM CHARACTERS

Alphabetic Characters	Numeric Characters
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	
L	
M	
N	Special Characters
O	
P	(blank)
Q	+
R	-
S	/
T	=
U	.
V	)
W	*
X	,
Y	(
Z	' (apostrophe)
$	&

The 49 characters listed above constitute the set of characters acceptable by FORTRAN, except in literal data where any valid card code is acceptable.

This appendix discusses those features of previously implemented FORTRAN IV languages that are incorporated into the System/360 FORTRAN IV language. The inclusion of these additional language facilities allows existing FORTRAN programs to be recompiled for use on the IBM System/360 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 locations in storage into which the data is placed.

This statement has the effect of a READ (n,b) list statement where b and list are defined as above, and the value of n is installation dependent.

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 locations in storage from which the data is taken.

This statement has the effect of a WRITE (n,b) list statement where b and list are defined as above, and the value of n is installation dependent.

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.

This statement has the effect of a WRITE (n,b) list statement where b and list are defined as above, and the value of n is installation dependent.

The FORTRAN-supplied subprograms are of two types: mathematical subprograms and service subprograms. The mathematical subprograms correspond to a FUNCTION subprogram; the service subprograms correspond to a SUBROUTINE subprogram. 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 in a library and the compiler generates an external reference to it. A detailed description of out-of-line mathematical subprograms and service subprograms is given in the publication IBM System/360: FORTRAN IV Library Subprograms, Form C28-6596. Table 4 shows mathematical function subprograms, and Table 5 shows out-of-line service subprograms.

Table 4. Mathematical Function Subprograms (Part 1 of 3)

Function	Entry Name	Definition	In-Line (I) Out-of-Line (O)	No. of Arg.	Type of Arguments	Type of Function Value
Exponential	EXP	e^{arg}	O	1	Real *4	Real *4
	DEXP	e^{arg}	O	1	Real *8	Real *8
	CEXP	e^{arg}	O	1	Complex *8	Complex *8
	CDEXP	e^{arg}	O	1	Complex *16	Complex *16
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
Common Logarithm	ALOG10	$\log_{10}$(Arg)	O	1	Real *4	Real *4
	DLOG10	$\log_{10}$(Arg)	O	1	Real *8	Real *8
Arcsine	ARSIN	arcsin (Arg)	O	1	Real *4	Real *4
	DARSIN	arcsin (Arg)	O	1	Real *8	Real *8
Arccosine	ARCOS	arccos (Arg)	O	1	Real *4	Real *4
	DARCOS	arccos (Arg)	O	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
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
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
Trigonometric Tangent (Argument in radians)	TAN	tan (Arg)	O	1	Real *4	Real *4
	DTAN	tan (Arg)	O	1	Real *8	Real *8

Table 4. Mathematical Function Subprograms (Part 2 of 3)

Function	Entry Name	Definition	In-Line (I) Out-of-Line (O)	No. of Arg.	Type of Arguments	Type of Function Value
Trigonometric Cotangent (Argument in radians)	COTAN	cotan (Arg)	O	1	Real *4	Real *4
	DCOTAN	cotan (Arg)	O	1	Real *8	Real *8
Square Root	SQRT	$(Arg)^{1/2}$	O	1	Real *4	Real *4
	DSQRT	$(Arg)^{1/2}$	O	1	Real *8	Real *8
	CSQRT	$(Arg)^{1/2}$	O	1	Complex *8	Complex *8
	CDSQRT	$(Arg)^{1/2}$	O	1	Complex *16	Complex *16
Hyperbolic Tangent	TANH	tanh (Arg)	O	1	Real *4	Real *4
	DTANH	tanh (Arg)	O	1	Real *8	Real *8
Hyperbolic Sine	SINH	sinh (Arg)	O	1	Real *4	Real *4
	DSINH	sinh (Arg)	O	1	Real *8	Real *8
Hyperbolic Cosine	COSH	cosh (Arg)	O	1	Real *4	Real *4
	DCOSH	cosh (Arg)	O	1	Real *8	Real *8
Error Function	ERF	$\frac{2}{\sqrt{\pi}}\int_0^x e^{-u^2} du$	O	1	Real *4	Real *4
	DERF		O	1	Real *8	Real *8
Complemented Error Function	ERFC	1-erf (x)	O	1	Real *4	Real *4
	DERFC		O	1	Real *8	Real *8
Gamma	GAMMA	$\int_0^\infty u^{x-1} e^{-u} du$	O	1	Real *4	Real *4
	DGAMMA		O	1	Real *8	Real *8
Log-gamma	ALGAMA	$\log_e \Gamma(x)$	O	1	Real *4	Real *4
	DLGAMA		O	1	Real *8	Real *8
Modular Arithmetic	MOD	Arg_1 (mod Arg_2)= $Arg_1-[x]*Arg_2$ Where: [x] is the largest integer whose magnitude does not exceed the magnitude of Arg_1/Arg_2. The sign of the integer is the same as the sign of Arg_1/Arg_2.	I	2	Integer *4	Integer *4
	AMOD		I	2	Real *4	Real*4
	DMOD		I	2	Real *8	Real *8
Absolute value	IABS	|Arg|	I	1	Integer *4	Integer *4
	ABS		I	1	Real *4	Real *4
	DABS		I	1	Real *8	Real *8
	CABS	$\sqrt{(a^2+b^2)}$ for a+bi	O	1	Complex *8	Real *4
	CDABS		O	1	Complex *16	Real *8
Truncation	INT	Sign of Arg times largest integer ≤|Arg|	I	1	Real *4	Integer *4
	AINT		I	1	Real *4	Real *4
	IDINT		I	1	Real *8	Integer *4

X = AMOD (10.0, 2.0)

X = 0

X = AMOD (9.0, 2.0)

X = 1

Table 4. Mathematical Function Subprograms (Part 3 of 3)

Function	Entry Name	Definition	In-Line (I) Out-of-Line (O)	No. of Arg.	Type of Arguments	Type of Function Value
Largest value[1]	AMAX0	Max (Arg_1,Arg_2,...)	O	≥2	Integer *4	Real *4
	AMAX1		O	≥2	Real *4	Real *4
	MAX0		O	≥2	Integer *4	Integer *4
	MAX1		O	≥2	Real *4	Integer *4
	DMAX1		O	≥2	Real *8	Real *8
value[1]st	AMIN0	Min (Arg_1,Arg_2,...)	O	≥2	Integer *4	Real *4
	AMIN1		O	≥2	Real *4	Real *4
	MIN0		O	≥2	Integer *4	Integer *4
	MIN1		O	≥2	Real *4	Integer *4
	DMIN1		O	≥2	Real *8	Real *8
Float	FLOAT	Convert from	I	1	Integer *4	Real *4
	DFLOAT	integer to real	I	1	Integer *4	Real *8
Fix	IFIX	Convert from	I	1	Real *4	Integer *4
	HFIX	real to integer	I	1	Real *4	Integer *2
Transfer of sign	SIGN	Sign of Arg_2 times $\|Arg_1\|$	I	2	Real *4	Real *4
	ISIGN		I	2	Integer *4	Integer *4
	DSIGN		I	2	Real *8	Real *8
Positive difference	DIM	Arg_1-Min(Arg_1, Arg_2)	I	2	Real *4	Real *4
	IDIM				Integer *4	Integer *4
Obtaining most significant part of a Real *8 argument	SNGL		I	1	Real *8	Real *4
Obtain real part of complex argument	REAL		I	1	Complex *8	Real *4
Obtain imaginary part of complex argument	AIMAG		I	1	Complex *8	Real *4
Express a Real *4 argument in Real *8 form	DBLE		I	1	Real *4	Real *8
Express two real arguments in complex form	CMPLX	$C=Arg_1,+iArg_2$	I	2	Real *4	Complex *8
Obtain conjugate of a complex argument	CONJG	C=X-iY	I	1	Complex *8	Complex *8
	DCONJG	For Arg=X+iY	I	1	Complex *16	Complex *16

[1]For the FORTRAN IV (H) compiler, these functions are in-line.

Table 5. Out-of-Line Service Subprograms

Function	CALL Statement	Argument Information
Alter status of sense lights	CALL SLITE(i)	i is an integer expression. If i = 0, the four sense lights are turned off. If i = 1, 2, 3, or 4, the corresponding sense light is turned on.
Test and record status of sense lights	CALL SLITET(i,j)	i is an integer expression that has a value of 1, 2, 3, or 4 and indicates which sense light to test. j is an integer variable that is set to 1 if the sense light was on, or to 2 if the sense light was off.
Dump storage on the output data set and terminate execution	CALL DUMP (a_1,b_1,f_1,...,a_n,b_n,f_n)	a and b are variables that indicate the limits of storage to be dumped. (Either a or b may be the upper or lower limits of storage, but both must be in the same program or subprogram or in COMMON.) f indicates the dump format and may be one of the following: 0 - hexadecimal 4 - integer 5 - real 6 - double precision
Dump storage on the output data set and continue execution	CALL PDUMP (a_1,b_1,f_1,...,a_n,b_n,f_n)	a, b, and f are as defined above for DUMP.
Test for divide check exception	CALL DVCHK(j)	j is an integer variable that is set to 1 if the divide-check indicator was on, or to 2 if the indicator was off. After testing, the divide-check indicator is turned off.
Test for exponent overflow or underflow	CALL OVERFL(j)	j is an integer variable that is set to 1 if an exponent overflow condition was the last to occur, to 2 if no overflow condition exists, or to 3 if an exponent underflow condition was the last to occur. After testing, the overflow indicator is turned off.
Terminate execution	CALL EXIT	None

SAMPLE PROGRAM 1

The sample program (Figure 1) is designed to find all of 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 evenly be divided by 3.

IBM FORTRAN Coding Form

PROGRAM: SAMPLE PROGRAM 1 | DATE: 6/68 | PUNCHING INSTRUCTIONS | GRAPHIC | PUNCH | PAGE 1 OF 1 | CARD ELECTRO NUMBER*

STATEMENT NUMBER	CONT	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
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	

*A standard card form, IBM electro 888157, is available for punching statements from this form.

Figure 1. Sample Program 1

SAMPLE PROGRAM 2

The n points (x_i, y_i) are to be used to fit an m degree polynomial by the least-squares method.

$$y = a_0 + a_1x + a_2x^2 + \ldots + a_mx^m$$

In order to obtain the coefficients $a_0, a_1, \ldots, a_m$, it is necessary to solve the normal equations:

$$\begin{aligned}
&(1) \quad W_0a_0 + W_1a_1 + \ldots + W_ma_m = Z_0\\
&(2) \quad W_1a_0 + W_2a_1 + \ldots + W_{m+1}a_m = Z_1\\
&\quad \vdots\\
&(m+1) \quad W_ma_0 + W_{m+1}a_1 + \ldots + W_{2m}a_m = Z_m
\end{aligned}$$

where:

$$W_0 = n \qquad Z_0 = \sum_{i=1}^{n} y_i$$

$$W_1 = \sum_{i=1}^{n} x_i \qquad Z_1 = \sum_{i=1}^{n} y_ix_i$$

$$W_2 = \sum_{i=1}^{n} x_i^2 \qquad Z_2 = \sum_{i=1}^{n} y_ix_i^2$$

$$\vdots \qquad\qquad \vdots$$

$$Z_m = \sum_{i=1}^{n} y_ix_i^m$$

$$W_{2m} = \sum_{i=1}^{n} x_i^{2m}$$

After the W's and Z's 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).

$$(1) \quad W_0a_0 + W_1a_1 + W_2a_2 = Z_0$$

$$(2) \quad W_1a_0 + W_2a_1 + W_3a_2 = Z_1$$

$$(3) \quad W_2a_0 + W_3a_1 + W_4a_2 = Z_2$$

The forward solution is as follows:

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:

$$(4) \quad a_0 + b_{12}a_1 + b_{13}a_2 = b_{14}$$

$$(5) \quad b_{22}a_1 + b_{23}a_2 = b_{24}$$

$$(6) \quad b_{32}a_1 + b_{33}a_2 = b_{34}$$

where:

$$b_{12} = W_1/W_0, \quad b_{13} = W_2/W_0, \quad b_1 = Z_0/W_0$$

$$b_{22} = W_2-b_{12}W_1, \quad b_{23} = W_3-b_{13}W_1, \quad b_2 = Z_1-b_{14}W_1$$

$$b_{32} = W_3-b_{12}W_2, \quad b_{33} = W_4-b_{13}W_2, \quad b_3 = Z_2-b_{14}W_2$$

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:

$$(7) \quad a_1 + c_{23}a_2 = c_{24}$$

$$(8) \quad c_{33}a_2 = c_{34}$$

where:

$$c_{23} = b_{23}/b_{22}, \quad c_{24} = b_{24}/b_{22}$$

$$c_{33} = b_{33}-c_{23}b_{32}, \quad c_{34} = b_{34}-c_{24}b_{32}$$

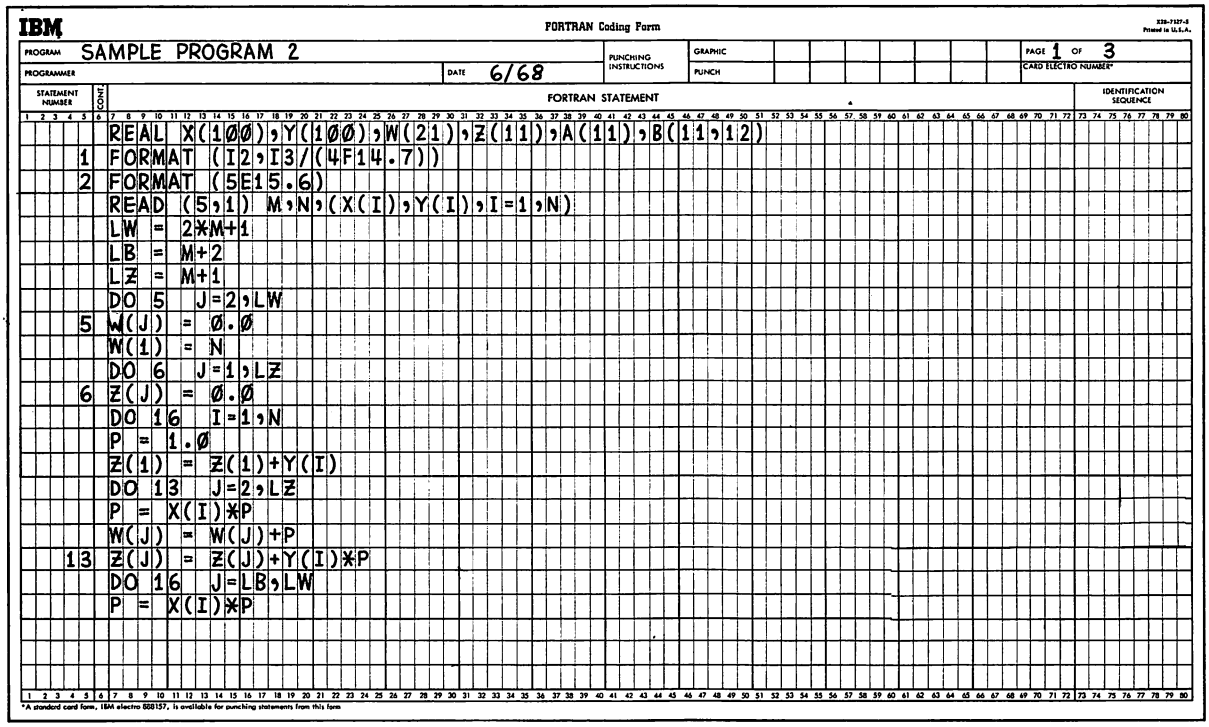
The backward solution is as follows:

$$(9) \quad a_2 = c_{34}/c_{33} \qquad \text{from equation (8)}$$

$$(10) \quad a_1 = c_{24}-c_{23}a_2 \qquad \text{from equation (7)}$$

$$(11) \quad a_0 = b_{14}-b_{12}a_1-b_{13}a_2 \qquad \text{from equation (4)}$$

Figure 2 is a possible FORTRAN program for carrying out the calculations for the case: n = 100, m ≤ 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, ..., Z_m are stored in Z (1), Z (2), Z (3), ..., Z(M+1), respectively.



IBM FORTRAN Coding Form

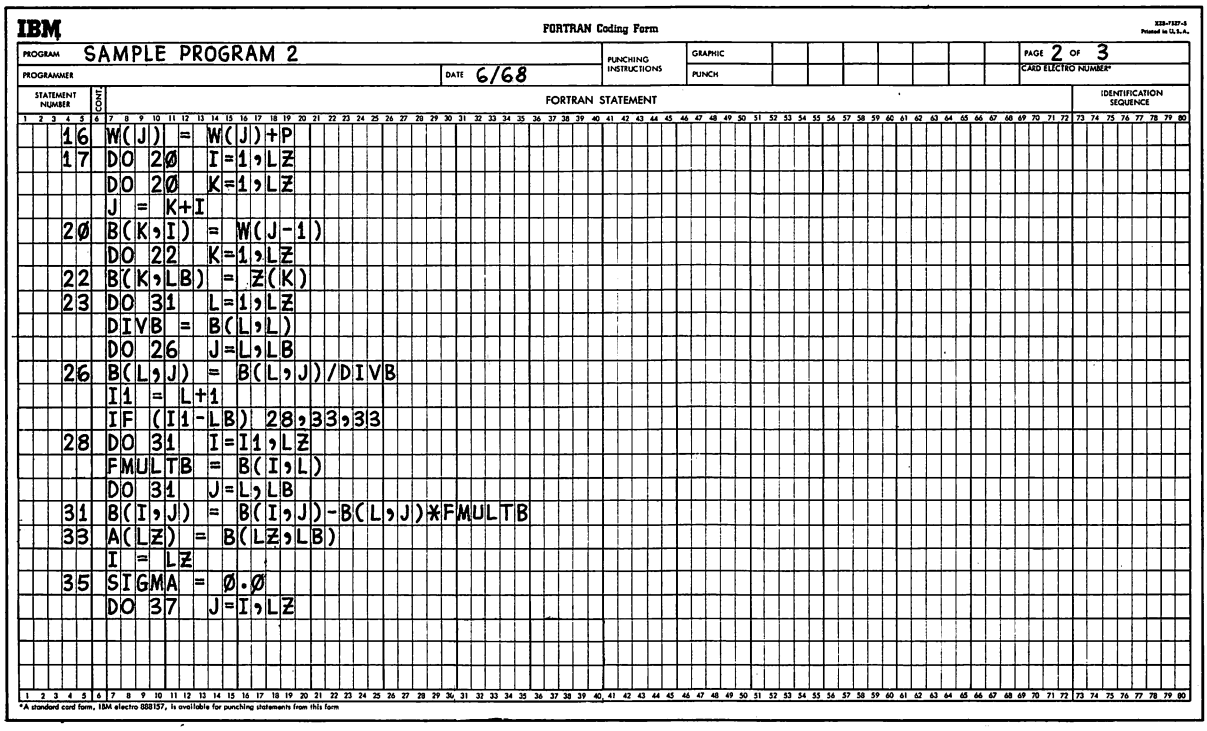
PROGRAM SAMPLE PROGRAM 2 | DATE 6/68 | PUNCHING INSTRUCTIONS | GRAPHIC | PUNCH | PAGE 1 OF 3 | CARD ELECTRO NUMBER*

PROGRAMMER

STATEMENT NUMBER	CONT	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
		REAL X(1ØØ),Y(1ØØ),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) = Ø.Ø	
		W(1) = N	
		DO 6 J=1,LZ	
6		Z(J) = Ø.Ø	
		DO 16 I=1,N	
		P = 1.Ø	
		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	

*A standard card form, IBM electro 888157, is available for punching statements from this form

Figure 2. Sample Program 2 (Part 1 of 2)



IBM FORTRAN Coding Form

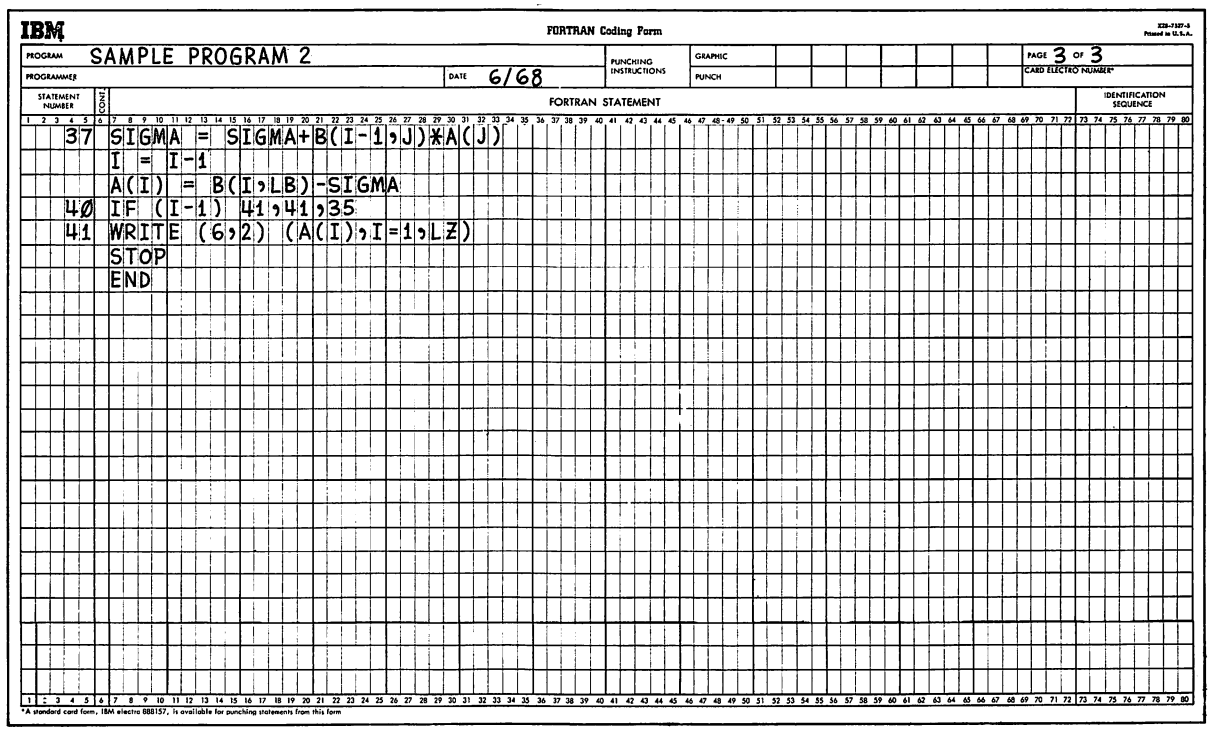
PROGRAM SAMPLE PROGRAM 2 | DATE 6/68 | PUNCHING INSTRUCTIONS | GRAPHIC | PUNCH | PAGE 2 OF 3 | CARD ELECTRO NUMBER*

PROGRAMMER

STATEMENT NUMBER	CONT	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
16		W(J) = W(J)+P	
17		DO 2Ø I=1,LZ	
		DO 2Ø K=1,LZ	
		J = K+I	
2Ø		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 = Ø.Ø	
		DO 37 J=I,LZ	

*A standard card form, IBM electro 888157, is available for punching statements from this form

Figure 2. Sample Program 2 (Part 2 of 2)



IBM FORTRAN Coding Form

PROGRAM SAMPLE PROGRAM 2 | DATE 6/68 | PUNCHING INSTRUCTIONS | GRAPHIC | PUNCH | PAGE 3 OF 3 | CARD ELECTRO NUMBER*

STATEMENT NUMBER	CONT	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,LZ)
		STOP
		END

* A standard card form, IBM electro 888157, is available for punching statements from this form

Figure 2. Sample 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, XI and YI are selected. The powers of XI are computed and accumulated in the correct W counters. The powers of XI are multiplied by YI, and the products are accumulated in the correct Z counters. In order to save machine time when the object program is being run, the previously computed power of XI is used when computing the next power of XI. Note the use of variables as index parameters. By the time control has passed to statement 17, the counters have been set as follows:

$$W(1) = N \qquad z(1) = \sum_{I=1}^{N} YI$$

$$W(2) = \sum_{I=1}^{N} XI \qquad z(2) = \sum_{I=1}^{N} YIXI$$

$$W(3) = \sum_{I=1}^{N} XI^2 \qquad z(3) = \sum_{I=1}^{N} YIXI^2$$

$$\vdots \qquad\qquad \vdots$$

$$z(M+1) = \sum_{I=1}^{N} YIXIM$$

$$\vdots$$

$$W(2M+1) = \sum_{I=1}^{N} XI^{2M}$$

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, ..., Z_m have been stored in the locations corresponding to the column M + 2 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 the following 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 the following computed values:

1	b_{12}	b_{13}	b_{14}
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) *AI terms have been stored in the M + 1 locations for the A array. For the illustrative problem, the A array would contain the following computed values for a_2, a_1, and a_0, respectively:

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.

The IBM System/360 Operating System FORTRAN IV (G) Debug Facility is a programming aid that enables the user to locate errors in a source program. The debug facility provides for tracing the flow within a program, tracing the flow between programs, displaying the values of variables and arrays, and checking the validity of subscripts.

The FORTRAN IV debug facility consists of a DEBUG specification statement, an AT debug packet identification statement, and three executable statements. These statements, alone or in combination with any FORTRAN IV source language statements, are used to state the desired debugging operations for a single program unit in source language. (A program unit is a single main program or a subprogram.)

The source deck arrangement consists of the source language statements that comprise the program, followed by the DEBUG specification statement, followed by the debug packets, followed by the END statement.

The statements that make up a program debugging operation must be grouped in one or more debug packets. A debug packet is preceded by the AT debug packet identification statement and consists of one or more executable debug facility statements, and/or FORTRAN IV source language statements. A debug packet is terminated by either another debug packet identification statement or the END statement of the program unit.

PROGRAMMING CONSIDERATIONS

The following precautions must be taken when setting up a debug packet:

1. Any DO loops initiated within a debug packet must be wholly contained within that packet.

2. Statement numbers within a debug packet must be unique. They must be different from statement numbers within other debug packets and within the program being debugged.

3. An error in a program should not be corrected with a debug packet; when the debug packet is removed, the error remains in the program.

4. The following statements must not appear in a debug packet:

   SUBROUTINE
   FUNCTION
   ENTRY
   IMPLICIT
   BLOCK DATA
   statement function definition

5. The program being debugged must not transfer control to any statement number defined in a debug packet; however, control may be returned to any point in the program from a packet. In addition, a debug packet may contain a RETURN, STOP, or CALL EXIT statement.

DEBUG FACILITY STATEMENTS

The specification statement (DEBUG) sets the conditions for operation of the debug facility and designates debugging operations that apply to the entire program unit (such as subscript checking). The debug packet identification statement (AT) identifies the beginning of the debug packet and the point in the program at which debugging is to begin. The three executable statements (TRACE ON, TRACE OFF, and DISPLAY) designate actions to be taken at specific points in the program. The following text explains each debug facility statement and contains several programming examples.

DEBUG SPECIFICATION STATEMENT

There must be one DEBUG statement for each program or subprogram to be debugged, and it must immediately precede the first debug packet.

General Form
DEBUG option,...,option Where: option may be any of the following: UNIT (a) where a is an integer constant that represents a data set reference number. All debugging output is placed in this data set, called the debug output data set. If this option is not specified, any debugging output is placed in the standard output data set. All unit definitions within an executable program must refer to the same unit. SUBCHK (n_1, n_2,...,n_n) where n is an array name. The validity of the subscripts used with the named arrays is checked by comparing the subscript combination with the size of the array. If the subscript exceeds its dimension bounds, a message is placed in the debug output data set. Program execution continues, using the incorrect subscript. If the list of array names is omitted, all arrays in the program are checked for valid subscript usage. If the entire option is omitted, no arrays are checked for valid subscripts. TRACE This option must be in the DEBUG specification statement of each program or subprogram for which tracing is desired. If this option is omitted, there can be no display of program flow by statement number within this program. Even when this option is used, a TRACE ON statement must appear in the first debug packet in which tracing is desired. INIT (m_1, m_2,...,m_n) where m is the name of a variable or an array that is to be displayed in the debug output data set only when the variable or the array values change. If m is a variable name, the name and value are displayed whenever the variable is assigned a new value in either an assignment, a READ, or an assigned GO TO statement. If m is an array name, the changed element is displayed. If the list of names is omitted, a display occurs whenever the value of a variable or an array element is changed. If the entire option is omitted, no display occurs when values change. SUBTRACE This option specifies that the name of this program or subprogram is to be displayed whenever it is entered. The message RETURN is to be displayed whenever execution of the subprogram is completed.

The options in a DEBUG specification statement may be given in any order and they must be separated by commas.

AT DEBUG PACKET IDENTIFICATION STATEMENT

The AT statement identifies the beginning of a debug packet and indicates the point in the program at which debugging is to begin. There must be one AT statement for each debug packet; there may be many debug packets for one program or subprogram.

General Form
AT statement number Where: statement number is an executable statement number in the program or subprogram to be debugged.

The debugging operations specified within the debug packet are performed prior to the execution of the statement indicated by the statement number in the AT statement.

TRACE ON STATEMENT

The TRACE ON statement initiates the display of program flow by statement number. Each time a statement with an external statement number is executed, a record of the statement number is made on the debug output data set. This statement has no effect unless the TRACE option was specified in the DEBUG specification statement.

General Form
TRACE ON

For a given debug packet, the TRACE ON statement takes effect immediately before the execution of the statement specified in the AT statement; tracing continues until a TRACE OFF statement is encountered. The TRACE ON stays in effect through any level of subprogram call or return. However, if a TRACE ON statement is in effect and control is given to a program in which the TRACE option was not specified, the statement numbers in that program are not traced. Trace output is placed in the debug output data set.

This statement may not appear as the conditional part of a logical IF statement.

TRACE OFF STATEMENT

The TRACE OFF statement may appear anywhere within a debug packet and stops the recording of program flow by statement number.

General Form
TRACE OFF

This statement may not appear as the conditional part of a logical IF statement.

DISPLAY STATEMENT

The DISPLAY statement may appear anywhere within a debug packet and causes data to be displayed in NAMELIST output format.

General Form
DISPLAY list Where: list is a series of variable or array names, separated by commas.

The DISPLAY statement eliminates the need for FORMAT or NAMELIST and WRITE statements to display the results of a debugging operation. The data is placed in the debug output data set.

The effect of a DISPLAY list statement is the same as the following FORTRAN IV source language statements:

```
NAMELIST /name/list
WRITE (n, name)
```

where:

name is the same in both statements. Note that subscripted variables or dummy arguments may not appear in the list.

This statement may not appear as the conditional part of a logical IF statement.

DEBUG PACKET PROGRAMMING EXAMPLES

The following examples show the use of a debug packet to test the operation of a program.

Example 1:

```
    INTEGER SOLON, GFAR, EWELL
       .
       .
       .
10  SOLON = GFAR * SQRT (EWELL)
11  IF (SOLON) 40, 50, 60
       .
       .
       .
    DEBUG UNIT (3)
    AT 11
    DISPLAY GFAR, SOLON, EWELL
    END
```

In example 1, the values of SOLON, GFAR, and EWELL are to be examined as they were at the completion of the arithmetic operation in statement 10. Therefore, the statement number entered in the AT statement is 11.

The debugging operation indicated is carried out just before execution of statement 11. If statement number 10 is entered in the AT statement, the values of SOLON, GFAR, and EWELL are displayed as they were before execution of statement 10.

Example 2:

```
    DIMENSION STOCK(1000),OUT(1000)
       .
       .
       .
    DO 30 I = 1, 1000
25  STOCK (I) = STOCK (I) - OUT (I)
30  CONTINUE
35  A = B + C
       .
       .
       .
    DEBUG UNIT (3)
    AT 35
    DISPLAY STOCK
    END
```

In example 2, all of the values of STOCK are to be displayed. When statement 35 is encountered, the debugging operation designated in the debug packet is executed. The value of STOCK at the completion of the DO loop is displayed.

Note: If the AT statement indicated statement 25 as the point of execution for the debugging operation, the value of STOCK is displayed for each iteration of the DO loop.

Example 3:

```
10  A = 1.5
12  L = 1
15  B = A + 1.5
20  DO 22 I = 1,5
       .
       .
       .
22  CONTINUE
25  C = B + 3.16
30  D = C/2
    STOP
       .
       .
       .
    DEBUG UNIT (3), TRACE
C   DEBUG PACKET NUMBER 1
    AT 10
    TRACE ON
C   DEBUG PACKET NUMBER 2
    AT 20
    TRACE OFF
    DO 35 I = 1,3
       .
       .
       .
35  CONTINUE
    TRACE ON
C   DEBUG PACKET NUMBER 3
    AT 30
    TRACE OFF
    END
```

When statement 10 is encountered, tracing begins as indicated by debug packet 1. When statement 20 is encountered, tracing stops as indicated by the TRACE OFF statement in debug packet 2 and no tracing occurs during the execution of the statements within this packet. Tracing begins again before leaving debug packet 2. When statement 30 is encountered, debug packet 3 is executed, and causes tracing to stop.

In this example, all trace information is placed in the data set associated with data set reference number 3. This data set contains trace information for the following statement numbers: 10, 12, 15, 20, 22, 22, 22, 22, 22, 25. Note that statement numbers 35 and 30 do not appear.

APPENDIX F: FORTRAN IV FEATURES NOT IN BASIC FORTRAN IV

The following features in FORTRAN IV are not in Basic FORTRAN IV:

ASSIGN
BLOCK DATA
Labeled COMMON
COMPLEX
DATA
Debug Facility
More than three dimensions
Object-time dimensions
Object-time FORMAT specifications
Assigned GO TO
Logical IF
LOGICAL
PRINT b, list
PUNCH b, list
READ b, list
END and ERR parameters in a READ
Generalized Type statement (But note that DOUBLE PRECISION is provided as an explicit type.)
IMPLICIT
Call by name
Literal as argument of CALL
ENTRY
RETURNi (i not a blank)
NAMELIST
PAUSE with literal
G, Z, and L format codes
Complex, logical, literal, and hexadecimal constants
Generalized subscript form

The following in-line subprograms in FORTRAN IV are not in Basic FORTRAN IV:

REAL	INT
AIMAG	AINT
DCMPLX	INDINT
CMPLX	
DCONJG	
CONJG	
HFIX	
CABS	
CDABS	

The following out-of-line subprograms in FORTRAN IV are not in Basic FORTRAN IV:

CEXP		DARSIN
CDEXP	ARCOS	DARCOS
CLOG	TAN	DTAN
CDLOG	COTAN	DCOTAN
CSIN	SINH	DSINH
CDSIN	COSH	DCOSH
CCOS	ERF	DERF
CDCOS		
CSQRT	ERFC	DERFC
CDSQRT	GAMMA	DGAMMA
DATAN2	ALGAMMA	DLGAMMA

Direct Access Input/Output Statements
Double Exponentiation
END and ERR parameters in READ
ENTRY
Generalized subscripts
Hexadecimal constant
IMPLICIT
Initial data values in type statement
Length of variables as part of type specifications
Literal enclosed in apostrophes
Mixed mode expressions
More than 3 dimensions in an array
NAMELIST
PAUSE 'message'
PRINT
PUNCH
READ b,list
T and Z format codes
RETURN i