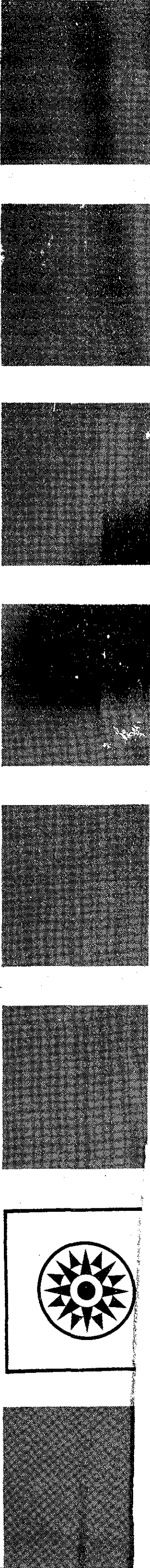
File No. S360-25
GC28-6629-2

OS
DOS
TOS
BPS

Systems Reference Library



IBM System/360

Basic FORTRAN IV Language

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

PREFACE

This publication is designed to support four implementations of the Basic FORTRAN IV language for the IBM System/360. The language described is implemented for the IBM System/360 Operating System, the IBM System/360 Disk Operating System, the IBM System/360 Tape Operating System, and the IBM System/360 Basic Programming Support Tape System. Differences among the language implementations are indicated in the programmer's guide for each system.

The material in this publication is arranged to provide a quick definition and syntactical reference to the Basic FORTRAN IV language by means of a box format. In addition, sufficient text to describe each element and examples of possible use are given.

Appendixes contain additional information useful in writing a FORTRAN program. This information consists of a table of source program characters, a comparison of the four language implementations, a list of FORTRAN-supplied subprograms, sample programs, a list of FORTRAN IV features and statements not available in Basic FORTRAN IV, and a list of Basic FORTRAN IV features not available in USAS Basic FORTRAN.

The reader should have some knowledge of an existing FORTRAN language before using this publication. A useful source of information is the FORTRAN IV For System/360 Programmed Instruction Course, Forms R29-0080 through R29-0087. This course is available through IBM representatives.

Compiler restrictions and programming aids are contained in the programmer's guide for the respective system. The appropriate programmer's guide and this language publication are both required publications. The programmer's guides are as follows:

IBM System/360 Operating System: FORTRAN IV (E) Programmer's Guide, Form C28-6603

IBM System/360 Disk and Tape Operating System: FORTRAN IV Programmer's Guide, Form C24-5038

IBM System/360 Basic Programming Support: FORTRAN IV (Tape) Programmer's Guide, Form C24-5038

References are made to information contained in the programmer's guides and in the following publications:

IBM System/360 FORTRAN IV Language, Form C28-6515

IBM System/360 Operating System: FORTRAN IV Library Subprograms, Form C28-6596

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

Third Edition (July, 1969)

This is a major revision of, and makes obsolete, Form C28-6629-1. This edition clarifies text and corrects errors that appeared in the previous edition, and should be reviewed for revised, added, and deleted material. Changes to text, and small changes to illustrations, are indicated by a vertical line to the left of the change; changed or added illustrations are denoted by the symbol ∘ to the left of the caption.

Changes are periodically made to the specifications herein; any such change will be reported in subsequent revisions or Technical Newsletters.

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

Address comments concerning the contents of this publication to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, New York 10020.

© International Business Machines Corporation 1966, 1969

INTRODUCTION 5

ELEMENTS OF THE LANGUAGE 6
Statements 6
Coding FORTRAN Statements 7
Constants 8
Integer Constants 8
Real and Double-Precision Constants . 9
Symbolic Names 10
Variables 11
Variable Names 11
Variable Types and Lengths 12
Type Declaration by the Predefined Specification 12
Type Declaration by Explicit Specification Statements 12
Arrays 13
Declaring the Size and Type of an Array 13
Arrangement of Arrays in Storage . . . 14
Subscripts 14
Expressions 16
Arithmetic Operators 16

ARITHMETIC ASSIGNMENT STATEMENT 20

CONTROL STATEMENTS 22
GO TO Statements 22
Unconditional GO TO Statement . . . 22
Computed GO TO Statement 23
Additional Control Statements 23
Arithmetic IF Statement 23
DO Statement 24
CONTINUE Statement 28
PAUSE Statement 29
STOP Statement 29
END Statement 29

INPUT/OUTPUT STATEMENTS 30
Sequential Input/Output Statements . . . 32
READ Statement 32
Formatted READ 33
Unformatted READ 33
WRITE Statement 33
Formatted WRITE 34
Unformatted WRITE 34
FORMAT Statement 35
Various Forms of a FORMAT Statement 36
I Format Code 37
D, E, and F Format Codes 38
Examples of Numeric Format Codes . . 38
Scale Factor - P 40
A Format Code 41
H Format Code and Literal Data . . . 42
X Format Code 43
T Format Code 43
Group Format Specification 44
END FILE Statement 44
REWIND Statement 45
BACKSPACE Statement 45
Direct Access Input/Output Statements . 45
DEFINE FILE Statement 47
Direct Access Programming Considerations 48
READ Statement 49
WRITE Statement 51
FIND Statement 52
General Examples -- Direct Access Operations 53

SPECIFICATION STATEMENTS 55
DIMENSION Statement 55
Explicit Specification Statements . . 56
COMMON Statement 57
Arrangement of Variables in Common . . 59
EQUIVALENCE Statement 60
Storage Arrangement of Variables in Equivalence Groups 61

SUBPROGRAMS 63
Naming Subprograms 63
Functions 64
Function Definition 64
Function Reference 64
Statement Functions 64
FUNCTION Subprograms 66
RETURN and END Statements in a FUNCTION Subprogram 67
SUBROUTINE Subprograms 68
CALL Statement 69
RETURN Statement in a SUBROUTINE Subprogram 70
Arguments in a FUNCTION or SUBROUTINE Subprogram 70
EXTERNAL Statement 71

APPENDIX A: SOURCE PROGRAM CHARACTERS . . 73

APPENDIX B: BASIC FORTRAN IV IMPLEMENTATION DIFFERENCES 74

APPENDIX C: FORTRAN-SUPPLIED SUBPROGRAMS 75

APPENDIX D: SAMPLE PROGRAMS 79
Sample Program 1 79
Sample Program 2 80

APPENDIX E: FORTRAN IV FEATURES NOT IN BASIC FORTRAN IV 86

APPENDIX F: IBM BASIC FORTRAN IV EXTENSIONS TO USAS BASIC FORTRAN . . . 87

INDEX 89

ILLUSTRATIONS

FIGURES

Figure 1. Sample Program 1 79
Figure 2. Sample Program 2
(Part 1 of 3). 82

TABLES

Table 1. Determining the Type of
the Result of + - * / ** 19
Table 2. Conversion Rules for
Arithmetic Assignment Statements . . . 20
Table 3. Implementation
Differences 74
Table 4. In-Line Mathematical
Function Subprograms 75
Table 5. Out-of-Line
Mathematical Function Subprograms . . 77
Table 6. Out-of-Line Service
Subprograms 78

IBM System/360 Basic FORTRAN IV for the Operating System, the Tape Operating System, the Disk Operating System, and the Basic Programming Support Tape System consists of a language, a library of subprograms, and a compiler.

The Basic 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 Basic 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 programmer's 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 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 Basic FORTRAN IV language is compatible with and encompasses the United States of America Standard (USAS) Basic FORTRAN, X3.10-1966, including its mathematical subroutine provisions. Basic FORTRAN IV is a subset of FORTRAN IV, as described in the publication IBM System/360 FORTRAN IV Language. Appendixes E and F contain lists of differences between FORTRAN IV, Basic FORTRAN IV, and USA Basic FORTRAN.

ELEMENTS OF THE LANGUAGE

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., add, multiply, branch)
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 Statements: These statements cause calculations to be performed and the result to replace the current value of a designated variable or subscripted variable.
2. Control Statements: These statements enable the user to govern the flow and 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 in which data appears on an input/output device.
5. Specification Statements: These statements are used to declare the properties of variables, arrays, and subprograms (such as type and amount of storage reserved).
6. Statement Function Definition Statement: This statement specifies operations to be performed whenever the statement function name appears in the program.
7. 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 chapter. The actual FORTRAN statements in which these elements are used are discussed in following sections. The term <u>program unit</u> refers to a main program or a subprogram. The phrase <u>executable statements</u> refers to those statements in categories 1, 2, and 3.

The order of a Basic FORTRAN program unit is:

1. Subprogram statement, if any.

2. 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.)

3. Statement function definitions, if any.

4. Executable statements, at least one of which must be present.

5. END statement.

FORMAT statements can appear anywhere in a program unit.

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, however, that are inserted in literal data are retained and treated as blanks within the data.[1]

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, etc.

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.

---

[1]E-level FORTRAN programmers may exercise a compiler option that permits blanks in key words, names, and constants.

CONSTANTS

A constant is a fixed, unvarying quantity. Three types of constants can be used: integer, real, and double-precision.

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
<u>Real or Double-Precision Constant</u> -- has one of three forms: a basic real or double-precision constant, a basic real or double-precision constant followed by a decimal exponent, or an integer constant followed by a decimal exponent. A basic real constant is a string of fewer than eight decimal digits with a decimal point. A basic double-precision constant is a string of eight or more decimal digits with a decimal point. A constant can be explicitly specified as real or double-precision by appending an exponent to a basic real constant, a basic double-precision 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 real constant; the letter D specifies a double-precision constant. A real constant occupies four storage locations (bytes); a double-precision constant occupies eight storage locations (bytes). Magnitude: (either real or double precision) 0 or 16^{-65} (approximately 10^{-78}) through 16^{63} (approximately 10^{75}) Precision: (real) 6 hexadecimal digits (approximately 7.2 decimal digits) (double precision) 14 hexadecimal digits (approximately 16.8 decimal digits)

A real or double-precision 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 or double-precision constant as the product of a basic real constant, or an integer constant multiplied by 10 raised to a desired power.

<u>Examples</u>:

<u>Valid</u> real constants:

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

<u>Valid</u> double-precision constants:

Constant	Meaning
21.98753829457168	
1.0000000	
79D3	
7.9D03 7.9D+3 7.9D+03	(i.e., $7.9 \times 10^{3} = 7900.0$)
7.9D-03	(i.e., $7.9 \times 10^{-3} = 0.0079$)
7.9D0	(i.e., $7.9 \times 10^{0} = 7.9$)
0.0D0	(i.e., $0.0 \times 10^{0} = 0.0$)

Invalid real and double-precision constants:

0	(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, $23.5 \times 10^{97} > 16^{63}$)
21.3D-90	(Magnitude outside the allowable range; that is, $21.3 \times 10^{-90} < 16^{-65}$)

SYMBOLIC NAMES

Definition
Symbolic Name -- consists of from one through six alphameric characters [i.e., numeric (0 through 9) or alphabetic (A through Z and $)], 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")
- 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 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, 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 following statement both A and B are variables:

```
A = 5.0+B
```

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, and a real variable represents real data, and a double-precision variable represents double-precision data.

The number of storage locations reserved for variables depends on the type of the variable. Integer and real variables have four storage locations (bytes) reserved; double-precision variables have eight storage locations (bytes) reserved.

A programmer may declare the type of a variable by using the:

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

The explicit specification statement overrides the predefined specification.

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.

2. If the first character of the variable name is any other alphabetic character, the variable is real.

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.

TYPE DECLARATION BY EXPLICIT SPECIFICATION STATEMENTS

Explicit specification statements differ from the predefined specification, in that an explicit specification statement declares the type of a particular variable by its <u>name</u>, rather than by its initial <u>character</u>.

For example, assume that an explicit specification statement declared that the variable named ITEM is real. Then ITEM is treated as a real variable but all other variables beginning with the character I are treated as integer variables.

These statements are discussed in greater detail in the chapter "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 that 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 be assigned a value of 1, 2, 3, 4, or 5.

To refer to any element in an array, the array name must be subscripted. The 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 reference would be coded as:

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

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 chapter "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.

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 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 three subscript quantities can appear in a subscript.

General Form
Subscript Quantities -- may be one of seven forms: v k v+k v-k c*v c*v+k c*v-k Where: v represents an unsigned, nonsubscripted, integer variable. c and k each represent an unsigned integer constant.

Whatever subscript form[1] is used, its evaluated result, as well as the intermediate result, must always be greater than 0 and less than or equal to 32,767. For example, when reference is made to the subscripted variable V(I-2), the value of I should be greater than 2 and less than or equal to 32,767. In any case, the evaluated result must be within the range of the array.

Examples:

Valid subscripted variables:

```
ARRAY (IHOLD)
NEXT (19)
MATRIX (I-5)
A(5*L)
W(4*M+3)
Z(2*I+3,6*J+8,3*K-2)
```

Invalid subscripted variables:

ARRAY (-I)	(The subscript quantity I may not be signed)
COST(A+2)	(A is not an integer variable unless defined as such by an explicit specification statement)
ARRAY(I+2.)	(The constant within a subscript quantity must be an integer)
NEXT(-7*J)	(The constant within a subscript quantity must be unsigned)
W(I(2))	(The subscript quantity I may not be subscripted)
LOT (0)	(A subscript quantity may neither be nor assume a value of zero)
TEST (K*2)	(If multiplication is indicated, the constant must precede the variable. Thus, TEST (2*K) is correct)
TOTAL (2+K)	(If addition is indicated, the variable must precede the constant. Thus, TOTAL (K+2) is correct)
Q(I,J,K,L)	(No more than three subscript quantities may be used)

---

[1]If more than one subscript form is used, the product of all subscript quantities must be less than or equal to 131,068 in Operating System FORTRAN IV (E) and less than or equal to 32,767 in the other three systems.

EXPRESSIONS

Basic FORTRAN IV provides only one kind of expression: the arithmetic expression. The value of an arithmetic expression is always a number whose type is integer, real, or double precision. Expressions may appear in arithmetic assignment statements and in certain control statements.

ARITHMETIC EXPRESSIONS

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

If the primary is of type integer, the expression is integer. If it is of type real, the expression is real, etc.

Examples:

Primary	Type of Primary	Type of Expression
3	Integer constant	Integer
A	Real variable	Real
3.14D3	Double-precision constant	Double precision
B(2*I)	Double-precision subscripted variable (specified as such in an explicit specification statement)	Double precision
SIN(X)	Real function reference	Real
(A+B*C)	Parenthesized real expression	Real

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

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, then 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 by the result. (For additional 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 it is 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:

1.	A*B	Call the result X (multiplication)	(X+C*D**I)
2.	D**I	Call the result Y (exponentiation)	(X+C*Y)
3.	C*Y	Call the result Z (multiplication)	(X+Z)
4.	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:

1.	B**C	Call the result Z
2.	A**Z	Final operation

A unary plus or minus has the same hierarchy as a plus or minus in addition or subtraction. Thus,

```
A=-B is treated as A=0-B

A=-B*C is treated as A=0-(B*C)

A=-B+C is treated as A=(0-B)+C
```

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:

```
1.  (A+B)     Call the result X   B+(X*C)+A**2
2.  (X*C)     Call the result Y   B+Y+A**2
3.  B+Y       Call the result W   W+A**2
4.  A**2      Call the result Z   W+Z
5.  W+Z       Final operation
```

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

   Assume that the type of the variables I, J, K, C, and D has been specified as follows:

Variable Names	Type
I,J,K	Integer variable
C	Real variable
D	Double-precision variable

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

Operations		Type
I*J	(Call the result X)	Integer
C**K	(Call the result Y)	Real
X/Y	(Call the result Z)	Real
Z+D		Double precision

   Thus, the final type of the entire expression is double precision, 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 may differ from that of the expression I*C*J.

5. The type of the result of an exponentiation (**) operation depends on 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. Note, however, that a negative real or double-precision quantity must not be followed by a real or double-precision exponent, since the result is, in general, complex and cannot be represented as a real or double-precision value.

• Table 1. Determining the Type of the Result of + - * / **

+ - * / **	INTEGER	REAL	DOUBLE PRECISION
INTEGER	Integer	Real	Double precision
REAL	Real	Real	Double precision
DOUBLE PRECISION	Double precision	Double precision	Double precision

Note: When one integer is divided by another, the quotient is also an integer. If necessary, the result is truncated. For example, 5/2 gives a quotient of 2.

General Form
a = b Where: a is a subscripted or nonsubscripted variable. b is an arithmetic 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.

The type of the variable, represented by a, is converted according to the type of the arithmetic expression b, as shown in Table 2.

• Table 2. Conversion Rules for Arithmetic Assignment Statements

Type of a \ Type of b	INTEGER	REAL	DOUBLE PRECISION
INTEGER	Assign	Fix and Assign	Fix and Assign
REAL	Float and Assign	Assign	Real Assign
DOUBLE PRECISION	DP Float and Assign	DP Float and Assign	Assign

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 data can contain.
3. Fix means truncate the fractional portion of the resulting value and transform to the form of an integer.
4. Float means transform the resulting value to the form of a real number retaining in the process as much precision of the value as a real number can contain.
5. DP Float means transform the resulting value to the form of a double-precision number.

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

Variable names	Type
I, J, W	Integer variables
A, B, D	Real variables
E	Double-precision variable
F	Real array

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.
B = I**J+D	The value of 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 then replaces the value of B.
A = B*D	The most significant part of the product of B and D replaces the value of A.
A = I+E	The value of I is converted to double precision and added to E. The result of the addition is truncated from double precision to real and replaces the value of A.
A = F(5,4)	The value of F(5,4) replaces the value of A.
	The value of I is converted to double precision, and this value replaces the value of E.
J = E	The value of E is truncated to an integer value, and this value replaces the value of J.
E = A	The value of A is converted to double precision, and this value replaces the value of E.

CONTROL STATEMENTS

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

Unconditional GO TO Statement

General Form
GO TO xxxxx Where: xxxxx is the number of an executable statement.

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 the above example, each time the GO TO statement is executed, control is transferred to statement 25.

Computed GO TO Statement

General Form
GO TO ($x_1, x_2, x_3, \ldots, x_n$), i
Where: $x_1, x_2, \ldots, x_n$, are the numbers of executable statements.
i is a nonsubscripted integer variable whose current value is in the range: $1 \le i \le n$

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.

ADDITIONAL CONTROL STATEMENTS

Arithmetic IF Statement

General Form
IF (a) x_1, x_2, x_3
Where: a is an arithmetic expression. x_1, x_2, x_3 are the numbers of executable statements.

This 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 the above 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.

DO Statement

General Form
DO x (End of Range) i (DO Variable) = m_1 (Initial Value), m_2 (Test Value), m_3 (Increment)
Where: x is the number of an executable statement which appears 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 $2^{31}-2$ (2,147,483,646) 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 should 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                          |
                                              |  Range of
      DO 50 J=2, 5          )  Range of       |  Outer DO
                            )  Inner DO       |
   50 C(J) = A(I)           )                 )
   ```

   Example 2:

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

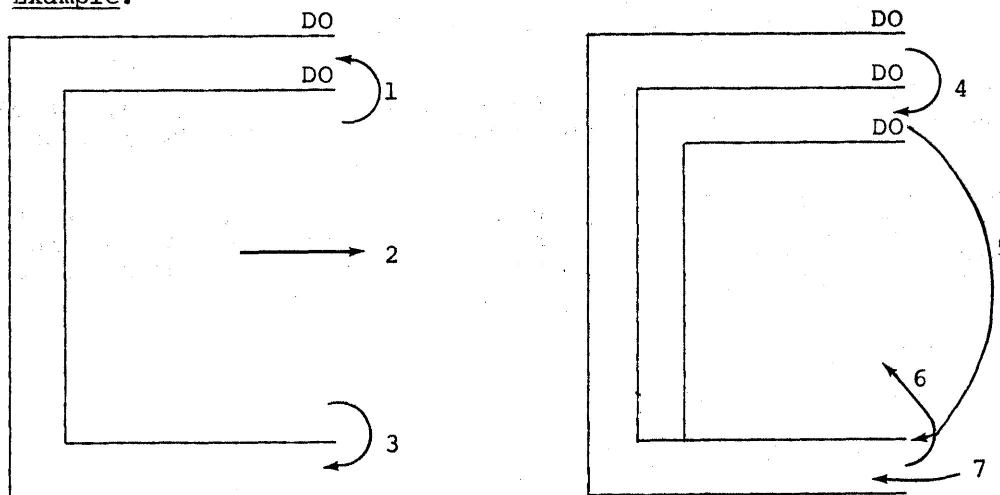
3. A transfer out of the range of any DO loop is permissible at any time.

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.
   - No DO statements are permitted in the extended range of the DO.
   - The indexing parameters $(\underline{i},\underline{m}_1,\underline{m}_2,\underline{m}_3)$ 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 loop. The statement label of such a terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

   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 $(\underline{i},\underline{m}_1,\underline{m}_2,\underline{m}_3)$ 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 $\underline{x}$) must be an executable statement. It cannot be a GO TO statement of any form, or a PAUSE, STOP, RETURN, arithmetic IF, or another DO statement.

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, or another DO statement.

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 Where: n is a string of from one through five decimal digits.

PAUSE n or PAUSE 00000 is displayed, depending on whether n was specified. 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 programmer's guides listed in the Preface.

STOP Statement

General Form
STOP STOP n Where: n is a string of from one through five decimal digits.

The STOP statement terminates the execution of the object program. STOP n is displayed if n is specified. For further information, see the FORTRAN programmer's 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. It must have blanks in columns 1 through 6 and 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 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: for example, READ, WRITE, or BACKSPACE.

Data Set Reference Number: A FORTRAN programmer refers to a data set by its data set reference number. (The FORTRAN programmer's 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 specify 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 that 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 (except expressions used as subscripts) 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 $i=m_1, m_2, m_3$. The indexing parameters are as defined for the DO statement. Their range is the list of the DO-implied list and, for input lists, i, m_1, m_2, and m_3 may appear within that range only in subscripts.

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 (3) 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 3 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 (1) ((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 storage locations.

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 can satisfy the needs of machine representation. The transformation for input takes the character codes and constructs a machine representation for 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 required by the input/output routines or Data Management. For further information, see the Basic FORTRAN IV programmer's 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" in this publication and the Basic FORTRAN IV programmer's guides listed in the Preface.

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 and are thus not available in Basic Programming Support Basic FORTRAN IV. They can be compiled but not executed by Tape Operating System Basic FORTRAN IV.

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 statement, although not an input/output statement, is used with certain forms of the READ and WRITE statements. The FORMAT statement is not executable and can be placed anywhere in the program.

READ STATEMENT

General Form
READ (a,b) list Where: a is an unsigned integer constant or an integer variable that represents a data set reference number. b is optional and is the statement number of the FORMAT statement describing the data being read. 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 and list can be omitted.

The basic forms of the READ statement are:

Form	Purpose
READ (a,b) list	Formatted READ
READ (a) list	Unformatted READ

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 (1,98) 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 1 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 and represents a data set reference number.
b is optional and is the statement number of the FORMAT statement describing the data being written.
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 basic forms of the WRITE statement are:

Form	Purpose
WRITE (a,b) list	Formatted WRITE
WRITE(a) list	Unformatted WRITE

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(3,75) 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), and C into the data set associated with data set reference number 3 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.

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)
paDw.d	(Describes double-precision data fields)
paEw.d	(Describes real data fields)
paFw.d	(Describes real data fields)
aAw	(Describes character 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)
Tr	(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 repeated. If a is omitted, the code is used only once.
	w is an unsigned integer constant that is less than or equal to 255 and 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.
	p is optional and represents a scale factor. It is specified as an unsigned or negatively signed integer constant followed by the letter P.
	r is an unsigned integer constant designating a character position in a record.
	(...) is a group format specification. Within the parentheses are format codes separated by commas or slashes. Group format specifications cannot be nested. The a preceding this form is called a group repeat count.
	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 a FORTRAN record and the form of the data fields within the record. In the FORMAT statement, the data fields are described with format codes; the order in which these format codes is specified gives the structure of the FORTRAN record. 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 ƀ represents a blank.

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

1. A FORMAT statement is not executed; its function is to supply information to the object program. It 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 device. 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. When formatted records are prepared for printing, the first character of the record is not printed. It is treated as a carriage control character. It can be specified in a FORMAT statement with either of two forms of literal data; either 'x' or 1Hx, 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

For devices other than the printer, the first character of the record is treated as data.

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.

Execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information provided jointly by the I/O list, if one exists, and the format specification. There is no I/O list item corresponding to the format descriptors X, H, and literals enclosed in apostrophes. These communicate information directly with the record.

Whenever an I, D, E, F, or A code is encountered, format control determines whether there is a corresponding element in the I/O list. If there is such an element, appropriately converted information is transmitted. If there is no corresponding element, the format control terminates.

If format control reaches the last outer right parenthesis of the format specification and another element is specified in the I/O list,

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 question of whether there are further elements in the I/O list is asked only when an I, D, E, F, or A code or the final right parenthesis of the format specification is encountered. Before this is done, T, X, and H codes, literals enclosed in apostrophes, and slashes are processed. If there are fewer elements in the I/O list than there are format codes, the remaining format codes are ignored.

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 parenthesis. 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 following statement describes two FORTRAN record formats:

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

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 following statement describes three FORTRAN record formats:

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

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.

D, E, and F Format Codes

The D, E, and F format codes are used in transmitting real or double precision data. The data must not exceed the maximum magnitude for a real or double-precision constant.

Input: Input must be a real or double-precision number which, optionally, may have a D or E exponent. An exponent may be expressed without the letter, simply as a signed integer constant, such as 02.380+02b for the value 2.380D+020. The decimal point also may be omitted. If it is present, its position overrides the position indicated by the d portion of the format field descriptor, and the number of positions specified by w must include a place for it. If the data has a D or E exponent and the format field descriptor includes a P scale factor, the scale factor has no effect. Each data item must be right justified in its field, since leading, trailing, and embedded blanks are treated as zeros. These three format codes are interchangable for input. It makes no difference, for example, whether D, E, or F is used to describe a field containing 12.42E+08.

Output: For data written under a D or E format code, unless a P scale factor is specified, output consists of an optional sign (required for negative values), a decimal point, the number of significant digits specified by d, and a D or E exponent requiring four positions. The w specification must provide for all these positions, including the one for a sign when the output value is negative. If additional space is available, a leading zero may be written before the decimal point.

For data written under an F format code, w must provide sufficient spaces for an integer segment if it is other than zero, a fractional segment containing d digits, a decimal point, and, if the output value is negative, a sign. If insufficient positions are provided for the integer portion, including the decimal point and sign (if any), asterisks are written instead of data. If excess positions are provided, the number is preceded by blanks.

For D, E, and F, fractional digits in excess of the number specified by d are dropped after rounding.

Examples of Numeric Format Codes

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

Example 1:

```
75 FORMAT (I3,F5.2,E10.3,D10.3)
      .
      .
      .
   READ (1,75) 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 1.

2. When an input card is read, the number in the first field of the card (three columns) is stored in integer format in location N. The number in the second field of the input card (five columns) is stored in real format, 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 card would be ignored.

4. If there were one less variable in the list (say C was omitted), the format specification D10.3 would be ignored.

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 were given:

```
 75    FORMAT  (I4,F6.3,D10.3)
       READ     (1,75)    N,A,B
```

where N is integer, A is real, and B is double precision and that the following input cards are read:

```
Column       1   5      11          21

           ƀ311 56.432 02.380+02ƀ
Input
           2343 155381 -7.654D-06
Cards
           331ƀ 346.18 ƀƀƀ4.673ƀƀ

Format       I4   F6.3     D10.3
```

Then the variables N, A, and B receive values as though the following had been punched:

N	A	B
0311	56.432	2.380D+020
2343	155.381	-7.654D-06
3310	346.18	4.673D+00

Explanation:

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

Example 3: Assume that the following statements were given:

```
 76 FORMAT   ('1',I3,F6.2,E10.3)
    WRITE    (3,76) N,A,B
```

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

N	A	B
031	034.40	123.380E+02
130	031.1	1156.1E+02
428	-354.32	834.621E-03
000	01.132	83.121E+06

Then, the following lines are printed:

```
Column  1  4     10        20

           31 34.40 0.123E 05

          130 31.10 0.115E 06

          428****** 0.835E 00

            0  1.13 0.831E 08
```

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.

Scale Factor - P

The P scale factor may be specified as the first part of a D, E, or F field descriptor to change the location of the decimal point in real numbers. The effect of the P scale factor is:

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

Input: A scale factor may be specified for any real data, but it is ignored for any data item that contains an exponent in the external field. 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 0.238Eb̸03. If it were printed according to the format 1PE10.3, it would appear as 2.380Eb̸02.

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 established, it applies to all subsequently interpreted D, E, and F codes in the same FORMAT statement until another scale factor is encountered. The new scale factor is then established. A factor of 0 may be used to discontinue the effect of a previous scale factor.

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 the A format code depends on the length of the corresponding variable in the I/O list, i.e., four for integer and real, and eight for double precision. 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 the number of characters of data in the field (w) is greater than the length of the variable (v), then w minus v characters are skipped from the left before the remaining input 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 (assume 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 double precision, that N and M are integers, and that the following statements are given:

```
25   FORMAT (3A7)
       .
       .
       .
     READ   (1,25) B, N, M
```

When the READ statement is executed, one input card is read from the data set associated with data set reference number 1 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 (b̸ represents a blank).

Input Card	B	N	M
ABCDEFG46b̸ATb̸11234567	ABCDEFGb̸	ATb̸1	4567
HIJKLMN76543213334445	HIJKLMNb̸	4321	4445

Example 2: Assume that A and B are real variables, that C is a double-precision variable, and that the following statements are given:

```
26   FORMAT   ('b̸',A6,A5,A6)

     WRITE    (3,26) A,B,C
```

When the WRITE statement is executed, one line is written on the data set associated with data set reference number 3 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 (b̸ represents a blank):

A	B	C	Printed Line
A1B2	C3D4	E5F6G7H8	b̸b̸A1B2b̸C3D4E5F6G7

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 (' 1969 INVENTORY REPORT')

25   FORMAT (22H 1969 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   (1,8)
```

cause the first nine characters of the next record to be read from the data set associated with data set reference number 1 into the FORMAT statement 8, replacing the blank and the eight 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  (14H0MEAN AVERAGE:, F8.4)

   WRITE   (3,8) AVRGE
```

cause the following record to be written if the value of AVRGE is 12.3456:

```
MEAN AVERAGE:  12.3456
```

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   (1,5) I,J,K,L,M
```

cause the first ten characters of the input card to be read into variable I, the next ten characters to be skipped over without transmission, and the next four fields of ten 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. (Note that for printed output, the first character of the output data record is used for carriage control and is not printed. Thus, if T50,'Z' is specified in a FORMAT statement, a Z will be the 50th character of the output record, but it will appear in the 49th print position.) For example, the following statements:

```
5   FORMAT (T40,'1969 INVENTORY REPORT',T80,
   X'DECEMBER',T1,0PART NO.  10095')

    WRITE  (3,5)
```

cause the following line to be printed:

```
Print             Print                    Print
Position 1        Position 39              Position 79
|                 |                        |
V                 V                        V
PART NO. 10095    1969 INVENTORY REPORT    DECEMBER
```

The T format code can be used in a FORMAT statement with any type of format code, as with FORMAT ('0',T40,I5).

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   (1,15) 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.

END FILE STATEMENT

General Form
END FILE a Where: a is an unsigned integer constant or integer variable that 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 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 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 Basic FORTRAN IV programmer's guides listed in the Preface.

DIRECT ACCESS INPUT/OUTPUT STATEMENTS

There are four direct access input/output statements[1]: 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.

---

[1]The direct access input/output statements are not available in Basic Programming Support Basic FORTRAN IV. They may be compiled but not executed by Tape Operating System Basic FORTRAN IV.

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 (i.e., must be "executed" prior to) any input/output statement referring to the data set described in the DEFINE FILE statement.

General Form
DEFINE FILE $\underline{a}_1(\underline{m}_1,\underline{r}_1,\underline{f}_1,\underline{v}_1),\underline{a}_2(\underline{m}_2,\underline{r}_2,\underline{f}_2,\underline{v}_2),\ldots,\underline{a}_n(\underline{m}_n,\underline{r}_n,\underline{f}_n,\underline{v}_n)$
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 indicates the storage location of the record that immediately follows the last record transmitted. At the conclusion of a FIND operation, _v_ is set to a value that indicates the storage location of 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.

Examples:

```
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 each 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 of the 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 only 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 refers to data associated with data set reference number 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) list Where: a is an unsigned integer constant or integer variable that 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 the statement number of the FORMAT statement that describes the data being read. 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 was:

```
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 unsigned integer constant or integer variable that 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 the statement number of the FORMAT statement that describes the data being written. 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 five 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 (seven variables, type real, each occupying four storage locations).

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)
```

As with the READ statement, a FORMAT statement also may 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 a FIND statement preceding a WRITE statement. If a WRITE statement for a record is executed between the time a FIND statement and a READ statement are executed for that record, the effect of the FIND statement is nullified.

General Form
FIND (a'r) Where: a is an unsigned integer constant or integer variable that 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. If a WRITE statement refers to this record between the issuing of the FIND statement and the READ statement, the FIND operation is nullified.

General Examples -- Direct Access Operations

Example 1:

```
    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
```

Explanation:

Example 1 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 statement.

Example 2:

```
C MAIN PROGRAM
      COMMON I
          .
          .
          .
    1 READ (1,2) I
    2 FORMAT (I4)
      I=IABS(I)
      IF (I) 10,20,10
   10 CALL SUB1 (A)
      GO TO 70
   20 CALL SUB2 (A)
   70 CONTINUE
          .
          .
          .
      WRITE (9'I+1,100) X,Y,Z
  100 FORMAT (3F10.3)
          .
          .
          .
      END
```

```
      SUBROUTINE SUB1 (AA)
      COMMON J
      DEFINE FILE 9(100,100,E,J)
          .
          .
          .
      RETURN
      END
```

```
      SUBROUTINE SUB2 (BB)
      COMMON K
      DEFINE FILE 9(125,80,L,K)
          .
          .
          .
      RETURN
      END
```

Explanation:

Example 2 illustrates the use of two different DEFINE FILE statements to describe the characteristics of the data set associated with data set reference number 9. If SUB1 is called, the data set contains 100 records, each with a maximum length of 100 characters; the data is to be transmitted under format control, and the associated variable is J. If SUB2 is called, the data set contains 125 records, each with a maximum length of 80 storage locations; the data is to be transmitted either with or without format control, and the associated variable is K. Because the associated variables are in COMMON with I, the information is shared by the main program and the two subroutines.

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 that part of the program containing at least one executable statement. Specification statements describing data must precede any statements that refer to that data. For example, if an element of an array is to be made equivalent to a variable, the specification statement that declares the size of the array (e.g., a DIMENSION statement) must precede the EQUIVALENCE statement.

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

DIMENSION STATEMENT

General Form
DIMENSION $\underline{a}_1(\underline{k}_1),\underline{a}_2(\underline{k}_2),\underline{a}_3(\underline{k}_3),\ldots,\underline{a}_n(\underline{k}_n)$
Where: $\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$ are array names.
$\underline{k}_1,\underline{k}_2,\underline{k}_3,\ldots,\underline{k}_n$ are each composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

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)
```

General Form
Type $a(k_1),b(k_2),\ldots,z(k_n)$ Where: Type is INTEGER, REAL, or DOUBLE PRECISION. a,b,...,z are variable, array, or function names (see the chapter "Subprograms") $(k_1),(k_2),\ldots,(k_n)$ are optional and give dimension information for arrays. Each k is composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

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

Example 1:

```
INTEGER DEV,SMALL
```

Explanation:

This statement declares the type of the variables DEV and SMALL as integer and, thus, overrides the implied declaration made by the predefined convention.

Example 2:

```
REAL ITA,JOB,MATRIX (5,2,6)
```

Explanation:

This statement declares the type of the array MATRIX and variables ITA and JOB as real. In addition, it declares the size (dimension) of the array MATRIX. This statement overrides the implied declaration made by the predefined convention.

Example 3:

```
DOUBLE PRECISION DOUB,TWIN
```

Explanation:

This statement declares the type of the variables DOUB and TWIN as double precision.

COMMON STATEMENT

General Form	
COMMON	$\underline{a}(\underline{k}_1),\underline{b}(\underline{k}_2),\underline{c}(\underline{k}_3),\ldots,\underline{z}(\underline{k}_n)$
Where:	$\underline{a},\underline{b},\underline{c},\ldots,\underline{z}$ are variable or array names.
	$\underline{k}_1,\underline{k}_2,\underline{k}_3,\ldots,\underline{k}_n$ are optional and are each composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

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 chapter "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.

The COMMON statement can be used to provide dimension information for an array if the array name first appears in the COMMON statement. If the array has appeared first in an explicit specification statement, the array cannot be dimensioned in the COMMON statement.

The length of a common area can be increased by using an EQUIVALENCE statement (see "EQUIVALENCE Statement").

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

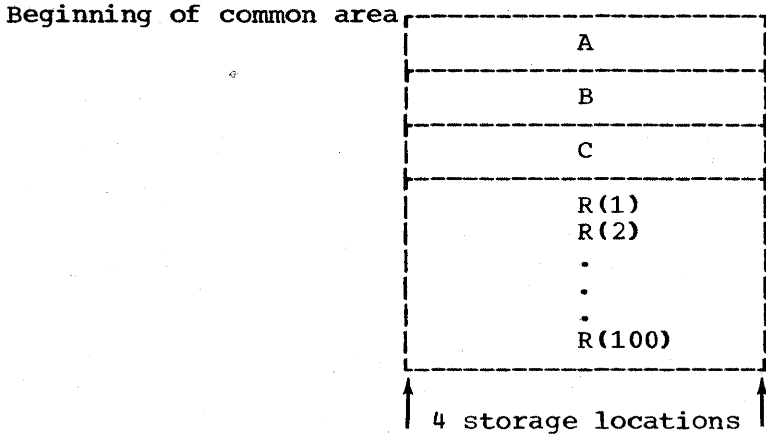
<u>Example 1</u>:

```
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:



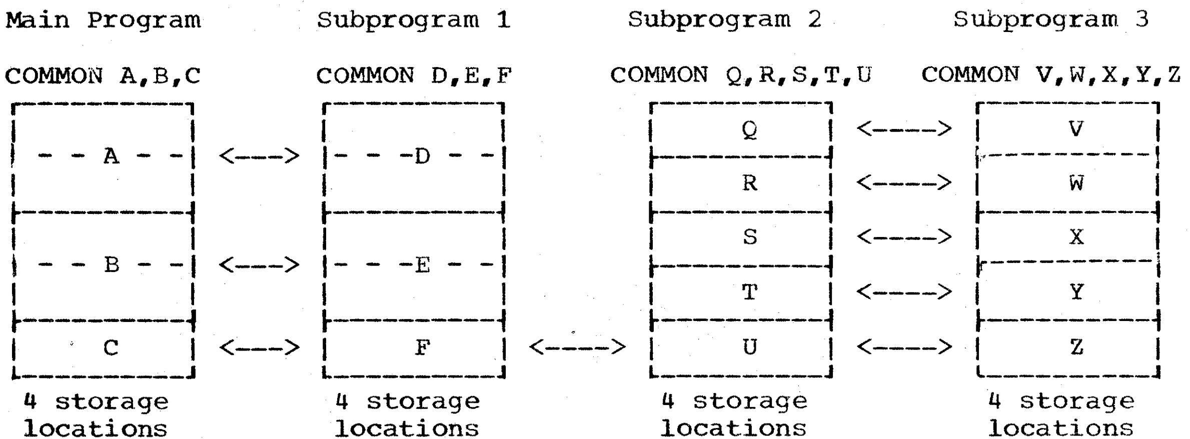
The statement COMMON X, Y, Z, S(100) would then cause the variables X, Y, Z, and S(1)...S(100) to share the same storage space as A, B, C, and R(1)...R(100), respectively. Note that values for X, Y, Z, and S(1) ...S(100), because they occupy the same storage locations as A, B, C, and R(1)...R(100), do not have to be transmitted in the argument list of a CALL statement.

Example 2:

Assume common is defined in a main program and in three subprograms as follows:

Main program:	COMMON	A,B,C, (A and B are 8 storage locations, C is 4 storage locations)
Subprogram 1:	COMMON	D,E,F (D and E are 8 storage locations, F is 4 storage locations)
Subprogram 2:	COMMON	Q,R,S,T,U (4 storage locations each)
Subprogram 3:	COMMON	V,W,X,Y,Z (4 storage locations each)

The correspondence of these variables within common can be illustrated as follows:



In this case, the variables A,B,C and D,E,F may be validly referred to in their respective programs, as may Q,R,S,T,U and V,W,X,Y,Z. In addition, subprogram 1 may implicitly refer to C, U, and Z by explicitly referring to F.

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.

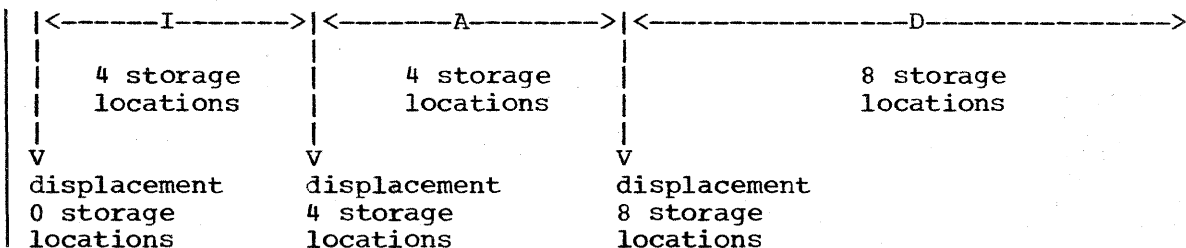
Proper alignment is achieved either by arranging the variables in a fixed descending order according to length, or by constructing the block so that dummy variables force proper alignment. If the fixed order is used, the variables must appear in the following order:

```
double precision    (length of 8)
real or integer     (length of 4)
```

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 its length. The first variable is positioned as though its length specification were eight. Therefore, a variable of any length can be the first assigned. To obtain the proper alignment for other variables, it may be necessary to insert a dummy variable. For example, if the variables I, A, and D are integer, real, and double precision, respectively, and form a common block that is defined as:

```
COMMON I, A, D
```

then the displacement of these variables within common is illustrated as follows:



The displacements of A and D are evenly divisible by their length. However if A were omitted, the displacement of D would not be evenly divisible by its length. In this case, proper alignment is ensured by inserting a dummy variable with a length of four between I and D.

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, 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 remaining 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 100 x 100 is needed; in the final stages of the program C is no longer used, but arrays A and B of sizes 50 x 50 and 100, respectively, are used. The elements of all three arrays are of the type real. 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))
```

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, 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) to 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)
```

STORAGE ARRANGEMENT OF VARIABLES IN EQUIVALENCE GROUPS

Variables in an equivalence group may be in any order in main storage. However, considerable object-time efficiency is lost unless the programmer ensures that all of the variables have proper boundary alignment, as in the COMMON statement.

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:

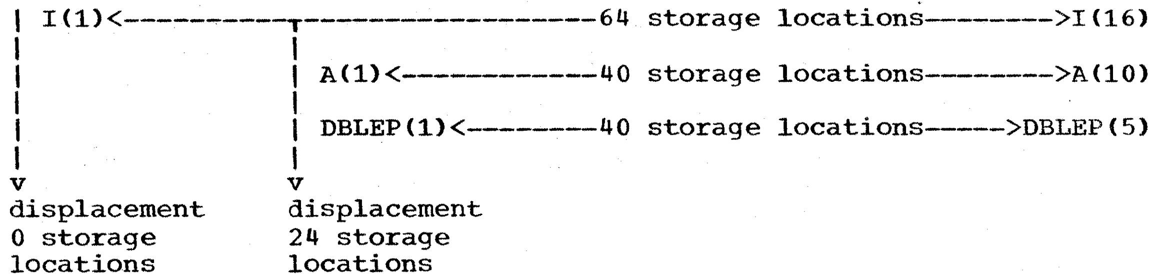
length of 8 (double precision)
length of 4 (real or integer)

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 length of 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 first variable in each group is positioned as if its length specification were 8.

For example, the variables A, I, and DBLEP are real, integer, and double precision, respectively, and are defined as:

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

Then, the displacement of these variables within the group is illustrated as follows:



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

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

then DBLEP is improperly aligned (its displacement of 20 is not evenly divisible by its length of 8).

Note that this discussion applies solely to the manner in which the equivalence group is arranged in storage. This arrangement is not affected by the order in which the variable and array names are listed in the EQUIVALENCE statement. For example, the EQUIVALENCE statement (A(1),I(7),DBLEP(1)) has exactly the same effect as EQUIVALENCE (DBLEP(1),A(1),I(7)).

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). FUNCTION subprograms differ from SUBROUTINE subprograms in that FUNCTION subprograms return at least one value to the calling program, whereas SUBROUTINE subprograms need not return any.

Statement functions are also discussed in this chapter since they are similar to FUNCTION subprograms. The difference is: subprograms are a different program unit from the program unit referring to them, while statement function definitions and references are in the same program unit.

NAMING SUBPROGRAMS

A subprogram name consists of from one through six alphameric characters, the first of which must be alphabetic. A subprogram name may not be a variable name and may not contain special characters other than blanks (see Appendix A). Blanks embedded in a subprogram name are ignored. The type of a function determines the type of the result that can be returned from it.

1. Type Declaration of a Statement Function: Such declaration may be accomplished in one of two ways: by the predefined convention or by the explicit specification statements. Thus, the rules for declaring the type of variables apply to statement functions.

2. Type Declaration of FUNCTION Subprograms: The declaration can be made by the predefined convention, 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. In order 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 (for example, "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.

General Form
$\underline{name}(\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n)$ = $\underline{expression}$ Where: name is the statement function name (see the section "Naming Subprograms"). $\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$ are dummy arguments. They must be unique (within the statement) nonsubscripted variables. There must be at least one argument in the argument list. expression is any arithmetic 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. A maximum of 15 variables appearing in <u>expression</u> can be used as arguments of the function. 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. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

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

<u>Example</u>:

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.

<u>Examples</u>:

<u>Valid</u> statement function definitions and statement function references:

Definition	Reference
SUM(A,B,C,D) = A+B+C+D	NET = GRDS - SUM(TAX,FICA,HOSP,STOCK)
FUNC(Z) = A+X*Y*Z	ANS = FUNC (RESULT)
	BIGSUM = SUM (A,B, SUM(C,D,E,F),G)

<u>Invalid</u> statement function <u>definitions:</u>

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 referred to in another program.

General Form
Type FUNCTION name (a_1,a_2,...,a_n) Where: Type is INTEGER, REAL, or DOUBLE PRECISION. The type declaration may be made by the predefined convention, by an explicit specification in the FUNCTION statement, or by an explicit specification statement within the FUNCTION subprogram. name is the name of the FUNCTION. a_1,a_2,...,a_n are dummy arguments. They must be nonsubscripted variable, array or dummy names of a SUBROUTINE or another FUNCTION subprogram. There must be at least one argument in the argument list.

A type declaration in the FUNCTION statement overrides any type declaration by an explicit specification statement within the FUNCTION subprogram. The function must also be typed in the calling program if the predefined convention is not used.

Since the FUNCTION subprogram is a separate program, the variable names 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 or another 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 (that is, a_1,a_2,a_3,...,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 dummy 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                     FUNCTION Subprogram
          .                             FUNCTION CALC (A,B,J)
          .                                       .
          .                                       .
    ANS = ROOT1*CALC(X,Y,I)                       .
          .                             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                     FUNCTION Subprogram

    INTEGER CALC                        INTEGER FUNCTION CALC(I,J,K)
        .                                   .
        .                                   .
        .                                   .
    ANS = ROOT1*CALC(N,M,P)             CALC = I+J+K**2
                                            .
                                            .
                                            .
                                        RETURN
                                        END
```

Explanation:

The FUNCTION subprogram CALC is declared as type INTEGER.

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 physical 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.

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 ($a_1,a_2,a_3,\ldots,a_n$) . . . 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. They must be nonsubscripted variable or array names, or the dummy names of other SUBROUTINE or FUNCTION subprograms. (There need not be any arguments.)

Since the SUBROUTINE subprogram is a separate program, the variable names 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 or another 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 on the left side 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 ($\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$) 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 variables 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

```
DIMENSION X(100),Y(100)
   .
   .
K = 100
CALL COPY (X,Y,K)
   .
   .
   .
```

SUBROUTINE Subprogram

```
   SUBROUTINE COPY(A,B,N)
   DIMENSION A(100),B(100)
   DO 10 I = 1, N
10 B(I) = A(I)
   RETURN
   END
```

CALL Statement

The CALL statement is used to call a SUBROUTINE subprogram.

General Form
CALL name ($\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$) Where: name is the name of a subroutine subprogram. $\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$ are the actual arguments that are being supplied to the 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 values of the actual arguments that appear in the CALL statement.

RETURN Statement in a SUBROUTINE Subprogram

General Form
RETURN

This is the exit from a subprogram. The RETURN statement signifies the conclusion of a computation. The subprogram transmits argument values and returns control to the statement following the CALL in the calling program. There may be several RETURN statements in a subprogram. In a main program, a RETURN statement performs the same function as a STOP statement.

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, in effect, 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, and type to the actual arguments. For example, if the actual argument is an integer, then the dummy argument must be an integer. If a dummy argument is an array, the corresponding actual argument must be (1) an array, or (2) an array element. In the first instance, the size of the dummy array must not exceed the size of the actual array. In the second, the size of the dummy array must not exceed the size of that portion of the actual array which follows and includes the designated element.

The actual arguments can be:

- Any type of constant
- Any type of subscripted or nonsubscripted variable
- An array name
- An arithmetic expression
- The name of a FUNCTION or SUBROUTINE subprogram

An actual argument that is the name of a subprogram must be identified in the calling program 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 nonsubscripted 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 so 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 (i.e., cannot appear to the left of an equal sign in an arithmetic statement) in the function DERIV because the actual argument list causes both A and B to be associated with more than one value.

EXTERNAL Statement

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

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

If the name of a FORTRAN supplied in-line function is used in an EXTERNAL statement, it 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

```
      .
      .
      .
EXTERNAL MULT
      .
      .
      .
CALL SUB (J, MULT,C)
      .
      .
      .
```

Subprogram

```
    SUBROUTINE SUB(K,Y,Z)
    IF (K) 4,6,6
4   D = Y (K,Z**2)
          .
          .
          .
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, and the variables J and C are passed to the dummy variables K and Z, respectively. Subprogram MULT is called and executed only if the value of K is negative.

Example 2:

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

In this example, an EXTERNAL statement is not required because subprogram MULT is not an argument; it is executed first and the result becomes the argument.

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	, (comma)
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.

APPENDIX B: BASIC FORTRAN IV IMPLEMENTATION DIFFERENCES

The differences among the four implementations of the Basic FORTRAN IV language are minor except for the absence of the direct access input/output statements in Basic Programming Support FORTRAN IV. These differences are indicated in the body of this publication by footnotes and, in addition, are summarized in Table 3.

The following abbreviations are used in Table 3:

OS - Operating System Basic FORTRAN IV (E)
DOS - Disk Operating System Basic FORTRAN IV
TOS - Tape Operating System Basic FORTRAN IV
BPS - Basic Programming Support Tape System FORTRAN IV

Table 3. Implementation Differences

	OS	DOS/TOS	BPS
Direct access input/output statements	Direct access input/output is available.	Direct access input/output is available. A program with direct access input/output may be compiled using either DOS or TOS, but it must be executed using DOS.	Direct access input/output is not available.
Key word and blank restriction	Control card option may be used to either keep or remove the restriction. See IBM System/360 Operating System: FORTRAN IV (E) Programmer's Guide.	The restriction has been removed; there is no option to retain it.	The restriction has been removed; there is no option to retain it.
Maximum array size	131,068 storage locations	32,767 storage locations.	32,767 storage locations.
Maximum sequential record size	Depends upon the input/output device in use. See the FORTRAN (E) Programmer's Guide above.	256 characters per record (TOS). 260 characters per record (DOS).	256 characters per record.
Subprogram names	Any valid FORTRAN name may be used unless key word restriction is retained when key words may not be used.	Any valid FORTRAN name may be used.	Any valid FORTRAN name may be used.

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.

Subprograms are either in-line or out-of-line. An in-line subprogram is inserted by the FORTRAN compiler at any point in the program where there is a reference to the subprogram. The in-line subprograms are mathematical function subprograms. These subprograms are listed in Table 4.

One asterisk in the Argument Type column of Table 4 identifies the function as an intrinsic function in USAS Basic FORTRAN.

• Table 4. In-Line Mathematical Function Subprograms

Function	Entry Name	Definition	No. of Arg.	Argument Type	Function Value Type
Absolute value	IABS ABS DABS	|Arg|	1 1 1	Integer * Real * Double precision	Integer Real Double precision
Float	FLOAT DFLOAT	Convert from integer to real	1 1	Integer * Integer	Real Double precision
Fix	IFIX	Convert from real to integer	1	Real *	Integer
Transfer of sign	SIGN ISIGN DSIGN	Sign of Arg_2 times $\|Arg_1\|$	2 2 2	Real * Integer * Double precision	Real Integer Double precision
Positive difference	DIM IDIM	Arg_1-Min(Arg_1, Arg_2)	2 2	Real Integer	Real Integer
Obtaining most significant part of a double-precision argument	SNGL		1	Double precision	Real
Express a real argument in double-precision form	DBLE		1	Real	Double precision

An out-of-line subprogram is located in a library and the compiler generates an external reference to it. These subprograms are mathematical function subprograms and service subprograms. Out-of-line mathematical function subprograms are listed in Table 5; out-of-line service subprograms are listed in Table 6. A detailed description of all out-of-line subprograms is contained in the publication IBM System/360: FORTRAN IV Library Subprograms, Form C28-6596.

Two asterisks in the Argument Type column of Table 5 identify the function as a basic external function of USAS Basic FORTRAN.

Note: A variable used as an argument of any mathematical function subprogram must be of the type specified in the table. For example, if a program uses FLOAT to convert integer to real, the argument must be of type integer.

• Table 5. Out-of-Line Mathematical Function Subprograms

Function	Entry Name	Definition	No. of Arg.	Argument Type	Function Value Type
Exponential	EXP DEXP	e^{Arg} e^{Arg}	1 1	Real ** Double precision	Real Double precision
Natural Logarithm	ALOG DLOG	ln(Arg) ln(Arg)	1 1	Real ** Double precision	Real Double precision
Common Logarithm	ALOG10 DLOG10	$\log_{10}$(Arg) $\log_{10}$(Arg)	1 1	Real Double precision	Real Double precision
Arctangent (in radians)	ATAN DATAN	arctan(Arg) arctan(Arg)	1 1	Real ** Double precision	Real Double precision
Trigonometric Sine	SIN DSIN	sin(Arg) sin(Arg) (Arg in radians)	1 1	Real ** Double precision	Real Double precision
Trigonometric Cosine	COS DCOS	cos(Arg) cos(Arg) (Arg in radians)	1 1	Real ** Double precision	Real Double precision
Square Root	SQRT DSQRT	$(Arg)^{1/2}$ $(Arg)^{1/2}$	1 1	Real ** Double precision	Real Double precision
Hyperbolic Tangent	TANH DTANH	tanh(Arg) tanh(Arg)	1 1	Real ** Double precision	Real Double precision
Modular Arithmetic[1] (Remaindering)	MOD AMOD DMOD	Arg_1(mod Arg_2)	2 2 2	Integer Real Double precision	Integer Real Double precision
Truncation[1]	INT AINT IDINT	Sign of Arg times largest integer ≤|Arg|	1 1 1	Real Real Double precision	Integer Real Integer
Largest value	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Max(Arg_1, ...,Arg_n)	≥2 ≥2 ≥2 ≥2 ≥2	Integer Real Integer Real Double precision	Real Real Integer Integer Double precision
Smallest value	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Min(Arg_1, ...,Arg_n)	≥2 ≥2 ≥2 ≥2 ≥2	Integer Real Integer Real Double precision	Real Real Integer Integer Double precision

[1]These functions are in-line in FORTRAN IV. A program written in Basic FORTRAN IV that passes these names via an argument list and an EXTERNAL statement may not be compiled correctly by a FORTRAN IV compiler.

Table 6. 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 previously defined 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

This sample program (Figure 1) is designed to find all of the prime numbers between 2 and 1000. A prime number is an integer greater than 1 that cannot be evenly divided by any integer except itself and 1. Thus 2, 3, 5, 7, 11,... are prime numbers. The number 9 is not a prime number since it can be evenly divided by 3.

IBM FORTRAN Coding Form — Program: SAMPLE PROGRAM 1 — Date: 8/69 — Page 1 of 1

```
C PRIME NUMBER GENERATOR
      WRITE (3,1)
    1 FORMAT ('1FOLLOWING IS A LIST OF PRIME NUMBERS FROM 2 TO 1000'/
     *19X,'2'/19X,'3')
      DO 4 I=5,1000,2
      K=SQRT(FLOAT(I))
      DO 2 J=3,K,2
      IF (MOD(I,J)) 2,4,2
    2 CONTINUE
      WRITE (3,3) I
    3 FORMAT (I20)
    4 CONTINUE
      WRITE (3,5)
    5 FORMAT (' THIS IS THE END OF THE PROGRAM')
      STOP
      END
```

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.

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

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

$$\begin{array}{ll} (1) & W_oa_o + W_1a_1 + \ldots + W_ma_m = Z_o \\ (2) & W_1a_o + W_2a_1 + \ldots + W_{m+1}a_m = Z_1 \\ \cdot & \\ \cdot & \\ \cdot & \\ (m+1) & W_ma + W_{m+1}a_1 + \ldots + W_{2m}a_m = Z_m \end{array}$$

where:

$$W_o = n \qquad Z = \sum_{i=1}^{n} y_i$$

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

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

$$\vdots \qquad \vdots$$

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

$$\vdots$$

$$W_2m = \sum_{i=1}^{n} x_i^{2m}$$

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).

$$\begin{array}{ll} (1) & W_oa_o + W_1a_1 + W_2a_2 = Z_o \\ (2) & W_1a_o + W_2a_1 + W_3a_2 = Z_1 \\ (3) & W_2a_o + W_3a_1 + W_4a_2 = Z_2 \end{array}$$

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) $a_0 + b_{12}a_1 + b_{13}a_2 = b_{14}$

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

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

where:

$b_{12} = W_1/W_0$, $b_{13} = W_2/W_0$, $b_{14} = Z_0/W_0$

$b_{22} = W_2-b_{12}W_1$, $b_{23} = W_3-b_{13}W_1$, $b_{24} = Z_1-b_{14}W_1$

$b_{32} = W_3-b_{12}W_2$, $b_{33} = W_4-b_{13}W_2$, $b_{34} = Z_2-b_{14}W_2$

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) $a_1 + c_{23}a_2 = c_{24}$

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

where:

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

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

The backward solution is as follows:

(9) $a_2 = c_{34}/c_{33}$ from equation (8)

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

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

Figure 2 is a sample FORTRAN program for carrying out the calculations for the case: $n = 100$, $m \le 10$. 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: 8/69 — PAGE 1 OF 3

```
      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 (1,1) 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
```

Figure 2. Sample Program 2 (Part 1 of 3)

IBM FORTRAN Coding Form

PROGRAM: SAMPLE PROGRAM 2 — DATE: 8/69 — PAGE 2 OF 3

```
   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
```

Figure 2. Sample Program 2 (Part 2 of 3)

IBM FORTRAN Coding Form

PROGRAM: SAMPLE PROGRAM 2
DATE: 8/69
PAGE 3 OF 3

```
   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 (3,2) (A(I),I=1,L2)
      STOP
      END
```

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, X_i and Y_i are selected. The powers of X_i are computed and accumulated in the correct W counters. The powers of X_i are multiplied by Y_i 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 X_i is used when computing the next power of X_i. Note the use of variables as index parameters. By the time control has passed to statement 17, the counters have been set as follows:

$$W(1) = N \qquad Z(1) = \sum_{I=1}^{N} Y_I$$

$$W(2) = \sum_{I=1}^{N} X_I \qquad Z(2) = \sum_{I=1}^{N} Y_I X_I$$

$$W(3) = \sum_{I=1}^{N} X_I^2 \qquad Z(3) = \sum_{I=1}^{N} Y_I X_I^2$$

$$\vdots \qquad \vdots$$

$$Z(M+1) = \sum_{I=1}^{N} Y_I X_I^M$$

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

By the time control has passed to statement 23, the values of W_0, W_1, ..., W_{2m+1} have been placed in the storage locations corresponding to columns 1 through M+1, row 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, in 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 manual 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:

$$\begin{matrix} 1 & b_{12} & b_{13} & b_{14} \\ 0 & 1 & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \end{matrix}$$

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 A(I) terms, the M+1 values of the A(I) 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 A(I) terms are then printed according to the FORMAT specification in statement 2.

APPENDIX E: FORTRAN IV FEATURES NOT IN BASIC FORTRAN IV

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

- ASSIGN
- BLOCK DATA
- Labeled COMMON
- COMPLEX
- DATA
- Debug facility
- More than three dimensions in arrays
- 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 (DOUBLE PRECISION is provided as an explicit type)
- IMPLICIT
- Call by name
- Literal as argument of CALL
- ENTRY
- RETURN i (i not a blank)
- Statement number as CALL argument
- NAMELIST
- PAUSE with literal
- G, Z, and L format codes
- Nesting of group format specifications
- Complex, logical, literal, and hexadecimal constants
- Generalized subscript form

The following in-line subprograms in FORTRAN IV are not in Basic FORTRAN IV. Asterisks indicate subprograms which are out-of-line in Basic FORTRAN IV and in-line in FORTRAN IV.

REAL	DCONJG	INT*
AIMAG	CONJG	AINT*
DCMPLX	HFIX	IDINT*
CMPLX		

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

ARSIN	ALGAMA	DSINH
ARCOS	DGAMMA	DCOSH
DARSIN	DLGAMA	TAN
DARCOS	CLOG	COTAN
ERF	CDLOG	DTAN
ERFC	CSIN	DCOTAN
DERF	CCOS	CSQRT
DERFC	CDSIN	CDSQRT
CEXP	CDCOS	ATAN2
CDEXP	SINH	DATAN2
GAMMA	COSH	CABS
		CDABS

APPENDIX F: IBM BASIC FORTRAN IV EXTENSIONS TO USAS BASIC FORTRAN

IBM Basic FORTRAN IV is compatible with and encompasses USAS Basic FORTRAN. The following list gives the IBM extensions to the USAS Basic FORTRAN.

Array dimension specification in COMMON statement

Carriage control

Direct access (except Basic Programming Support System FORTRAN IV)

Double-precision constants

Explicit specification statements

Explicit type-specification in FUNCTION statement

EXTERNAL

A, D, P, and T format codes and literals enclosed in single quotation marks

Mixed-mode expressions

Return of values via the argument list in a FUNCTION subprogram

Three-dimensional arrays (instead of two)

(Where more than one page reference is given, the major reference is first.)

+ (addition) 16-19
- (subtraction) 16-19
* (multiplication) 16-19
** (exponentiation) 16-19
/ division 16-19
   in FORMAT statement 35,36-37

A format code 41-42,35
ABS 75
absolute value 75
addition (+) 16-19
AINT 77
alignment
   COMMON 59
   EQUIVALENCE 61
ALOG 77
ALOG10 77
alphabetic characters 73
alphameric characters 10
AMAX0 77
AMAX1 77
AMIN0 77
AMIN1 77
AMOD 77
apostrophe, literal data 42-43
arctangent 77
argument, subprogram 70-71,67-69
arithmetic assignment statement 20-21
arithmetic expression 16-19
arithmetic IF statement 23-24,27
arithmetic statement, definition 6
array 13-15
   declaring
      DIMENSION statement 55
      explicit specification 56
   maximum size 74
assignment statement, arithmetic 20-21
associated variable 47
asterisks, in input/output 37,40
ATAN 77

BACKSPACE statement 45
blanks
   FORMAT statements 35,43
   in statements 7
boundary alignment
   COMMON statement 59
   EQUIVALENCE statement 61

CALL statement 69,68
carriage control character 36,43
characters 73
character data, FORMAT statement 35,41-42
coding form 7
coding statements 7
comma, FORMAT statement 35,36-37

comments 7
common block
   DEFINE FILE variables 48
   EQUIVALENCE restrictions 60
COMMON statement 57-59
compile 5
computation, order of 17-18
computed GO TO statement 23
constants 8-10
   double-precision 9-10
   integer 8
   real 9-10
continuation card 7
CONTINUE statement 28
control character, carriage 36,43
control statement
   definition 6
   examples 22-29
conversion, arithmetic 20-21
COS 77
cosine 77

D format code 38-40,35
DABS 75
data
   allocation EQUIVALENCE 60
   by use of FORMAT 35
data set reference number 30
   (see also programmer's guides in Preface)
DATAN 77
DBLE 75
DCOS 77
DEFINE FILE statement 47-49,45-46
DEXP 77
DFLOAT 75
difference, positive 75
DIM 75
DIMENSION statement 55,13
dimensions
   COMMON statement 57
   explicit specification 58
direct access statements 45-54
   examples 53-54
   explanation 47-52
divide check exception 78
division (/) 16-19
DLOG 77
DLOG10 77
DMAX1 77
DMIN1 77
DMOD 77
DO statement 24-27
DOUBLE PRECISION 56
   constant 9-10
   FORMAT statement 35,38-40
DSIGN 75
DSIN 77
DSQRT 77
DTANH 77
DVCHK 78

dummy argument
statement function definition 65
subprograms 70-71,66,69
dummy statement 28
DUMP 78

E format code 38-40,35
E in DEFINE FILE 47
element, array 13
END FILE statement 44
END statement 29
FUNCTION subprogram 67-68
program unit 7
equation 20
EQUIVALENCE statement 60-62
with COMMON statement 60-61
storage arrangements 61-62
executable statement, definition 7
EXIT 78
EXP 77
explicit specification statement 12-13,56
exponentiation (**) 16-19
exponentiation, subprogram 77
expression 16-19,6
EXTERNAL statement 71-72,70

F format code 38-40,35
FIND statement 52,45-46
fix 20,75
float 20,75
FORMAT statement 35-44,6
input/output
READ 32-33,49-50
WRITE 33-34,51
order in program 7
formatted records 35-44
READ 33
WRITE 34
FORTRAN IV features 86-87
FUNCTION subprogram
arguments in 70-71
definition 66-68,64

GO TO statement 22-23
computed 23
restriction of 27
unconditional 22
group format specification 35,44

H format code 42-43,35
hyperbolic tangent 77

I format code 37,35
IABS 75
IDIM 75
IDINT 77
IF statement 23-24
IFIX 75

implementation, differences in systems 74
in-line subprograms 75-77
input/output statement 30-54,6
direct access 45-54
sequential 32-45
INT 77
integer
data
constants 8
FORMAT statements 35,37
explicit specification of 56
ISIGN 75

key words 74,6

L in DEFINE FILE 47
length, of variable 12
listing, source program
comments 7
literal data
in FORMAT statement 35,42-43
restriction 73
logarithm 77
looping 24-27

mathematical function subprograms 75-77
in-line 75
out-of-line 77
MAX0 77
MAX1 77
maximum 77
MIN0 77
MIN1 77
minimum 77
MOD 77
modular arithmetic 77
multiplication (*) 16-19

name
subprogram 63
variable 10-11
number, statement 7
numeric characters 73
numeric format code 38-40,35

operating system 5,74
operator, arithmetic 16
out-of-line subprograms 75,78
OVERFL 78

P scale factor 40,35,38
parentheses, arithmetic expression 18
PAUSE statement 29,27
PDUMP 78
positioning record 35,43
predefined specification 12,56

printer, carriage control 36
program, samples of 79-85
program unit 7,10

READ statement
   direct access 49-50,45-46
   sequential 32-33
real
   data
      constants 9-10
      FORMAT statements 35,38-40
   explicit specification 56
record size, maximum 74
RETURN statement
   FUNCTION subprogram 67-68
   restriction 27
   SUBROUTINE subprogram 70
REWIND statement 45

scale factor, P 40,35,38
sense light 78
sequential input/output 32-45
service subprograms 78,75-76
SIGN 75
sign, transfer 75
SIN 77
sine 77
size
   array 13-14
   reduction by EQUIVALENCE 60
slash (/) 35,36-37
   (see also division)
SLITE 78
SLITET 78
SNGL 75
source program 6,7
special characters 73
specification
   explicit 12
   predefined 12
specification statement 55-62
   definition 6-7
   EXTERNAL 71-72
SQRT 77
square root 77
statement
   control 22-29
   types 6-7
statement number 7
statement function definition 63
   definition 6,64-65
   order in program 7
STOP statement 29,27
storage dump 78
subprograms 63-72
   FUNCTION 63,75-77
   service 75,78
   SUBROUTINE 63
subprogram statement
   definition 6
   order in program 7
SUBROUTINE subprogram
   arguments in 70-71
   definition 68-69
subscript 14-15
subscript quantity 13
subscripted variable 13,60
subtraction (-) 16-19
symbolic names 10

T format code 43,35
tangent, hyperbolic 77
TANH 77
truncation 77
type 12-14
   result of arithmetic 18-19
   subprogram 63

U in DEFINE FILE 47
unconditional GO TO statement 22
unformatted records
   READ 33
   WRITE 34
USAS FORTRAN 87

variable 11-12,6
   names 11
   subscripted 13

WRITE statement
   direct access 51,45-46
   sequential 33-34

X format code 43,35

GC28-6629-2

S/360 Basic FORTRAN IV Lang. Printed in U.S.A. GC28-6629-2

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
[U.S.A. only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]