GC28-6515-10

Systems

IBM System/360 and System/370 FORTRAN IV Language

This publication describes and illustrates the use of the FORTRAN IV language for IBM System/360 and System/370. It is primarily a reference manual for programmers who are familiar with the elements of the language.

FORTRAN IV is a mathematically-oriented language useful in writing programs for applications that involve manipulation of numerical data.

IBM

PREFACE

This publication describes the IBM System/360 and System/370 FORTRAN IV language. It is intended to be used as a reference manual by persons writing programs in the FORTRAN IV language. A reader should have some knowledge of FORTRAN before using this publication. A useful source for this information is the set of programmed instruction texts, FORTRAN IV for IBM System/360 and System/370, Order Nos. SR29-0081, SR29-0084, SR29-0085, SR29-0086, and SR29-0087.

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

Appendixes contain additional information useful in writing a FORTRAN IV program. This information consists of a table of source program characters; a list of other FORTRAN statements accepted by FORTRAN IV; a list of FORTRAN-supplied mathematical and service subprograms; lists of differences between FORTRAN IV and Basic FORTRAN IV and between FORTRAN IV and ANS FORTRAN; sample programs; extensions to the FORTRAN IV language supported by the FORTRAN IV (H Extended), FORTRAN IV (G1), and Code and Go FORTRAN compilers, and a glossary.

Information pertaining to the FORTRAN IV libraries, and compiler restrictions and programming considerations, will be found in the System Reference Library publication for the respective library or compiler. A list of such publications is contained in IBM System/360 and System/370 Bibliography, Order No. GA22-6822, or the General Information manual for the program product.

As this book is revised, a summary of amendments will be included with the TNL or complete revision. It will be inserted immediately following the cover page and will highlight the changes made. As this book changes over a period of time, these summaries of amendments will form, as the the first part of the book, a permanent section that will trace, in reverse chronological order, the development of this publication. Any revision of the complete book will include a reprinting of all previous summaries of amendments.

Eleventh Edition (May 1974)

This is a major revision of, and makes obsolete, the previous editions, GC28-6515-8 (as amended by Technical Newsletters GN28-0595 and GN28-0610) and GC28-6515-9 (as amended by Technical Newsletter GN28-0610).

Changes are periodically made to the specifications herein. Any such changes will be reported in subsequent revisions or Technical Newsletters. Before using this publication in connection with the operation of IBM systems, refer to the latest IBM System/360 and System/370 Bibliography, Order No. GA22-6822 for editions that are applicable and current.

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

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, New York 10020. Comments should mention the compiler and level being used.

Summary of Amendments Number 3

Date of Publication: May 15, 1974
Form of Publication: Revision, GC28-6515-10

Miscellaneous Changes

New: Documentation Only

Certain errors in the shading of IBM extensions to ANS FORTRAN have been corrected.

The discussion of FORTRAN IV statement coding has been clarified.

The explanation of literal constant restrictions has been clarified.

The maximum number of dimensions permitted when declaring the size of an array is three in ANS FORTRAN and seven in FORTRAN IV.

The restriction that a logical IF statement may not have a statement number has been explicitly stated in the description of its general form.

An explanation of the effect of a sequential WRITE or END FILE statement has been added.

The illustration of NAMELIST output has been revised to show the correct type of output.

The P scale factor output example has been revised.

The discussion of sharing associated variables has been revised.

The restriction on repeating specification statement information has been explicitly stated.

The explanation of the COMMON statement example has been revised.

The explanation of the effect of the RETURN statement on storage entities in subprograms has been revised.

The description of the DISPLAY statement list has been revised.

An example of an invalid statement function reference has been added.

The mathematical function tables in Appendix C have been revised and clarified.

Editorial changes having no technical significance are not noted here.

Specific changes to the text of this publication are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Date of Publication: March 31, 1973

Form of Publication: TNL GN28-0610 to GC28-6515-8, -9

Miscellaneous Changes

Maintenance: Documentation Only

Shading indicating IBM extensions to ANS FORTRAN has been corrected.

The description of the order of computation has been changed slightly to reflect that left to right computation is within a hierarchical level.

The rules governing data types in exponentiation operations have been revised to indicate that negative operands may not have real exponents.

The examples illustrating integer division have been clarified to better show the effects of truncation.

The description of the use of subscripts in an implied DO has been clarified to indicate that unsubscripted array names refer to entire arrays.

The description of the scale factor when used with G formats has been revised to indicate the range outside which the effect of the scale factor is suspended.

The description of the DATA statement has been revised to indicate that real, integer, and complex constants may be signed and that storage entities may only be initialized once.

An explanation of the effect of the RETURN statement on storage entities in subprograms has been added.

The definition of BLOCK DATA subprograms has been clarified to indicate that they are not called but only supply initial data values.

The definitions of certain FORTRAN library functions have been updated.

FLOAT has been deleted from the list of GENERIC names for library functions.

The lists of IBM extensions to ANS Basic FORTRAN and ANS FORTRAN have been revised to include statement numbers as arguments in CALL statements and asterisks as dummy arguments in SUBROUTINE statements.

Editorial changes of no technical significance are not noted here.

Specific changes to text made as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Summary of Amendments — Number 1

Date of Publication: March 1971
Form of Publication: TNL GN28-0595 to GC28-6515-8

Automatic Function Selection (GENERIC Statement)

Change: Specification

Eight new function names, referred to as aliases, are recognized by the FORTRAN IV (H Extended) compiler. These names are the generic names for their respective families, and also may be used in program units in which automatic function selection has not been requested.

Miscellaneous Changes

Maintenance: Documentation Only

Certain errors in the shading of IBM extensions to ANS FORTRAN have been corrected.

The discussion of NAMELIST output data has been modified to emphasize that character data cannot be produced.

The explanation of the general form of the direct-access WRITE statement has been amended to clarify the circumstances in which the I/O list is optional.

Integer variables of length 4 may be specified in the subscript declarator of a DOUBLE PRECISION type-statement.

The explanation of receipt by location has been rewritten to clarify how it differs from receipt by value.

A sentence specifying the order of certain data initialization statements in a program unit has been revised.

Two items have been added to the list of IBM extensions to ANS FORTRAN in Appendix G.

A correction has been made to the DINT entry in Appendix C.

Editorial changes of no technical significance are not noted here.

Specific changes to text made as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

ILLUSTRATIONS

FIGURES

TABLES

IBM System/360 and System/370 FORTRAN IV consists of a language, a library of subprograms, and a compiler.

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

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

In a process called compilation, a program called the FORTRAN compiler analyzes the source program statements and translates them into a machine language program called the object program, which will be suitable for execution on IBM System/360 and System/370. In addition, when the FORTRAN compiler detects errors in the source program, it produces appropriate diagnostic error messages. The FORTRAN IV programmer's guides contain information about compiling and executing FORTRAN programs.

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

The IBM System/360 and System/370 FORTRAN IV language is compatible with and encompasses the American National Standard (ANS) FORTRAN, X3.9-1966, including its mathematical function provisions. It also contains, as a proper subset, Basic FORTRAN IV. Appendixes F and G contain lists of differences between FORTRAN IV and Basic FORTRAN IV and ANS FORTRAN. In the body of this manual, IBM extensions to ANS FORTRAN are demarcated by shading of the text, in the manner of this sentence.

STATEMENTS

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

1. It causes certain operations to be performed (e.g., addition, multiplication, branching)

2. It specifies the nature of the data being handled

3. It 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. Assignment Statements: These statements cause calculations to be performed. The result replaces the current value of a designated variable or array element.

2. Control Statements: These statements enable the user to govern the order of execution of the object program and terminate its execution.

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 in a FORTRAN record on an input/output device.

5. NAMELIST Statement: This statement is used in conjunction with certain input/output statements to specify data appearing in a special kind of record.

6. DATA Initialization Statement: This statement is used to assign initial values to variables and array elements.

7. Specification Statements: These statements are used to declare the properties of variables, arrays, and functions (such as type and amount of storage reserved) and, in addition, can be used to assign initial values to variables and arrays.

8. Statement Function Definition Statement: This statement specifies operations to be performed whenever the statement function name appears in an executable statement.

9. Subprogram Statements: These statements enable the user to name and to specify arguments for functions and subroutines.

The basic elements of the language are discussed in this section. The actual FORTRAN statements in which these elements are used are discussed in following sections. The term program unit refers to a main program or a subprogram; the term executable statements refers to those statements in categories 1, 2, and 3, above. An executable program consists of a main program plus any number of subprograms and/or external procedures.

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

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

FORMAT and DATA statements may appear anywhere after the IMPLICIT statement, if present, and before the END statement. DATA statements, however, must follow any specification statements that contain the same variable or array names. A NAMELIST statement declaring a NAMELIST name must precede the use of that name in any input/output statement.

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

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.

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. The FORTRAN compiler does not process comments other than to print them as part of the source program listing. Comments may appear anywhere in the program except between the cards of a FORTRAN statement that is contained on more than one card.

FORTRAN statements are written on one or more cards within columns 7 through 72. The first card of a statement may have a statement number in columns 1 through 5 and must have a blank or zero in column 6. The statement number, which must not be zero, consists of from 1 to 5 decimal digits. Blanks and leading zeros in a statement number are ignored. The values of the statement numbers do not affect the order in which the statements are executed.

A FORTRAN statement that is not confined to one card may be continued onto as many as 19 additional cards. A continuation card has any character other than a blank or zero in column 6. The statement is then continued within columns 7 through 72. Columns 1 through 5 may contain any characters, except that the letter C must not appear in column 1. The characters in columns 1 through 5 are ignored.

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

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

CONSTANTS

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

Numerical constants are integer, real, or complex numbers; logical constants are .TRUE. or .FALSE.; literal constants are a string of alphameric and/or special characters; and hexadecimal constants are hexadecimal (base 16) numbers.

INTEGER CONSTANTS

Definition
Integer Constant - a whole number written without a decimal point. It occupies four locations of storage (i.e., four bytes).
Maximum Magnitude: 2147483647 (i.e., $2^{31}-1$).

An integer constant may be positive, zero, or negative. If unsigned and nonzero, it is assumed to be positive. (A zero may be written with a preceding sign, which has no effect on the value zero.) 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 maximum magnitude)
5,396               (Contains an embedded comma)
```

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

A real constant may be positive, zero, or negative (if unsigned and nonzero, it is assumed to be positive) and must be within the allowable range. It may not contain embedded commas. A zero may be written with a preceding sign, which has no effect on the value zero. The decimal exponent permits the expression of a real constant as the product of a basic real constant or integer constant times 10 raised to a desired power.

Examples:

Valid Real Constants (four storage locations):

```
+0.
-999.9999
7.0E+0              (i.e., 7.0 x 10^0 = 7.0)
19761.25E+1         (i.e., 19761.25 x 10^1 = 197612.5)
7.E3
7.0E3               (i.e., 7.0 x 10^3 = 7000.0)
7.0E+03
7E-03               (i.e., 7.0 x 10^-3 = 0.007)
21.98753829457168   (Note: this level of precision cannot be
                     accommodated in four storage locations)
```

Valid Real Constants (eight storage locations):

```
1234567890123456.D-93    (Equivalent to .1234567890123456x10^-77)
7.9D03
7.9D+03                  (i.e., 7.9 x 10^3 = 7900.0)
7.9D+3
7.9D0                    (i.e., 7.9 x 10^0 = 7.9)
7D03                     (i.e., 7.0 x 10^3 = 7000.0)
```

Invalid Real Constants:

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

COMPLEX CONSTANTS

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

The real constants in a complex constant may be positive, zero, or negative (if unsigned and nonzero, they are assumed to be positive), and must be within the allowable range. A zero may be written with a preceding sign, which has no effect on the value zero.

Examples:

Valid Complex Constants

(3.2,-1.86)	(Has the value 3.2 - 1.86i)
(-5.0E+03,.16E+02)	(Has the value -5000. + 16.0i)
(4.7D+2,1.973614D4)	(Has the value 470. + 19736.14i)
(47D+2,38D+3)	(Has the value 4700. + 38000.i)

Where $i = \sqrt{-1}$

Invalid Complex Constants:

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

LOGICAL CONSTANTS

Definition
Logical Constant - a constant that specifies a logical value "true" or "false." There are two logical constants: .TRUE. .FALSE. Each occupies four storage locations. The words TRUE and FALSE must be preceded and followed by periods.

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

LITERAL CONSTANTS

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

Each character requires one byte of storage. The number of characters in the string, including blanks, may not be less than 1 or greater than 255. If apostrophes delimit the literal, a single apostrophe within the literal is represented by two apostrophes. If wH precedes the literal, a single apostrophe within the literal is represented by a single apostrophe.

Literals can be used only in the argument list of a CALL statement or function reference (other than a statement function), as data initialization values, or in FORMAT statements. The first form, a string enclosed in apostrophes, may be used in PAUSE statements.

Examples:

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

HEXADECIMAL CONSTANTS

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

Hexadecimal constants may be used only as data initialization values.

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

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

Examples:

Z1C49A2F1 represents the bit string: 00011110001001001101000101111000 01

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

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

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

If the number of digits is greater than the maximum, the leftmost hexadecimal digits are truncated; if the number of digits is less than the maximum, hexadecimal zeros are supplied on the left.

SYMBOLIC NAMES

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

Symbolic names are used in a program unit to identify elements in the following classes:

- An array and the elements of that array (see "Arrays")
- A variable (see "Variables")
- A statement function (see "Statement Functions")
- An intrinsic function (see Appendix C)
- A FUNCTION subprogram (see "FUNCTION Subprograms")
- A SUBROUTINE subprogram (see "SUBROUTINE Subprograms")
- A common-block name (see "BLOCK DATA Subprogram")
- An external procedure name that cannot be classified in that program unit as either a SUBROUTINE or FUNCTION subprogram name (see "EXTERNAL Statement")
- A NAMELIST name (see "READ and WRITE Using NAMELIST")

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

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

Once a symbolic name is used as a FUNCTION subprogram name, a SUBROUTINE subprogram name, a block name, or an external procedure name in any unit of an executable program, no other program unit of that executable program can use that name to identify an entity of these classes in any other way.

VARIABLES

A FORTRAN variable is a data item, identified by a symbolic name, that occupies a storage area. The value specified by the name is always the current value stored in the area.

For example, in the statement

```
A = 5.0+B
```

both A and B are variables. The value of B has been determined by some previously executed statement. The value of A is calculated when this statement is executed and depends on the previously calculated value of B.

Before a variable has been assigned a value its contents are undefined, and the variable may not be referred to except to assign it a value.

VARIABLE NAMES

FORTRAN variable names must follow the rules governing symbolic names. The use of meaningful variable names can serve as an aid in documenting a program.

Examples:

Valid Variable Names:

```
B292S
RATE
$VAR
```

Invalid Variable Names:

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

VARIABLE TYPES AND LENGTHS

The type of a variable corresponds to the type of data the variable represents. Thus, an integer variable represents integer data, a real variable represents real data, etc. There is no variable type associated with literal or hexadecimal data. These types of data are identified by a name of one of the other types.

For every type of variable, there is a corresponding standard and optional length specification which determines the number of storage locations (bytes) that are reserved for each variable. The following list shows each variable type with its associated standard and optional length:

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

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

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

An explicit specification statement overrides an IMPLICIT statement, which, in turn, overrides the predefined specification. The optional length specification of a variable may be declared only by the IMPLICIT or explicit specification statements. If, in these statements, no length specification is stated, the standard length is assumed (see the section "Type Statements").

Type Declaration by the Predefined Specification

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

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

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

This convention is the traditional FORTRAN method of implicitly specifying the type of a variable as being either integer or real. In all examples that follow in this publication it is presumed that this specification applies unless otherwise noted. Variables defined with this convention are of standard length.

Type Declaration by the IMPLICIT Statement

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

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

Type Declaration by Explicit Specification Statements

Explicit specification statements differ from the first two ways of specifying the type of a variable, in that an explicit specification statement declares the type of a particular variable by its name rather than as a group of variable names beginning with a particular letter.

For example, assume that an IMPLICIT statement overrode the predefined convention for variable names beginning with the letter I by declaring them to be real and that a subsequent explicit specification statement declared that the variable named ITEM is complex. Then, the variable ITEM is complex and all other variable names beginning with the character I are real. Note that variable names beginning with the letters J through N are specified as integer by the predefined convention.

The explicit specification statements are discussed in greater detail in the section "Specification Statements."

ARRAYS

A FORTRAN array is a set of data items identified by a symbolic name, called the array name. The data items which the array comprises are called array elements. A particular element in the array is identified by the array name and its position in the array (e.g., first element, third element, seventh element, etc.).

Consider the array named NEXT, which consists of five elements, each currently representing the following values: 273, 41, 8976, 59, and 2. Each 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.

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

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

To refer to any element in an array, the array name plus a parenthesized subscript must be used. In particular, the array name alone does not represent the first element.

Before an array element has been assigned a value its contents are undefined, and the array element may not be referred to except to assign it a value.

The following array named LIST is 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

The element in row 2, column 3 would be referred to as LIST (2,3), and has the value 14. Ordinary mathematical notation might use $LIST_{ij}$ to represent any element of the array LIST. In FORTRAN, this is written as LIST(I,J), where I equals 1, 2, 3, 4, or 5 and J equals 1, 2, or 3.

SUBSCRIPTS

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

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

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

2. Subscript quantities may contain function references.

3. Subscript quantities may contain array elements.

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

5. The evaluated result of a subscript quantity should always be greater than zero.

Examples:

Valid Array Elements:

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

Invalid Array Elements:

```
ARRAY (-5)          (A subscript quantity may not be negative)
LOT (0)             (A subscript quantity may not be nor assume a
                      value of zero)
ALL(.TRUE.)         (A subscript quantity may not assume a true or
                      false value)
NXT (1+(1.3,2.0))   (A subscript quantity may not assume a complex
                      value)
```

DECLARING THE SIZE AND TYPE OF AN ARRAY

The size (number of elements) of an array is declared by specifying in a subscript declarator the number of dimensions in the array, and the size of each dimension. Each dimension is represented by an integer constant or integer variable. A maximum of seven dimensions (three in ANS FORTRAN) is permitted. The size of each dimension is equal to the value of its respective constant or variable.

Size information must be given for all arrays in a FORTRAN program so that an appropriate amount of storage may be reserved. Declaration of this information is made by a DIMENSION statement, a COMMON statement, or by one of the explicit specification statements; these statements are discussed in detail in the section "Specification Statements." The type of an array name is determined by the conventions for specifying the type of a variable name. Each element of an array is of the type specified for the array name.

ARRANGEMENT OF ARRAYS IN STORAGE

An array is stored in ascending storage locations, with the value of the first of its subscript quantities increasing most rapidly and the value of the last increasing least rapidly.

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

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

The array named B, described by two subscript quantities with the first subscript quantity varying 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.

EXPRESSIONS

FORTRAN IV provides two kinds of expressions: arithmetic and logical. The value of an arithmetic expression is always a number whose type is integer, real, or complex. The value of a logical expression is always a truth value: true or false. Expressions may appear in assignment statements and in certain control statements.

ARITHMETIC EXPRESSIONS

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

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

Examples:

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

Arithmetic Operators

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

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

   In fact, AB is regarded as a single variable with a two-letter name.

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

   A*B or B*A

2. No two arithmetic operators may appear consecutively 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

   A*(-B)

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

3. Order of Computation: Computation is performed according to the hierarchy of operations shown in the following list. Within a hierarchical level, computation is performed from left to right.

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 sequential operations is performed first. If the first operator is higher than or equal to the second, the first operation is performed. If not, the second operator is compared to the third, etc. When the end of the expression is encountered, all of the remaining operations are performed in reverse order.

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

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 sequential 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=-(B*C)

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

Parentheses may be used in arithmetic expressions, as in algebra, to specify the order in which the arithmetic operations are to be performed. Where parentheses are used, the expression within the parentheses is evaluated before the result is used. 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 W	B/(W*C)+A**2
2.	W*C	Call the result X	B/X+A**2
3.	B/X	Call the result Y	Y+A**2
4.	A**2	Call the result Z	Y+Z
5.	Y+Z	Final operation	

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

First Operand \ Second Operand	INTEGER (2)	INTEGER (4)	REAL (4)	REAL (8)	COMPLEX (8)	COMPLEX (16)
INTEGER (2)	Integer (2)	Integer (4)	Real (4)	Real (8)	Complex (8)	Complex (16)
INTEGER (4)	Integer (4)	Integer (4)	Real (4)	Real (8)	Complex (8)	Complex (16)
REAL (4)	Real (4)	Real (4)	Real (4)	Real (8)	Complex (8)	Complex (16)
REAL (8)	Real (8)	Real (8)	Real (8)	Real (8)	Complex (16)	Complex (16)
COMPLEX (8)	Complex (8)	Complex (8)	Complex (8)	Complex (16)	Complex (8)	Complex (16)
COMPLEX (16)	Complex (16)	Complex (16)	Complex (16)	Complex (16)	Complex (16)	Complex (16)

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

5. A REAL*4 or REAL*8 operand may have an INTEGER*2, INTEGER*4, REAL*4, or REAL*8 exponent. An INTEGER*2 or INTEGER*4 operand may have an INTEGER*2, INTEGER*4, REAL*4, or REAL*8 exponent. However, a negative operand (either REAL or INTEGER) may not have a REAL exponent. The exponent of a complex operand must be an integer value. No other combinations are allowed. The type of the result depends upon the types 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.

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

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

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

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

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

6. When division is performed using two integers, any remainder is truncated (without rounding) and an integer answer is given. The sign is determined normally.

   Examples:

I	J	I/J
9	2	4
-5	2	-2
1	-4	0

LOGICAL EXPRESSIONS

The simplest form of logical expression consists of a single logical primary. A logical primary can be a logical constant, logical variable, logical array element, logical function reference, relational expression, or logical expression enclosed in parentheses. A logical primary, when evaluated, always has the value true or false.

More complicated logical expressions may be formed by using logical operators to combine logical primaries.

Relational Expressions

Relational expressions are formed by combining two arithmetic expressions with a relational operator. The six relational operators, each of which must be preceded and followed by a period, are as follows:

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

The relational operators express an arithmetic condition which can be either true or false. The relational operators may be used to compare two integer expressions, two real expressions, or a real and an integer expression.

Examples:

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

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

Then the following examples illustrate valid and invalid relational expressions.

Valid Relational Expressions:

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

Invalid Relational Expressions:

C .GE. (2.7,5.9E3)	(Complex quantities may never appear in logical expressions)
L .EQ. (A+F)	(Logical quantities may never be joined by relational operators)
E**2 .LT 97.1E1	(Missing period immediately after the relational operator)
.GT. 9	(Missing arithmetic expression before the relational operator)

Logical Operators

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

Logical Operator	Use	Meaning
.NOT.	.NOT.A	If A is true, then .NOT.A has the value false; if A is false, then .NOT.A has the value true.
.AND.	A.AND.B	If A and B are both true, then A.AND.B has the value true; if either A or B or both are false, then A.AND.B has the value false.
.OR.	A.OR.B	If either A or B or both are true, then A.OR.B has the value true; if both A and B are false, then A.OR.B has the value false.

The only valid sequences of two logical operators are .AND..NOT. and .OR..NOT.; the sequence .NOT..NOT. is invalid.

Only those expressions which, when evaluated, have the value true or false may be combined with the logical operators to form logical expressions.

Examples:

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

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

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

Valid Logical Expressions:

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

Invalid Logical Expressions:

A .AND. L	(A is not a logical expression)
.OR. W	(.OR. must be preceded by a logical expression)
NOT. (A .GT. F)	(Missing period before the logical operator .NOT.)
(C .EQ. I) .AND. L	(A complex quantity may never be an operand of a relational operator)
L .AND. .OR. W	(The logical operators .AND. and .OR. must always be separated by a logical expression)
.AND. L	(.AND. must be preceded by a logical expression)

Order of Computations in Logical Expressions: The order in which the operations are performed is:

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

For example, the expression:

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

is effectively evaluated in the following order:

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

Note: Logical expressions may not require that all parts be evaluated. Functions within logical expressions may or may not be called. For example, in the expression A.OR.LGF(.TRUE.), it should not be assumed that the LGF function is always invoked, since it is not necessary to do so to evaluate the expression when A has the value true.

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

```
.NOT.((B.GT.C.OR.K).AND.L)
```

is evaluated in the following order:

```
1.  B.GT.C     Call the result X    .NOT.((X.OR.K).AND.L)
2.  X.OR.K     Call the result Y    .NOT.(Y.AND.L)
3.  Y.AND.L    Call the result Z    .NOT.Z
4.  .NOT.Z     Final operation
```

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

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

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

In the second expression, .NOT.A is evaluated first. The result is true; but .TRUE..OR.B is the equivalent of .TRUE.. Therefore, the value of the second expression is true. Note that the value of B is irrelevant to the result in this example. Thus, if B were a function reference, it would not have to be evaluated.

Length of a Relational Expression: A relational expression is always evaluated to a LOGICAL*4 result.

Results of the various logical operations are shown in Table 2.

Table 2. Determining the Type and Length of the Result of Logical Operations

First operand \ Second operand	LOGICAL (1)	LOGICAL (4)
LOGICAL (1)	LOGICAL (1)	LOGICAL (4)
LOGICAL (4)	LOGICAL (4)	LOGICAL (4)

General Form
a = b Where: a is a variable or array element. b is an arithmetic or logical expression.

This FORTRAN statement closely resembles a conventional algebraic equation; however, the equal sign specifies replacement rather than equality. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable or array element to the left of the equal sign.

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

Assume that the type of the following data items has been specified as:

Symbolic Name	Type	Length Specification
I, J, W	Integer variables	4,4,2
A, B, C, D	Real variables	4,4,8,8
E	Complex variable	8
F(1),...,F(5)	Real array elements	4
G, H	Logical variables	4,4

Then the following examples illustrate valid arithmetic statements using constants, variables, and array elements of different types:

Statements	Description
A = B	The value of A is replaced by the current value of B.
W = B	The value of B is truncated to an integer value, and this value replaces the value of W.
A = I	The value of I is converted to a real value, and this result replaces the value of A.
I = I + 1	The value of I is replaced by the value of I + 1.
E = I**J+D	I is raised to the power J and the result is converted to a real value to which the value of D is added. This result replaces the real part of the complex variable E. The imaginary part of the complex variable is set to zero.
A = C*D	The most significant part of the product of C and D replaces the value of A.
A = E	The real part of the complex variable E replaces the value of A.

Statements	Description
E = A	The value of A replaces the value of the real part of the complex variable E; the imaginary part is set equal to zero.
G = .TRUE.	The value of G is replaced by the truth value true.
H = .NOT.G	If G is true, the value of H is replaced by the truth value false. If G is false, the value of H is replaced by the truth value true.
G = 3..GT.I	The value of I is converted to a real value; if the real constant 3. is greater than this result, the value true replaces the value of G. If 3. is not greater than I, the truth value false replaces the value of G.
E = (1.0,2.0)	The value of the complex variable E is replaced by the value of the complex constant (1.0,2.0). Note that the statement E = (A,B), where A and B are real variables, is invalid. The mathematical function subprogram CMPLX can be used for this purpose. See Appendix C.
F(1) = A	The value of element 1 of array F is replaced by the value of A.
E = F(5)	The real part of the complex constant E is replaced by the value of array element F(5). The imaginary part is set equal to zero.

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

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

Notes:

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

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

GO TO STATEMENTS

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

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

UNCONDITIONAL GO TO STATEMENT

General Form
GO TO xxxxx Where: xxxxx is the number of an executable statement in the same program unit.

This GO TO statement causes control to be transferred to the statement specified by the statement number. Every subsequent execution of this GO TO statement results in a transfer to that same statement. Any executable statement immediately following this statement should have a statement number; otherwise it can never be referred to or executed.

Example:

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

Explanation:

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

COMPUTED GO TO STATEMENT

General Form
GO TO (x_1,x_2,x_3,...,x_n), i Where: Each x is the number of an executable statement in the program unit containing the GO TO statement. i is an integer variable (not an array element) which must be given a value before the GO TO statement is executed.

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 range $1 \leq i \leq n$, the next statement is executed.

Example:

```
      GO TO (25,10,7,10), ITEM
345 C = 7.02
        .
        .
        .
  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 or 4 statement 10 is executed next, and so on. If ITEM is less than 1 or greater than 4, statement 345 is executed next.

General Form
ASSIGN i TO m . . . GO TO m, (x_1,x_2,x_3,...,x_n) Where: i is the number of an executable statement. It must be one of the numbers x_1,x_2,x_3,...,x_n. Each x is the number of an executable statement in the program unit containing the GO TO statement. m is an integer variable (not an array element) of length 4 which is assigned one of the statement numbers: x_1,x_2,x_3,...,x_n.

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

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

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

At the time of execution of an assigned GO TO statement, the current value of m must have been defined to be one of the values x_1, x_2,...,x_n by the previous execution of an ASSIGN statement. The value of the integer variable m is not the integer statement number; ASSIGN 10 TO I is not the same as I = 10.

Any executable statement immediately following this statement should have a statement number; otherwise it can never be referred to or executed.

Example 1:

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

Explanation:

In Example 1, statement 50 is executed immediately after statement 10.

Example 2:

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

Explanation:

In Example 2, the first time statement 13 is executed, control is transferred to statement 10. On the second execution of statement 13, control is transferred to statement 25.

ADDITIONAL CONTROL STATEMENTS

ARITHMETIC IF STATEMENT

General Form
IF (a) x_1,x_2,x_3 Where: a is an arithmetic expression of any type except complex. Each x is the number of an executable statement in the program unit containing the IF statement.

The arithmetic IF statement causes control to be transferred to the statement numbered x_1,x_2, or x_3 when the value of the arithmetic expression (a) is less than, equal to, or greater than zero, respectively.

Any executable statement immediately following this 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
40    D = C**2
        .
        .
        .
4     D = B + C
        .
        .
        .
30    C = D**2
        .
        .
        .
10    E = (F*B)/D+1
        .
        .
        .
```

Explanation:

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

LOGICAL IF STATEMENT

General Form
IF (a) s Where: a is any logical expression. s is any executable statement except a DO statement or another logical IF statement. The statement s may not have a statement number.

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

Example 1:

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

Explanation:

In the first statement, if the value of the expression is true (i.e., A is less than or equal to 0.0), the statement GO TO 25 is executed next and control is passed to the statement numbered 25. If the value of the expression is false (i.e., A is greater than 0.0), the statement GO TO 25 is ignored and control is passed to the second statement.

In the third statement, if the value of the expression is true (i.e., A is equal to B), the value of ANSWER is replaced by the value of the expression (2.0*A/C) and then the fourth statement is executed. If the value of the expression is false (i.e., A is not equal to B), the value of ANSWER remains unchanged and the fourth statement is executed next.

Example 2:

Assume that P and Q are logical variables.

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

Explanation:

In the first statement, if the value of the expression is true, the value of A is replaced by the value of B and the second statement is executed next. If the value of the expression is false, the statement A = B is skipped and the second statement is executed.

DO STATEMENT

General Form

	End of Range	DO Variable		Initial Value	Test Value	Increment
DO	x	i	=	m_1,	m_2,	m_3

Where: x is the number of an executable statement appearing after the DO statement in the program unit containing the DO.

i is an integer variable (not an array element) called the DO variable.

m_1, m_2, and m_3, are either unsigned integer constants greater than zero or unsigned integer variables (not array elements) whose value is greater than zero. The value of m_1 should not exceed that of m_2. m_2 may not exceed $2^{31}-2$ in value. m_3 is optional; if it is omitted, its value is assumed to be 1. In this case, the preceding comma must also be omitted.

The DO statement is a command to execute, at least once, the statements that physically follow the DO statement, up to and including the statement numbered x. These statements are called the range of the

DO. The first time the statements in the range of the DO are executed, i is initialized to the value m_1; each succeeding time i is increased by the value m_3. When, at the end of the iteration, i is equal to the highest value that does not exceed m_2, control passes to the statement following the statement numbered x. Upon completion of the DO, the DO variable is undefined and may not be used until assigned a value (e.g., in an arithmetic assignment statement).

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

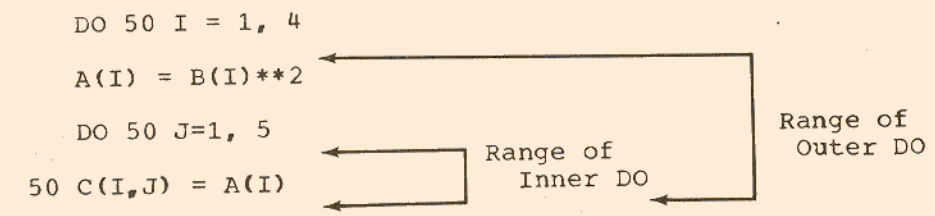
In Example 3, statement 25 is the end of the range of the DO loop. The DO variable, I, is set to the initial value of 1. Before the second execution of the DO loop, I is increased by the increment, 2, and the second and third statements are executed a second time. After the fifth execution of the DO 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

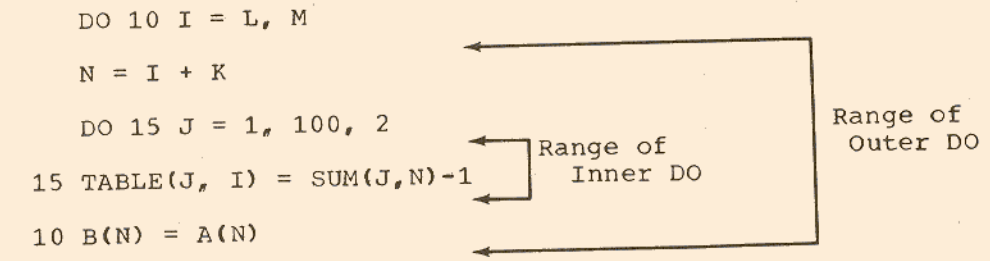
1. The indexing parameters of a DO statement (i, m_1, m_2, m_3) should not be changed by a statement within the range of the DO loop.

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

   Example 1:



   Example 2:



3. A transfer out of the range of any DO loop is permissible at any time. The DO variable is defined when such a transfer is executed, and its value is the value it has when the transfer is executed.

4. The extended range of a DO is defined as those statements that are executed between the transfer out of the innermost DO of a set of

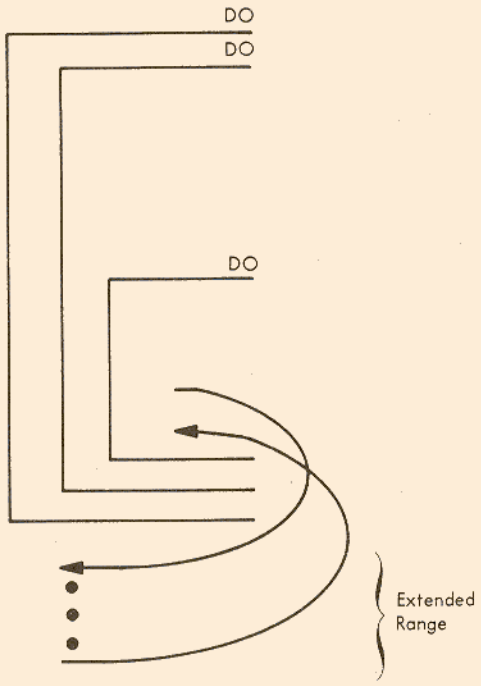
completely nested DO's and the transfer back into the range of this innermost DO. In a set of completely nested DO's, the first DO is not in the range of any other DO, and each succeeding DO is in the range of every DO which precedes it. The following restrictions apply:

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

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

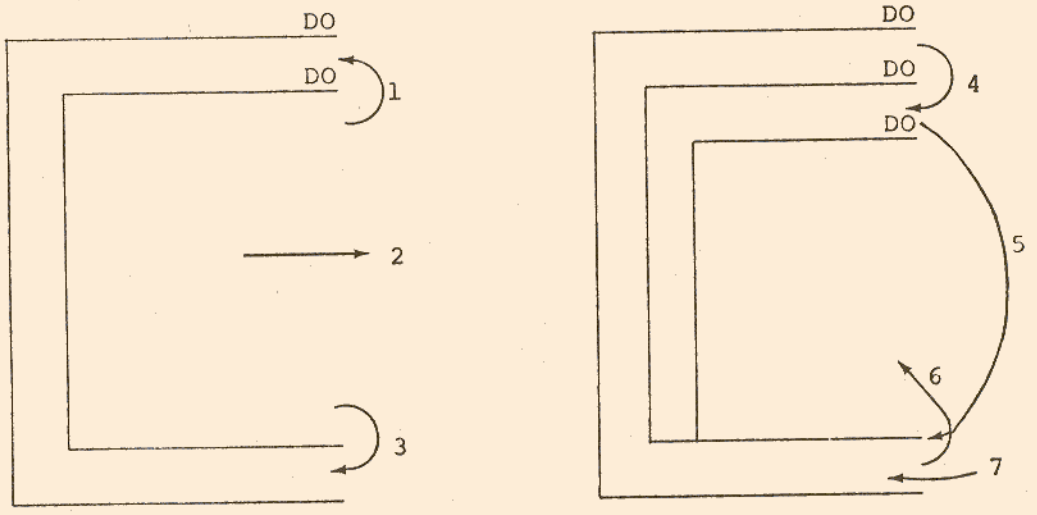
- The indexing parameters ($\underline{i}$, $\underline{m}_1$, $\underline{m}_2$, $\underline{m}_3$) cannot be changed in the extended range of the DO.

Example 3:



5. A statement that is the end of the range of more than one DO statement is within the innermost DO. 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 4:



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.

6. 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.

7. 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 statement, another DO statement, or a logical IF statement containing any of these forms.

8. The use of, and return from, a subprogram from within any DO loop in a nest of DO's, or from within an extended range, is permitted.

CONTINUE STATEMENT

General Form
CONTINUE

CONTINUE is a statement that may be placed anywhere in the source program (where an executable statement may appear) without affecting the sequence of execution. It may be used as the last statement in the range of a DO in order to avoid ending the DO loop with a GO TO, PAUSE, STOP, RETURN, arithmetic IF, another DO statement, or a logical IF statement containing any of these forms.

Example 1:

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

Explanation:

In Example 1, the CONTINUE statement is used as the last statement in the range of the DO in order to avoid ending the DO loop with the statement GO TO 7.

Example 2:

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

Explanation:

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

PAUSE STATEMENT

General Form
PAUSE PAUSE n PAUSE 'message' Where: n is a string of 1 through 5 decimal digits. 'message' is a literal constant enclosed in apostrophes and containing alphameric and/or special characters. Within the literal, an apostrophe is indicated by two successive apostrophes.

PAUSE n, PAUSE message, or PAUSE 00000 is displayed, depending upon whether n, 'message' or no parameter was specified, and the program waits until operator intervention causes it to resume execution, starting with the next statement after the PAUSE statement. For further information, see the FORTRAN IV programmer's guide for the respective system.

STOP STATEMENT

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

The STOP statement terminates the execution of the object program and displays STOP n if n is specified. For further information, see the FORTRAN IV programmer's guide for the respective system.

END STATEMENT

General Form
END

The END statement is a nonexecutable statement that defines the end of a main program or subprogram. Physically, it must be the last statement of each program unit. It may not have a statement number, and it may not be continued. The END statement does not terminate program execution. To terminate execution, a STOP statement or a RETURN statement in the main program is required.

Input/output statements 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.

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.

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

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

I/O List: Input/output statements in FORTRAN are primarily concerned with the transfer of data between storage locations defined in a FORTRAN program and records which are external to the program. On input, data is taken from a record and placed into storage locations that are not necessarily contiguous. On output, data is gathered from diverse storage locations and placed into a record. An I/O list is used to specify which storage locations are used. The I/O list can contain variable names, array elements, array names, or a form called an implied DO (see below). No function references or arithmetic expressions are permitted in an I/O list, except in subscripts of array elements in the list. If a function reference is used in a subscript, the function may not perform input/output.

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

If an 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.")

Implied DO: If an implied DO appears in the I/O list, the variables, array elements, or arrays specified by the implied DO are transmitted. The implied DO specification is enclosed in parentheses. Within the parentheses are one or more variables, array elements, or array names, separated by commas, with a comma following the last name, followed by indexing parameters $\underline{i} = \underline{m}_1, \underline{m}_2, \underline{m}_3$. The indexing parameters are as defined for the DO statement. Their range is the list of the DO-implied list and, for input lists, $\underline{i}$, $\underline{m}_1$, $\underline{m}_2$, and $\underline{m}_3$ may appear within that range only in subscripts.

For example, assume that A is a variable and that B, C, and D are 1-dimensional arrays each containing 20 elements. Then the statement:

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

writes the current value of variable A, the entire array B, the first four elements of the array C, and the fourth element of D. (The 6 following the WRITE is the data set reference number.) If the subscript (I) were not included with array C, the entire array would be written four times.

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

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

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

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

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

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

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

For unformatted output data, the I/O list determines the length of the record. An output record is complete when the current values of all the items in the I/O list have been placed in it, plus any control words supplied by the input/output routines or Data Management. For further information, see the FORTRAN IV programmer's guide for the respective system.

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

SEQUENTIAL INPUT/OUTPUT STATEMENTS

There are five sequential input/output statements: READ, WRITE, END FILE, REWIND, and BACKSPACE. The READ and WRITE statements cause transfer of records of sequential data sets. The END FILE statement defines the end of a data set; the REWIND and BACKSPACE statements control the positioning of data sets. In addition to these five statements, the FORMAT and NAMELIST statements, although not input/output statements, are used with certain forms of the READ and WRITE statements.

After execution of a sequential WRITE or END FILE statement, no record exists in the data set following the last record transferred by that statement.

READ STATEMENT

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

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

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

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

The basic forms of the READ statements are:

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

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

Formatted READ

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

Example:

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

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

Unformatted READ

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

Example:

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

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

WRITE STATEMENT

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

The three basic forms of the WRITE statement are:

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

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

Formatted WRITE

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

Example:

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

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

Unformatted WRITE

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

Example:

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

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

READ AND WRITE USING NAMELIST

The NAMELIST statement is a specification statement used in conjunction with the READ(a,x) and WRITE(a,x) statements. It provides for reading and writing data without including the list specification in the READ and WRITE statements. The NAMELIST statement declares a name x to refer to a particular list of variables or array names. Neither a dummy variable name nor a dummy array name may appear in the list. Thereafter, the forms READ(a,x) and WRITE(a,x) are used to transmit data between the data set associated with the reference number a and the variables specified by the NAMELIST name x. A BACKSPACE statement may not be executed for a data set which is read or written using NAMELIST.

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

General Form
NAMELIST /x_1/$list_1$/x_2/$list_2$/.../x_n/$list_n$
Where: Each x is a NAMELIST name
Each list is a list of the form
$a_1, a_2, \ldots, a_n$
where each a is a variable or array name.

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

1. A NAMELIST name is a symbolic name which must not be the same as a variable or array name.
2. A NAMELIST name is enclosed in slashes. The list of variable or array names belonging to a NAMELIST name ends with a new NAMELIST name enclosed in slashes or with the end of the NAMELIST statement.

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

4. A NAMELIST name must be declared in a NAMELIST statement and may be declared only once. The name may appear only in input/output statements.

5. The NAMELIST statement must precede any statement function definitions and all executable statements.

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

NAMELIST Input Data

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

The form of the data items in an input record is:

- symbolic name = constant

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

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

  The set of constants consists of constants of the type integer, real, literal, complex, or logical. The number of constants must be less than or equal to the number of elements in the array. Successive occurrences of the same constant can be represented in the form k*constant, where k is a nonzero integer constant specifying the number of times the constant is to occur.

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

Each data record must begin with a blank followed by a complete variable or array name or constant. Embedded blanks are not permitted in names or constants. Trailing blanks after integers and exponents are treated as zeros.

NAMELIST Output Data

When output data is written using a NAMELIST list, it is written in a form that can be read using a NAMELIST list. All variable and array names specified in the NAMELIST list and their values are written out, each according to its type. Literal data cannot be produced. The fields for the data are made large enough to contain all the significant digits. The values of a complete array are written out in columns.

Example: Assume that A is a 3-element array, I and L are 3 X 3 arrays, and that the following statements are given:

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

READ (5,NAM1)

WRITE (6,NAM2)
```

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

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

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

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

```
                        Column 2
                        |
                        v
First line              &NAM2

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

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

Fourth line             &END
```

Note: The blanks that would normally separate the variables and elements have been omitted in reproducing this example because of space limitations.

General Form

xxxxx FORMAT (c_1,c_2,...,c_n)

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

c_1,c_2,...,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.)
aZw	(Describes hexadecimal data fields.)
paGw.s	(Describes integer, real, or logical data fields.)
aLw	(Describes logical data fields.)
aAw	(Describes character data fields.)
'literal'	(Indicates literal data.)
wH	(Indicates 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 a repeat count, an unsigned integer constant used to denote the number of times the format code or group is to be used. If a is omitted, the code or group is used only once.

w is an unsigned nonzero integer constant that specifies the number of characters in the field.

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

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

r is an unsigned integer constant designating a character position in a record.

p is optional and represents a scale factor designator of the form nP where n is an unsigned or negatively signed integer constant.

(...) is a group format specification. Within the parentheses are format codes or an additional level of groups, separated by commas or slashes.

The FORMAT statement is used in conjunction with the I/O list in the READ and WRITE statements to specify the structure of FORTRAN records and the form of the data fields within the records. In the FORMAT statement, the data fields are described with format codes, and the order in which these format codes are specified gives the structure of the FORTRAN records. The I/O list gives the names of the data items to make up the record. The length of the list, in conjunction with the FORMAT statement, specifies the length of the record (see the section "Various Forms of a FORMAT Statement").

Throughout this section, the examples show punched card input and printed line output. However, the concepts apply to all input/output media. In the examples, the character b represents a blank.

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

1. FORMAT statements are not executed; their function is to supply information to the object program. They may be placed anywhere in a program unit other than a BLOCK DATA subprogram, subject to the rules for the placement of the FUNCTION, SUBROUTINE, IMPLICIT, and END statements.

2. Complex data in records require two successive D, E, F, or G format codes.

3. Either one comma or any number of slashes can be used as separators between format codes (see the section "Various Forms of a FORMAT Statement").

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

5. When formatted records are prepared for printing at a printer or terminal, 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 1H*x*, where *x* is one of the following:

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

   For media other than a printer or terminal, the first character of the record is treated as data.

6. 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").

7. To illustrate the nesting of group format specifications, FORMAT (...,a(...,a(...),...,a(...),...),...) is permitted, but FORMAT (...,a(...,a(...,a(...),...),...),...) is invalid, because a group within another group cannot itself contain a group.

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, slash and comma. The slash indicates the end of the physical record; the comma indicates the end of a data item within the record.

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 codes T, X, H, and literals enclosed in apostrophes. These communicate information directly with the record.

Whenever an I, D, E, F, G, A, L, or Z 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, even if there is an unsatisfied repeat count.

If, however, format control reaches the last (outer) right parenthesis of the format specification, a test is made to determine if another element is specified in the I/O list. If not, control terminates. However, if another list element is specified, the format control demands that a new record start. Control therefore reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification.

Given the following FORMAT statements:

```
70 FORMAT (2(I3,F5.2),I4,F3.1)
80 FORMAT (I3,F5.2,2(I3,2F3.1))
90 FORMAT (I3,F5.2,2I4,5F3.1)
```

With additional elements in the I/O list after control has reached the last right parenthesis of each, control would revert to the 2(I3,F5.2) specification in the case of statement 70; to 2(I3,2F3.1) in the case of statement 80; and to the beginning of the format specification, I3,F5.2,... in the case of statement 90.

The question of whether there are further elements in the I/O list is asked only when an I, D, E, F, G, A, L, or Z 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. The format codes, separated by commas, are enclosed in a pair of parentheses. One FORTRAN record is defined within a single pair of left and right parentheses. For an example, see the section "Examples of Numeric Format Codes."

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

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

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

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

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

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

I Format Code

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

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

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

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 number which, optionally, may have a D, E, or signed-integer-constant exponent. The D or E may be omitted from the exponent if the exponent is signed. All exponents must be preceded by a constant: i.e., an optional sign followed by at least one decimal digit with or without decimal point. If the decimal point 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, E, or signed-integer-constant 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.

The D, E, and signed-integer-constant exponent specifications for input data are interchangeable. For example, given a REAL*4 item in an input list, and an E or F FORMAT specification, it makes no difference whether the exponent specification of the data item is a D, an E, or a signed integer constant. Similarly, if the list item is REAL*8 or DOUBLE PRECISION and the format specification is D, the exponent specification of the data item may likewise be a D, an E, or a signed integer constant.

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.

Z Format Code

The Z format code is used in transmitting hexadecimal data.

Input: Scanning of the input field proceeds from right to left. Leading, embedded, and trailing blanks in the field are treated as zeros. One storage location (byte) in internal storage contains two hexadecimal digits; thus, if an input field contains an odd number of digits, the number will be padded on the left with a hexadecimal zero when it is stored. If the storage area is too small for the input data, the data is truncated and high-order digits are lost.

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

G Format Code

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

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

Output: The s portion of the format code can be omitted for integer and logical data and the numbers are printed according to the rules for I and L format codes. For real data, the s is used to determine the number of digits to be printed and whether the number should be printed with or without a decimal exponent. If the number, say n, is in the range 0.1≤ n < 10**s, the number is printed without a decimal exponent. Otherwise, it is printed with an E or D decimal exponent, depending on the length specification of the variable in the I/O list. The w specification for real data must include a position for a decimal point, four positions for a decimal exponent, and, if the value is negative, a position for a minus sign. All other rules for output are the same as those for the individual format codes.

Examples of Numeric Format Codes

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

Example 1:

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

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

Explanation:

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

2. When an input card is read, the number in the first field of the card (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 of that card would be stored in integer format in location M. The rest of the card would be ignored.

4. If there were one fewer variable in the list (say C is omitted), format specification G10.3 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 are given:

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

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

where A, C, and D are REAL*4 and B is REAL*8 and that on successive executions of the READ statement, the following input cards are read:

```
Column:    1    5          15         25         35
           |    |          |          |          |
           v    v          v          v          v
           b3F1156432D+02276.38E+15bbbbbbbbbb
Input
           2AF3155381+02b382506E+28276.38E+15
Cards
           3ACb346.18D-03485.322836276.38E+15

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

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

A	B	C	D
03F1	156.432D02	276.38E+15	000000.000
2AF3	155.381+20	382.506E28	276.38E+15
3AC0	346.18D-03	485.322836	276.38E+15

Explanation:

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

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

Example 3: Assume that the following statements are given:

```
76 FORMAT    ('0',F6.2,E12.3,G14.6,I5)

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

and that the variables A, B, C, and N have the following values on successive executions of the WRITE statement:

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

Then, the following lines are printed by successive executions of the WRITE statement:

```
Print
Column:     1         9              21              35
            |         |              |               |
            v         v              v               v
             34.40     0.123E 05      12338.0         31

             31.10     0.116E 06     0.123457E 09     130

            ******     0.835E 00      1234.57         428

              1.13     0.831E 08     0.123380E 08      0
```

Explanation:

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

2. Note that for the variable B the decimal point is printed to the left of the first significant digit and that only three significant digits are printed because of the format specification E12.3. Excess digits are rounded off from the right.

3. The values of the variable C are printed according to the format specification G14.6. The s specification, which in this case is 6, determines the number of digits to be printed and whether the number should be printed with a decimal exponent. Values greater than or equal to 0.1 and less than 1000000 are printed without a decimal exponent in this example. Thus, the first and third values have no exponent. The second and fourth values are greater than 1000000, so they are printed with an exponent.

Scale Factor - P

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

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

Input: A scale factor 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 .xxxxxx and is to be written out in the form xx.xxxx, the format code used to effect this change is 2PF7.4. For real numbers written under the G format code, the effect of the scale factor is suspended unless the magnitude of the number (n) to be converted is outside the range (.1>n>10**s, where s is the number of significant digits specified in the G format code, Gw.s) that permits the effective use of the F format code.

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 238.47 were printed according to the format E10.3, it would appear as 0.238Eb03. If it were printed according to the format 1PE10.3, it would appear as 2.385Eb02.

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

Note: Once a scale factor has been established, it applies to all subsequently interpreted D, E, F, and G 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.

L Format Code

The L format code is used in transmitting logical variables.

Input: The input field must consist of optional blanks, followed by a T or F, followed by optional characters, for true and false respectively. The T or F causes a value of true or false to be assigned to the logical variable in the input list.

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

A Format Code

The A format code is used in transmitting data that is stored internally in character format. The number of characters transmitted under A format code depends on the length of the corresponding variable in the I/O list. Each alphabetic or special character is given a unique internal code. Numeric characters are transmitted without alteration; they are not converted into a form suitable for computation. Thus, the A format code can be used for numeric fields, but not for numeric fields requiring arithmetic.

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

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

Example 1: Assume that B has been specified as REAL*8, that N and M are INTEGER*4, and that the following statements are given:

```
25    FORMAT (3A7)

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

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

Input Card	B	N	M
ABCDEFG46bATb11234567	ABCDEFGb	ATb1	4567
HIJKLMN76543213334445	HIJKLMNb	4321	4445

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

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

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

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

A	B	C	Printed Line
A1B2	C3D4	E5F6G7H8	bbA1B2bC3D4E5F6G7

H Format Code and Literal Data

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

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

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

No item in the I/O list corresponds to the literal data. The data is read directly into or written directly 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, the number of characters read is the number of characters between the apostrophes.) For example, the following statements:

```
8     FORMAT (' HEADINGS')

      READ    (5,8)
```

cause the first nine characters of the next record to be read from the data set associated with data set reference number 5 into the FORMAT statement 8, replacing the blank and the 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 (6,8) AVRGE
```

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

```
MEAN AVERAGE: 12.3456
```

The first character of the output data record in this example is used for carriage control of printed output and does not appear in the printed line.

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

X Format Code

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

```
5   FORMAT (I10,10X,4I10)
    READ   (5,5) I,J,K,L,M
```

cause the first 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, for example, 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.)

The following illustrates the use of the T code.

```
 5   FORMAT (T40,'1968 STATISTICAL REPORT',T80,
X  'DECEMBER',T1,'0PART NO. 10095')
      WRITE  (6,5)
```

cause the following line to be printed:

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

The T format code can be used in a FORMAT statement with any type of format code, as, for example, 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    (5,15)  N,A,B,C,D,E
```

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

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

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

The first physical record, containing two data items, is transmitted according to the specification 2I3; the second, fourth, etc., records, each containing four data items, are transmitted according to the specification 3F6.2,F6.3; and the third, fifth, etc., records, each also containing four data items, are transmitted according to the specification D10.3,3D10.2, until the I/O list is exhausted.

Reading Format Specifications at Object Time

FORTRAN provides for variable FORMAT statements by allowing a format specification to be read into an array in storage. The data in the array may then be used as the format specification for subsequent input/output operations. The format specification may also be placed

into the array by a DATA statement or an explicit specification statement in the source program. The following rules are applicable:

1. The format specification must be in an array, even if the array size is only 1.

2. The format codes entered into the array must have the same form as a source program FORMAT statement, except that the word FORMAT and the statement number are omitted.

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

Example: Assume that the following statements are given:

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

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

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

END FILE STATEMENT

General Form
END FILE a Where: a is an unsigned integer constant or an integer variable (not an array element) that is of length 4 and represents a data set reference number.

The END FILE statement defines the end of the data set associated with a by causing an end-of-file record to be written.

REWIND STATEMENT

General Form
REWIND a Where: a is an unsigned integer constant or an integer variable (not an array element) that is of length 4 and represents a data set reference number.

The REWIND statement causes a subsequent READ or WRITE statement referring to a to read data from or write data into the first record of the data set associated with a.

BACKSPACE STATEMENT

General Form
BACKSPACE a Where: a is an unsigned integer constant or an integer variable (not an array element) that is of length 4 and represents a data set reference number.

The BACKSPACE statement causes the data set associated with a to backspace one record. If the data set associated with a is already at its beginning, execution of this statement has no effect. This statement may not be executed for direct-access or NAMELIST data sets. For further information, see the FORTRAN IV programmer's guide for the respective system.

DIRECT-ACCESS INPUT/OUTPUT STATEMENTS

The direct-access statements permit a programmer to read and write records randomly from any location within a data set. They contrast with the sequential input/output statements, described previously, that process records, one after the other, from the beginning of a data set to its end. With the direct-access statements, a programmer can go directly to any point in the data set, process a record, and go directly to any other point without having to process all the records in between.

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

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

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

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

DEFINE FILE STATEMENT

To use the direct-access READ, WRITE, and FIND statements in a program, the data set(s) to be operated on must be described with a DEFINE FILE statement. Each direct-access data set must be described once, in either the main program or a subprogram. The DEFINE FILE statement must logically precede any input/output statement referring to the data set being described.

The first DEFINE FILE statement encountered for a data set is the one used during program execution. Subsequent descriptions are ignored.

General Form

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

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

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

Each *r* is 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*.

Each *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, and that the maximum record size is measured in number of storage locations (bytes).

E indicates that the data set is to be read or written with format control (as specified by a FORMAT statement), and that 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, and that the maximum record size is measured in number of storage units (words).

Each *v* is an integer variable (not an array element) called an associated variable. At the conclusion of each read or write operation, *v* is set to a value that points to the record that immediately follows the last record transmitted. At the conclusion of a find operation, *v* is set to a value that points to the record found.

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

Example:

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

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

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

If an E is substituted for 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 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 referring to the data set. For example, if the I/O list of the WRITE statement specifies transmission of the contents of 100 storage locations, the DEFINE FILE statement can be either:

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

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

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

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

An associated variable may also be passed to a subprogram as an argument in a function reference or CALL statement. In the FUNCTION or SUBROUTINE statement, the corresponding dummy argument should be received by location (i.e., enclosed in slashes). See the section "Dummy Arguments in a FUNCTION or SUBROUTINE Subprogram."

If a READ, WRITE, or FIND statement that uses the associated variable occurs in a different program unit from the corresponding DEFINE FILE statement, the associated variable must be shared between the program units through either common or argument association.

READ STATEMENT

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

General Form

READ (a'r, b, ERR=c) list

Where: a is an unsigned integer constant or an integer variable (not an array element) that is of length 4 and represents a data set reference number; a must be followed by an apostrophe (').

r is an integer expression that represents the relative position of a record within the data set associated with a.

b is optional and, if given, is either the statement number of the FORMAT statement that describes the data being read or the name of an array that contains an object-time format specification.

ERR=c is optional and c is the number of a statement in the same program unit as the READ statement to which control is given when a device error condition is encountered during data transfer from device to storage.

list is optional and is an I/O list.

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

The relative record number of the first record of a direct-access data set is 1.

Example:

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

READ statement 9 transmits data from the data set associated with data set reference number 8, under control of FORMAT statement 10; transmission begins with record 16. Ten data items of 20 characters each are read as specified by the I/O list and FORMAT statement 10. Two records are read to satisfy the I/O list, because each record contains only five data items (100 characters). The associated variable ID1 is set to a value of 18 at the conclusion of the operation.

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

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

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

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

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

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

WRITE STATEMENT

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

General Form

WRITE (a'r,b) list

Where: a is an unsigned integer constant or an integer variable (not an array element) that is of length 4 and represents a data set reference number; a must be followed by an apostrophe (').

r is an integer expression that represents the relative position of a record within the data set associated with a.

b is optional and, if given, is either the statement number of the FORMAT statement that describes the data being written or the name of an array that contains an object-time format.

list is an I/O list. It is optional if b is specified.

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)
    8 WRITE (8'16,10) (M(K),K=1,10)
              .
              .
              .
   11 WRITE (9'ID2+5) A,B,C,D,E,F,G
```

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

WRITE statement 11 transmits data into the data set associated with data set reference number 9, without format control; transmission begins with record 26. The contents of 28 storage locations are written as specified by the I/O list for statement 11. The associated variable ID2 is set to a value of 27 at the conclusion of the operation. Note the

correspondence between the records described (28 storage locations per record) and the number of items called for by the I/O list (7 variables, type real, each occupying four storage locations).

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

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

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

FIND STATEMENT

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

General Form
FIND (a'r) Where: a is an unsigned integer constant or an integer variable (not an array element) that is of length 4 and represents a data set reference number; a must be followed by an apostrophe ('). r is an integer expression that represents the relative position of a record within the data set associated with a.

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

Example:

```
      DEFINE FILE 8(1000,80,L,IVAR)
   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. After the FIND statement is executed, the value of IVAR is 50. After the READ statement is executed, the value is 51.

General Example -- Direct Access Operations

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

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

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

General Form	
DATA	$k_1/d_1/,k_2/d_2/,\ldots,k_n/d_n/$
Where:	Each k is a list containing the names of variables, array elements (in which case the subscript quantities must be unsigned integer constants), or arrays. Dummy argument names may not appear in the list. Each d is a list of constants (integer, real, or complex, which may be optionally signed, or hexadecimal, logical, or literal). Any of these constants may be preceded by i*, where i is an unsigned integer constant. When the form i* appears before a constant, it indicates that the constant is to be specified i times.

A DATA initialization statement is not executable. It is used to define initial values of variables, array elements, and arrays. There must be a one-to-one correspondence between the total number of elements specified or implied by the list k and the total number of constants specified by the corresponding list d after application of any replication factors, i.

For real, integer, complex, and logical types, each constant must agree in type with the variable or array element it is initializing. Any type of variable or array element may be initialized with a literal or hexadecimal constant.

This statement cannot precede any specification statement that refers to the same variables, array elements, or arrays, but it can precede specification statements for other variables, array elements, or arrays. It also cannot precede an IMPLICIT statement. Otherwise, a DATA statement can appear anywhere in the program.

A storage entity may not be assigned an initial value more than once. For purposes of this constraint, entities that are associated with each other through COMMON or EQUIVALENCE statements are considered as the same entity.

Example 1:

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

Explanation:

The DATA statement indicates that the variables A, B, and C are to be initialized to the values 5.0, 6.1, and 7.3 respectively. In addition, the statement specifies that the first 25 elements of the array D are to be initialized to the value 1.0, the remaining 25 elements of D to the value 2.0, and the variable E to the value 5.1.

Example 2:

```
      DIMENSION A(5), B(3,3), L(4)
      LOGICAL L
      DATA A/5*1.0/, B/9*2.0/, L/4*.TRUE./, C/'FOUR'/
```

Explanation:

The DATA statement specifies that all the variables in the arrays A and B are to be initialized to the values 1.0 and 2.0, respectively. All the logical variables in the array L are initialized to the value true. The letters T and F may be used as an abbreviation for .TRUE. and .FALSE., respectively. In addition, the variable C is initialized with the literal data constant FOUR.

An initially defined variable or array element may not be in blank common. In a labeled common block, they may be initially defined only in a BLOCK DATA subprogram. (See the section "Subprograms.")

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.

The IMPLICIT statement, if used, must be the first specification statement. Specification statements must precede statement function definitions, which must precede the program part containing at least one executable statement. Explicit specification statements that initialize variables or arrays must follow other specification statements that contain the same variable or array names.

Information describing a variable or array in one specification statement should not be repeated in other specification statements that refer to the same variable or array.

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

DIMENSION STATEMENT

General Form
DIMENSION $a_1(k_1), a_2(k_2), a_3(k_3), \ldots, a_n(k_n)$
Where: Each a is an array name. Each k is composed of one through seven unsigned integer constants, separated by commas, that represent the maximum value of each subscript in the array. When the DIMENSION statement in which it appears is in a subprogram, each k may contain integer variables of length 4, provided that the array is a dummy argument.

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

Examples:

```
DIMENSION A(10), ARRAY(5,5,5), LIST(10,100)
DIMENSION B(25,50), TABLE(5,8,4)
```

TYPE STATEMENTS

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

The IMPLICIT statement enables the user to:

- Specify the type (including length) of all variables, arrays, and user-supplied functions whose names begin with a particular letter

The explicit specification statements enable the user to:

- Specify the type (including length) of a variable, array, or user-supplied function of a particular name
- Specify the dimensions of an array
- Assign initial data values for variables and arrays

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

IMPLICIT STATEMENT

General Form

IMPLICIT $type_1$*s_1(a_{11},a_{12},...),...,$type_n$*s_n(a_{n1},a_{n2},...)

Where: *type* is one of the following: INTEGER, REAL, COMPLEX, or LOGICAL.

Each *s* is optional and represents one of the permissible length specifications for its associated type.

Each *a* is a single alphabetic character or a range of characters drawn from the set A, B,...,Z, $, in that order. The range is denoted by the first and last characters of the range separated by a minus sign (e.g., (A-D)).

The IMPLICIT specification statement must be the first statement in a main program and the second statement in a subprogram. There can be only one IMPLICIT statement per program or subprogram. The IMPLICIT specification statement enables the user to declare the type of the variables appearing in his program (i.e., integer, real, complex, or logical) by specifying that variables beginning with certain designated letters are of a certain type. Furthermore, the IMPLICIT statement allows the programmer to declare the number of locations (bytes) to be allocated for each in the group of specified variables. The types that a variable may assume, along with the permissible length specifications, are as follows:

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

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

Note: The type DOUBLE PRECISION may not appear in an IMPLICIT statement. The type REAL*8 should be used.

Example 1:

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

Explanation:

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

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

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

Example 2:

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

Explanation:

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

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

Example 3:

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

Explanation:

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

General Form

Type*s a_1*s_1(k_1)/x_1/,a_2*s_2(k_2)/x_2/,...,a_n*s_n(k_n)/x_n/

Where: Type is INTEGER, REAL, LOGICAL, or COMPLEX.

Each s is optional and represents one of the permissible length specifications for its associated type.

Each a is a variable, array, or function name (see the section "Subprograms")

Each k is optional and gives dimension information for arrays. Each k is composed of one through seven unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array. When the type statement in which it appears is in a subprogram, each k may contain integer variables of length 4, provided that the array is a dummy argument.

Each /x/ is optional and represents initial data values. Dummy arguments may not be assigned initial values.

The explicit specification statements declare the type (INTEGER, REAL, COMPLEX, or LOGICAL) of a particular variable or array by its name, rather than by its initial character. This differs from the other ways of specifying the type of a variable or array (i.e., predefined convention and the IMPLICIT statement). In addition, the information necessary to allocate storage for arrays (dimension information) may be included within the statement.

Initial data values may be assigned to variables or arrays by use of /x_n/, where x_n is a constant or list of constants separated by commas. The x_n provides initialization only for the immediately preceding variable or array. The data must be of the same type as the variable or array, except that literal or hexadecimal data may also be used. Lists of constants are used only to assign initial values to array elements. Successive occurrences of the same constant can be represented by the form i*constant, as in the DATA statement. If initial data values are assigned to an array in an explicit specification statement, the dimension information for the array must be in the explicit specification statement or in a preceding DIMENSION or COMMON statement. An initial data value may not be assigned to a function name, but a function name may appear in an explicit specification statement.

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

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

Example 1:

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

Explanation:

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

Example 2:

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

Explanation:

This statement declares that the variables C, D, and E are of type complex. Since no length specification was explicitly given, the standard length is assumed. Thus, C and D each have eight storage locations reserved (four for the real part, four for the imaginary part) and D is initialized to the value (2.1,4.7). In addition, 16 storage locations are reserved for the variable E. Thus, if a length specification is explicitly written, it overrides the assumed standard length.

Example 3:

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

Explanation:

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

Example 4:

```
REAL A(5,5)/20*6.9E2,5*1.0/,B(100)/100*0.0/,TEST*8(5)/5*0.0D0/
```

Explanation:

This statement declares the size of each array, A and B, and their type (real). The array A has 100 storage locations reserved (four for each element in the array) and the array B has 400 storage locations reserved (four for each element). In addition, the first 20 elements in the array A are initialized to the value 6.9E2 and the last five elements are initialized to the value 1.0. All 100 elements in the array B are initialized to the value 0.0. The array TEST has 40 storage locations reserved (eight for each element). In addition, each element is initialized to the value 0.0D0.

DOUBLE PRECISION STATEMENT

General Form

DOUBLE PRECISION $a_1(k_1),a_2(k_2),a_3(k_3),\ldots,a_n(k_n)$

Where: Each *a* represents a variable, array, or function name (see the section "Subprograms").

Each *k* is optional and is composed of one through seven unsigned integer constants, separated by commas, that represent the maximum value of each subscript in the array. When the DOUBLE PRECISION statement in which it appears is in a subprogram, each *k* may contain integer variables of length 4, provided that the array is a dummy argument.

The DOUBLE PRECISION statement explicitly specifies that each of the variables *a* is of type double precision. This statement overrides any specification of a variable made by either the predefined convention or the IMPLICIT statement. The specification is identical to that of type REAL*8, but it cannot be used to define initial data values.

COMMON STATEMENT

General Form

COMMON $/r_1/a_{11}(k_{11}),a_{12}(k_{12}),\ldots/r_n/a_{n1}(k_{n1}),a_{n2}(k_{n2}),\ldots$

Where: Each *a* is a variable name or array name that is not a dummy argument.

Each *k* is optional and is composed of one through seven unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

Each *r* represents an optional common block name consisting of one through six alphameric characters, the first of which is alphabetic. These names must always be enclosed in slashes.

The form // (with no characters except possibly blanks between the slashes) may be used to denote blank common. If r_1 denotes blank common, the first two slashes are optional.

The COMMON statement is used to cause the sharing of storage by two or more program units, and to specify the names of variables and arrays that are to occupy this area. Storage sharing can be used for two purposes: to conserve storage, by avoiding more than one allocation of storage for variables and arrays used by several program units; and to implicitly transfer arguments between a calling program and a subprogram. Arguments passed in a common area are subject to the same rules with regard to type, length, etc., as arguments passed in an argument list (see the section "Subprograms").

A given common block name may appear more than once in a COMMON statement, or in more than one COMMON statement in a program unit. All

entries within such blocks are strung together in order of their appearance.

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

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

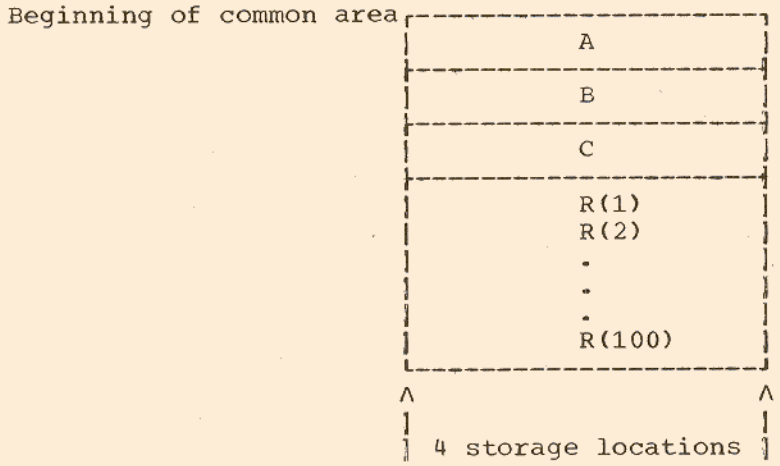
Since the entries in a common area share storage locations, the order in which they are entered is significant when the common area is used to transmit arguments. Consider the following example:

Example 1:

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

Explanation:

The statement COMMON A,B,C,R(100) in the calling program would cause 412 storage locations (four locations per variable) to be reserved in the following order:



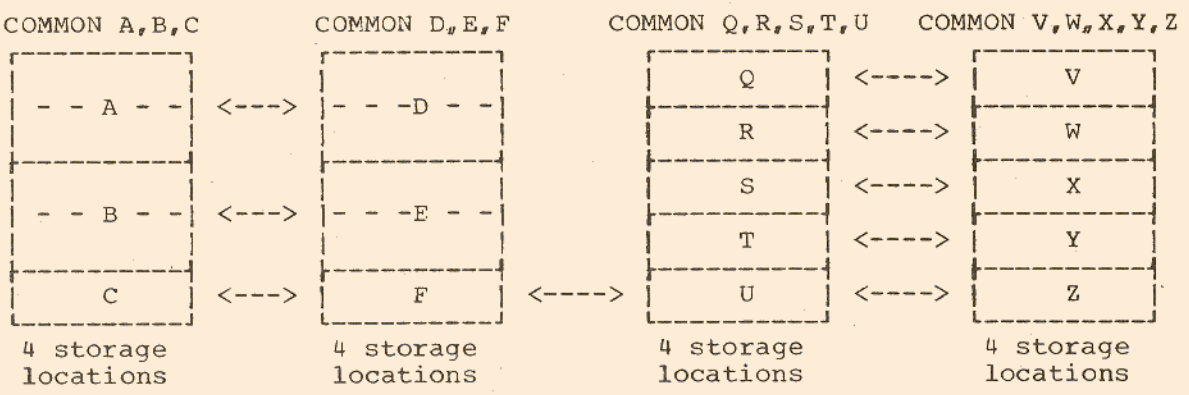
The statement COMMON X, Y, Z, S(100) in the subprogram would then cause the variables X, Y, Z, and S(1),...,S(100) to share the same storage space as A, B, C, and R(1),...,R(100), respectively. Note that values for X, Y, Z, and S(1),...,S(100), because they occupy the same storage locations as A, B,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

Example 2:

Assume a common area 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:



The main program can transmit values for A, B, and C to subprogram 1, provided that A is of the same type as D, B is of the same type as E, and C is of the same type as F. However, the main program and subprogram 1 cannot, by assigning values to the variables A and B, or D and E, respectively, transmit values to the variables Q, R, S, and T in subprogram 2, or V, W, X, and Y in subprogram 3, because the lengths of their common variables differ. Likewise, subprograms 2 and 3 cannot transmit values to variables A and B, or D and E.

Values can be transmitted between variables C, F, U, and Z, assuming that each is of the same type. With the same assumption, values can be transmitted between A and D, and B and E, and between Q and V, R and W, S and X, and T and Y. Note, however, that assignment of values to A or D destroys any values assigned to Q, R, V, and W, (and vice versa) and that assignment to B or E destroys the values of S, T, X, and Y (and vice versa).

BLANK AND LABELED COMMON

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

occupy the same storage space. This permits a calling program to share one common block with one subprogram and another common block with another subprogram, and also facilitates program documentation.

The differences between blank and labeled common are:

- There is only one blank common in an executable program, and it has no programmer-assigned name; there may be many labeled commons, each with its own name.
- Each program unit which uses a given labeled common must define it to be of the same length; blank common may have different lengths in different program units.
- Variables and array elements in blank common cannot be assigned initial values; variables and array elements in labeled common may be assigned initial values by DATA statements or explicit specification statements, but only in a BLOCK DATA subprogram.

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

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

In a COMMON statement, blank common is distinguished from labeled common by placing two consecutive slashes before the variables in blank common or, if the variables appear at the beginning of the COMMON statement, by omitting any block name. For example, in the following statement:

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

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

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

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

These two statements have the same effect as the single statement:

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

Example:

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

Calling Program

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

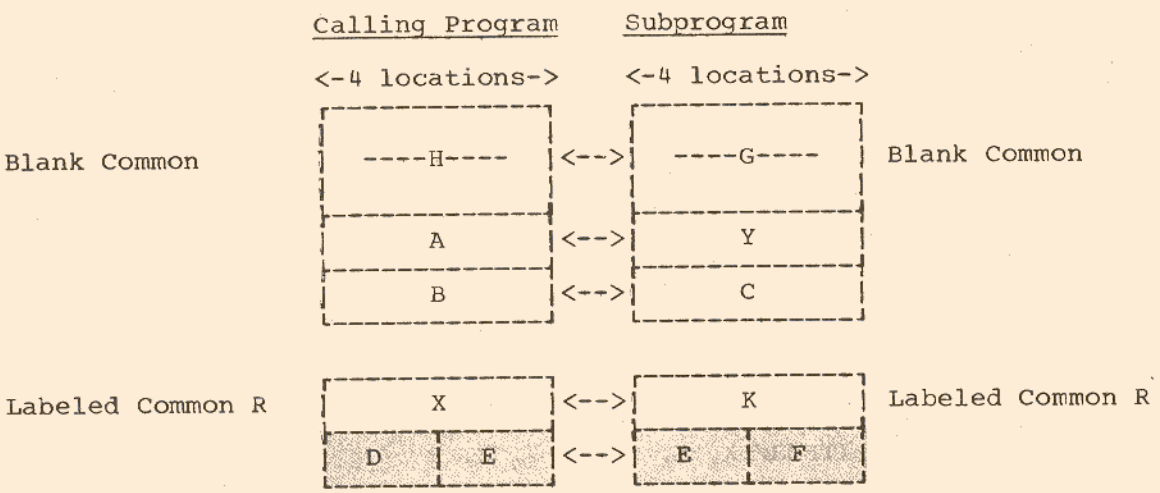
Subprogram

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

Explanation:

In the calling program, the statement COMMON H, A /R/ X, D, E //B causes 16 locations (four locations each for A and B, and eight for H) to be reserved in blank common, and eight locations in labeled common (four for X and two each for D and E).

The statement COMMON G,Y,C/R/K,E,F appearing in the subprogram MAPMY would then cause the variables G, Y, and C to share the same storage space (in blank common) as H, A, and B, respectively. It would also cause the variables K, E, and F to share the same storage space (in labeled common area R) as X, D, and E, respectively, as follows:



ARRANGEMENT OF VARIABLES IN COMMON

Variables in a common block need not be aligned properly. However, on some machines, the System/360 in particular, considerable object-time efficiency is lost unless the programmer ensures that all of the variables have proper boundary alignment. For System/370 machines, the loss in object-time efficiency is much less.

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

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

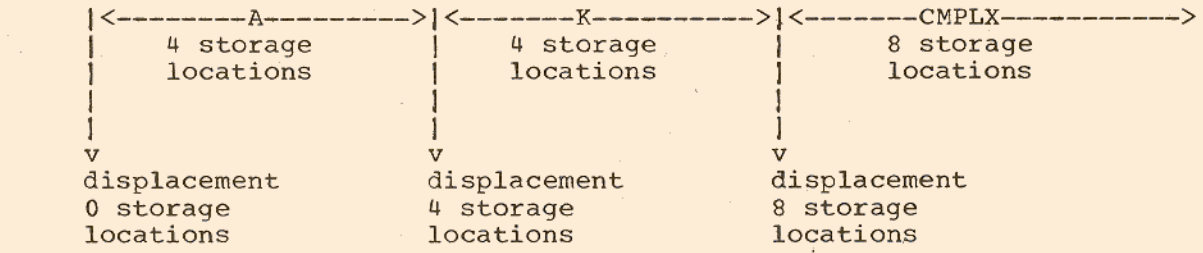
If the fixed order is not used, proper alignment can be ensured by constructing the block so that the displacement of each variable can be evenly divided by the reference number associated with the variable. (Displacement is the number of storage locations, or bytes, from the beginning of the block to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

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

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

```
COMMON A, K, CMPLX
```

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



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

General Form

EQUIVALENCE ($\underline{a}_{11}$,$\underline{a}_{12}$,$\underline{a}_{13}$,...),($\underline{a}_{21}$,$\underline{a}_{22}$,$\underline{a}_{23}$,...),...

Where: Each $\underline{a}$ is a variable or array element and may not be a dummy argument. The subscripts of array elements may have either of two forms:

If the array element has a single subscript quantity, it refers to the linear position of the element in the array (i.e., its position relative to the first element in the array: 3rd element, 17th element, 259th element).

If the array element is multi-subscripted (with the number of subscript quantities equal to the number of dimensions of the array), it refers to position in the same manner as in an arithmetic statement (i.e., its position relative to the first element of each dimension of the array). In either case, the subscripts themselves must be integer constants.

All the elements within a single set of parentheses share the same storage locations. The EQUIVALENCE statement provides the option for controlling the allocation of data storage within a single program unit. In particular, when the logic of the program permits it, the number of storage locations used can be reduced by causing locations to be shared by two or more variables of the same or different types. Equivalence between variables implies storage sharing. Mathematical equivalence of variables or array elements is implied only when they are of the same type, when they share exactly the same storage, and when the value assigned to the storage is of that type.

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.

Note that the EQUIVALENCE statement is the only statement in which a single subscript may be used to refer to an element (or elements) in a multi-dimensional array.

Two variables in one common block or in two different common blocks cannot be made equivalent. However, a variable in a program unit can be made equivalent to a variable in a common block. If the variable that is equivalenced to a variable in the common block is an element of an array, the implicit equivalencing of the rest of the elements of the array can extend the size of the common block (see Example 3, below). The size of the common block cannot be extended so that elements are added before the beginning of the established common block.

Example 1:

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

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

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

Example 2:

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

This EQUIVALENCE statement specifies that the variables A, B(1), and C(5,3) are assigned the same storage locations and that variables D(5,10,2) and E are assigned the same storage locations. It also implies that the array elements 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.

Example 3:

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

Explanation:

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

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

The following EQUIVALENCE statement is invalid:

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

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

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

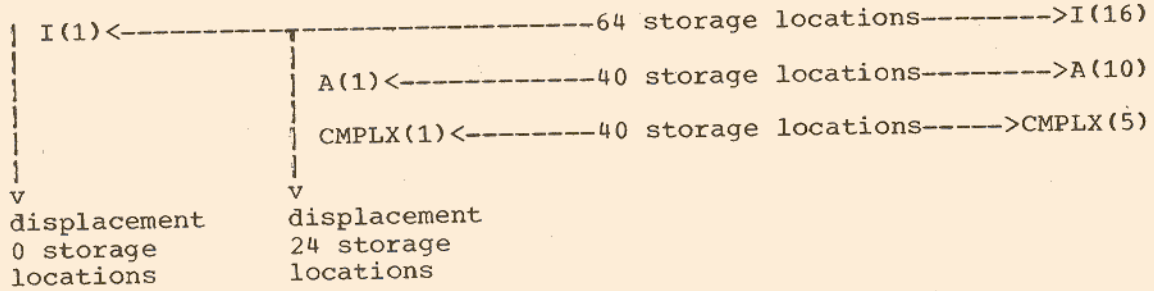
STORAGE ARRANGEMENT OF VARIABLES IN EQUIVALENCE GROUPS

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. 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. The displacement of a variable, say X, is the number of storage locations, or bytes, from the first byte of the first variable in the group to the first byte of variable X. The first variable in the group is the variable or array element with the lowest storage address. The reference numbers for each type of variable are given in the section "COMMON Statement." The first variable in each group is positioned as if its length specification were eight.

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

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

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



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

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

then CMPLX is not properly aligned (its displacement of 20 is not evenly divisible by its 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 statement EQUIVALENCE (A(1),I(7),CMPLX(1)) has exactly the same effect as EQUIVALENCE (CMPLX(1),A(1),I(7)), and in either case, I(1) is the first variable in the group even though it does not appear in the EQUIVALENCE statement.

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.

A subprogram must never refer to itself directly or indirectly or through any of its entry points.

Statement functions are also discussed in this section since they are similar to FUNCTION subprograms. The difference is that subprograms are not in the same program unit as the program unit referring to them, whereas 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 contain special characters (see Appendix A). The type of a function determines the type of the result that can be returned from it.

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

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

No type is associated with a SUBROUTINE name because the results that are returned to the calling program are dependent only on the type of the variable names appearing in the argument list of the calling program and/or the implicit arguments in COMMON.

FUNCTIONS

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

1. Define the function (i.e., specify which 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 name by which it can 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 subprograms, "Statement Functions" and "FUNCTION Subprograms."

Function Reference

When the name of a function, followed by a list of its arguments, appears in any FORTRAN expression, it refers to the function and causes the computations to be performed as indicated by the function definition. The resulting quantity (the function value) replaces the function reference in the expression and assumes the type of the function. The type of the name used for the reference must agree with the type of the name used in the definition.

STATEMENT FUNCTIONS

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

General Form
name(a_1,a_2,a_3,...,a_n) = *expression*
Where: *name* is the statement function name (see the section "Naming Subprograms").
Each *a* is a dummy argument. It must be a distinct variable (i.e., it may appear only once within the list of arguments). There must be at least one dummy argument.
expression is any arithmetic or logical expression that does not contain array elements. 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 a statement elsewhere in the program unit. The expression defining the function must not contain a reference to the function it is defining.

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

Example: The statement:

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

defines the statement function FUNC, where FUNC is the function name and A and B are the dummy arguments. The expression to the right of the equal sign defines the operations to be performed when the function reference appears in an arithmetic statement.

The function reference might appear in a statement as follows:

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

This is equivalent to:

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

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

Examples:

Valid statement function definitions and statement function references:

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

Invalid statement function definitions:

`SUBPRG(3,J,K)=3*I+J**3`	(Arguments must be variables)
`SOMEF(A(I),B)=A(I)/B+3.`	(Arguments must not be array elements)
`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)`	(Expression may not contain an array element)
`BAD(A,B)=A+B+BAD(C,D)`	(Recursive definition not permitted)
`NOGOOD(A,A)=A*A`	(Arguments are not distinct variable names)

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

```
WRONG = SUM(TAX,INSUR)        (Number of arguments does not agree
                                with above definition)
MIX = FUNC(I)                 (Type of argument does not agree with
                                above definition)
ALPHA = FUNC('DATA')          (Arguments must not be literals)
```

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*s (a_1,a_2,a_3,...,a_n)

Where: Type is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL. Its inclusion is optional.

name is the name of the FUNCTION.

s represents one of the permissible length specifications for its associated type. It may be included optionally only when Type is specified. It must not be used when DOUBLE PRECISION is specified.

Each a is a dummy argument. It must be a distinct variable or array name (i.e., it may appear only once within the statement) or dummy name of a SUBROUTINE or other FUNCTION subprogram. There must be at least one argument in the argument list. Any of the variable or array names may be enclosed in slashes.

A type declaration for a function name may be made by the predefined convention, by an IMPLICIT statement, by an explicit specification in the FUNCTION statement, or by an explicit specification statement within the FUNCTION subprogram. The function name must also be typed in the program units which refer to it if the predefined convention is not used.

Since the FUNCTION is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

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

The name of the function (or one of the ENTRY names; see the section, "Multiple Entries Into A Subprogram," below) must be assigned a value at least once in the subprogram -- as the variable name to the left of the equal sign in an arithmetic or logical assignment statement, as an argument of a CALL statement or an external function reference that is assigned a value by the function or subroutine referred to, or in the list of a READ statement within the subprogram.

The FUNCTION subprogram may also use one or more of its arguments to return values to the calling program. An argument so used will appear on the left side of an arithmetic or logical assignment statement, in the list of a READ statement within the subprogram, or as an argument in a CALL statement or function reference that is assigned a value by the subroutine or function referred to.

The dummy arguments of the FUNCTION subprogram (e.g., $a_1, a_2, a_3, \ldots, a_n$) may be considered to be dummy names. These are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. Additional information about arguments is in the section "Dummy Arguments in a FUNCTION or SUBROUTINE Subprogram."

When a RETURN statement in a FUNCTION subprogram is executed, all local storage entities become undefined, except those given initial values in a DATA statement or explicit specification statement and whose initial values were not changed.

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

Example 1:

Calling Program

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

FUNCTION Subprogram

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

Explanation:

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

Example 2:

Calling Program

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

FUNCTION Subprogram

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

Explanation:

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

RETURN and END Statements in a FUNCTION Subprogram

All FUNCTION subprograms must contain an END statement and at least one RETURN statement. The END statement specifies the physical end of the subprogram; the RETURN statement signifies a logical conclusion of the computation and returns the computed function 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+D**2
         .
         .
      RETURN
   30 DAV = D**2
         .
         .
         .
      RETURN
      END
```

Explanation:

If the result of (D-E) is negative or zero, the first RETURN statement will be executed. If the result is positive, the second RETURN will be executed.

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.

General Form

SUBROUTINE name (a_1,a_2,a_3,...,a_n)

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

Each a is a distinct dummy argument (i.e., it may appear only once within the statement). There need not be any arguments, in which case the parentheses must be omitted. Each argument used must be a variable or array name (optionally enclosed in slashes), the dummy name of another SUBROUTINE or FUNCTION subprogram, or an asterisk, where the character "*" denotes a return point specified by a statement number in the calling program. See the section "Dummy Arguments in a FUNCTION or SUBROUTINE Subprogram."

Since the SUBROUTINE is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

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

The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. An argument so used will appear on the left side of an arithmetic or logical assignment statement, in the list of a READ statement within the subprogram, or as an argument in a CALL statement or function reference that is assigned a value by the subroutine or function referred to. The subroutine name must not appear in any other statement in the SUBROUTINE subprogram.

The dummy arguments (a_1, a_2, a_3,...,a_n) may be considered dummy 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 "Dummy Arguments in a FUNCTION or SUBROUTINE Subprogram."

When a RETURN statement in a SUBROUTINE subprogram is executed, all local storage entities become undefined, except those given initial values in a DATA statement or explicit specification statement and whose initial values were not changed.

CALL Statement

The CALL statement is used to call a SUBROUTINE subprogram.

General Form

CALL name (a_1,a_2,a_3,....,a_n)

Where: name is the name of a SUBROUTINE subprogram or ENTRY point.

Each a is an actual argument that is being supplied to the SUBROUTINE subprogram. The argument may be a variable, array element, or array name, a literal, an arithmetic or logical expression or a function or subroutine name. Each may also be of the form &n where n is a statement number (see "RETURN Statements in a SUBROUTINE Subprogram").

Examples:

```
CALL OUT
CALL MATMPY (X,5,40,Y,7,2)
CALL QDRTIC (X,Y,Z,ROOT1,ROOT2)
CALL SUB1(X+Y*5,ABDF,SINE)
CALL SUB2(A,B,&10,&20,&30)
```

The CALL statement transfers control to the SUBROUTINE subprogram, and replaces the dummy variables with the value of the actual arguments that appear in the CALL statement.

Example:

Calling Program

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

SUBROUTINE Subprogram

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

Explanation:

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.

Subroutine COPY "copies" array A into array B within the subprogram. In this particular call, the subroutine arrays A and B are associated with the calling program arrays X and Y, respectively, and the variable N in the subroutine is associated with the value 100. Thus a call to subroutine COPY in this instance results in the 100 elements of array X being copied into the 100 elements of array Y.

RETURN Statements in a SUBROUTINE Subprogram

General Form
RETURN
RETURN i
Where: i is an integer constant or variable of length 4 whose value, say n, denotes the nth statement number in the argument list of a SUBROUTINE statement; i may be specified only in a SUBROUTINE subprogram.

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

Example:

```
      Calling Program                        Subprogram

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

Explanation:

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

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

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

is equivalent to:

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

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

DUMMY ARGUMENTS IN A FUNCTION OR SUBROUTINE SUBPROGRAM

The dummy arguments of a subprogram appear after the FUNCTION or SUBROUTINE name and are enclosed in parentheses. They are replaced at the time of execution by the actual arguments supplied in the function reference or CALL statement in the calling program. The dummy arguments must correspond in number, order, and type to the actual arguments. For example, if an actual argument is an integer constant, then the corresponding dummy argument must be an integer of length 4. 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:

- A literal, arithmetic or logical constant
- Any type of variable or array element
- Any type of array name
- Any type of arithmetic or logical expression
- The name of a FUNCTION or SUBROUTINE subprogram
- A statement number (for a SUBROUTINE subprogram only; see the section "RETURN Statements in a SUBROUTINE Subprogram")

If a literal is passed to a function or subroutine, the argument passed is the literal as defined, without delimiting apostrophes or the preceding wH specification. An actual argument which is the name of a subprogram must be identified by an EXTERNAL statement in the calling program unit containing that name. Hexadecimal constants cannot be actual arguments.

A dummy argument is an array when an appropriate DIMENSION or explicit specification statement appears in the subprogram. None of the dummy arguments may appear in an EQUIVALENCE, COMMON, DATA, or NAMELIST statement.

If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, an array element, or an array. A constant or expression should not be written 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 assign new values to dummy arguments which are 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 assigned new values by the function DERIV. Dummy arguments X and Z cannot be defined because they are both associated with the same argument, A; dummy argument Y, because it is associated with an argument, B, which is in COMMON; and the variable W, because it is also associated with B.

Enclosing a dummy argument in slashes (e.g., (/A/,/B/)) ensures that it will be received by location in the subprogram. With receipt by location, the subprogram reserves no storage for the dummy argument, using the corresponding actual argument in the calling program for its calculations. Thus the value of the actual argument changes as soon as the dummy argument changes. By contrast, with receipt by value, the dummy argument is assigned storage in the subprogram, and when the subprogram is entered, the value of the actual argument is brought in from the calling program. Only when the subprogram terminates is the final value transmitted back to the actual argument.

This is important in the case where an associated variable is passed as an argument to a subprogram and used there to denote a direct-access record to be processed. Unless it is received by location in the subprogram statement, the dummy associated variable will not automatically be updated after execution of a direct-access I/O statement. This is because the system recognizes only the first definition of an associated variable for each data set; in this case, the one in the calling program. It is this variable that is incremented, although it is the one in the subprogram I/O statement that is used to locate the record. Thus, within the subprogram, the same record will be continuously processed unless the "local" associated variable is incremented by FORTRAN statements.

Receipt by value versus receipt by location is also important when there are ENTRY statements in a subprogram. This is described in the following section.

MULTIPLE ENTRY INTO A SUBPROGRAM

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

It is also possible to enter a subprogram (either SUBROUTINE or FUNCTION) by a CALL statement or a function reference that references an ENTRY statement in the subprogram. Entry is made at the first executable statement following the ENTRY statement.

General Form
ENTRY name (a_1,a_2,a_3,...,a_n) Where: name is the name of an entry point (see the section "Naming Subprograms"). Each a is a dummy argument corresponding to an actual argument in a CALL statement or in a function reference. A dummy argument may be enclosed in slashes. See the section "Dummy Arguments in a FUNCTION or SUBROUTINE Subprogram."

An entry in a subroutine must be referred to by a CALL statement; an entry in a function must be referred to by a function reference.

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

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

Entry into a subprogram associates actual arguments with the dummy arguments of the referenced ENTRY statement. Thus, all appearances of these arguments in the whole subprogram become associated with actual arguments. Arguments that were received by value at some previous use of the subprogram need not appear in the argument list of the ENTRY statement. Arguments that were received by location must appear. In this case, the reference will not transmit new values for the arguments not listed. A function reference, and hence any ENTRY statement in a FUNCTION subprogram, must have at least one argument.

A dummy argument must not be used in any executable statement in the subprogram unless it has been previously defined as a dummy argument in an ENTRY, SUBROUTINE, or FUNCTION statement.

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

In a FUNCTION subprogram, the types of the function name and entry name are determined by the predefined convention, by an IMPLICIT statement, by an explicit type-statement, or by a type in the FUNCTION statement. The types of these variables (i.e., the function name and entry names) can be different; the variables are treated as if they were equivalenced. After one of these variables is assigned a value in the subprogram, any others of different type become indeterminate in value.

When there is an ENTRY statement in a function subprogram, either the function name or one of the entry names must be assigned a value.

Upon exit from a FUNCTION subprogram, the value returned is the value last assigned to the function name or any entry name. It is returned as

though it were assigned to the name in the current function reference. If the last value is assigned to a different entry name, and that entry name differs in type from the name in the current function reference, the value of the function is undefined.

Example 1:

```
    Calling Program                 Subprogram

          .                         FUNCTION FUNC(T,A,B,C)
          .                                .
          .                                .
    TABLE(1) = FUNC(W,X,Y,Z)               .
    DO 5 I=2,100                    ENTRY ENT(T)
    TABLE(I) = ENT(U)                      .
          .                                .
          .                                .
          .                         FUNC = A * B + C ** T
5   CONTINUE                        RETURN
          .                                .
          .                                .
          .                                .
                                    END
```

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

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

Example 2:

Calling Program

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

Subprogram

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

Explanation:

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

EXTERNAL STATEMENT

General Form
EXTERNAL $a_1, a_2, a_3, \ldots, a_n$ Where: Each a is a name of a subprogram that is passed as an argument to other subprograms.

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

If the name of a FORTRAN supplied in-line function is used in an EXTERNAL statement, the function is not expanded in-line when it appears as a function reference. Instead, it is assumed that the function is supplied by the user.

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

Example 1:

```
Calling Program                    Subprogram

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

Explanation:

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

Example 2:

```
Calling Program                    Subprogram

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

Explanation:

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

OBJECT-TIME DIMENSIONS

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

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

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

The name of an array with object-time dimensions cannot appear in a COMMON statement, although variables containing the dimensions may be placed in a common block.

Example 1:

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

Explanation:

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

Example 2:

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

Explanation:

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

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

BLOCK DATA SUBPROGRAMS

To initialize variables in a labeled (named) common block, a separate subprogram must be written. This separate subprogram contains only the BLOCK DATA, DATA, COMMON, DIMENSION, EQUIVALENCE, and explicit specification statements associated with the data being defined. This subprogram is not called; its presence suffices to provide initial data values for references in main and subprograms to labeled common blocks. Data may not be initialized in unlabeled common.

General Form
BLOCK DATA

1. The BLOCK DATA subprogram may not contain any executable statements, statement function definitions, or FORMAT, NAMELIST, DEFINE FILE, FUNCTION, SUBROUTINE, or ENTRY statements.

2. The BLOCK DATA statement must be the first statement in the subprogram. If an IMPLICIT statement is used in a BLOCK DATA subprogram, it must immediately follow the BLOCK DATA statement. Statements which provide initial values for data items cannot precede the COMMON statements which define those data items.

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

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

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

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

6. A particular common block may not be defined in more than one BLOCK DATA subprogram.

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

This appendix discusses those features of previously implemented FORTRAN languages that are incorporated into the System/360 and 370 FORTRAN IV language. The inclusion of these additional language facilities allows existing FORTRAN programs to be recompiled for use on the IBM System/360 or 370 with little or no reprogramming.

READ STATEMENT

General Form
READ b, list Where: b is the statement number of the FORMAT statement describing the data, or the name of an array containing a format specification. list is a series of variable or array names, separated by commas, which may be indexed and incremented. They specify the number of items to be read and the locations in storage into which the data is placed.

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

PUNCH STATEMENT

General Form
PUNCH b, list Where: b is the statement number of the FORMAT statement describing the data, or the name of an array containing a format specification. list is a series of variable or array names, separated by commas, which may be indexed and incremented. They specify the number of items to be written and the locations in storage from which the data is taken.

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

PRINT STATEMENT

General Form
PRINT b, list Where: b is the statement number of the FORMAT statement describing the data, or the name of an array containing a format specification. list is a series of variable or array names, separated by commas which may be indexed and incremented. They specify the number of items to be written and the locations in storage from which the data is taken.

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

APPENDIX C: FORTRAN-SUPPLIED PROCEDURES

The FORTRAN-supplied procedures are of two types: mathematical functions and service subroutines. An in-line function is inserted by the FORTRAN compiler at any point in the program where the function is referenced. An out-of-line function is located in a library, and the compiler generates an external reference to it. Table 4 shows mathematical functions, and Table 5 shows service subroutines. Detailed descriptions of out-of-line mathematical functions and service subroutines are given in the FORTRAN IV library publications.

Table 4. Mathematical Functions (Part 1 of 6)

General Function	Entry Name	Definition	Argument(s) No.	Argument(s) Type[5]	Argument(s) Length	Argument(s) Range	Function Value Returned Type[5]	Function Value Returned Length	Function Value Returned Range[3]	In-line (I) Out-of-line (O)[1] DOS	G,G1	C&G	H	H Ext
Natural and common logarithm	LOG†	$y=\log_e x$	1	Real	4	$x > 0$	Real	4	$-180.218 \leqq y \leqq 174.673$	O	O	O	O	O
	ALOG	or	1	Real	4		Real	4		O	O	O	O	O
	DLOG	$y=\ln x$	1	Real	8		Real	8						
	QLOG		1	Real	16		Real	16						O
	CLOG	$y=PV \log_e z$	1	Complex	8	$z \neq 0 + 0i$	Complex	8	$-180.218 \leqq y_1 \leqq 175.021$ $-\pi < y_2 \leqq \pi$	O	O	O	O	O
	CDLOG		1	Complex	16		Complex	16		O	O	O	O	O
	CQLOG	See Notes 2 and 4.	1	Complex	32		Complex	32						O
	LOG10†	$y=\log_{10} x$	1	Real	4	$x > 0$	Real	4	$-78.268 \leqq y \leqq 75.859$	O	O	O	O	O
	ALOG10		1	Real	4		Real	4		O	O	O	O	O
	DLOG10		1	Real	8		Real	8						
	QLOG10		1	Real	16		Real	16						O
Exponential	EXP	$y=e^x$	1	Real	4	$-180.218 \leqq x \leqq 174.673$	Real	4	$0 \leqq y \leqq \gamma$	O	O	O	O	O
	DEXP		1	Real	8		Real	8		O	O	O	O	O
	QEXP		1	Real	16		Real	16						O
	CEXP	$y=e^z$	1	Complex	8	$x_1 \leqq 174.673$ $\lvert x_2\rvert < (2^{18} \cdot \pi)$	Complex	8	$-\gamma \leqq y_1, y_2 \leqq \gamma$	O	O	O	O	O
	CDEXP	See Note 4.	1	Complex	16	$x_1 \leqq 174.673$ $\lvert x_2\rvert < (2^{50} \cdot \pi)$	Complex	16		O	O	O	O	O
	CQEXP		1	Complex	32	$x_1 \leqq 174.673$ $\lvert x_2\rvert < (2^{100})$	Complex	32						O
Square root	SQRT	$y=\sqrt{x}$ or	1	Real	4	$x \geqq 0$	Real	4	$0 \leqq y \leqq \gamma^{1/2}$	O	O	O	O	O
	DSQRT	$y=x^{1/2}$	1	Real	8		Real	8		O	O	O	O	O
	QSQRT		1	Real	16		Real	16						O
	CSQRT	$y=\sqrt{z}$ or	1	Complex	8	Any COMPLEX argument	Complex	8	$0 \leqq y_1 \leqq 1.0987 \cdot (\gamma^{1/2})$ $\lvert y_2\rvert \leqq 1.0987 \cdot (\gamma^{1/2})$	O	O	O	O	O
	CDSQRT	$y=z^{1/2}$	1	Complex	16		Complex	16		O	O	O	O	O
	CQSQRT	See Note 4.	1	Complex	32		Complex	32						O

Notes:

1. No entry = not provided.
2. PV = Principal Value. The answer given $(y_1 + y_2 i)$ is that one whose imaginary part (y_2) lies between $-\pi$ and π; more specifically, $-\pi < y_2 \leqq \pi$, unless $x_1 < 0$ and $x_2 = 0$, in which case $y_2 = -\pi$.
3. $\gamma = 16^{63} \cdot (1 - 16^{-6})$ for single precision, $16^{63} \cdot (1 - 16^{-14})$ for double precision, and $16^{63} \cdot (1 - 16^{-28})$ for extended precision routines.
4. z is a complex number of the form $x_1 + x_2 i$.
5. Type real of length 8 exists in ANS FORTRAN as type double precision.

† Alias. This name may be used in place of the REAL *4 function name when the H Extended compiler is used. See Appendix H.

General Function	Entry Name	Definition	Argument(s) No.	Argument(s) Type[a]	Argument(s) Length	Argument(s) Range	Function Value Returned Type[a]	Function Value Returned Length	Function Value Returned Range[a]	In-line (I) Out-of-line (O)[1] DOS	G,G1	C&G	H	H Ext
Arc sine and arc cosine	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	Real (in radians)	4	$-\frac{\pi}{2} \leqq y \leqq \frac{\pi}{2}$					O
	[illegible]		[illegible]	[illegible]	[illegible]		Real (in radians)	8		O	O	O	O	O
	[illegible]		[illegible]	[illegible]	[illegible]		Real (in radians)	8		O	O	O	O	O
	[illegible]		[illegible]	[illegible]	[illegible]		Real (in radians)	16						O
	ACOS[1]	[illegible]	[illegible]	[illegible]	[illegible]		Real (in radians)	4	$0 \leqq y \leqq \pi$					O
	ARCOS		1	Real	4		Real (in radians)	4		O	O	O	O	O
	DARCOS		1	Real	8		Real (in radians)	8		O	O	O	O	O
	QARCOS		1	Real	16		Real (in radians)	16						O
Arc tangent	ATAN	$y=\arctan x$	1	Real	4	Any REAL Argument	Real (in radians)	4	$-\frac{\pi}{2} \leqq y \leqq \frac{\pi}{2}$	O	O	O	O	O
	DATAN		1	Real	8		Real (in radians)	8		O	O	O	O	O
	QATAN		1	Real	16		Real (in radians)	16						O
	ATAN2	$y=\arctan \frac{x_1}{x_2}$	2	Real	4	Any REAL arguments except (0, 0)	Real (in radians)	4	$-\pi < y \leqq \pi$	O	O	O	O	O
	DATAN2		2	Real	8		Real (in radians)	8		O	O	O	O	O
	QATAN2		2	Real	16		Real (in radians)	16						O
Sine and cosine	SIN	$y=\sin x$	1	Real (in radians)	4	$\lvert x \rvert < (2^{18} \cdot \pi)$	Real	4	$-1 \leqq y \leqq 1$	O	O	O	O	O
	DSIN		1	Real (in radians)	8	$\lvert x \rvert < (2^{50} \cdot \pi)$	Real	8		O	O	O	O	O
	QSIN		1	Real (in radians)	16	$\lvert x \rvert < (2^{100})$	Real	16						O
	COS	$y=\cos x$	1	Real (in radians)	4	$\lvert x \rvert < (2^{18} \cdot \pi)$	Real	4	$-1 \leqq y \leqq 1$	O	O	O	O	O
	DCOS		1	Real (in radians)	8	$\lvert x \rvert < (2^{50} \cdot \pi)$	Real	8		O	O	O	O	O
	QCOS		1	Real (in radians)	16	$\lvert x \rvert < (2^{100})$	Real	16						O

Notes:

1. No entry = not provided.
2. $\gamma = 16^{63} \cdot (1 - 16^{-8})$ for single precision, $16^{63} \cdot (1 - 16^{-14})$ for double precision, and $16^{63} \cdot (1 - 16^{-28})$ for extended precision routines.
3. Type real of length 8 exists in ANS FORTRAN as type double precision.

† Alias. This name may be used in place of the REAL * 4 function name when the H Extended compiler is used. See Appendix H.

General Function	Entry Name	Definition	Argument(s)				Function Value Returned			In-line (I) Out-of-line (O)[1]				
			No.	Type[6]	Length	Range	Type[6]	Length	Range[5]	DOS	G,G1	C&G	H	H Ext
Sine and cosine (continued)	CSIN	$y = \sin z$ See Note 4.	1	Complex (in radians)	8	$\lvert x_1 \rvert < (2^{18} \cdot \pi)$ $\lvert x_2 \rvert \leqq 174.673$	Complex	8	$-\gamma \leqq y_1, y_2 \leqq \gamma$	O	O	O	O	O
	CDSIN		1	Complex (in radians)	16	$\lvert x_1 \rvert < (2^{50} \cdot \pi)$ $\lvert x_2 \rvert \leqq 174.673$	Complex	16		O	O	O	O	O
	CQSIN		1	Complex (in radians)	32	$\lvert x_1 \rvert < (2^{100})$ $\lvert x_2 \rvert \leqq 174.673$	Complex	32						O
	CCOS	$y = \cos z$ See Note 4.	1	Complex (in radians)	8	$\lvert x_1 \rvert < (2^{18} \cdot \pi)$ $\lvert x_2 \rvert \leqq 174.673$	Complex	8	$-\gamma \leqq y_1, y_2 \leqq \gamma$	O	O	O	O	O
	CDCOS		1	Complex (in radians)	16	$\lvert x_1 \rvert < (2^{50} \cdot \pi)$ $\lvert x_2 \rvert \leqq 174.673$	Complex	16		O	O	O	O	O
	CQCOS		1	Complex (in radians)	32	$\lvert x_1 \rvert < (2^{100})$ $\lvert x_2 \rvert \leqq 174.673$	Complex	32						O
Tangent and cotangent	TAN	$y = \tan x$	1	Real (in radians)	4	$\lvert x \rvert < (2^{18} \cdot \pi)$ See Note 2.	Real	4	$-\gamma \leqq y \leqq \gamma$	O	O	O	O	O
	DTAN		1	Real (in radians)	8	$\lvert x \rvert < (2^{50} \cdot \pi)$ See Note 2.	Real	8		O	O	O	O	O
	QTAN		1	Real (in radians)	16	$\lvert x \rvert < (2^{100})$ See Note 2.	Real	16						O
	COTAN	$y = \text{cotan } x$	1	Real (in radians)	4	$\lvert x \rvert < (2^{18} \cdot \pi)$ See Note 3.	Real	4	$-\gamma \leqq y \leqq \gamma$	O	O	O	O	O
	DCOTAN		1	Real (in radians)	8	$\lvert x \rvert < (2^{50} \cdot \pi)$ See Note 3.	Real	8		O	O	O	O	O
	QCOTAN		1	Real (in radians)	16	$\lvert x \rvert < (2^{100})$ See Note 3.	Real	16						O
Hyperbolic sine and cosine	SINH	$y = \frac{e^x - e^{-x}}{2}$	1	Real	4	$\lvert x \rvert < 175.366$	Real	4	$-\gamma \leqq y \leqq \gamma$	O	O	O	O	O
	DSINH		1	Real	8		Real	8		O	O	O	O	O
	QSINH		1	Real	16		Real	16						O
	COSH	$y = \frac{e^x + e^{-x}}{2}$	1	Real	4		Real	4	$1 \leqq y \leqq \gamma$	O	O	O	O	O
	DCOSH		1	Real	8		Real	8		O	O	O	O	O
	QCOSH		1	Real	16		Real	16						O
Hyperbolic tangent	TANH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1	Real	4	Any REAL argument	Real	4	$-1 \leqq y \leqq 1$	O	O	O	O	O
	DTANH		1	Real	8		Real	8		O	O	O	O	O
	QTANH		1	Real	16		Real	16						O

Notes:

1. No entry = not provided.
2. The argument of the tangent functions may not approach an odd multiple of $\pi/2$.
3. The argument of the cotangent functions may not approach a multiple of π.
4. z is a complex number of the form $x_1 + x_2 i$.
5. $\gamma = 16^{63} \cdot (1 - 16^{-6})$ for single precision, $16^{63} \cdot (1 - 16^{-14})$ for double precision, and $16^{63} \cdot (1 - 16^{-28})$ for extended precision routines.
6. Type real of length 8 exists in ANS FORTRAN as type double precision.

General Function	Entry Name	Definition	Argument(s)				Function Value Returned			In-line (I) Out-of-line (O)[1]				
			No.	Type[4]	Length	Range	Type[4]	Length	Range[2]	DOS	G,G1	C&G	H	H Ext
Absolute value	IABS*	$y=\|x\|$	1	Integer	4	Any INTEGER argument	Integer	4		I	I	I	I	I
	ABS*		1	Real	4	Any	Real	4		I	I	I	I	I
	DABS*		1	Real	8	REAL	Real	8		I	I	I	I	I
	QABS		1	Real	16	argument	Real	16						I
	CABS	$y=\|z\|=(x_1^2+x_2^2)^{1/2}$	1	Complex	8	Any COMPLEX	Real	4	$0 \leqq y_1 \leqq \gamma$	O	O	O	O	O
	CDABS		1	Complex	16	argument	Real	8	$y_2 = 0$	O	O	O	O	O
	CQABS		1	Complex	32	See Note 3.	Real	16						O
Error function	ERF	$y=\frac{2}{\sqrt{\pi}}\int_0^x e^{-u^2}\,du$	1	Real	4	Any REAL argument	Real	4	$-1 \leqq y \leqq 1$	O	O	O	O	O
	DERF		1	Real	8		Real	8		O	O	O	O	O
	QERF		1	Real	16		Real	16						O
	ERFC	$y=\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-u^2}\,du$ $y=1-\operatorname{erf} x$	1	Real	4		Real	4	$0 \leqq y \leqq 2$	O	O	O	O	O
	DERFC		1	Real	8		Real	8		O	O	O	O	O
	QERFC		1	Real	16		Real	16						O
Maximum and minimum values	MAX1	$y=\max(x_1, \ldots, x_n)$	$\geqq 2$	Integer	4	Any INTEGER argument	Integer	4						I
	MAX0*		$\geqq 2$	Integer	4		Integer	4		O	O	O	I	I
	AMAX0*		$\geqq 2$	Integer	4		Real	4		O	O	O	I	I
	MAX1*		$\geqq 2$	Real	4	Any	Integer	4		O	O	O	I	I
	AMAX1*		$\geqq 2$	Real	4	REAL	Real	4		O	O	O	I	I
	DMAX1*		$\geqq 2$	Real	8	argument	Real	8		O	O	O	I	I
	QMAX1		$\geqq 2$	Real	16		Real	16						I
	MIN1	$y=\min(x_1, \ldots, x_n)$	$\geqq 2$	Integer	4	Any INTEGER	Integer	4						I
	MIN0*		$\geqq 2$	Integer	4	argument	Integer	4		O	O	O	I	I
	AMIN0*		$\geqq 2$	Integer	4		Real	4		O	O	O	I	I
	MIN1*		$\geqq 2$	Real	4	Any	Integer	4		O	O	O	I	I
	AMIN1*		$\geqq 2$	Real	4	REAL	Real	4		O	O	O	I	I
	DMIN1*		$\geqq 2$	Real	8	argument	Real	8		O	O	O	I	I
	QMIN1		$\geqq 2$	Real	16		Real	16						I
Truncation	AINT*	$y=$ (sign of x) $\cdot$ n where n is the largest integer $\leqq \|x\|$	1	Real	4	Any REAL argument	Real	4		I	I	I	I	I
	DINT		1	Real	8		Real	8						I
	QINT		1	Real	16		Real	16						I
	INT*		1	Real	4		Integer	4		I	I	I	I	I
	IDINT*		1	Real	8		Integer	4		I	I	I	I	I
	HFIX		1	Real	16		Integer	4						I

Notes:

1. No entry = not provided.
2. $\gamma = 16^{63} \cdot (1 - 16^{-6})$ for single precision, $16^{63} \cdot (1 - 16^{-14})$ for double precision, and $16^{63} \cdot (1 - 16^{-28})$ for extended precision routines.
3. Floating-point overflow can occur.
4. Type real of length 8 exists in ANS FORTRAN as type double precision.

5. [illegible] may be used in place of the INTEGER*4 function name when the H Extended compiler is used. See Appendix H.

* ANS FORTRAN Intrinsic Function.

General Function	Entry Name	Definition	Argument(s)				Function Value Returned			In-line (I) Out-of-line (O)[1]				
			No.	Type[4]	Length	Range	Type[4]	Length	Range[5]	DOS	G,G1	C&G	H	H Ext
Gamma and log-gamma	GAMMA	$y=\int_0^\infty u^{x-1}e^{-u}\,du$	1	Real	4	$2^{-252} < x < 57.5744$	Real	4	$0.88560 \leqq y \leqq \gamma$	O	O	O	O	O
	DGAMMA		1	Real	8		Real	8		O	O	O	O	O
	LGAMMA†	$y=\log_e\Gamma(x)$ or $y=\log_e\int_0^\infty u^{x-1}e^{-u}\,du$	1	Real	4	$0 < x < 4.2913 \cdot 10^{73}$	Real	4	$-0.12149 \leqq y \leqq \gamma$					O
	ALGAMA		1	Real	4		Real	4		O	O	O	O	O
	DLGAMA		1	Real	8		Real	8		O	O	O	O	O
Modulo arithmetic	MOD*	y=remainder $\left(\frac{x_1}{x_2}\right)$, i.e., $y=x_1$ (modulo x_2) See Note 2.	2	Integer	4	$x_2 \neq 0$ See Note 3.	Integer	4		I	I	I	I	I
	AMOD*		2	Real	4		Real	4		I	I	I	I	I
	DMOD		2	Real	8		Real	8		I	I	I	I	I
	QMOD		2	Real	16		Real	16						I
Conversion from INTEGER to REAL	FLOAT*		1	Integer	4	Any INTEGER argument	Real	4		I	I	I	I	I
	DFLOAT		1	Integer	4		Real	8		I	I	I	I	I
	QFLOAT		1	Integer	4		Real	16						I
Conversion from REAL to INTEGER	IFIX*	y=(sign of x) · n where n is the largest integer $\leqq \lvert x \rvert$	1	Real	4	Any REAL argument	Integer	4		I	I	I	I	I
	HFIX		1	Real	4		Integer	2		I	I	I	I	I
Transfer of sign	ISIGN*	$y=(\text{sign of } x_2) \cdot \lvert x_1 \rvert$	2	Integer	4	$x_2 \neq 0$ See Note 3.	Integer	4		I	I	I	I	I
	SIGN*		2	Real	4		Real	4		I	I	I	I	I
	DSIGN*		2	Real	8		Real	8		I	I	I	I	I
	QSIGN		2	Real	16		Real	16						I
Positive difference	IDIM*	$y=x_1-\min(x_1, x_2)$	2	Integer	4	Any INTEGER argument	Integer	4		I	I	I	I	I
	DIM*		2	Real	4	Any REAL argument	Real	4		I	I	I	I	I
	DDIM		2	Real	8		Real	8						I
	QDIM		2	Real	16		Real	16						I
Obtain most significant part of a REAL argument	SNGL*		1	Real	8	Any REAL argument	Real	4		I	I	I	I	I
	SNGLQ		1	Real	16		Real	4						I
	DBLEQ		1	Real	16		Real	8						I

Notes:

1. No entry = not provided.
2. x_1 (modulo x_2) is defined as $x_1 - \left[\frac{x_1}{x_2}\right] \cdot x_2$, where the brackets indicate that the largest integer whose magnitude does not exceed the magnitude of $\frac{x_1}{x_2}$ is used. The sign of the integer is the same as the sign of $\frac{x_1}{x_2}$.
3. If $x_2 = 0$, then the modulus and transfer-of-sign functions are mathematically undefined.
4. Type real of length 8 exists in ANS FORTRAN as type double precision.
5. $\gamma = 16^{63} \cdot (1 - 16^{-6})$ for single precision, $16^{63} \cdot (1 - 16^{-14})$ for double precision, and $16^{63} \cdot (1 - 16^{-28})$ for extended precision routines.

* ANS FORTRAN Intrinsic Function.

† Alias. This name may be used in place of the six (6) character function name when the H Extended Compiler is used. See Appendix H.

General Function	Entry Name	Definition	Argument(s)				Function Value Returned			In-line (I) Out-of-line (O)[1]				
			No.	Type[2]	Length	Range	Type[2]	Length	Range	DOS	G,G1	C&G	H	H Ext
Obtain real part of a COMPLEX argument	REAL*		1	Complex	8	Any COMPLEX argument	Real	4		I	I	I	I	I
	DREAL		1	Complex	16		Real	8						I
	QREAL		1	Complex	32		Real	16						I
Obtain imaginary part of a COMPLEX argument	IMAG†		1	Complex	8	Any COMPLEX argument	Real	4						I
	AIMAG*		1	Complex	8		Real	4		I	I	I	I	I
	DIMAG		1	Complex	16		Real	8						I
	QIMAG		1	Complex	32		Real	16						I
Precision increase	DBLE*		1	Real	4	Any REAL argument	Real	8		I	I	I	I	I
	QEXT		1	Real	4		Real	16						I
	QEXTD		1	Real	8		Real	16						I
Express two REAL arguments in COMPLEX form	CMPLX*	$y=x_1+x_2i$	2	Real	4	Any REAL argument	Complex	8		I	I	I	I	I
	DCMPLX		2	Real	8		Complex	16		I	I	I	I	I
	QCMPLX		2	Real	16		Complex	32						I
Obtain conjugate of a COMPLEX argument	CONJG*	$y=x_1-x_2i$ for $\arg=x_1+x_2i$	1	Complex	8	Any COMPLEX argument	Complex	8		I	I	I	I	I
	DCONJG		1	Complex	16		Complex	16		I	I	I	I	I
	QCONJG		1	Complex	32		Complex	32						I

Notes:

1. No entry = not provided.
2. Type real of length 8 exists in ANS FORTRAN as type double precision.

* ANS FORTRAN Intrinsic Function.

† Alias. This name may be used in place of the COMPLEX *8 function name when the H Extended compiler is used. See Appendix H.

Table 5. Service Subroutines

Purpose	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 (After the test, the sense light that was tested is turned off.)	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 1 - LOGICAL*1 2 - LOGICAL*4 3 - INTEGER*2 4 - INTEGER*4 5 - REAL*4 6 - REAL*8 7 - COMPLEX*8 8 - COMPLEX*16 9 - literal 10 - REAL*16 11 - COMPLEX*32
Dump storage on the output data set and continue execution	CALL PDUMP (a_1,b_1,f_1, ...,a_n,b_n,f_n)	a, b, and f are as defined above for DUMP.
Test for divide check exception	CALL DVCHK(j)	j is an integer variable that is set to 1 if the divide-check indicator was on, or to 2 if the indicator was off. After testing, the divide-check indicator is turned off.
Test for exponent overflow or underflow	CALL OVERFL(j)	j is an integer variable that is set to 1 if an exponent overflow condition was the last to occur, to 2 if no overflow condition exists, or to 3 if an exponent underflow condition was the last to occur. After testing, the overflow indicator is turned off.
Terminate execution	CALL EXIT	None

SAMPLE PROGRAM 1

The sample program shown in 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, 7 is a prime number. The number 9 is not prime, since it can be divided evenly by 3.

IBM FORTRAN Coding Form GX28-7327-6 U/M050 Printed in U.S.A.

PROGRAM: SAMPLE PROGRAM 1 | PUNCHING INSTRUCTIONS | GRAPHIC | PUNCH | PAGE 1 OF 1
PROGRAMMER | DATE 10/70 | CARD ELECTRO NUMBER*

COMMENT	STATEMENT NUMBER	CONT	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
C			PRIME NUMBER GENERATOR	
			WRITE (6,1)	
	1		FORMAT ('1FOLLOWING IS A LIST OF PRIME NUMBERS FROM 2 TO 1000'/	
		*	19X,'2'/19X,'3'/19X,'5'/19X,'7')	
			DO 4 I=11,1000,2	
			K=SQRT(FLOAT(I))	
			DO 2 J=3,K,2	
			IF (MOD(I,J) .EQ. 0) GO TO 4	
	2		CONTINUE	
			WRITE (6,3) I	
	3		FORMAT (I20)	
	4		CONTINUE	
			WRITE (6,5)	
	5		FORMAT (' THIS IS THE END OF THE LIST')	
			STOP	
			END	

* A standard card form, IBM electro 859157, is available for punching statements from this form

Figure 1. Sample Program 1

SAMPLE PROGRAM 2

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

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

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

$$\begin{aligned}
&(1) \quad w_0a_0 + w_1a_1 + \cdots + w_ma_m = z \\
&(2) \quad w_1a_0 + w_2a_1 + \cdots + w_{m+1}a_m = z_1 \\
&\quad \vdots \\
&(m+1) \quad w_ma_0 + w_{m+1}a_1 + \cdots + w_{2m}a_m = z_m
\end{aligned}$$

where:

$$w_0 = n \qquad z_0 = \sum_{i=1}^{n} y_i$$

$$w_1 = \sum_{i=1}^{n} x_i \qquad z_1 = \sum_{i=1}^{n} y_i x_i$$

$$w_2 = \sum_{i=1}^{n} x_i^2 \qquad z_2 = \sum_{i=1}^{n} y_i x_i^2$$

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

$$w_{2m} = \sum_{i=1}^{n} x_i^{2m} \qquad z_m = \sum_{i=1}^{n} y_i x_i^m$$

After the w's and z's have been computed, the normal equations are solved by the method of elimination which is illustrated by the following solution of the normal equations for a second-degree polynomial (m = 2).

$$(1) \quad w_0a_0 + w_1a_1 + w_2a_2 = z_0$$

$$(2) \quad w_1a_0 + w_2a_1 + w_3a_2 = z_1$$

$$(3) \quad w_2a_0 + w_3a_1 + w_4a_2 = z_2$$

The forward solution is as follows:

1. Divide equation (1) by w_0.
2. Multiply the equation resulting from step 1 by w_1 and subtract from equation (2).
3. Multiply the equation resulting from step 1 by w_2 and subtract from equation (3).

The resulting equations are:

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

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

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

where:

$$b_{12} = w_1/w_0, \quad b_{13} = w_2/w_0, \quad b_{14} = z_0/w_0$$

$$b_{22} = w_2 - b_{12}w_1, \quad b_{23} = w_3 - b_{13}w_1, \quad b_{24} = z_1 - b_{14}w_1$$

$$b_{32} = w_3 - b_{12}w_2, \quad b_{33} = w_4 - b_{13}w_2, \quad b_{34} = z_2 - b_{14}w_2$$

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:

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

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

where:

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

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

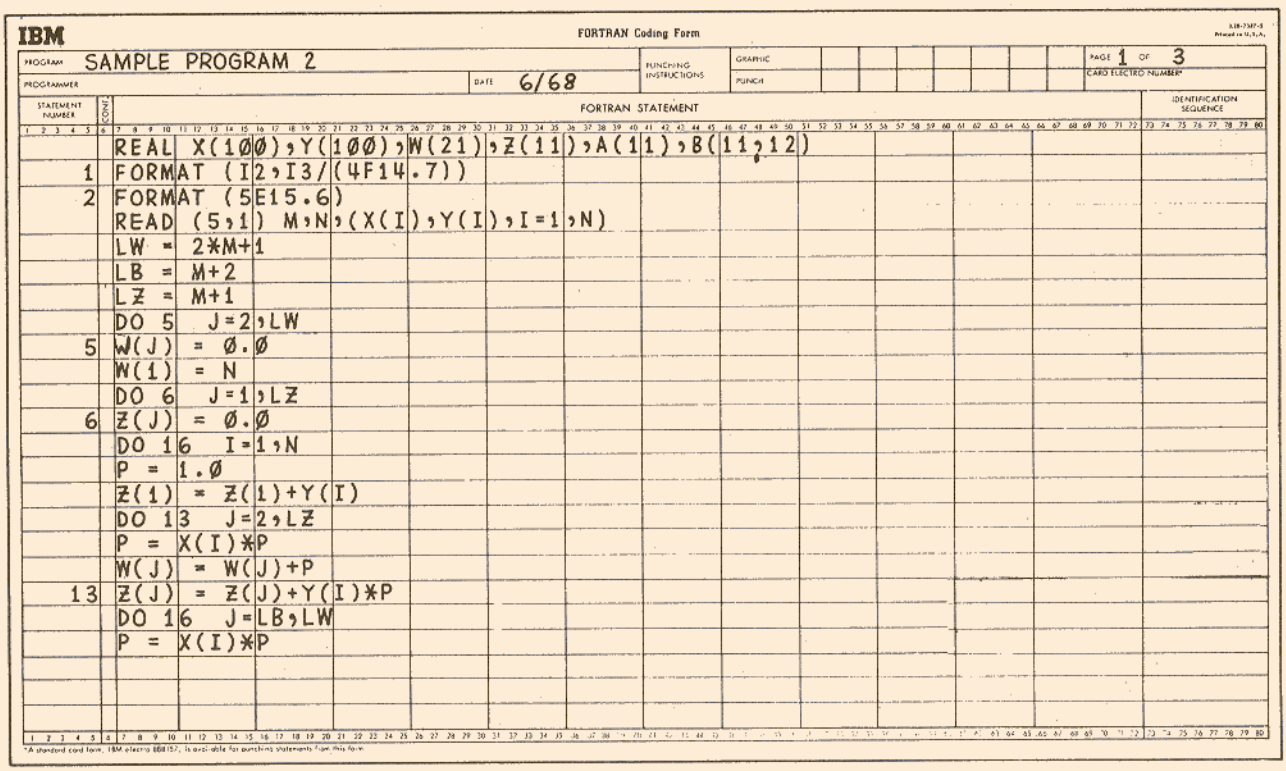
The backward solution is as follows:

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

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

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

Figure 2 is a possible FORTRAN program for carrying out the calculations for the case: $n = 100$, $m \leq 10$. $w_0, w_1, w_2, \ldots, w_{2m}$ are stored in W(1), W(2), W(3),..., W(2M+1), respectively. $z_0, z_1, z_2, \ldots, z_m$ are stored in Z(1), Z(2), Z(3),..., Z(M+1), respectively.



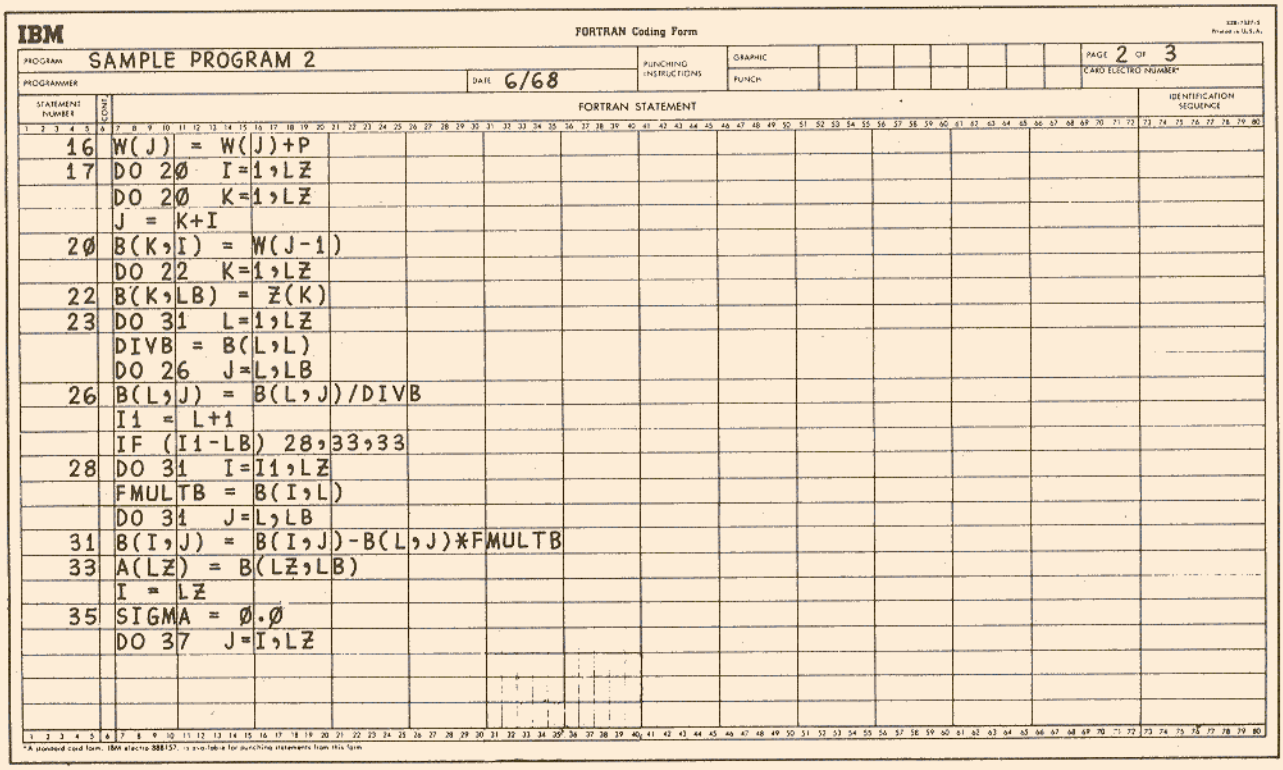
IBM — FORTRAN Coding Form

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

STATEMENT NUMBER	CONT.	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
		REAL X(1ØØ),Y(1ØØ),W(21),Z(11),A(11),B(11,12)	
1		FORMAT (I2,I3/(4F14.7))	
2		FORMAT (5E15.6)	
		READ (5,1) M,N,(X(I),Y(I),I=1,N)	
		LW = 2*M+1	
		LB = M+2	
		LZ = M+1	
		DO 5 J=2,LW	
5		W(J) = Ø.Ø	
		W(1) = N	
		DO 6 J=1,LZ	
6		Z(J) = Ø.Ø	
		DO 16 I=1,N	
		P = 1.Ø	
		Z(1) = Z(1)+Y(I)	
		DO 13 J=2,LZ	
		P = X(I)*P	
		W(J) = W(J)+P	
13		Z(J) = Z(J)+Y(I)*P	
		DO 16 J=LB,LW	
		P = X(I)*P	

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

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



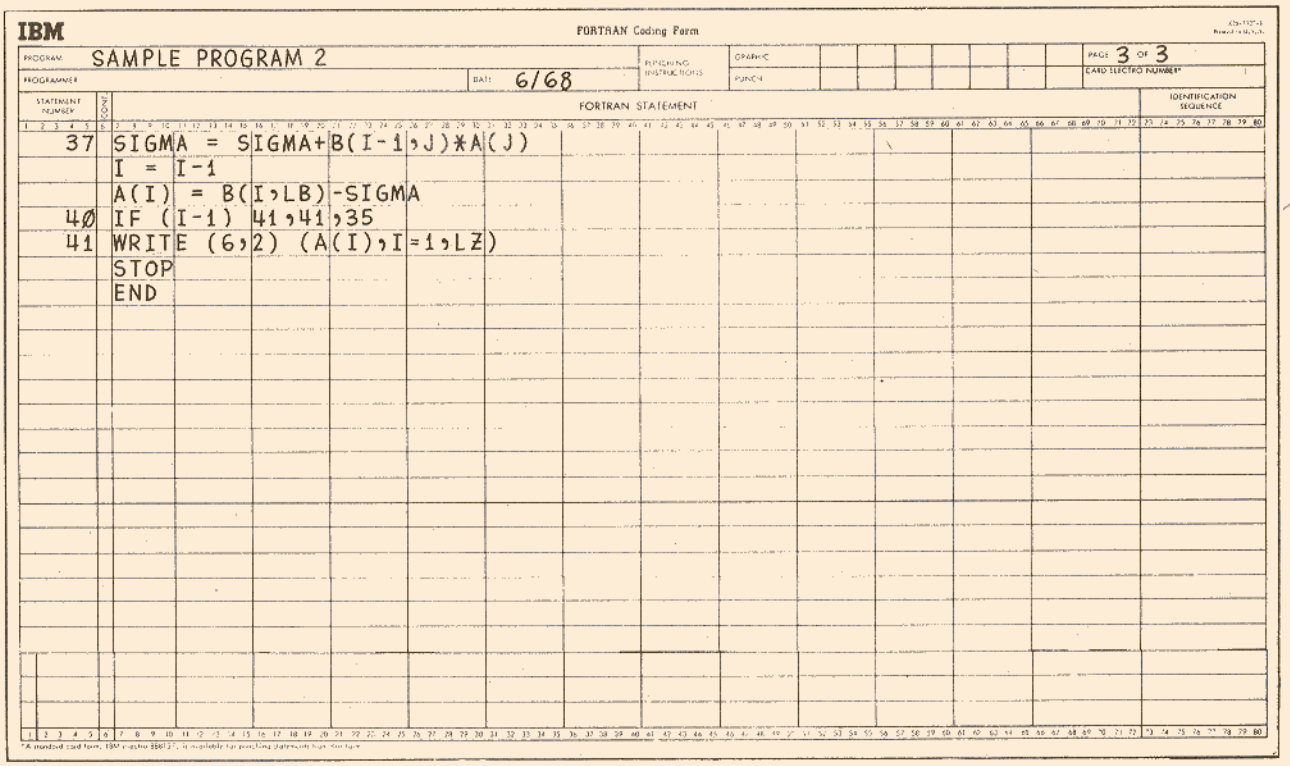
IBM — FORTRAN Coding Form

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

STATEMENT NUMBER	CONT.	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
16		W(J) = W(J)+P	
17		DO 2Ø I=1,LZ	
		DO 2Ø K=1,LZ	
		J = K+I	
2Ø		B(K,I) = W(J-1)	
		DO 22 K=1,LZ	
22		B(K,LB) = Z(K)	
23		DO 31 L=1,LZ	
		DIVB = B(L,L)	
		DO 26 J=L,LB	
26		B(L,J) = B(L,J)/DIVB	
		I1 = L+1	
		IF (I1-LB) 28,33,33	
28		DO 31 I=I1,LZ	
		FMULTB = B(I,L)	
		DO 31 J=L,LB	
31		B(I,J) = B(I,J)-B(L,J)*FMULTB	
33		A(LZ) = B(LZ,LB)	
		I = LZ	
35		SIGMA = Ø.Ø	
		DO 37 J=I,LZ	

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

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



IBM — FORTRAN Coding Form

PROGRAM: SAMPLE PROGRAM 2 — PUNCHING INSTRUCTIONS: GRAPHIC, PUNCH — PAGE 3 OF 3

PROGRAMMER — DATE: 6/68 — CARD ELECTRO NUMBER*

STATEMENT NUMBER	CONT.	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
37		SIGMA = SIGMA+B(I-1,J)*A(J)	
		I = I-1	
		A(I) = B(I,LB)-SIGMA	
40		IF (I-1) 41,41,35	
41		WRITE (6,2) (A(I),I=1,LZ)	
		STOP	
		END	

*A standard card form, IBM Electro 88812, is available for punching statements from coding form.

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

The elements of the W array, except W(1), are set equal to zero. W(1) is set equal to N. For each value of I, 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:

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

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

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

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

$$Z(M+1) = \sum_{I=1}^{N} Y(I)X(I)^m$$

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

By the time control has passed to statement 23, the values of w_0, $w_1,\ldots,w_{2m}$ have been placed in the storage locations corresponding to columns 1 through M+1, rows 1 through M+1, of the B array, and the values of $z_0,z_1,\ldots,z_m$ have been stored in the locations corresponding to the column M+2 of the B array. For example, for the illustrative problem (M = 2), columns 1 through 4, rows 1 through 3, of the B array would be set to the following computed values:

w_0	w_1	w_2	z_0
w_1	w_2	w_3	z_1
w_2	w_3	w_4	z_2

This matrix represents equations (1), (2), and (3), the normal equations for M = 2.

The forward solution, which results in equations (4), (7), and (8) in the illustrative problem, is carried out by statements 23 through 31. By the time control has passed to statement 33, the coefficients of the A(I) terms in the M+1 equations which would be obtained in hand calculations have replaced the contents of the locations corresponding to columns 1 through M+1, rows 1 through M+1, of the B array, and the constants on the right-hand side of the equations have replaced the contents of the locations corresponding to column M+2, rows 1 through M+1, of the B array. For the illustrative problem, columns 1 through 4, rows 1 through 3, of the B array would be set to the following computed values:

1	b_{12}	b_{13}	b_{14}
0	1	c_{23}	c_{24}
0	0	c_{33}	c_{34}

This matrix represents equations (4), (7), and (8).

The backward solution, which results in equations (9), (10), and (11) in the illustrative problem, is carried out by statements 33 through 40. By the time control has passed to statement 41, which prints the values of the A(I) terms, the values of the M+1 A(I) terms have been stored in the M+1 locations of 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: DEBUG FACILITY

The debug facility is a programming aid that enables the user to locate errors in a FORTRAN source program. It is available under Operating System/360 only with the FORTRAN IV (G), FORTRAN IV (G1), and Code and Go FORTRAN compilers, and under Disk Operating System/360 with the DOS FORTRAN IV compiler. The debug facility provides for tracing the flow within a program, tracing the flow between programs, displaying the values of variables and arrays, and checking the validity of subscripts.

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

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

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

PROGRAMMING CONSIDERATIONS

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

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

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

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

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

   SUBROUTINE
   FUNCTION
   ENTRY
   IMPLICIT
   BLOCK DATA
   statement function definition

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

DEBUG FACILITY STATEMENTS

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

DEBUG SPECIFICATION STATEMENT

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

General Form

DEBUG option,...,option

Where: option may be any of the following:

UNIT (a)
: where a is an integer constant that represents a data set reference number. All debugging output is placed in this data set, called the debug output data set. If this option is not specified, any debugging output is placed in the standard output data set. All unit definitions within an executable program must refer to the same unit.

SUBCHK (n_1, n_2,...,n_n)
: where n is an array name. The validity of the subscripts used with the named arrays is checked by comparing the subscript combination with the size of the array. If the subscript exceeds its dimension bounds, a message is placed in the debug output data set. Program execution continues, using the incorrect subscript. If the list of array names is omitted, all arrays in the program are checked for valid subscript usage. If the entire option is omitted, no arrays are checked for valid subscripts.

TRACE
: This option must be in the DEBUG specification statement of each program or subprogram for which tracing is desired. If this option is omitted, there can be no display of program flow by statement number within this program. Even when this option is used, a TRACE ON statement must appear in the first debug packet in which tracing is desired.

INIT (m_1, m_2,...,m_n)
: where m is the name of a variable or an array that is to be displayed in the debug output data set only when the variable or the array values change. If m is a variable name, the name and value are displayed whenever the variable is assigned a new value in either an assignment, a READ, or an assigned GO TO statement. If m is an array name, the changed element is displayed. If the list of names is omitted, a display occurs whenever the value of a variable or an array element is changed. If the entire option is omitted, no display occurs when values change.

SUBTRACE
: This option specifies that the name of this subprogram is to be displayed whenever it is entered. The message RETURN is to be displayed whenever execution of the subprogram is completed.

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

AT DEBUG PACKET IDENTIFICATION STATEMENT

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

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

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

TRACE ON STATEMENT

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

General Form
TRACE ON

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

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

TRACE OFF STATEMENT

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

General Form
TRACE OFF

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

DISPLAY STATEMENT

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

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

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

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

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

where:

name is the same in both statements.

Note that array elements and dummy arguments may not appear in the list.

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

DEBUG PACKET PROGRAMMING EXAMPLES

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

Example 1:

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

Explanation:

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

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

Example 2:

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

Explanation:

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

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

Example 3:

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

```
35  CONTINUE
    TRACE ON
C   DEBUG PACKET NUMBER 3
    AT 30
    TRACE OFF
    END
```

Explanation:

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

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

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

ASSIGN
Assigned GO TO
Asterisk as dummy argument in SUBROUTINE
BLOCK DATA
Call by name
COMPLEX
Complex, logical, literal, and hexadecimal constants
DATA
Debug facility
ENTRY
ERR and END parameters in a READ
G, Z, and L format codes
Generalized subscript form
IMPLICIT
Initial data values in explicit specification statements
Length of variables and arrays as part of type specifications
Labeled COMMON
Literal as actual argument in CALL and function reference
LOGICAL
Logical IF
More than three dimensions in an array
NAMELIST
Object-time dimensions
Object-time format specifications
PAUSE with literal
PRINT b, list
PUNCH b, list
READ b, list
RETURN i (i not a blank)
Statement number as actual argument in CALL

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

AIMAG	DCONJG
AINT	HFIX
CMPLX	IDINT
CONJG	INT
DCMPLX	REAL

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

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

APPENDIX G: IBM FORTRAN IV FEATURES NOT IN ANS FORTRAN

Asterisk as dummy argument in SUBROUTINE
Data types COMPLEX*16, INTEGER*2, and LOGICAL*1
Direct access input/output statements
Dummy arguments enclosed in slashes
ENTRY
ERR and END parameters in a READ
Function subprogram name in an explicit specification statement within the FUNCTION subprogram
Generalized subscripts
Hexadecimal constant
IMPLICIT
Initial data values in explicit specification statements
Length of variables and arrays as part of type specifications
Length specification in FUNCTION statement
Literal as actual argument in function reference
Literal enclosed in apostrophes
Mixed-mode expressions
More than three dimensions in an array
Multiple exponentiation without parentheses to indicate order of computation
NAMELIST
Object-time dimensions transmitted in COMMON
PAUSE 'message'
PRINT b, list
PUNCH b, list
READ b, list
RETURN i
Statement number as actual argument in CALL
T and Z format codes, and extension to G format code

APPENDIX H: FORTRAN IV (H EXTENDED) FEATURES

This appendix describes four extensions to the IBM System/360 and 370 FORTRAN IV language available with the FORTRAN IV (H Extended) compiler. The extensions are:

- Asynchronous Input/Output Statements, providing asynchronous reading and writing. The feature allows the high-speed transmission of unformatted data between external data sets and arrays in main storage. (See the section "Asynchronous Input/Output Statements.")

- Extended Precision, allowing the new length specifications REAL*16 and COMPLEX*32. The built-in functions and library subroutines required to support these data types are also provided. Extended precision permits the programmer to satisfy increased precision requirements. (See the section "Extended Precision.")

- EXTERNAL Statement Extension, providing a means of declaring a user-supplied subprogram to be executed in place of an IBM FORTRAN library function or subroutine of the same name. This is achieved by the use of an ampersand character (&) prefixed to the function name in its appearance in the EXTERNAL statement. (See the section "EXTERNAL Statement Extension.")

- Automatic Function Selection, allowing the use of a generic name in the FORTRAN program in place of certain specific built-in and library function names. Use of the generic name with arguments of a permissible data type results in the appropriate function's being selected by the compiler, based on the data type of the arguments. This facility is provided by a new statement, the GENERIC statement. (See the section "Automatic Function Selection.")

An additional extension, the list-directed READ and WRITE statements, is described in Appendix I.

ASYNCHRONOUS INPUT/OUTPUT STATEMENTS

The asynchronous READ and WRITE statements, and the WAIT statement, provide high-speed input/output. They transmit unformatted sequential data between direct access or sequential storage devices and arrays in main storage. (The asynchronous forms of the READ and WRITE statements, and the WAIT statement, may not be used with data sets associated with unit-record devices -- card reader, card punch, or printer.) The READ and WRITE statements are asynchronous in that while data transfer is taking place, other program statements may be executed. (The WAIT statement is not asynchronous; i.e., while it is being executed, no other FORTRAN statements can be executed.)

The asynchronous READ and WRITE statements initiate a transmission. The WAIT statement, which must be executed for each asynchronous READ and WRITE, terminates the transmission cycle. When executed after an asynchronous READ, the WAIT statement enables the program to refer to the transmitted data. This process ensures that a program will not refer to a data field while transmission to it is still in progress. When the WAIT is executed after an asynchronous WRITE, new values may be assigned to the transmitting area.

The asynchronous READ and WRITE statements differ in form from standard and direct access READ and WRITE statements in that a special parameter, ID=n, is specified within the parentheses of the statement. The ID establishes a unique identification for the READ or WRITE statement.

The END FILE, REWIND, and BACKSPACE statements may be used in conjunction with the asynchronous READ and WRITE statements, provided that any input/output operation on the data set has been completed by the execution of a WAIT. The syntax and function of these statements are the same as when they are used with the standard sequential READ and WRITE statements. A WAIT statement is not required to complete their operation.

Synchronous READ and WRITE statements may be executed for the data set only after all asynchronous READ and WRITE operations have been completed and a REWIND has been executed for the data set. Conversely, asynchronous READ and WRITE statements may be executed for a data set previously read or written synchronously after a REWIND has been executed.

ASYNCHRONOUS READ STATEMENT

The asynchronous READ statement is used to transmit data from an external data set to an array in main storage.

General Form

READ (a,ID=n) list

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

n is an integer constant or integer expression which is of length 4 and is the identifier for the READ statement.

list, which is optional, is an asynchronous-I/O list, which may have any of four forms:

e
$e_1 \ldots e_2$
$e_1 \ldots$
$\ldots e_2$

Where: e is the name of an array.

e_1 and e_2 are the names of elements in the same array.

Note that the ellipsis (...) is an integral part of the syntax of the list and must appear in the positions indicated.

The forms ERR=c and END=d are not permitted in an asynchronous READ statement.

The data set specified by a must reside on a sequential or direct-access device. The array (e) or array elements (e_1 through e_2) constitute the receiving area for the data to be read.

Execution of an asynchronous READ statement initiates reading of the next record on the specified data set. The record may contain more or

less data than there are bytes in the receiving area. If there is more, the excess data is not transmitted to the receiving area; if there is less, the values of the excess array elements remain unaltered. The extent of the receiving area is determined as follows:

- If e is specified, the entire array is the receiving area.
- If $e_1...e_2$ is specified, the receiving area begins at array element e_1 and includes every element up to and including e_2. The subscript value of e_1 must not exceed that of e_2.
- If $e_1...$ is specified, the receiving area begins at element e_1 and includes every element up to and including the last element of the array.
- If $...e_2$ is specified, the receiving area begins at the first element of the array and includes every element up to and including e_2.

If no list is specified, there is no receiving area, and no data is transmitted. Execution of such an asynchronous READ statement causes a record to be skipped.

Subscripts in the list of the asynchronous READ may not refer to array elements in the receiving area. If a function reference is used in a subscript, the function reference may not cause I/O to be performed on any data set.

Given an array with elements of length p, transmission begins with the first p bytes of the record being placed in the first specified (or implied) array element. Each successive p bytes of the record are placed in the array element with the next highest subscript value. Transmission ceases after all elements of the receiving area have been filled, or the entire record has been transmitted -- whichever occurs first. If the record length is less than the receiving area size, the last array element to receive data may receive fewer than p bytes.

The specified array may be multi-dimensional. Array elements are filled sequentially. Thus, during transmission, the leftmost subscript quantity increases most rapidly, and the rightmost least rapidly. (See the section "Arrangement of Arrays in Storage" elsewhere in this publication.)

Since the asynchronous READ statement reads records sequentially from the data set, a DEFINE FILE statement should not be specified, even though the data set may reside on a direct-access device.

There is no FORMAT statement associated with the input data, and no conversion takes place. Data is assumed to be of the type and length indicated by the array.

Any number of program statements may be executed between an asynchronous READ and its corresponding WAIT, subject to the following rules:

1. No array element in the receiving area may appear in any such statement. This and the following rules apply also to indirect references to such array elements; i.e., reference to or redefinition of any variable or array element associated by COMMON or EQUIVALENCE statements, or argument association with an array element in the receiving area.
2. No executable statement may appear which redefines or undefines a variable or array element appearing in the subscript of e_1 or e_2.

3. The above restriction notwithstanding, an array element in the receiving area, or a variable associated with such an element, may be written as an actual argument in a CALL statement or function reference, provided that the corresponding dummy argument is not an object-time dimension and that the referenced subroutine or function receives it by location (i.e., the dummy argument must be enclosed in slashes).

4. If a function reference appears in the subscript expression of e_1 or e_2, the function may not be referred to by any statements executed between the asynchronous READ and the corresponding WAIT. Also, no subroutines or functions may be referenced which directly or indirectly refer to the subscript function, or to which the subscript function directly or indirectly refers.

5. No function or subroutine may be executed which performs input or output on the data set being manipulated, or which contains object-time dimensions that are in the receiving area (whether they be dummy arguments or in a common block).

ASYNCHRONOUS WRITE STATEMENT

The asynchronous WRITE statement is used to transmit data from an array in main storage to an external data set.

General Form

WRITE (a,ID=n) list

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

n is an integer constant or integer expression which is of length 4 and is the identifier for the WRITE statement.

list is an asynchronous-I/O list, which may have any of four forms:

e
e_1...e_2
e_1...
...e_2

Where: e is the name of an array.
e_1 and e_2 are names of elements in the same array.

Note that the ellipsis (...) is an integral part of the syntax of the list and must appear in the positions indicated.

The data set specified by a must reside on a sequential or direct access device. The array or array elements specified by e (or e_1 and e_2) constitute the transmitting area for the data to be written. The extent of the transmitting area is determined as follows:

- If e is specified, the entire array is the transmitting area.
- If e_1...e_2 is specified, the transmitting area begins at array element e_1 and includes every element up to and including e_2. The subscript value of e_1 must not exceed that of e_2.

- If e_1... is specified, the transmitting area begins at element e_1 and includes every element up to and including the last element of the array.

- If ...e_2 is specified, the transmitting area begins at the first element of the array and includes every element up to and including e_2.

If a function reference is used in a subscript of the list, the function reference may not cause I/O to be performed on any data set.

Execution of an asynchronous WRITE statement initiates writing of the next record on the specified data set. The size of the record is equal to the size of the transmitting area. All the data in the area is written.

Given an array with elements of length p, the number of bytes transmitted will be p times the number of elements in the array. Elements are transmitted sequentially from the smallest-subscript element to the highest. If the array is multidimensional, the leftmost subscript quantity increases most rapidly, and the rightmost least rapidly. (See the section "Arrangement of Arrays in Storage" elsewhere in this publication.)

Since the asynchronous WRITE statement writes records sequentially, a DEFINE FILE statement should not be specified even though the data set may be resident on a direct-access device.

There is no FORMAT statement associated with the output data, and no conversion takes place.

Any number of program statements may be executed between an asynchronous WRITE and its corresponding WAIT, subject to the following rules:

1. No such statement may in any way assign a new value to any array element in the transmitting field. This and the following rules apply also to indirect references to such array elements; i.e., assigning a new value to a variable or array element associated by COMMON or EQUIVALENCE statements, or argument association with an array element in the receiving area.

2. No executable statement may appear which redefines or undefines a variable or array element appearing in the subscript of e_1 or e_2.

3. If a function reference appears in the subscript expression of e_1 or e_2, the function may not be referred to by any statements executed between the asynchronous WRITE and the corresponding WAIT. Also, no subroutines or functions may be referenced which directly or indirectly refer to the subscript function, or to which the subscript function directly or indirectly refers.

4. No function or subroutine may be executed which performs input or output on the data set being manipulated.

WAIT STATEMENT

Execution of a WAIT statement completes the data transmission begun by the corresponding asynchronous READ or WRITE statement. The WAIT redefines a receiving area and makes it available for reference, or makes a transmitting area available for redefinition.

The corresponding asynchronous READ or WRITE, which need not appear in the same program unit as the WAIT, is that statement which:

1. Was not completed by the execution of another WAIT
2. Refers to the same data set as the WAIT
3. Contains the same value for n in the ID=n form as did the asynchronous READ or WRITE when it was executed

The correspondence between a WAIT and an asynchronous READ or WRITE holds for a particular execution of the statements. Different executions may establish different correspondences.

General Form
WAIT (a,p) list Where: a is an unsigned integer constant or integer variable that is of length 4 and represents a data set reference number. p is a parameter list which contains (in any order) one or more of the following forms: ID=n where n is an integer constant or integer expression which is of length 4. COND=i_1 where i_1 is an integer variable name which is of length 4. This form is optional. NUM=i_2 where i_2 is an integer variable name which is of length 4. This form is optional. list, which is optional, is an asynchronous-I/O list as specified for the asynchronous READ and WRITE statements.

If a list is included, it must specify the same receiving or transmitting area as the corresponding asynchronous READ or WRITE statement. It must not be specified if the asynchronous READ did not specify a list.

When the WAIT is completing an asynchronous READ, the subscripts in the list may not refer to array elements in the receiving area. If a function reference is used in a subscript, the function reference may not cause I/O to be performed on any data set.

If COND=i_1 is specified, the variable i_1 is assigned a value of 1 if the input or output operation was completed successfully; 2 if an error condition was encountered; and 3 if an end-of-file condition was encountered while reading. In case of an error or end-of-file condition, the data in the receiving area may be meaningless.

If NUM=i_2 is specified, the variable i_2 is assigned a value representing the number of bytes of data transmitted to the elements specified by the list. If the list requires more data from the record than the record contains, this parameter must be specified. If the WAIT is completing an asynchronous WRITE, i_2 remains unaltered.

If the WAIT is completing an asynchronous READ, the expression n (in the ID=n parameter) is subject to the following rules:

1. No array element in the receiving area of the read may appear in the expression. This also includes indirect references to such

elements; i.e., reference to or redefinition of any variable or array element associated by COMMON or EQUIVALENCE statements, or argument association with an array element in the receiving area.

2. If a function reference appears in the subscript expression of e_1 or e_2, the function may not be referred to in the expression n. Also, no functions or subroutines may be referred to by the expression which directly or indirectly refer to the subscript function, or to which the subscript function directly or indirectly refers.

Example 1:

```
      .
      .
      .
   READ (8,ID=1) ARRAY(101)...ARRAY(500)
      .
      .
      .
   WAIT (8,ID=1) ARRAY(101)...ARRAY(500)
      .
      .
      .
```

After execution of the WAIT, the specified array elements may be referred to.

Example 2:

```
        DIMENSION ARRAY1(100), ARRAY2(100)
        ISW=0
   10   READ(9,ID=1) ARRAY1
        IF (ISW.EQ.0) GO TO 30
   20   WRITE (8,ID=2)ARRAY2
   30   WAIT (9,ID=1,COND=ITEST)
        GO TO (35,7777,8888),ITEST
   35           .
                .
                .
        IF (ISW.EQ.0) GO TO 60
   40   WAIT (8,ID=2,COND=JTEST)
        IF (JTEST.GT.1) GO TO 9999
   50   DO 55 I=1,100
   55   ARRAY2(I)=ARRAY1(I)
        GO TO 10
   60   ISW=1
        GO TO 50
 7777           .
 8888           .
 9999           .
        STOP
        END
```

This routine reads a record into main storage, processes the data, moves it to a transmitting area, and writes a new record. After the first READ, reading and writing of records take place simultaneously.

Example 3:

```
          DIMENSION A(1000)              SUBROUTINE ASYNCI (ARRAY,I)
                  .                      DIMENSION ARRAY(I)
                  .                      READ(8,ID=1) ARRAY
          CALL ASYNCI(A,1000)            RETURN
                  .                      END
                  .
          WAIT (8,ID=1,NUM=N)
          N=N/4
                  .
                  .
          CALL ASYNCO(A,N)               SUBROUTINE ASYNCO(ARRAY,I)
          WAIT (9,ID=2)                  DIMENSION ARRAY(I)
                  .                      WRITE (9,ID=2) ARRAY
                  .                      RETURN
          STOP                           END
          END
```

In this routine, the main program calls a subprogram to execute an asynchronous READ and returns. The corresponding WAIT, in the main program, specifies that the number of bytes transmitted be placed in the variable N. The program calculates the number of elements completely filled (N/4), and transmits the array, with a new dimension, to another subroutine, ASYNCO, which writes the array and returns.

EXTENDED PRECISION

REAL*16 CONSTANTS

Definition
REAL*16 constant - a string of decimal digits with or without a decimal point, followed by an extended precision exponent. The exponent consists of the letter Q followed by a signed or unsigned 1- or 2-digit integer constant. A REAL*16 constant occupies 16 storage locations (bytes).
Magnitude: 0, or 16^{-65} (approximately 10^{-78}) through 16^{63} (approximately 10^{75}).
Precision: 28 hexadecimal digits (approximately 35 decimal digits).

The REAL*16 constant may be positive, negative, or zero. If unsigned and not zero, it is assumed to be positive. A zero may be written with a preceding sign, which has no effect on the value zero.

Examples:

Valid REAL*16 constants:

```
.23452345345634564567456756780+43
5.001Q08
```

Invalid REAL*16 constants:

88.6321574801111	(the exponent contains too many digits)
.879434607227421133400091	(the constant does not have an extended precision exponent, and is treated as a valid basic real constant of length 4)

The explicit specification statement REAL*16, and the IMPLICIT statement, may be used to declare one or more variables or arrays to be type REAL*16.

COMPLEX*32 CONSTANTS

Definition
COMPLEX*32 constant - an ordered pair of signed or unsigned REAL*16 constants, separated by a comma and enclosed in parentheses. The first real constant represents the real part of the complex number; the second, the imaginary part. The complex constant occupies 32 storage locations (bytes).

The real constants in a complex constant may be positive, negative, or zero. If unsigned and not zero, they are assumed to be positive. A zero may be written with a preceding sign, which has no effect on the value zero. Each of the real constants must be within the allowable range for a REAL*16 constant.

Examples:

Valid COMPLEX*32 constants:

```
(238.3054565676787Q59,-1.0Q-5)
(45Q6,6Q45)
```

Invalid COMPLEX*32 constants:

(.0009Q-1,7643.Q+1199)	(too many digits in the exponent of the imaginary part)
(43.34,34Q43)	(the real part is not a REAL*16 constant)

The explicit specification statement COMPLEX*32, and the IMPLICIT statement, may be used to declare one or more variables or arrays to be type COMPLEX*32.

Descriptions of the FORTRAN-supplied mathematical and service procedures supporting extended precision are given in Appendix C.

Table 6 shows the type and length of the result of the operations +, -, *, and /.

Table 6. Determining the Type and Length of the Result of +, -, *, and / Operations Used with the FORTRAN IV (H Ext.) Compiler

First Operand \ Second Operand	INTEGER (2)	INTEGER (4)	REAL (4)	REAL (8)	REAL (16)	COMPLEX (8)	COMPLEX (16)	COMPLEX (32)
REAL (16)	REAL (16)	REAL (16)	REAL (16)	REAL (16)	REAL (16)	COMPLEX (32)	COMPLEX (32)	COMPLEX (32)
COMPLEX (32)	COMPLEX (32)	COMPLEX (32)	COMPLEX (32)	COMPLEX (32)	COMPLEX (32)	COMPLEX (32)	COMPLEX (32)	COMPLEX (32)

The exponent of a REAL*16 operand must be a real or integer value; the result is a REAL*16 value. The exponent of a COMPLEX*32 operand must be an integer value; the result is a COMPLEX*32 value.

Q Format Code

The letter Q is used in the FORMAT statement to represent extended precision REAL data fields. The G format code may also be used.

Example:

```
80 FORMAT (Q25.17,2Q12.5,3Q23.16)
```

This statement specifies the format for six REAL*16 data fields as follows:

The first field contains 25 characters, of which 17 are the fractional portion.

The second and third are identical fields of 12 characters, 5 of which are the fractional portion.

The fourth, fifth, and sixth are identical fields and contain 23 characters, of which 16 are the fractional portion.

A scale factor, P, may be used with the Q format code.

EXTERNAL STATEMENT EXTENSION

An extension to the EXTERNAL statement allows the programmer to declare names of the FORTRAN library functions or subroutines to be the names of user-supplied functions or subroutines. The normal usage of the EXTERNAL statement is unchanged. The added facility is obtained by writing an ampersand (&) preceding the library function or subroutine name in an EXTERNAL statement. A built-in function name appearing in an EXTERNAL statement is assumed to be user supplied regardless of whether the name is preceded by an ampersand. An ampersand has no effect if it precedes a name other than that of a FORTRAN library function or subroutine.

This special declaration is made in program units which refer to the user-supplied subprogram, but not in the subprogram itself. When any program unit in the program declares a library name to be the name of a user-supplied subprogram (and when the subprogram is actually supplied), the library subprogram is not available to any program unit in the program.

It is not possible for some program units in a program to refer to a user-supplied routine while other program units in the same program refer to the FORTRAN library routine of the same name. However, the name of a built-in function may be used as the name of a user-supplied routine in some program units (by using that name in an EXTERNAL statement or conflicting type-statement), and as the name of a built-in function in other program units (where the name does not appear in an EXTERNAL statement or conflicting type-statement).

General Form
EXTERNAL &subprogram-name$_1$,&subprogram-name$_2$,...,&subprogram-name$_n$

Names without a preceding ampersand may also be included in the list. Elsewhere in the program unit, references to the function or subroutine are written in the normal manner, i.e., without the preceding ampersand.

A user-supplied function is regarded as of the type and length predefined by its initial character, unless its type is declared in an IMPLICIT statement or an explicit specification statement. (No type is associated with a subroutine name.) Compiler checking of the type, number, and length of the arguments of both functions and subroutines is suspended.

Example:

```
      EXTERNAL &CDSIN
        .
        .
        .
      A = CDSIN(B,C,D)
```

Explanation: Because of its appearance with an ampersand in an EXTERNAL statement, CDSIN is treated as a user-supplied external procedure of type REAL*4 (the predefined type and length for a variable name beginning with the letter C), rather than as a FORTRAN-supplied library subprogram of type COMPLEX*16. Note also that, since argument checking is suspended for the function, the three REAL*4 arguments may be used in place of the one COMPLEX*16 argument normally required.

AUTOMATIC FUNCTION SELECTION (GENERIC STATEMENT)

The automatic function selection facility allows the programmer to use a single generic name when requesting a FORTRAN-supplied function which has several names, depending on argument type. The proper function is selected by the FORTRAN compiler, based on the type and length of the argument(s) of the function.

With this facility the programmer can, for example, use the generic name SIN to refer to any sine routine, rather than explicitly calling SIN for REAL*4 arguments, DSIN for REAL*8 arguments, CSIN for COMPLEX*8 arguments, etc. The feature is requested by including the following statement in each executable program unit in which it is to be used.

General Form
GENERIC

As a specification statement, GENERIC must precede statement function definitions and all executable program statements, and must follow any FUNCTION, SUBROUTINE, or IMPLICIT statement.

The appearance of the GENERIC statement declares the set of names in the first column of Table 7 to be generic. Specific built-in and library function names may be interspersed with generic names in the same program unit.

The appearance of a generic name in an explicit specification statement invalidates its generic status, since generic names have no type. If a generic name which coincides with a built-in function name appears in an EXTERNAL statement, its generic status is also invalidated, since it is thereby considered a user-supplied external procedure. A generic name which does not coincide with a built-in function name may appear without a preceding ampersand in an EXTERNAL statement and still be considered generic. However, if the name appears with a preceding ampersand, it can no longer be generic since, again, it is considered a user-supplied external procedure.

Several new function names are recognized by the FORTRAN IV (H Extended) compiler. These names, here referred to as aliases, do not provide new functions; rather, they are common abbreviations for certain existing function names. (For example, the alias LOG may be used in place of the function name ALOG for the natural logarithm function.) The aliases, eight in number, are indicated by an asterisk following the function name in the first column of Table 7. The alias is the generic name for the respective function in program units in which the GENERIC statement has been included. However, in other program units, they are still recognized as aliases, that is, they may be used specifically in place of the respective function name. For further information see the FORTRAN IV (H Extended) programmer's guide.

Specific function names, rather than generic names, must be passed as arguments to external procedures. The automatic function selection facility will not substitute the appropriate function name for the generic name in an argument list. (There is no way to make such a selection, since the name being passed as an argument has no arguments of its own.) Thus, a function name is specific for use as an argument, even if the same name is generic for use as a function reference.

Example:

```
      GENERIC
      EXTERNAL COS
      REAL*8 A,B,C,D
      C=COS(A)
      D=DCOS(B)
      CALL SUB(COS)
               .
               .
               .
```

Explanation: Because automatic function selection has been invoked, the function DCOS is called to calculate the value of C, as well as D. The specific name COS is passed to the subroutine SUB.

Table 7. Generic Names for Built-in and Library Functions

Generic Name	Definition	Number, Type, and Length of Arguments[1]								Function Value[2]	
		No.	I*4	R*4	R*8	R*16	C*8	C*16	C*32	Type	Length
ABS	Absolute value	1	X	X	X	X				Argument	Argument
							X	X	X	Real	1/2 Argument
ACOS*	Arc cosine	1		X	X	X				Real	Argument
AINT	Truncation	1		X	X	X				Real	Argument
ASIN*	Arc sine	1		X	X	X				Real	Argument
ATAN	Arc tangent	1		X	X	X				Real	Argument
ATAN2	Arc tangent (2 arguments)	2		X	X	X				Real	Argument
CMPLX	Complex	2		X	X	X				Complex	2* Argument
CONJG	Conjugate	1					X	X	X	Complex	Argument
COS	Cosine	1		X	X	X	X	X	X	Argument	Argument
COSH	Hyperbolic cosine	1		X	X	X				Real	Argument
COTAN	Cotangent	1		X	X	X				Real	Argument
DBLE	Express as R*8	1		X		X				Real	8
DIM	Positive difference	2	X	X	X	X				Argument	Argument
ERF	Error function	1		X	X	X				Real	Argument
ERFC	1 - Error function	1		X	X	X				Real	Argument
EXP	Exponentiation	1		X	X	X	X	X	X	Argument	Argument
GAMMA	Gamma function	1		X	X					Real	Argument
IMAG*	Imaginary part	1					X	X	X	Real	1/2 Argument
INT	Express as I*4	1		X	X	X				Integer	4
LGAMMA*	Log of gamma function	1		X	X					Real	Argument
LOG*	Natural logarithm	1		X	X	X	X	X	X	Argument	Argument
LOG10*	Common logarithm	1		X	X	X				Real	Argument
MAX*	Maximum value	≥2	X	X	X	X				Argument	Argument
MIN*	Minimum value	≥2	X	X	X	X				Argument	Argument
MOD	Remainder	2	X	X	X	X				Argument	Argument
QEXT	Express as R*16	1		X	X					Real	16
REAL	Real part	1					X	X	X	Real	1/2 Argument
SIGN	Transfer of sign	2	X	X	X	X				Argument	Argument
SIN	Sine	1		X	X	X	X	X	X	Argument	Argument
SINH	Hyperbolic sine	1		X	X	X				Real	Argument
SNGL	Express as R*4	1			X	X				Real	4
SQRT	Square root	1		X	X	X	X	X	X	Argument	Argument
TAN	Tangent	1		X	X	X				Real	Argument
TANH	Hyperbolic tangent	1		X	X	X				Real	Argument

1. "X" indicates a permissible mode of argument. For argument ranges, see Appendix C.

2. "Argument" indicates that the type or length of the result is the same as that of the argument(s).

* The function name is an alias. In the following list, function names in the left column are aliases for those in the right column.

Alias	Function
ACOS	ARCOS
ASIN	ARSIN
IMAG	AIMAG
LGAMMA	ALGAMA
LOG	ALOG
LOG10	ALOG10
MAX	MAX0
MIN	MIN0

APPENDIX I: FORTRAN IV (H EXTENDED), (G1), AND CODE AND GO FEATURES

This appendix describes the list-directed READ and WRITE statements, extensions to the IBM System/360 and 370 FORTRAN IV language, available with the FORTRAN IV (H Extended), FORTRAN IV (G1), and Code and Go FORTRAN compilers.

List-directed I/O simplifies data entry by freeing the user from FORMAT statement restrictions. A list-directed READ, WRITE, PRINT or PUNCH statement substitutes an asterisk for the FORMAT statement number. No FORMAT statement is used. Data to be read or written by execution of the statement may be entered (or is written) without regard for column, card, or line boundaries.

LIST-DIRECTED READ STATEMENT

General Form

READ (a,*,ERR=b,END=c) list

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

ERR=b is optional and b is the number of a statement in the same program unit as the READ statement to which transfer is made if a transmission error occurs during data transfer.

END=c is optional and c is the number of a statement in the same program unit as the READ statement to which transfer is made upon encountering the end of the data set.

list is an I/O list.

Execution of the list-directed READ statement causes data to be read from the data set corresponding to a. The values are assigned to the elements specified by the list. The ERR and END parameters may appear in any order.

If ERR=b is specified and a transmission error occurs, control is transferred to the statement numbered b. No indication is given of which record or records could not be read. If a transmission error occurs, and the ERR parameter is not specified, object program execution terminates.

If END=c is specified and an end-of-file record is read, control is transferred to the statement numbered c. No indication is given of the number of list items read into before the end of the data set was encountered. If the END parameter is not specified, object program execution is terminated when the end of the data set is encountered.

The list may not specify fewer elements than are contained in the record, and may specify more elements only if the record contains a slash after the last element. (See the section "List-directed Input Data," below.) The BACKSPACE statement may not be used with list-directed data.

An alternate form,

```
READ *, list
```

may also be used. This has the same effect as

```
READ (a,*) list
```

where a is installation dependent. The ERR=b and END=c parameters may not be used with this form.

Example:

```
READ (5,*,ERR=98,END=99) (ARRAY(I),I=1,25),B,C(6)
```

Execution of this statement causes data to be read from the data set with reference number 5 into the 27 elements specified by the list. If an error occurs during transmission, control is transferred to the statement numbered 98. If an end-of-file record is read, control is passed to the statement numbered 99.

LIST-DIRECTED WRITE STATEMENT

General Form
WRITE (a,*) list Where: a is an unsigned integer constant or integer variable that is of length 4 and represents a data set reference number. list is an I/O list.

Execution of the list-directed WRITE statement causes the next record to be created on the data set specified by a. The list specifies the sequence and locations from which the data is to be taken. (See the section "List-Directed Output Data," below.)

Example:

```
WRITE (6,*) A,B,C(I),(D(N),N=1,5)
```

Execution of this statement causes a record containing the eight elements specified by the list to be written on the data set with reference number 6.

The following forms of the PUNCH and PRINT statements may be used for list-directed data:

```
PUNCH *, list

PRINT *, list
```

These statements have the same effect as

```
WRITE (a,*) list
```

where a is installation dependent.

LIST-DIRECTED INPUT DATA

A record containing list-directed input data consists of an alternation of constants and separators. The record may be read from any input device, including a keyboard terminal.

An input constant may be of any valid FORTRAN data type, except that a literal constant must be enclosed in quotation marks. Blanks may not be embedded in any list-directed constant except a literal constant, since they would be interpreted as separators. Numeric constants may optionally be signed, but there must be no embedded blanks between the sign and the constant.

Each constant (except a literal constant) must agree in type with the corresponding list element. The decimal point may be omitted from a real constant. If omitted, it is assumed to follow the rightmost digit of the constant.

With the exceptions noted below, a separator is either a comma or a blank. In addition, for terminal input, a horizontal tab or a carriage return is a separator. For punched card input, an end-of-card condition is also a separator.

A carrier return or an end-of-card condition is not considered a separator if it occurs anywhere within a literal constant, or immediately before or immediately after the internal comma in a complex constant. Blanks may optionally occur between the comma and the carrier return or end-of-card.

A separator may be surrounded by any number of blanks, horizontal tabs, carrier returns, or end-of-card conditions. Any such combination (with no intervening constants) constitutes a single separator. At the inception of execution of a list-directed READ, a preceding separator is assumed; and initial blanks, horizontal tabs, carrier returns, or end-of-card conditions, if present, are considered part of that separator.

A null item is represented by two consecutive commas with no intervening constant. Any number of blanks, horizontal tabs, carrier returns, or end-of-card conditions may be embedded between the commas. If a null item is specified, the corresponding list item is skipped; its current value remains unaltered.

A repetition factor may be specified for a constant or null item. For a constant, the form is

*i**constant

and for a null item, the form is

*i**

In each instance, *i* is a nonzero integer constant, which indicates that the following constant or null item is to occur *i* times. Neither of these forms may contain embedded blanks (except for blanks in a literal constant). The separators surrounding a repeated null item need not be commas.

A slash (/) serves as a special-purpose separator, indicating that no more data is to be read during the current execution of a READ statement. The slash may be surrounded by any number of horizontal tabs, carrier returns, or end-of-card conditions, which do not constitute additional separators. If the list has not been satisfied,

the values of the remaining list elements remain unaltered. If the list has been satisfied, the slash is optional.

Example:

In the following example, a list-directed READ statement is used to read a record containing constants of various types into main storage. For this example, the character [is used within the data stream to represent a horizontal tab, and the character] is used to represent a carrier return. Note that a carrier return is the terminal equivalent of an end-of-card condition. Correct type specification is assumed throughout.

```
READ (5,*) (ARRAY(I),I=1,50), HEAD1,HEAD2,A,B,C,D,E,F,G,H,J,P

      50*0.  ' HEADING' [ 'FOOTING']
Data
      (2.17E+15,3.14E0) , 1.,2.0,0.125D-3[2*,.TRUE.,,8/
```

Explanation: When this READ is executed, the value 0. is read into each of the first 50 elements of ARRAY; text is read into the locations HEAD1 and HEAD2; a complex value is read into location A (note that no blanks are permitted within the complex constant specification); three values are read into locations B, C and D; locations E and F are skipped because of the repeated null specification; the logical value true is placed in location G; location H is skipped; location J receives the value 8; and location P receives no data because of the terminating slash.

LIST-DIRECTED OUTPUT DATA

List-directed output may contain any producible form of data which is readable as list-directed input. However, certain forms which are permissible as list-directed input are not produced as list-directed output. These forms are:

- D and Q forms of exponent
- literal constants
- null items
- i* repeat factor
- / special-purpose separator

If a REAL*8, DOUBLE PRECISION, or REAL*16 data item is to be written, an E is produced instead of the D or Q exponent in the converted number. The appropriate level of precision is maintained. A separator is generated after each data item, including the last item of a record.

alphabetic character: a character of the set A, B, C, ..., Z, $.

alphameric character: a character of the set which includes the alphabetic characters and the numeric characters.

argument: a parameter passed between a calling program and a subprogram or statement function.

arithmetic expression: a combination of arithmetic operators and arithmetic primaries.

arithmetic operator: one of the symbols +, -, *, /, **, used to denote, respectively, addition, subtraction, multiplication, division, and exponentiation.

arithmetic primary: an irreducible arithmetic unit; a single constant, variable, array element, function reference, or arithmetic expression enclosed in parentheses.

array: an ordered set of data items identified by a single name.

array declarator: the part of a statement which describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension. An array declarator may appear in a DIMENSION, COMMON, or explicit specification statement.

array element: a data item in an array, identified by the array name followed by a subscript indicating its position in the array.

array name: the name of an ordered set of data items.

assignment statement: an arithmetic or logical variable or array element, followed by an equal sign (=), followed by an arithmetic or logical expression.

basic real constant: a string of decimal digits containing a decimal point.

blank common: an unlabeled (unnamed) common block.

common block: a storage area that may be referred to by a calling program and one or more subprograms.

complex constant: an ordered pair of real constants separated by a comma and enclosed in parentheses. The first real constant represents the real part of the complex number; the second represents the imaginary part.

constant: a fixed and unvarying quantity. The four classes of constants specify numbers (numerical constants), truth values (logical constants), literal data (literal constants), and hexadecimal data (hexadecimal constants).

control statement: any of the several forms of GO TO, IF and DO statements, or the PAUSE, CONTINUE, and STOP statements, used to alter the normally sequential execution of FORTRAN statements, or to terminate the execution of the FORTRAN program.

data item: a constant, variable, or array element.

data set: an ordered collection of one or more records.

data set reference number: a constant or variable in an input/output statement, which specifies the data set which is to be operated upon.

data type: the mathematical properties and internal representation of data and functions. The four basic types are integer, real, complex, and logical.

DO loop: repetitive execution of the same statement or statements by use of a DO statement.

DO variable: a variable, specified in a DO statement, which is initialized or incremented prior to each execution of the statement or statements within a DO loop. It is used to control the number of times the statements within the DO loop are executed.

dummy argument: a variable within a FUNCTION or SUBROUTINE statement, or statement function definition, with which actual arguments from the calling program or function reference are associated.

executable program: a program that can be used as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures or both.

executable statement: a statement which specifies action to be taken by the program; e.g., causes calculations to be performed, conditions to be tested, flow of control to be altered.

extended range of a DO statement: those statements that are executed between the transfer out of the innermost DO of a completely nested nest of DO statements and the transfer back into the range of this innermost DO.

external function: a function whose definition is external to the program unit which refers to it.

external procedure: a procedure subprogram or a procedure defined by means other than FORTRAN statements.

formatted record: a record which is transmitted with the use of a FORMAT statement.

FUNCTION subprogram: an external function defined by FORTRAN statements and headed by a FUNCTION statement. It returns a value to the calling program unit at the point of reference.

hexadecimal constant: the character Z followed by a hexadecimal number, formed from the set 0 through 9 and A through F.

hierarchy of operations: relative priority assigned to arithmetic or logical operations which must be performed.

implied DO: the use of an indexing specification similar to a DO statement (but without specifying the word DO and with a list of data elements, rather than a set of statements, as its range).

integer constant: a string of decimal digits containing no decimal point.

I/O list: a list of variables in an I/O statement, specifying the storage locations into which data is to be read or from which data is to be written.

labeled common: a named common block.

length specification: an indication, by the use of the form *s, of the number of storage locations (bytes) to be occupied by a variable or array element.

literal constant: a string of alphameric and/or special characters enclosed in quotation marks or preceded by a wH specification.

logical constant: a constant that specifies a truth value: true or false.

logical expression: a combination of logical primaries and logical operators.

logical operator: any of the set of three operators .NOT., .AND., .OR..

logical primary: an irreduceable logical unit: a logical constant, logical variable, logical array element, logical function reference, relational expression, or logical expression enclosed in parentheses, having the value true or false.

looping: repetitive execution of the same statement or statements, usually controlled by a DO statement.

main program: a program unit not containing a FUNCTION, SUBROUTINE, or BLOCK DATA statement and containing at least one executable statement. A main program is required for program execution.

name: a string of from one through six alphameric characters, the first of which must be alphabetic, used to identify a variable, an array, a function, a subroutine, a common block, or a namelist.

nested DO: a DO loop whose range is entirely contained by the range of another DO loop.

nonexecutable statement: a statement which describes the use or extent of the program unit, the characteristics of the operands, editing information, statement functions, or data arrangement.

numeric character: any one of the set of characters 0,1,2,...,9.

numeric constant: an integer, real, or complex constant.

predefined specification: the FORTRAN-defined type and length of a variable, based on the initial character of the variable name in the absence of any specification to the contrary. The characters I-N are typed INTEGER*4; the characters A-H, O-Z and $ are typed REAL*4.

procedure subprogram: a FUNCTION or SUBROUTINE subprogram.

program unit: a main program or a subprogram.

range of a DO statement: those statements which physically follow a DO statement, up to and including the statement specified by the DO statement as being the last to be executed in the DO loop.

real constant: a string of decimal digits which must have either a decimal point or a decimal exponent, and may have both.

relational expression: an arithmetic expression, followed by a relational operator, followed by an arithmetic expression. The expression has the value true or false.

relational operator: any of the set of operators which express an arithmetic condition that can be either true or false. The operators are: .GT., .GE., .LT., .LE., .EQ., .NE., and are defined as greater than, greater than or equal to, less than, less than or equal to, equal to, and not equal to, respectively.

scale factor: a specification in a FORMAT statement whereby the location of the decimal point in a real number (and, if there is no exponent, the magnitude of the number) can be changed.

specification statement: one of the set of statements which provide the compiler with information about the data used in the source program. In addition, the statement supplies information required to allocate storage for this data.

specification subprogram: a subprogram headed by a BLOCK DATA statement and used to initialize variables in labeled (named) common blocks.

statement: the basic unit of a FORTRAN program, composed of a line or lines containing some combination of names, operators, constants, or words whose meaning is predefined to the FORTRAN compiler. Statements fall into two broad classes: executable and nonexecutable.

statement function: a function defined by a function definition within the program unit in which it is referred to.

statement function definition: a name, followed by a list of dummy arguments, followed by an equal sign (=), followed by an arithmetic or logical expression.

statement function reference: a reference in an arithmetic or logical expression to a previously defined statement function.

statement number: a number of from one through five decimal digits placed within columns 1 through 5 of the initial line of a statement. It is used to identify a statement uniquely, for the purpose of transferring control, defining a DO loop range, or referring to a FORMAT statement.

subprogram: a program unit headed by a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

SUBROUTINE subprogram: a subroutine consisting of FORTRAN statements, the first of which is a SUBROUTINE statement. It optionally returns one or more parameters to the calling program unit.

subscript: a subscript quantity or set of subscript quantities, enclosed in parentheses and used in conjunction with an array name to identify a particular array element.

subscript quantity: a component of a subscript: a positive integer constant, integer variable, or expression which evaluates to a positive integer constant. If there is more than one subscript quantity in a subscript, the quantities must be separated by commas.

type declaration: the explicit specification of the type and, optionally, length of a variable or function by use of an explicit specification statement.

unformatted record: a record for which no FORMAT statement exists, and which is transmitted with a one-to-one correspondence between internal storage locations (bytes) and external positions in the record.

variable: a data item that is not an array or array element, identified by a symbolic name.

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

GC28-6515-10

S/360 & S/370 FORTRAN IV Lang. Printed in U.S.A. GC28-6515-10

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)

READER'S COMMENTS

TITLE: IBM System/360 & System/370 FORTRAN IV Language

ORDER NO. GC28-6515-10

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM Branch Office serving your locality.

Corrections or clarifications needed:

Page *Comment*

Please include your name and address in the space below if you wish a reply.

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

GC28-6515-10

cut along this line

fold fold

FIRST CLASS
PERMIT NO. 33504
NEW YORK, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM CORPORATION
1271 Avenue of the Americas
New York, New York 10020

Attention: PUBLICATIONS

fold fold

S/360 & S/370 FORTRAN IV Lang. Printed in Denmark. GC28-65 15-10

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)