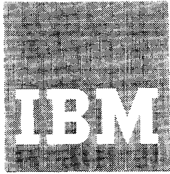
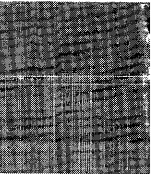
File No. S360-25
Form C28-6515-4

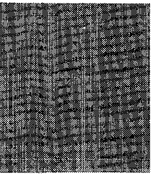
OS

44PS



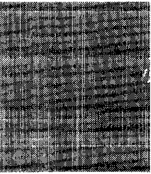
Systems Reference Library



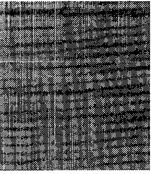


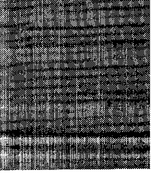
IBM System/360

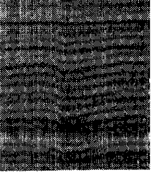
FORTRAN IV Language

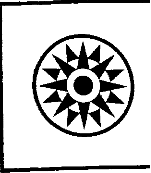


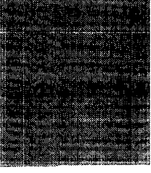
This publication describes and illustrates the use of the FORTRAN IV language for the IBM System/360 Operating System and the IBM System/360 Model 44 Programming System.











PREFACE

This publication describes the IBM System/360 FORTRAN IV language for the IBM System/360 Operating System and the IBM System/360 Model 44 Programming System. A reader should have some knowledge of an existing FORTRAN language before using this publication.

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

Appendixes contain additional information useful in writing a FORTRAN IV program. This information consists of a table of source program characters, a list of other FORTRAN statements accepted by FORTRAN IV, a list of FORTRAN supplied mathematical subprograms, and sample programs. Out-of-line mathematical subprograms and service subprograms are described in the publication IBM System/360 Operating System: FORTRAN IV (E) Library Subprograms, Form C28-6596.

TEMPORARY RESTRICTION

The ordering of variables in COMMON blocks and equivalence groups is subject to the following restriction:

> The programmer must always ensure proper boundary alignment of all variables in COMMON blocks and equivalence groups.

The methods of ensuring proper alignment are given in the descriptions of the COMMON and EQUIVALENCE statements in this publication.

The restriction will be removed in the near future.

Fourth Edition
This is a major revision of, and obsoletes, the previous edition, Form C28-6515-3 and Technical Newsletter N28-2104. Automatic function typing has been removed from this edition which also contains minor technical corrections and additions to the previous edition. Technical changes to the text are indicated by a vertical line to the left of the change; revised illustrations are indicated by the symbol • to the left of the caption.

Significant changes or additions to the specifications contained in this publication will be reported in subsequent revisions or Technical Newsletters.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special print chain.

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

A form is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to IBM Corporation, Programming Systems Publications, Department D58, PO Box 390, Poughkeepsie, N. Y. 12602

© International Business Machines Corporation 1965, 1966

INTRODUCTION 5

ELEMENTS OF THE LANGUAGE 7

Statements 7
Coding FORTRAN Statements 7

Constants. 8
Integer Constants 9
Real Constants. 9
Complex Constants 11
Logical Constants 11
Literal Constants 12
Hexadecimal Constants 12

Variables. 13
Variable Names. 13
Variable Types and Lengths. 14
Type Declaration by the Predefined Specification. 14
Type Declaration by the Implicit Specification Statement. 15
Type Declaration by Explicit Specification Statements 15

Expressions. 16
Arithmetic Expressions. 16
Arithmetic Operators 16
Logical Expressions 20
Relational Operators 20
Logical Operators. 21

Arrays 23

Subscripts 25
Declaring the Size of an Array. . . . 25
Arrangement of Arrays in Storage. . . 25

ARITHMETIC AND LOGICAL ASSIGNMENT STATEMENT 27

CONTROL STATEMENTS 29

The GO TO Statements 29
Unconditional GO TO Statement. . . 29
Computed GO TO Statement 30
The ASSIGN and Assigned GO TO Statements. 31

Additional Control Statements. 32
Arithmetic IF Statement. 32
Logical IF Statement 33
DO Statement 34
CONTINUE Statement 37
PAUSE Statement. 38
STOP Statement 39
END Statement 39

INPUT/OUTPUT STATEMENTS. 40

Sequential Input/Output Statements . . . 40
READ Statement. 41
The Form READ (a,x). 42
The Form READ (a,b) list 44
The Form READ (a) list 45
Indexing I/O Lists 46
Reading Format Statements. 47
WRITE Statement 47
The Form WRITE (a,x) 48
The Form WRITE (a,b) list. 48
The Form WRITE (a) list. 49
FORMAT Statement. 50
G Format Code. 54
Numeric Format Codes (I,F,E,D) . . 58
I Format Code. 59
F Format Code. 59
D and E Format Codes 60
Z Format Code. 60
L Format Code. 61
A Format Code. 61
Literal Data in a Format Statement 64
H Format Code. 65
X Format Code. 65
T Format Code. 66
Scale Factor - P 66
Carriage Control 69
END FILE Statement. 69
REWIND Statement. 69
BACKSPACE Statement 70

Direct Access Input/Output Statements. . 70
DEFINE FILE Statement. 70
Programming Considerations 72
READ Statement 73
WRITE Statement. 74
FIND Statement 75

SPECIFICATION STATEMENTS 77

The Type Statements. 77
IMPLICIT Statement 77
Explicit Specification Statements. 79
Adjustable Dimensions. 81

Additional Specification Statements. . . 82
DIMENSION Statement. 82
COMMON Statement 83
Blank and Labeled Common 84
Programming Considerations 86
EQUIVALENCE Statement. 87
Programming Considerations 89

SUBPROGRAMS. 90
Naming Subprograms. 90
Functions 91
Function Definition. 91
Function Reference 91

Statement Functions 91
FUNCTION Subprograms. 93
Type Specification of the FUNCTION Subprogram 94
RETURN and END Statements in a Function Subprogram 95
SUBROUTINE Subprograms. 96
CALL Statement 97
Arguments in a FUNCTION and SUBROUTINE Subprogram. 98
RETURN Statement in a SUBROUTINE Subprogram. 100
Multiple ENTRY into a Subprogram .101
Additional Rules for Using ENTRY .103
EXTERNAL Statement. 103
Block Data Subprogram 104

APPENDIX A: SOURCE PROGRAM CHARACTERS .105

APPENDIX B: OTHER FORTRAN STATEMENTS ACCEPTED BY FORTRAN IV. 106
READ Statement 106
PUNCH Statement. 106
PRINT Statement. 107
DATA Initialization Statement. . .107
DOUBLE PRECISION Statement 108

APPENDIX C: FORTRAN-SUPPLIED SUBPROGRAMS 109

APPENDIX D: SAMPLE PROGRAMS 112

Sample Program 1 112

Sample Program 2 113

INDEX. 119

ILLUSTRATIONS

FIGURES

Figure 1. FORTRAN Coding Form 8
Figure 2. Sample Program 1. 112
Figure 3. Sample Program 2. 115

TABLES

Table 1. Determining the Mode of an Expression Containing Operands of Different Types 17
Table 2. Valid Combinations with Respect to the Arithmetic Operator, **. 18
Table 3. Insurance Premium Codes. . . . 24
Table 4. Mathematical Function Subprograms 109

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

The FORTRAN IV language is especially useful in writing programs for scientific and engineering applications that involve mathematical computations. In fact, the name of the language -FORTRAN- is derived from its primary use: FORmula TRANslating.

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

The FORTRAN compiler analyzes the source program statements and transforms them into an object program that is suitable for execution on the IBM System/360. In addition, when the FORTRAN compiler detects errors in the source program, appropriate error messages are produced.

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

The IBM System/360 FORTRAN IV language is compatible with and encompasses the American Standards Association (ASA) FORTRAN, including its mathematical subroutine provisions. It also contains, as proper subsets, Operating System FORTRAN IV (E), Basic Operating System FORTRAN IV, and Basic Programming Support FORTRAN IV.

Any valid programs compiled and executed using any System/360 subset FORTRAN may also be compiled and executed by FORTRAN IV compilers. Equivalent results are assured by:

1. Common data formats.
2. Common format code routines.
3. Common mathematical subroutines.

All of the features and facilities in Operating System FORTRAN IV (E) also exist in System/360 FORTRAN IV. Equivalent results from valid programs compiled by either FORTRAN IV (E) and FORTRAN IV are assured by:

1. Common data formats.
2. Common format code routines.
3. Common calling sequences.
4. Common libraries.

The following features facilitate the writing of source programs and reduce the possibility of coding errors:

1. Mixed-Mode: Expressions may consist of constants and variables, of the same and/or different types.

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

3. Literal Format Code: Apostrophes may be used to enclose literal data in a FORMAT Statement.

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

   a. Specify storage length.
   b. Explicitly type a variable as integer, real, complex, or logical.
   c. Specify the dimension of arrays.
   d. Specify data initialization values for variables.

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

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

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

STATEMENTS

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

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

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

1. Arithmetic and Logical Assignment Statements: Upon execution of an arithmetic or logical assignment statement, the result of calculations performed or conditions tested replaces the current value of a designated variable or subscripted variable.

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

3. Input/Output Statements: These statements, in addition to controlling input/output (I/O) devices, enable the user to transfer data between internal storage and an I/O medium.

4. Specification Statements: These statements are used to declare the properties of variables, arrays, and subprograms (such as type and amount of storage reserved) and to describe the format of data on input or output.

5. Subprogram Statements: These statements enable the user to name and define 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 phrase executable statements refers to those statements in groups 1, 2, and 3.

CODING FORTRAN STATEMENTS

The statements of a FORTRAN source program can be written on a standard FORTRAN coding form, Form No. X28-7327 (see Figure 1). FORTRAN statements are written one to a line from columns 7 through 72. If a statement is too long for one line, it may be continued on as many as 19 successive lines by placing any character, other than a blank or zero, in column 6 of each continuation line. For the first line of a statement, column 6 must be blank or zero.

Columns 1 through 5 of the first line of a statement may contain a statement number consisting of from 1 through 5 decimal digits. Leading zeros in a statement number are ignored. Statement numbers may appear anywhere in columns 1 through 5 and may be assigned in any order; the

value of statement numbers does not affect the order in which the statements are executed in a FORTRAN program.

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

IBM FORTRAN Coding Form X28-7327-5 Printed in U.S.A.

PROGRAM | PUNCHING INSTRUCTIONS | GRAPHIC | PAGE OF
PROGRAMMER | DATE | PUNCH | CARD ELECTRO NUMBER*

STATEMENT NUMBER | CONT. | FORTRAN STATEMENT | IDENTIFICATION SEQUENCE

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

• Figure 1. FORTRAN Coding Form

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

CONSTANTS

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

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

INTEGER CONSTANTS

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

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

Examples:

Valid Integer Constants:

0
91
173
-2147483647
-12

Invalid Integer Constants:

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

REAL CONSTANTS

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

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

Examples:

Valid Real Constants (4 storage locations):

Constant	
+0.	
-999.9999	
0.0	
5764.1	
7.0E+0	(i.e., $7.0 \times 10^{0} = 7.0$)
19761.25E+1	(i.e., $19761.25 \times 10^{1} = 197612.5$)
7.E3 7.0E3 7.0E03 7.0E+03	(i.e., $7.0 \times 10^{3} = 7000.0$)
7.0E-03	(i.e., $7.0 \times 10^{-3} = 0.007$)

Valid Real Constants (8 storage locations):

Constant	
21.98753829457168	
1.0000000	
7.9D3 7.9D03 7.9D+03 7.9D+3	(i.e., $7.9 \times 10^{3} = 7900.0$)
7.9D-03	(i.e., $7.9 \times 10^{-3} = .0079$)
7.9D0	(i.e., $7.9 \times 10^{0} = 7.9$)
00000001.	

Invalid Real Constants:

Constant	
0	(missing a decimal point)
3,471.1	(embedded comma)
1.E	(missing a one- or two-digit integer constant following the E. Note that it is not interpreted as 1.0×10^{0})
1.2E+113	(E is followed by a 3 digit integer constant)
23.5E+97	(value exceeds the magnitude permitted; that is, $23.5 \times 10^{97} > 16^{63}$)
7.9D	(missing a one- or two-digit integer constant following the D)
21.3E90	(value exceeds the magnitude permitted; that is, $21.3 \times 10^{90} > 16^{63}$)

COMPLEX CONSTANTS

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

The real constants in a complex constant may be positive, zero, or negative (if unsigned, they are assumed to be positive), but they must be in the given range. The first real constant in a complex constant represents the real part of the complex number; the second represents the imaginary part of the complex number. If the exponent is given, the decimal point is not required.

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)
(4.0E+03,.16E+02)	(has the value 4000.+16.0i)
(2.1,0.0)	(has the value 2.1+0.0i)
(4.7D+2,1.9736148)	(has the value 470.+1.9736148i)

Where $i = \sqrt{-1}$

Invalid Complex Constants:

(292704,1.697)	(the real part does not contain a decimal point)
(1.2E113,279.3)	(the real part contains an invalid decimal exponent)
(.003E4,.005D6)	(the parts differ in length)

LOGICAL CONSTANTS

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

The logical constants .TRUE. and .FALSE. specify that the value of the logical variable they are associated with is true or false, respectively. (See the section, "Logical Expressions.")

LITERAL CONSTANTS

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

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

Examples:

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

HEXADECIMAL CONSTANTS

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

Hexadecimal constants may be used only as data initialization values.

One storage location contains two hexadecimal digits. If a constant is specified as an odd number of digits, a leading hexadecimal zero is added 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
ZBADFAD
```

The maximum number of digits allowed in a hexadecimal constant depends upon the length specification of the variable being initialized (see "Variable Types and 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.

VARIABLES

A FORTRAN variable is a symbolic representation of a quantity that is assigned a value. The value may either be unchanged (i.e., constant) or may change either for different executions of a program or at different stages within the program.

For example, in the statement:

```
A = 5.0+B
```

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

VARIABLE NAMES

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

Variable names are symbols used to distinguish one variable from another. A name may be used in a source program in one and only one way (e.g., the name of a variable and that of a subprogram may not be identical in the same source program). A variable name may not contain special characters (see Appendix A).

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

```
X = Y * Z
```

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

```
DIST = RATE * TIME
```

Examples:

Valid Variable Names:

```
B292
RATE
SQ704
$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.

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

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

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

1. Predefined specification contained in the FORTRAN language.
2. Explicit specification statements.
3. IMPLICIT Specification statement.

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, "The 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.

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

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

TYPE DECLARATION BY THE IMPLICIT SPECIFICATION STATEMENT

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

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

TYPE DECLARATION BY EXPLICIT SPECIFICATION STATEMENTS

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

For example, assume:

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

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

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

These statements are discussed in greater detail in the section, "Specification Statements."

EXPRESSIONS

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

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. However, the evaluation of a logical expression always yields a truth value: .TRUE. or .FALSE..

ARITHMETIC EXPRESSIONS

The simplest arithmetic expression consists of a single constant, variable, or subscripted variable (see the discussion of arrays). The constant or variable may be one of the following types:

1. Integer
2. Real
3. Complex

If the constant, variable, or subscripted variable is of the type integer, the expression is in the integer mode. If it is of the type real, the expression is in the real mode, etc. The mode of the expression is determined solely by the type of constant, variable, or subscripted variable appearing in that expression.

Examples:

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

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

Arithmetic Operators

The arithmetic operators are as follows:

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

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

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

   AxB or AB or A•B

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

   A*B or B*A

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

   A*/B and A*-B

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

   A*(-B)

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

3. The mode of an arithmetic expression is determined by the type of the operands (where an operand is a variable, constant, function reference, or another expression) in the expression. Table 1 indicates how the mode of an expression that contains operands of different types may be determined using the operators: +, -, *, /.

Table 1. Determining the Mode of an Expression Containing Operands of Different Types

+ - * /	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)

From Table 1 it can be seen that there is a hierarchy of type and length specification (see the section, "The Type Statements") that determines the mode of an expression. For example, complex data that has a length specification of 16 when combined with any other types of constants and variables results in complex data of length 16.

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

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

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

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

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

• Table 2. Valid Combinations with Respect to the Arithmetic Operator, **

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

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

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

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

Examples:

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

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

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

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

For example, the expression:

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

is effectively evaluated in the following order:

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

Note: This order of computation is used in determining the mode of an expression (see Table 1).

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

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

is evaluated as follows:

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

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

For example, the following expression:

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

is effectively evaluated in the following order:

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

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

LOGICAL EXPRESSIONS

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

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

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

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

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

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

Relational Operators

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

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

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

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

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

Examples:

Valid Logical Expressions Using Relational Operators:

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

Invalid Logical Expressions Using Relational Operators:

C.LT.ROOT	(Complex quantities may never appear in logical expressions)
C.GE.(2.7,5.9E3)	(Complex quantities may never appear in logical expressions)
L.EQ.(A+F)	(Logical quantities may never be joined by relational operators)
E**2.EQ97.1E9	(Missing period immediately after 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: (A and B represent logical constants or variables, or expressions containing relational operators).

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

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

Only those expressions which, when evaluated, have the value .TRUE. or .FALSE. may be combined with the logical operators to form logical expressions. For example, assume that the type of the following variables has been specified as follows:

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

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

Examples:

Valid Logical Expressions:

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

Invalid Logical Expressions:

A.AND.L	(A is not a logical expression)
.OR.W	(.OR. must be preceded by a logical expression)
NOT.(A.GT.F)	(missing period before the logical operator .NOT.)
(C.EQ.I).AND.L	(a complex variable may never appear in a logical expression)
L.AND..OR.W	(the logical operators .AND. and .OR. must always be separated by a logical expression)
.AND.L	(.AND. must be preceded by a logical expression)

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

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

For example, the expression:

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

is effectively evaluated in the following order.

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

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

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

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

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

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

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

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

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

ARRAYS

A FORTRAN array is a set of variables identified by a single variable name. A particular variable in the array may be referred to by its position in the array (e.g., first variable, third variable, seventh variable, etc.). Consider the array named NEXT which consists of five variables, each currently representing the following values: 273, 41, 8976, 59, and 2.

NEXT(1) is the representation of 273
NEXT(2) is the representation of 41
NEXT(3) is the representation of 8976
NEXT(4) is the representation of 59
NEXT(5) is the representation of 2

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

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

To refer to the first element in an array, the array name must be subscripted. The array name itself does not represent the first element.

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

	Column1	Column2	Column3
Row1	82	4	7
Row2	12	13	14
Row3	91	1	31
Row4	24	16	10
Row5	2	8	2

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

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

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

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

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

Table 3. Insurance Premium Codes

AGE		SEX	
Age in Yrs.	Code	Sex	Code
		Male	ISEX=1
1 - 5	IAGE=1	Female	ISEX=2
6 - 10	IAGE=2		
11 - 15	IAGE=3	POLICY SIZE	
16 - 20	IAGE=4		
21 - 25	IAGE=5	Dollars	Code
26 - 30	IAGE=6		
31 - 35	IAGE=7		
36 - 40	IAGE=8	1,000	ISIZE=1
41 - 45	IAGE=9	3,000	ISIZE=3
46 - 50	IAGE=10	5,000	ISIZE=4
.	.	10,000	ISIZE=5
.	.	25,000	ISIZE=6
.	.	50,000	ISIZE=7
96 - 100	IAGE=20	100,000	ISIZE=8

Suppose an applicant were 14 years old, male, and desired a policy of $25,000. From Table 3, these statistics could be represented by the codes:

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

Thus, COST (3, 1, 6) represents the premium for a policy given the statistics above. Note that "IAGE" can vary from 1 to 20, "ISEX" from 1 to 2, and "ISIZE" from 1 to 8. (The number of subscripted variables in the array COST is the number of combinations that can be formed for different ages, sex, and policy size available - a total of 20x2x8 or 320. Therefore, there may be up to 320 different premiums stored in the array named COST.)

SUBSCRIPTS

A subscript is a number used to refer to a particular variable within an array. There may be a maximum of seven subscripts used with an array name. If more than one subscript is used they must be separated by commas. All of the subscripts used with a particular array name must be enclosed in parentheses.

The following rules apply to the construction of subscripts:

1. Subscripts may contain arithmetic expressions that use any of the arithmetic operators: +, -, *, /, **.
2. Subscripts may contain function references.
3. Subscripts may contain subscripted names.
4. Mixed mode expressions (integer and real only) within subscripts 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 must always be greater than zero and less than or equal to the size of the corresponding dimension.

Examples:

Valid Subscripted Variables:

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

Invalid Subscripted Variables

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

DECLARING THE SIZE OF AN ARRAY

The size of an array is determined by the number of subscript parameters of the array and the maximum value of each subscript. This information must be given for all arrays before using them in a FORTRAN program so that 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 further detail in the section, "Specification Statements."

ARRANGEMENT OF ARRAYS IN STORAGE

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

The array named A, consisting of one subscript parameter 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, consisting of two subscript parameters, whose first subscript varies over the range from 1 to 5, and second varies 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, consisting of three subscript parameters, whose first subscript varies from 1 to 3, second varies from 1 to 2, and third varies 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.

General Form
a = b Where: a is any subscripted or nonsubscripted variable. b is any arithmetic expression or logical expression.

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

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

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

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

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

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

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

THE GO TO STATEMENTS

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

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

Unconditional GO TO Statement

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

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

Example:

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

Explanation:

In the above example, every time statement 50 is executed, control is transferred to statement 25.

Computed GO TO Statement

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

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

Example:

```
      GO TO (25, 10, 50, 7), ITEM
        .
        .
        .
   50 A = B+C
        .
        .
        .
   7  C = E**2+A
        .
        .
        .
   25 L = C
        .
        .
        .
   10 B = 21.3E02
```

Explanation:

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

The ASSIGN and Assigned GO TO Statements

General Form
ASSIGN *i* TO *m* . . . GO TO *m*, ($x_1, x_2, x_3, \ldots, x_n$) Where: *i* is an executable statement number. $x_1, x_2, x_3, \ldots, x_n$ are executable statement numbers. *m* is a nonsubscripted integer variable which is of length 4 and is assigned one of the following statement numbers: $x_1, x_2, x_3, \ldots x_n$.

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

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

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

The current assignment of the integer variable *m* is determined by the last ASSIGN statement executed. Only an ASSIGN statement may be used to initialize or change the value of the integer variable *m*. The value of the integer variable *m* is not the integer statement number; ASSIGN 10 TO I is not the same as I = 10.

Example 1:

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

Explanation:

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

ADDITIONAL CONTROL STATEMENTS

Arithmetic IF Statement

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

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

Example:

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

Explanation:

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

Logical IF Statement

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

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

Example 1:

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

Explanation:

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

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

Example 2:

Assume that P and Q are logical variables.

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

Explanation:

In statement 5, if the value of the expression is .TRUE., the value of A is replaced by the value of B and statement 10 is executed next. If the value of the expression is .FALSE., the statement A = B is skipped and statement 10 is executed.

DO Statement

General Form
DO $\underbrace{x}_{\text{End of Range}}$ $\underbrace{i}_{\text{DO Variable}}$ = $\underbrace{m_1}_{\text{Initial Value}}$, $\underbrace{m_2}_{\text{Test Value}}$, $\underbrace{m_3}_{\text{Increment}}$
Where: *x* is an executable statement number that is not defined before the DO statement.
i is a nonsubscripted integer variable.
m_1, m_2, m_3, are either unsigned integer constants greater than zero or unsigned nonsubscripted integer variables whose value is greater than zero. m_2 may not exceed $2^{31}-2$ in value. m_3, is optional; if it is omitted, its value is assumed to be 1. In this case, the preceding comma must also be omitted.

The DO Statement is a command to execute repeatedly the statements that follow, up to and including the statement numbered *x*. The first time the statements in the range of the DO are executed, *i* is initialized to the value m_1; each succeeding time *i* is increased by the

value m_3. When, at the end of the iteration, i is equal to the highest value that does not exceed m_2, control passes to the statement following the statement numbered x. Thus, the number of times the statements in the range of the DO is executed is given by the expression:

$$\left\lfloor \frac{m_2 - m_1}{m_3} \right\rfloor + 1$$

where the brackets represent the largest integral value not exceeding the value of the expression. If m_2 is less than m_1, the statements in the range of the DO are executed once. Upon completion of the DO, the DO variable is undefined, and may not be used until redefined (e.g., in a READ list).

There are several ways in which looping (repetitively executing the same statements) may be accomplished when using the FORTRAN language. For example, assume that a manufacturer carries 1,000 different machine parts in stock. Periodically, he may find it necessary to compute the amount of each different part presently available. This amount may be calculated by subtracting the number of each item used, OUT(I), from the previous stock on hand, STOCK(I).

Example:

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

Explanation:

The three statements (5, 10, and 15) required to control the previously shown loop could be replaced by a single DO statement as shown in Example 1.

Example 1:

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

Explanation:

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

Example 2:

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

Explanation:

In the preceding example, 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 exexution of the DO loop, I is increased by the increment, 2, and statements 15 and 25 are executed a second time. After the fifth execution of the DO loop, I equals 9. Since I is now equal to the highest value that does not exceed the test value, 10, control passes out of the DO loop and statement 30 is executed next. Note that the DO variable I is now undefined; its value is not necessarily 9 or 11.

Programming Considerations in Using a DO Loop

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

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

   Example 1:

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

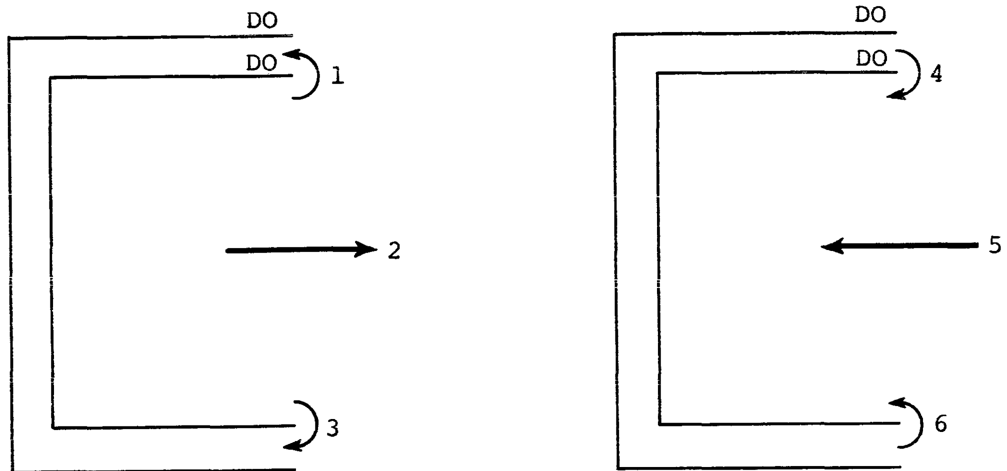
   Example 2:

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

3. A transfer out of the range of any DO loop is permissible at any time; a transfer into the range of a DO loop is permissible only as described in item 4.

4. When a transfer is made out of the range of an innermost DO loop, transfer back into the range of that DO loop is allowed if and only if none of the indexing parameters (i,m_1,m_2,m_3) are changed outside the range of the DO.

Example:



Explanation:

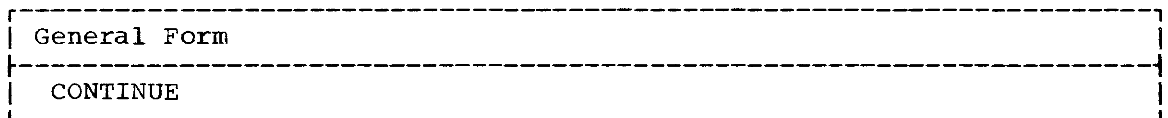
In the preceding example, the transfers specified by the numbers 1,2, and 3 are permissible, whereas those specified by 4,5, and 6 are not.

5. The indexing parameters ($\underline{i}, \underline{m}_1, \underline{m}_2, \underline{m}_3$) may be changed by statements outside the range of the DO statement only if no transfer is made back into the range of the DO statement that uses those parameters.

6. The last statement in the range of a DO loop (statement $\underline{x}$) must be an executable statement, not of the form GO TO, PAUSE, STOP, RETURN, Arithmetic IF, or another DO.

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

CONTINUE Statement



General Form
CONTINUE

CONTINUE is a dummy statement which may be placed anywhere in the source program without affecting the sequence of execution. It may be used as the last statement in the range of a DO in order to avoid ending the DO loop with a GO TO, PAUSE, STOP, RETURN, Arithmetic IF or another DO statement.

Example 1:

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

Explanation:

In the preceding example, 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 an unsigned 1 through 5 digit integer constant. 'message' is a literal constant.

Information is displayed and the program waits until operator intervention causes it to resume execution, starting with the next statement after the PAUSE statement. The particular form of the PAUSE statement used determines the nature of the information that is displayed. The PAUSE statement causes PAUSE 00000 to be displayed. If _n_ is specified, PAUSE _n_ is displayed. If '_message_' is specified, PAUSE '_message_' is displayed.

STOP Statement

General Form
STOP STOP _n_ Where: _n_ is an unsigned 1 through 5 digit integer constant.

This statement terminates the execution of the object program and displays _n_ if specified.

END Statement

General Form
END

The END statement is a nonexecutable statement that defines the end of a source program or source subprogram for the compiler. Physically, it must be the last statement of each program or subprogram, and it may not be continued.

INPUT/OUTPUT STATEMENTS

The input/output statements enable a user to transfer data, belonging to a named collection of data, between input/output devices (such as disk units, card readers, and magnetic tape units) and internal storage. The named collection of data is called a data set and is a continuous string of data that may be divided into FORTRAN records.

A data set is referred to by an integer constant or integer variable. Formerly, this reference was called a symbolic unit number. However, because the reference is to the data rather than any specific device, this number is called the data set reference number.

Two types of I/O statements are available to the FORTRAN IV user: sequential I/O statements and direct access I/O statements. The sequential statements provide facilities for the sequential selection and placement of data. These statements are device independent because a given statement may be applicable to a data set on any number of devices or device types.

The direct access I/O statements provide facilities for the selection and placement of data in an order specified by the user. These statements are only valid when the data set will be or is already resident on a direct access storage device.

SEQUENTIAL INPUT/OUTPUT STATEMENTS

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

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

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

The READ statement may take many forms. Either the list parameter or the b parameter may be omitted.

In addition, the parameters END=c and ERR=d are optional and, therefore, may or may not appear in a READ statement.

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

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

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

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

The basic forms of the READ statement involve formatted and unformatted data. They are:

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

The Form READ (a,x)

This form is used to read data from the data set associated with a into the locations in storage specified by the NAMELIST name x. The NAMELIST name x is a single variable name that refers to a specific list of variables or array names into which the data is placed. Neither a dummy variable name nor a dummy array name may appear in the list. A specific list of variable or array names receives a NAMELIST name by use of a NAMELIST statement. The programmer need only use the NAMELIST name in the READ (a,x) statement to reference that list thereafter in the program.

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

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

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

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

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

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

4. A NAMELIST name may be defined only once by its appearance in a NAMELIST statement and must be so defined before its use. After it is defined in the NAMELIST statement, the NAMELIST name may appear only in input or output statements thereafter in the program.

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

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

Example:

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

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

Explanation:

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

When a READ statement references a NAMELIST name, input data in the form described in the following text is read from the designated input data set.

Input Data

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

The form the data items may take is:

1. Variable name = constant

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

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

   The set of constants may be in the form "k* constant" where k is an unsigned integer called the repeat constant. It represents the number of successive elements in the array to be initialized by the specified constant. The number of constants must be equal to the number of elements in the array.

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

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

Example:

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

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

Explanation:

If this data were input to be used with the NAMELIST and READ statements previously illustrated, the following actions would take place. The first data card would be read and examined to verify that its name (and the data items that follow) 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; the real constant 4.0 is placed in A(3). Also, since L is an array not followed by a subscript, the entire array is filled with the succeeding constants. Therefore, the integer constants 2 and 3 are placed in L(1) and L(2), respectively, and the integer constant 4 is placed in L(3), L(4),...,L(10).

The Form READ (a,b) list

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

Example 1:

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

```
      .
      .
      .
75 FORMAT (I10, I8, I9)
      .
      .
      .
   READ (J, 75) A, B, C
      .
      .
      .
```

Explanation:

The above READ statement causes input data from the data set associated with data set reference number J to be read into the locations A, B, and C according to the FORMAT statement referenced (statement 75). That is, the first 10 positions of the record are read into storage location A; the next 8 positions are read into storage location B; and the next 9 positions are read into storage location C.

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

Example 2:

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

Explanation:

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

Example 3:

```
         .
         .
         .
  98 FORMAT (I10,'HEADING')
         .
         .
         .
     READ (5,98)
         .
         .
         .
```

Explanation:

The above statements would cause the next record in the data set associated with data set reference number 5 to be skipped. No data is transferred into internal storage because there is no list item which corresponds with format code I10.

The Form READ (a) list

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

Example 1:

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

Explanation:

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

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

Example 2:

```
READ (5)
```

Explanation:

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

Indexing I/O Lists

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

```
15 FORMAT (F10.3)
     .
     .
     .
   READ (2,15) (A(I),I=1,5)
```

This is equivalent to:

```
15 FORMAT (F10.3)
     .
     .
     .
   DO 12 I = 1,5
12 READ (2,15) A(I)
```

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

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

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

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

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

would transmit data in the following order:

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

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

As another example, consider the following:

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

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

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

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

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

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

Reading Format Statements

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

For example, the following statements result in A, B, and the array C being read, converted, and stored according to the FORMAT specifications (2E10.8,5F10.8), which are read into the array FMT at object time:

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

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

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

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

WRITE STATEMENT

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

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

The basic forms of the WRITE statement involve formatted and unformatted data. They are:

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

The Form WRITE (a,x)

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

Example:

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

Explanation:

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

When a WRITE statement references a NAMELIST name:

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

2. The output data will be written such that:
   a. The fields for the data will be large enough to contain all the significant digits.
   b. The output can be read by an input statement referencing the NAMELIST name.

Example:

Assume that A is a 3 by 3 array.

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

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

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

The Form WRITE (a,b) list

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

Example 1:

In the following example, assume that the variables A, B, and C have been declared as integer variables.

```
75  FORMAT (I10, I8, I9)
     .
     .
     .
WRITE (J, 75) A, B, C
```

Explanation:

The above WRITE statement causes output data to be written in the data set associated with the data set reference number J, from the locations A, B, C, according to the FORMAT statement referred to (statement 75). That is, the 10 rightmost digits in A are written in the data set associated with the data set reference number J; the next 8 positions in the data set will contain the 8 rightmost digits in B; and the next 9 positions in the data set will contain the 9 rightmost digits in C.

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

Example 2:

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

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

Example 3:

```
98   FORMAT (I10, 'HEADING')
       .
       .
       .
     WRITE (5,98)
```

Explanation:

The above statements would cause a blank record to be written in the data set associated with data set reference number 5. No data is transferred into the data set because there is no list item which corresponds with the format code I10.

The Form WRITE (a) list

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

Example:

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

Explanation:

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

FORMAT STATEMENT

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

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

The format codes are:

G - to transfer integer real, complex, or logical data
I - to transfer integer data
F - to transfer real data that does not contain a decimal exponent
D - to transfer real data that contains a D decimal exponent
E - to transfer real data that contains an E decimal exponent
L - to transfer logical data
Z - to transfer hexadecimal data
A - to transfer alphameric data
Literal - to transfer a string of alphameric and special characters
H - to transfer literal data
X - to either skip data when reading or insert blanks when writing
T - to specify the position in a FORTRAN record where transfer of data is to start
P - to specify a scale factor

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

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

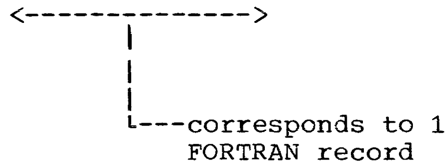
1. FORMAT statements are nonexecutable and may be placed anywhere in the source program.

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

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

      Example:

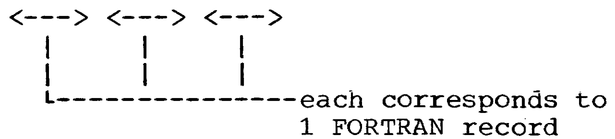
      ```
      xxxxx FORMAT (----, ----, ----)
      ```

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

      Example:

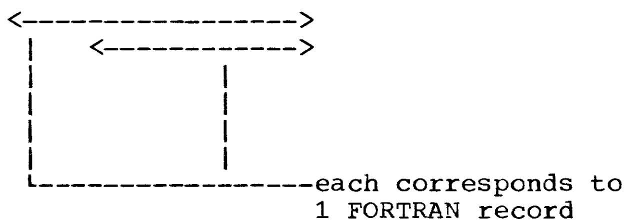
      ```
      xxxxx FORMAT (----/ ----/ ----)
      ```

   c. If more than one level of parentheses appear within a FORMAT statement, a FORTRAN record is defined by the beginning of the FORMAT statement to the end of the FORMAT statement. At this point, the definition of the FORTRAN record continues at the first-level left parenthesis that precedes the end of the FORMAT statement.

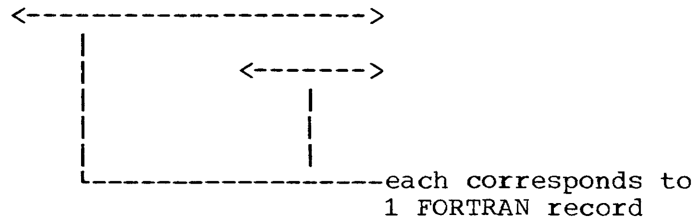
      Example 1:

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

Example 2:

```
                    0    1   1     1   1    0
xxxxx FORMAT (--- (---) --- (---) ---)
```



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

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

```
10  FORMAT (///I6)
        .
        .
        .
    READ (INPUT,10) MULT
```

   cause three records to be skipped on the data set associated with INPUT before data is read into MULT.

   The statements, where 'x' is a carriage control character (see, "Carriage Control"):

```
15  FORMAT ('x',I5,////'x',F5.2,I2//)
        .
        .
        .
    WRITE (IOUT,15) K,A,J
```

   result in the following output:

```
Integer
(blank line)
(blank line)
(blank line)
Real, Integer
(blank line)
(blank line)
```

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

```
10  FORMAT (F10.3,E12.4,F12.2)
        .
        .
        .
    WRITE (3,10) A,B,C,D,E,F,G
```

The following table shows the data transmitted in the column on the left and its corresponding format code.

Data Transmitted	Format Codes	
A	F10.3	first data record
B	E12.4	
C	F12.2	
D	F10.3	second data record
E	E12.4	
F	F12.2	
G	F10.3	third data record

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

```
(2F10.4)
```

is equivalent to:

```
(F10.4,F10.4)
```

6. A limited parenthetical expression is permitted to enable repetition of data fields according to certain format codes within a longer FORMAT statement. Two levels of parentheses, in addition to the parentheses required by the FORMAT statement, are permitted. The second level of parentheses facilitates the transmission of complex quantities.

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

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

G Format Code

General Form
aGw.s Where: a is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced. w is an unsigned integer constant specifying the total field length. s is an unsigned integer constant specifying the number of significant digits.

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

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

If the real data, say n, is in the range 0.1≤n<10**s where s is the s portion of the format code Gw.s, then this exponent field is blank. Otherwise, the real data is transferred with an E or D decimal exponent depending on the length specification (either 4 or 8 storage locations, respectively) of the real data.

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

Example 1:

Assume that the variables A, B, C, and D are of type real whose values are 292.7041, 82.43441, 136.7632, .8081945, respectively.

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

Explanation:

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

```
Print Position 1               Print Position 48
↑                                              ↑
bbb292.7bbbbbb82.434bbbbbb136.7bbbb.8081945bbbb
```

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

```
Print Position 1          Print Position 39
↑                         ↑
bbbb292.7bbbbbbb82.434bbbbbbbb136.7bbbb        Line 1
bbb0.8081bbbb                                  Line 2
```

From the above example, it can be seen that by increasing the field width reserved (w), blanks are inserted.

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

```
Print Position 1
↑
bbbb292.7bbbb                                  Line 1
bbbb82.43bbbb                                  Line 2
bbbb136.7bbbb                                  Line 3
bbb0.8081bbbb                                  Line 4
```

From the above example, it can be seen that the same format code was used for each variable in the list. Each repetition of the same format code caused a new line to be printed.

Example 2:

Assume that the variables I, J, K, and L are of type integer whose values are 292, 443428, 4908081, and 40018, respectively.

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

Explanation:

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

```
Print Position 1          Print Position 29
↑                         ↑
bbbbbbb292b443428490808140018                  Line 1
```

The same results may be achieved, had FORMAT statement 1 been written as follows:

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

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

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

```
Print Position 1
↑
bbb292                                            Line 1
443428                                            Line 2
******                                            Line 3
b40018                                            Line 4
```

Note that the second format code G6 is an incorrect specification for the third variable K, i.e., 4908081. Thus, the left-most digit is lost. In general, when the width specification w is insufficient, the left-most characters are not printed.

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

```
Print Position 1                   Print Position 40
↑                                          ↑
bbbbbbb292bbbb443428bbb4908081bbbbb40018          Line 1
```

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

Example 3:

Assume that the variable I is integer (length 2), A and B are real (length 4), D is real (length 8), C is complex (length 8), and L is logical (length 1) whose values are 292, 471.93, 81.91, 6.9310072, (2.1,3.7), and .TRUE. respectively.

```
  1  FORMAT  ('x',G3,2G9.2,G13.7,2G8.2,G3)
  2  FORMAT  ('x',G3/'x',2G10.2/'x',G9.1/'x',2G8.2,G3)
  3  FORMAT  (//'x',G3,2G9.2//'x',G13.7,2G8.2,G3///)
        .
        .
        .
     WRITE (5,n) I,A,B,D,C,L
        .
        .
        .
```

Explanation:

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

```
Print Position 1                           Print Position 53
↑                                                   ↑
292b0.47Eb03bb81.bbbbb6.931007bbbbb2.1bbbbb3.7bbbbbbT
```

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

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

```
Print Position 1
↑
292                                               Line 1
bb0.47Eb03bbb81.bbbb                              Line 2
bbb6.bbbb                                         Line 3
b2.1bbbbb3.7bbbbbbT                               Line 4
```

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

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

```
Print Position 1
↑
(blank line)                                      Line 1
(blank line)                                      Line 2
292b0.47Eb03bb81.bbbb                             Line 3
(blank line)                                      Line 4
b6.931007bbbbb2.1bbbbb3.7bbbbbbT                  Line 5
(blank line)                                      Line 6
(blank line)                                      Line 7
(blank line)                                      Line 8
```

In the above example, note that 2 consecutive slashes appearing at the beginning and 3 at the end of the series of format codes causes blank lines to be inserted as shown. However, the 2 consecutive slashes appearing elsewhere in the FORMAT statement causes the insertion of a blank line as shown in Line 4.

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

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

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

   is equivalent to:

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

2. If the data exists with a D decimal exponent, then it is transferred with this D decimal exponent.

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

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

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

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

As another example, consider the following statement:

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

If there are thirteen data items to be transmitted, then the printed output on a WRITE statement would take the following form:

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

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

Four types of format codes are available for the transfer of numeric data. These are specified in the following form:

General Form
<u>a</u>I<u>w</u> <u>a</u>F<u>w</u>.<u>d</u> <u>a</u>E<u>w</u>.<u>d</u> <u>a</u>D<u>w</u>.<u>d</u> Where: <u>a</u> is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced. I,F,E,D are format codes. <u>w</u> is an unsigned integer constant that is less than or equal to 255 and specifies the number of characters of data. <u>d</u> is an unsigned integer constant specifying the number of decimal places to the right of the decimal point, i.e., the fractional portion.

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

I Format Code

The I format code is used to transmit integer data. If the number of characters to be transmitted is greater than w, on input, the excess leftmost characters are lost; on output, asterisks are given. If the number of characters is less than w, on input, leading blanks are not significant, embedded and trailing blanks are treated as zeros; on output, the leftmost print positions are filled with blanks. If the quantity is negative, the position preceding the leftmost digit contains a minus sign. In this case, an additional position should be specified in w for the minus sign.

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

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

F Format Code

The F format code is used in conjunction with the transferral of real or double precision data that does not contain a decimal exponent. For F format codes, w is the total field length reserved and d is the number of places to the right of the decimal point (the fractional portion). The total field length reserved must include sufficient positions for a sign (if any) and a decimal point. The sign, if negative, is printed.

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

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

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

D and E Format Codes

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

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

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

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

Internal Value	Printed Value	
238.	b0.238Db03	
-.002	-0.200E-02	
.00000000004	b0.400D-10	
-21.0057	-0.210Eb02	(Last three digits of accuracy lost because of insufficient specification)

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

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

Z Format Code

General Form
aZw Where: a is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced. w is an unsigned integer constant that is less than or equal to 255 and specifies the number of hexadecimal digits to be read or written.

Hexadecimal numbers may be read or written by means of the format code Z*w*.

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

L Format Code

General Form
a*L*w Where: *a* is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced. *w* is an unsigned integer constant that is less than or equal to 255 and specifies the number of characters of data.

Logical variables may be read or written by means of the format code L*w*.

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

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

A Format Code

General Form
*a*A*w* Where: *a* is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced. *w* is an unsigned integer constant that is less than or equal to 255 and specifies the number of characters of data.

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

On input, if *w* is less than the length of the storage reserved for each item in the I/O list, *w* characters are read and the remaining right-most characters in the item are replaced with blanks. If *w* is greater than the length the number of characters equal to the difference between *w* and the length are skipped and the remaining characters are read.

On output, if *w* is less than the length of the storage reserved for each item, the printed line will consist of the left-most *w* characters of the item. If *w* is greater than the length the printed line will consist of the characters right-justified in the field and be preceded by blanks. Therefore it is important to always allocate enough area in storage to handle the characters being written (see the section "Specification Statements").

Example 1:

Assume that the array ALPHA consists of one subscript parameter ranging from 1 through 20. The following statements could be written to "copy" a record from one data set to another whose ultimate destination is a card punch.

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

Explanation:

In this example, the READ statement would cause 20 groups of characters to be read from the data set associated with data set reference number 5. Each group of four characters would be placed into one of the 20 positions in storage starting with ALPHA(1) and ending with ALPHA(20). The WRITE statement would cause the 20 groups of four characters to be written on the data set associated with data set reference number 6.

Example 2:

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

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

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

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

Explanation:

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

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

Literal Data in a Format Statement

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

```
25  FORMAT  (' 1964 INVENTORY REPORT')
```

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

```
DON''T
```

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

INPUT

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

For example, the statements:

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

would cause the next 9 characters to be read from the data set associated with data set reference number 3; these characters would replace the blank and the 8 characters H,E,A,D,I,N,G, and S in storage.

OUTPUT

All characters (including blanks) within the apostrophes are written as part of the output data. Thus, the statements:

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

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

```
THIS IS ALPHAMERIC DATA
```

H Format Code

General Form
wH Where: w is an unsigned integer constant that is less than or equal to 255 and specifies the number of characters following H.

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

The format code wH is followed in the FORMAT statement by w characters. For example,

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

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

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

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

X Format Code

General Form
wX Where: w is an unsigned integer constant that is less than or equal to 255 and specifies the number of blanks to be inserted on output or the number of characters to be skipped on input.

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

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

may be used, along with the appropriate READ statement.

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

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

may be used with an appropriate WRITE statement to print a line as follows:

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

T Format Code

General Form
Tw Where: w is an unsigned integer constant that is less than or equal to 255 and specifies the position in a FORTRAN record where the transfer of data is to begin.

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

```
5  FORMAT (T40, '1964 INVENTORY REPORT', T80, 'DECEMBER', T1, ' PART
   NO. 10095')
```

The preceding FORMAT statement would result in a printed line as follows:

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

The following statements:

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

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

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

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

Scale Factor - P

The representation of the data, internally or externally, may be modified by the use of a scale factor followed by the letter P preceding a format code. The scale factor is defined for input and output as follows:

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

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

INPUT

As another example, consider the following input data:

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

where b represents a blank.

The following statements:

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

would cause the variables in the list to assume the following values:

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

The following statements:

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

would cause the variables in the list to assume the following values:

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

The following statements:

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

would cause the variable in the list to assume the following values:

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

where the x represents an extraneous digit.

OUTPUT

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

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

then the following statements:

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

would cause the variables in the list to output the following values:

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

where the x represents an extraneous digit.

The following statements:

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

would cause the variables in the list to output the following values:

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

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

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

Then the statement FORMAT (1X,I2,1P3E13.3) used with the same WRITE statement results in the following printed output:

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

The statement FORMAT (1X,I2,-1P3E13.3) used with the same WRITE statement results in the following printed output:

```
27bbb-0.093Eb03bbb-0.017Eb04bbbb0.055Eb03
```

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

Carriage Control

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

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

The first character of the output record may be used for carriage control and is not printed. It appears in all other media as data.

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

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

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

END FILE STATEMENT

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

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

REWIND STATEMENT

General Form
REWIND a Where: a is an unsigned integer constant or integer variable that represents a data set reference number.

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

General Form
BACKSPACE *a* Where: *a* is an unsigned integer constant or integer variable that represents a data set reference number.

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

DIRECT ACCESS INPUT/OUTPUT STATEMENTS

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

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

DEFINE FILE Statement

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

General Form

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

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

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

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

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

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

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

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

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

Example:

```
DEFINE FILE 2(50,100,L,I2),3(100,50,L,J3)
```

This DEFINE FILE statement describes two data sets, referred to by data set reference numbers 2 and 3. The data in the first data set consists of 50 records, each with a maximum length of 100 storage locations. The L specifies that the data is to be transmitted either with or without format control. I2 is the associated variable that serves as a pointer to the next record.

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

If an E is substituted for the L in the preceding DEFINE FILE statement, a FORMAT statement is required and the data is transmitted

under format control. If the data is to be transmitted without format control, the DEFINE FILE statement can be written as:

```
DEFINE FILE 2(50,25,U,I2),3(100,13,U,J3)
```

Programming Considerations

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

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

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

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

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

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

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

To process a direct access data set without format control, the number of storage locations specified for each record must be greater than or equal to the maximum number of storage locations in a record to be written by any WRITE statement referencing the data set. For example, if the input/output 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, i.e., as a COMMON variable or as an argument. The following example shows how this can be accomplished.

```
    COMMON IUAR                              SUBROUTINE SUBI(A,B)
    DEFINE FILE 3(100,10,L,IUAR)             COMMON IUAR
         .                                        .
         .                                        .
         .                                        .
    ITEMP=IUAR
    CALL SUBI(ANS,ARG)
8   IF (IUAR-ITEMP) 20,16,20
         .
         .
         .
```

In this example, the program and the subprogram share the associated variable IUAR. An input/output operation that references data set 3 and is performed in the subroutine causes the value of the associated variable to be changed. The associated variable is then tested in the main program in statement 8.

READ Statement

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

General Form
READ (a'r, b, ERR=d) list Where: a is an integer constant or unsigned integer variable that 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. d is the statement number to which control is given when a device error condition is encountered during data transfer from device to storage. list is a series of variable or array names, separated by commas, that may be indexed and incremented. They specify the number of items to be read and the storage locations into which the data is to be placed. The list has the same forms and conventions as the list for the sequential READ statements.

Example:

```
    DEFINE FILE 1(500,100,L,ID1),2(100,28,L,ID2)
    DIMENSION M(10)
          .
          .
          .
    ID2 = 21
          .
          .
          .
10  FORMAT (5I20)
 9  READ (1'16,10) (M(K),K=1,10)
          .
          .
          .
13  READ (2'ID2+5) A,B,C,D,E,F,G
```

READ statement 9 transmits data from the data set associated with data set reference number 1, 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 2, 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 2 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 1(500,100,E,ID1),2(100,7,U,ID2)
```

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

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

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

WRITE Statement

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

General Form
WRITE (_a_'_r_,_b_) _list_ Where: _a_ is an integer constant or unsigned integer variable that 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 a series of variable or array names, separated by commas, that may be indexed or incremented. They specify the number of items to be written and the locations in storage from which the data is to be taken. The _list_ has the same forms and conventions as the I/O list for the sequential WRITE statements.

Example:

```
    DEFINE FILE 1(500,100,L,ID1),2(100,28,L,ID2)
    DIMENSION M(10)
            .
            .
            .
    ID2=21
            .
            .
            .
10  FORMAT (5I20)
 8  WRITE (1'16,10) (M(K),K=1,10)
            .
            .
            .
11  WRITE (2'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 1, 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 2, 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 previous example can also be written as

```
    DEFINE FILE 1(500,100,E,ID1),2(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 permits record retrieval to proceed concurrently with computation. By using the FIND statement, the user can increase the object program execution speed. There is no advantage to a FIND statement preceding a WRITE statement. The data set from which a record is being retrieved must be defined with a DEFINE FILE statement.

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

Example:

```
10  FIND (3'50)
      .
      .
      .
15  READ (3'50) A,B
```

While the statements between statements 10 and 15 are executed, record 50, in the data set associated with data set reference number 3, is retrieved.

General Example -- Direct Access Operations

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

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

The second DO loop in the example groups the data items from each array, as specified by the I/O list of the WRITE statement and FORMAT statement 15. Each group of data items is placed in the data set associated with data set reference number 8. Writing begins at the 505th record and continues at intervals of five, until record 1000 is written.

The specification statements provide the compiler with information about the nature of the data used in the source program. In addition, they supply the information required to allocate locations in storage for this data. Specification statements describing data may appear anywhere in the source program, but must precede any statements which refer to that data.

THE TYPE STATEMENTS

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

The IMPLICIT specification statement enables the user:

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

The Explicit specification statements enable the user:

1. to specify the type of a variable or array according to their particular name.
2. to specify the amount of storage to be allocated for each variable according to the associated type.
3. to specify the dimensions of an array.
4. to assign initial data values for variables and arrays.

IMPLICIT Statement

General Form
IMPLICIT <u>type</u>*<u>s</u>($\underline{a}_1$,$\underline{a}_2$,...),...,<u>type</u>*<u>s</u>($\underline{a}_1$,$\underline{a}_2$,...)
Where: <u>type</u> represents one of the following: INTEGER, REAL, COMPLEX, or LOGICAL.
*<u>s</u> is optional and represents one of the permissible length specifications for its associated type.
$\underline{a}_1$, $\underline{a}_2$,... represent single alphabetic characters each separated by commas, or a range of characters (in alphabetic sequence) denoted by the first and last characters of the range separated by a minus sign (e.g., (A-D)).

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

specifying that variables beginning with certain designated letters are of a certain type. Furthermore, the IMPLICIT statement allows the programmer to declare the number of locations to be allocated for each in the group of specified variables. The type a variable may assume, along with the permissible length specifications are as follows:

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

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

Example 1:

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

Explanation:

All variables beginning with the characters I through N are declared as INTEGER. Since no length specification was explicitly given (i.e., the *s was omitted), 4 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 4 storage locations allocated for each.

Note that the statement in Example 1 performs the exact 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 2 storage locations allocated for each. All variables beginning with the characters I through K are declared as real with 8 storage locations allocated for each. All variables beginning with the characters L, M, and N are declared as logical with 4 locations allocated for each.

Since the remaining letters of the alphabet, namely, O through Z and $, were left undefined by the IMPLICIT statement, the predefined convention will take effect. Thus, all variables beginning with the characters O through Z and $ are declared as real, each with a standard length of 4 locations.

Example 3:

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

Explanation:

All variables beginning with the characters C through F are declared as complex, each with 8 storage locations reserved for the real part of the complex data and 8 storage locations reserved for the imaginary part. The types of the variables beginning with the characters A, B, G through Z, and $ are determined by the predefined convention.

Explicit Specification Statements

General Form
Type*s $a{*}s_1(k_1)/x_1/,b{*}s_2(k_2)/x_2/,\ldots,z{*}s_n(k_n)/x_n/$
Where: Type is INTEGER, REAL, LOGICAL, or COMPLEX.
$*s,*s_1,*s_2,\ldots,*s_n$ are optional. Each s represents one of the permissible length specifications for its associated type.
a,b,...,z represent variable, array, or function names (see the section, "SUBPROGRAMS")
$(k_1),(k_2),\ldots,(k_n)$ are optional. Each k is composed of 1 through 7 unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array. Each k may be an unsigned integer variable only when the Type statement in which it appears is in a subprogram.
$/x_1/,/x_2/,\ldots,/x_n/$ are optional and represent initial data values.

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

Initial data values may be assigned to variables or arrays by use of $/x_n/$ where x_n is a constant or list of constants separated by commas. This set of constants may be in the form "r* constant", where r is an unsigned integer, called the repeat constant. An initial data value may not be assigned to a function name.

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

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

Example 1:

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

Explanation:

This statement declares that the variables ITEM and VALUE are of type integer, each with 2 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 8 storage locations reserved (4 for the real part, 4 for the imaginary part) and D is initialized to the value (2.1,4.7). In addition, 16 storage locations are reserved for the variable E. Thus, if a length specification is explicitly written, it overrides the assumed standard length.

Example 3:

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

Explanation:

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

Example 4:

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

Explanation:

This statement declares the size of each array, A and B, and their type (real). The array A has 100 storage locations reserved (4 for each variable in the array) and the array B has 400 storage locations reserved (4 for each variable). In addition, the first 20 variables in the array A are initialized to the value 6.9E2 and the last 5 variables are initialized to the value 1.0. All 100 variables in the array B are initialized to the value 0.0. The array TEST has 40 storage locations reserved (8 for each variable). In addition, each variable is initialized to the value 0.0.

Adjustable Dimensions

As shown in the previous examples, the maximum value of each subscript in an array was specified by a numeric value. These numeric values (maximum value of each subscript) are known as the absolute dimensions of an array and may never be changed. However, if an array is used in a subprogram (see the section, "Subprograms") and is not in Common, the size of this array does not have to be explicitly declared in the subprogram by a numeric value. That is, the Explicit specification statement, appearing in a subprogram, may contain integer variables of length 4 that specify the size of the array. These integer variables must be either actual or implicit subprogram arguments. When the subprogram is called, these integer variables then receive their values from the calling program. Thus, the dimensions (size) of a dummy array appearing in a subprogram are adjustable and may change each time the subprogram is called. Integer variables that provide dimension information may not be redefined within the subprogram.

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

The following example illustrates the use of adjustable dimensions:

Example:

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

Explanation:

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

```
R(1,1) R(2,1) R(1,2) R(2,2) R(1,3) R(2,3)

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

Thus, in the calling program the subscripted variable A(1,2) refers to the sixth subscripted variable in the array A. However, in the subprogram MAPMY the subscripted variable R(1,2) refers to the third

subscripted variable in the array A, namely, A(3,1). This is so because the dimensions of the array R as declared in the subprogram are not the same as those in the calling program.

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

The numbers 2 and 3, which became the adjusted dimension of the dummy array R, could also have been variables in the argument list or implicit arguments in a COMMON block. For example, assume that the following statement appeared in the calling program:

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

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

Dummy variables (e.g., L and M) may be used as dimensions of an array only in a FUNCTION or SUBROUTINE subprogram.

ADDITIONAL SPECIFICATION STATEMENTS

DIMENSION Statement

General Form
DIMENSION $\underline{a}_1(\underline{k}_1)$,$\underline{a}_2(\underline{k}_2)$, $\underline{a}_3(\underline{k}_3)$,...,$\underline{a}_n(\underline{k}_n)$ Where: $\underline{a}_1$, $\underline{a}_2$, $\underline{a}_3$,..., $\underline{a}_n$ are array names. $\underline{k}_1$, $\underline{k}_2$, $\underline{k}_3$,...,$\underline{k}_n$ are each composed of 1 through 7 unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array. $\underline{k}_1$ through $\underline{k}_n$ may be integer variables of length 4 only when the DIMENSION statement in which they appear is in a subprogram.

The information necessary to allocate storage for arrays used in the source program may be provided by the DIMENSION statement. Allocation information should be given to an array on its first appearance in a source program; however, for subprograms, the SUBROUTINE or FUNCTION statement may include a dummy argument that is dimensioned later. 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)
```

COMMON Statement

General Form	
COMMON	$/\underline{r}/\underline{a}\ (\underline{k}_1),\underline{b}(\underline{k}_2),\ldots/\underline{r}/\underline{c}(\underline{k}_3),\underline{d}(\underline{k}_4),\ldots$
Where:	$\underline{a},\underline{b},\ldots,\underline{c},\underline{d}\ldots$ are variable or array names.
	$\underline{k}_1,\underline{k}_2,\ldots\underline{k}_3,\underline{k}_4\ldots$ are optional and are each composed of 1 through 7 unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.
	$/\underline{r}/\ldots$ represent optional common block names consisting of 1 through 6 alphameric characters, the first of which is alphabetic. These names must always be embedded in slashes.

Although the COMMON statement may be used to provide dimension information, adjustable dimensions may never be used.

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

```
COMMON TABLE
```

and a second program contains the statement:

```
COMMON TREE
```

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

If the main program contains the statements:

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

and a subprogram contains the statements:

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

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

Common entries appearing in COMMON statements are cumulative in the given order throughout the program; that is, they are cumulative in the sequence in which they appear in all COMMON statements. For example, consider the following two COMMON statements:

```
COMMON A, B, C
COMMON G, H
```

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

```
COMMON A, B, C, G, H
```

Redundant entries are not allowed. For example, the following statement is invalid:

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

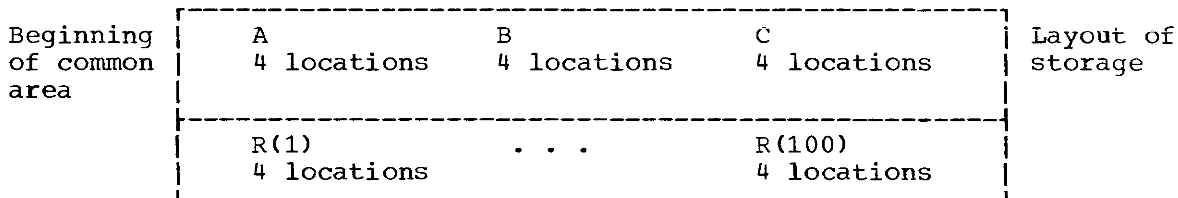
Consider the following example:

Example:

```
Calling Program              Subprogram

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

Explanation:

In the calling program, the statement COMMON A,B,C,R(100) would cause 412 storage locations (4 locations per variable) to be reserved in the following order:



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

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

Blank and Labeled Common

In the preceding example, the common storage area (common block) established is called a blank common area. That is, no particular name was given to that area of storage. The variables that appeared in the COMMON statements were assigned locations relative to the beginning of this blank common area. However, variables and arrays may be placed in separate common areas. Each of these separate areas (or blocks) is given a name consisting of 1 through 6 alphameric characters (the first of which is alphabetic); those blocks which have the same name occupy the same storage space.

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

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

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

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

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

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

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

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

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

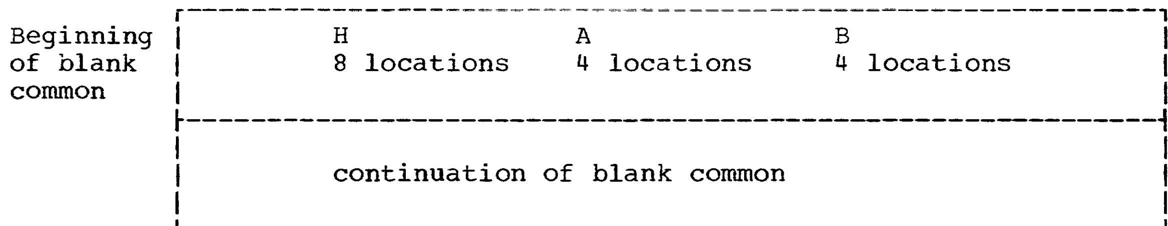
Example:

Assume that A, B, C, K, X, and Y each occupy 4 locations of storage, H and G each occupy 8 locations, and D and E each occupy 2 locations.

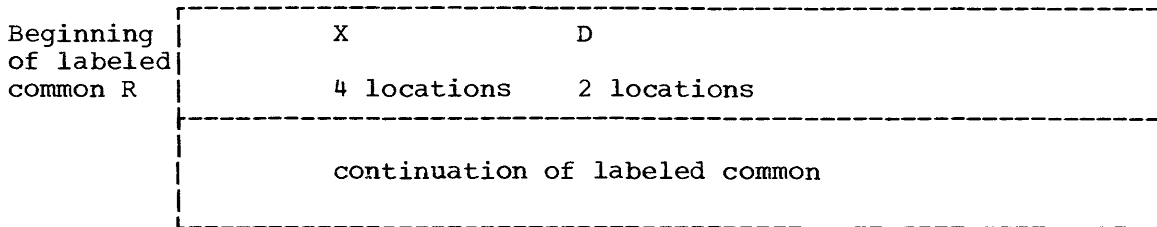
```
Calling Program                Subprogram

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

Explanation:

In the calling program, the statement COMMON H, A /R/ X, D //B causes 16 locations (4 locations each for A and B, and 8 for H) to be reserved in blank common in the following order:



and also causes 6 locations (4 for X and 2 for D) to be reserved in the labeled common area R in the following order:



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

Programming Considerations

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

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

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

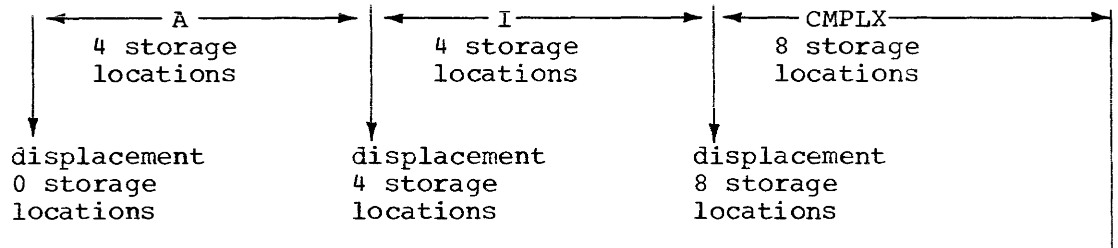
If the fixed order is not used, proper alignment can be ensured by constructing the block so that the displacement of each variable can be evenly divided by the reference number associated with the variable. (Displacement is the number of storage locations from the beginning of the block to the first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

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

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

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

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



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

EQUIVALENCE Statement

General Form
EQUIVALENCE (<u>a</u>, <u>b</u>, <u>c</u>, ...), (<u>d</u>, <u>e</u>, <u>f</u>,...)
Where: <u>a</u>, <u>b</u>, <u>c</u>, <u>d</u>, <u>e</u>, <u>f</u>,... are variables that may be subscripted. The subscripts may have two forms: If the variable is singly subscripted it refers to the position of the variable in the array (i.e., first variable, 25th variable, etc). If the variable is multi-subscripted it refers to the position in the array in the same fashion as the position is referred to in an arithmetic statement.

The EQUIVALENCE statement provides the option for controlling the allocation of data storage within a single program or subprogram. It is analogous to (but not identical to) the option of using the COMMON statement to control the allocation of data storage among several programs. In particular, when the logic of the program permits it, the number of storage locations used can be reduced by causing locations to be shared by two or more variables of the same or different types. Equivalence between variables implies storage sharing only, not mathematical equivalence.

<u>Example 1</u>:

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

Explanation:

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

Example 2:

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

Explanation:

This EQUIVALENCE statement indicates that the variable A, the first variable in the array B, namely B(1), and the 25th variable in the array C, namely C(5,3), are to be assigned the same storage locations. In addition, it also specifies that D(100) (i.e., D(5,10,2)) and E are to share the same storage locations. Note that the effect of the EQUIVALENCE statement in Examples 1 and 2 is the same.

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

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

This would cause a common area to be established containing the variables A, B, and C. The EQUIVALENCE statement would then cause the variable D(1) to share the 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)
```

Since arrays are stored in consecutive forward locations, a variable may not be made equivalent to another variable of an array in such a way as to cause the array to extend before the beginning of the common area. For example, 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)
```

Programming Considerations

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

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

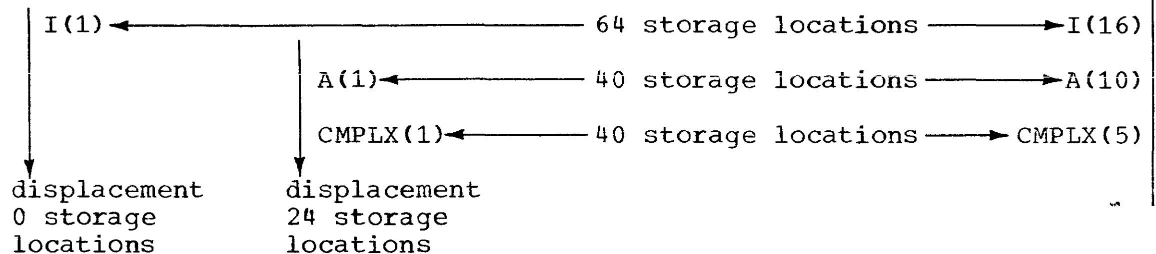
length of 16 (complex)
length of 8 (complex or real)
length of 4 (real or integer or logical)
length of 2 (integer)
length of 1 (logical)

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

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

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

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



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

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

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

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

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

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

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

NAMING SUBPROGRAMS

A subprogram name consists of from 1 through 6 alphameric characters, the first of which must be alphabetic. A subprogram name may not contain special characters (see Appendix A).

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

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

3. Type Declaration of a SUBROUTINE Subprogram: The type of a SUBROUTINE subprogram can not be defined because the results that are returned to the calling program are dependent only on the type of the variable names appearing in the argument list of the calling program and/or the implicit arguments in COMMON.

FUNCTIONS

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

1. Define the function (i.e., specify what calculations are to be performed).
2. Refer to the function by name where required in the program.

Function Definition

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

1. The function must be assigned a unique name by which it may be called (see the section "Naming Subprograms").
2. The arguments of the function must be stated.
3. The procedure for evaluating the function must be stated.

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

Function Reference

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

STATEMENT FUNCTIONS

Statement functions are defined by a single arithmetic statement within the program in which they appear. For 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 function arguments.

The expression on the right defines those computations which are to be performed when the function is used in an arithmetic statement. This function might be used in a statement as follows:

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

which is equivalent to:

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

Note the correspondence between A and B in the function definition statement and D and E in the arithmetic statement. The quantities A and B enclosed in parentheses following the function name are the arguments

of the function. They are dummy variables for which the quantities D and E, respectively are substituted when the function is used in an arithmetic statement.

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

The actual arguments must correspond in order, number, and type to the dummy arguments. There must be at least one argument.

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

Examples:

Valid statement function definitions:

```
SUM(A,B,C,D) = A+B+C+D
FUNC(Z) = A+X*Y*Z
AVG(A,B,C,D) = (A+B+C+D)/4
ROOT(A,B,C) = SQRT(A**2+B**2+C**2)
```

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

Invalid statement function definitions:

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

Valid statement function references:

```
NET = GROS - SUM(TAX, FICA, HOSP, STOCK)
ANS = FUNC(RESULT)
GRADE = AVG(ALAB, TERM, SUM(TEST1, TEST2, TEST3, TEST4), FACTOR)
```

Invalid statement function references:

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

FUNCTION SUBPROGRAMS

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

General Form
FUNCTION name ($a_1, a_2, a_3, \ldots, a_n$) . . . RETURN . . . END Where: name is subprogram name (see the section "Naming Subprograms"). $a_1, a_2, a_3, \ldots, a_n$ are nonsubscripted variable or array names, or the dummy names of SUBROUTINE or other FUNCTION subprograms. (There must be at least one argument in the argument list.)

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

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

The arguments of the FUNCTION subprogram (i.e., $a_1, a_2, a_3, \ldots, a_n$) may be considered to be dummy variable names. These are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. The actual arguments may be any of the following: any type of constant except literal or hexadecimal, any type of subscripted or nonsubscripted variable, an array name, an arithmetic expression, or the name of another subprogram. The actual arguments must correspond in number, order, type, and length to the dummy arguments. For example, if the actual argument is an integer constant, then the dummy argument in the FUNCTION statement must be of length 4. The array size must also be the same, except when adjustable dimensions are used.

The name of the FUNCTION subprogram cannot be typed with an Explicit specification statement in the subprogram.

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

Example 1:

```
    Calling Program                 FUNCTION Subprogram
          .
          .                         FUNCTION SOMEF(X,Y)
          .                         SOMEF = X/Y
    A = SOMEF(B,C)                  RETURN
          .                         END
          .
          .
```

Explanation:

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

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

Example 2:

```
    Calling Program                 FUNCTION Subprogram

                                    FUNCTION CALC (A,B,J)
                                               .
          .                                    .
          .                                    .
          .                         I = J*2
    ANS = ROOT1*CALC(X,Y,I)                    .
          .                                    .
          .                                    .
          .                         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.

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

Type Specification of the FUNCTION Subprogram

In addition to declaring the type of a FUNCTION name by either the predefined convention or the IMPLICIT statement, there exists the option of explicitly specifying the type of a FUNCTION name within the FUNCTION statement.

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

Example 1:

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

Example 2:

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

Explanation:

The FUNCTION subprograms SOMEF and CALC in Examples 1 and 2 are declared as type REAL and INTEGER, respectively.

RETURN and END Statements in a Function Subprogram

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

Example:

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

SUBROUTINE SUBPROGRAMS

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

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

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

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

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

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

The arguments ($\underline{a}_1$, $\underline{a}_2$, $\underline{a}_3$,...,$\underline{a}_n$) may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. The actual arguments must correspond in number, order, type, and length to the dummy arguments. The array size must also be the same. Dummy arguments may not appear in an EQUIVALENCE or COMMON statement within the subprogram.

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

```
Calling Program                 SUBROUTINE Subprogram

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

CALL Statement

The CALL statement is used to call a SUBROUTINE subprogram.

General Form
CALL name ($\underline{a}_1$,$\underline{a}_2$,$\underline{a}_3$,...,$\underline{a}_n$) Where: name is the name of a subroutine subprogram. $\underline{a}_1$,$\underline{a}_2$,$\underline{a}_3$,...,$\underline{a}_n$ are the actual arguments that are being supplied to the subroutine subprogram. Each may be of the form &n where n is a statement number (see, "RETURN statements in a SUBROUTINE Subprogram").

Examples:

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

The CALL statement transfers control to the subroutine subprogram and replaces the dummy variables with the value of the actual arguments that appear in the CALL statement. The arguments in a CALL statement may be any of the following:

1. Any type of constant except hexadecimal
2. Any type of subscripted or nonsubscripted variable
3. An array name
4. An arithmetic expression
5. The name of a FUNCTION or SUBROUTINE subprogram
6. A statement number (see the section, "RETURN Statements in a SUBROUTINE Subprogram).

The arguments in a CALL statement must agree in number, order, type, and length with the corresponding arguments in the subroutine subprogram. For example, if the actual argument is an integer constant, then the dummy argument in the SUBROUTINE statement must be of length 4. The array sizes must also be the same in the subroutine and the calling programs except when adjustable dimensions are used (see the section "Adjustable Dimensions"). If an actual argument corresponds to a dummy argument that is defined or re-defined in the referenced subprogram, the actual argument must be a variable name, subscripted variable name, or array name.

If a literal constant is passed as an argument, the actual argument passed is the literal as defined, without delimiting apostrophes or the preceding _w_H specification.

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

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

and if the following statements are included in the calling source program

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

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

ARGUMENTS IN A FUNCTION AND SUBROUTINE SUBPROGRAM

Arguments may be referred to in a subprogram in one of two ways: by value or by name. The method of reference depends on the nature of the dummy argument.

Reference by value: The value of the actual argument is brought from the calling program to the subprogram. This value is loaded into the location of the corresponding dummy argument. (During execution, all intermediate values are also stored in this location.) Upon return to the calling program, the final value is transmitted from the dummy argument to the actual argument.

An argument is referenced by value when the corresponding dummy argument is enclosed only in commas.

Reference by name: The address of the actual argument is brought to the subprogram. During execution of the subprogram, all intermediate values and the final value are referenced using this address.

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

All arithmetic or logical expressions that appear in the argument list of the calling program are evaluated and placed in a temporary storage location. It is either the value or the address of this storage location which is referenced by value if the dummy arguments are separated only by commas, and by name if each is enclosed in slashes.

Example:

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

Explanation:

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

Example:

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

Explanation:

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

RETURN Statement in a SUBROUTINE Subprogram

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

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

Example:

```
       Calling Program                  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
    30 Y = A + C                        END
          .
          .
          .
    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.

Multiple ENTRY into a Subprogram

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

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

General Form
ENTRY <u>name</u> ($\underline{a}_1, \underline{a}_2, \underline{a}_3, \ldots, \underline{a}_n$) Where: <u>name</u> is the name of an entry point (see the section, "Naming Subprograms"). $\underline{a}_1, \underline{a}_2, \underline{a}_3, \ldots, \underline{a}_n$ are the dummy arguments corresponding to an actual argument in a CALL statement or in a function reference.

ENTRY statements do not affect control sequencing during normal execution of a subprogram. The order, type, and number of arguments need not agree between the SUBROUTINE or FUNCTION statement and the ENTRY statements, nor do the ENTRY statements have to agree among themselves in these respects. Each CALL or function reference however, must agree in order, type, and number with the SUBROUTINE, FUNCTION, or ENTRY statement that it references. Entry may not be made into the range of a DO; further, a subprogram may not reference itself directly or through any of its entry points.

<u>Example 1</u>:

<u>Calling Program</u>	<u>Subprogram</u>
. . .	SUBROUTINE SUB1 (U,V,W,X,Y,Z) . . .
1 CALL SUB1 (A,B,C,D,E,F) . . .	U = V . . .
2 CALL SUB2 (G,H,P) . . .	ENTRY SUB2 (T,U,V) . . .
3 CALL SUB3 . . .	ENTRY SUB3 . . . END

Explanation:

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

Entry into a subprogram initializes all references in the whole subprogram to items in the argument list of the referenced ENTRY. Return from a subprogram is made by way of the entry point referenced. ENTRY statements may only appear in FUNCTION or SUBROUTINE subprograms. The following is a valid example:

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

Example 2:

```
    Calling Program                  Subprogram

         .                           SUBROUTINE SUB1 (U,V,W,X,Y,Z)
         .                           RETURN
         .                           ENTRY SUB2 (T,*,*)
    CALL SUB1 (A,B,C,D,E,F)          U = V* W+T
         .                           ENTRY SUB3 (*,*)
         .                           X = Y**Z
         .                      50   IF (J-K) 100, 200, 300
    CALL SUB2(G,&10,&20)        100  RETURN 1
         .                      200  RETURN 2
         .                      300  RETURN
         .                           END
    CALL SUB3(&10,&20)
5   Y =A+B
         .
         .
         .
10  Y = C+D
20  Y = E+F
         .
         .
         .
```

Explanation:

In the example above, a call to SUB1 merely performs initialization. Subsequent calls to SUB2 and SUB3 result in execution of different sections of the subroutine SUB1. Then, depending upon the result of the arithmetic IF at statement 50, return is made to the calling program at statement 10, 20, or 5.

Additional Rules for Using ENTRY

1. Reference to an ENTRY will not transmit new values for arguments which are referenced by value at some previous use of the subprogram unless those arguments are in the argument list of this ENTRY.

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

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

4. If a dummy argument is listed at more than one entry, whenever it appears the dummy argument must be referenced consistently either by name or by value.

5. A name which is defined as a dummy argument name may not appear in any executable statement unless it has been previously defined (as a dummy argument) in an ENTRY, SUBROUTINE, or FUNCTION statement.

EXTERNAL Statement

General Form
EXTERNAL _a_,_b_,_c_,...   Where: _a_,_b_,_c_,... are names of subprograms that are passed as arguments to other subprograms.

If a FORTRAN supplied in-line function is used in an EXTERNAL statement, it is not expanded in-line. It is assumed that the function is part of a library. (The FORTRAN supplied in-line and out-of-line functions are given in Appendix C.)

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

Example 1:

Calling Program

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

Subprogram

```
   SUBROUTINE SUB(K,Y,Z)
   IF (K) 4,6,6
4  D = Y (K,Z**2)
     .
     .
     .
6  RETURN
   END
```

Explanation:

In this example, the subprogram name MULT is used as an argument in the subprogram SUB. The subprogram name MULT is passed to the dummy variable Y 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:

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

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.

BLOCK DATA SUBPROGRAM

In order to initialize variables in a COMMON block, a separate subprogram must be written. This separate subprogram contains only the DATA, COMMON, DIMENSION, EQUIVALENCE, and Type statements associated with the data being defined. Data may be initialized in labeled (named), but not unlabeled, COMMON by the BLOCK DATA subprogram.

General Form
BLOCK DATA . . . END

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

2. The 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.

3. 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
```

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

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

The 49 characters listed above comprise 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 IV languages that are incorporated into the System/360 FORTRAN IV language. The inclusion of these additional language facilities allows existing FORTRAN programs to be re-compiled for use on the IBM System/360 with little or no re-programming.

READ Statement

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

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

PUNCH Statement

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

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

PRINT Statement

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

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

DATA Initialization Statement

General Form
DATA $\underline{v}_1,\ldots,\underline{v}_n/\underline{i}_1*\underline{d}_1,\ldots,\underline{i}_n*\underline{d}_n/,\underline{v}_{n+1},\ldots,\underline{v}\ /\underline{i}_{n+1}*\underline{d}_{n+1},\ldots,\underline{i}\ *\underline{d}\ /,\ldots$ Where: $\underline{v}_1,\ldots,\underline{v}*$ are variables, subscripted variables (in which case the subscripts must be integer constants), or array names. $\underline{d}_1,\ldots,\underline{d}*$ are values representing integer, real, complex, hexadecimal, logical, or literal data constants. $\underline{i}_1,\ldots,\underline{i}*$ represent unsigned integer constants indicating the number of consecutive variables that are to be assigned the value of $\underline{d}_1,\ldots,\underline{d}$.

A data initialization statement is used to define initial values of variables, array elements, and arrays. There must be a one-to-one correspondence between these variables (i.e., $\underline{v}_1,\ldots,\underline{v}*$) and the data constants (i.e., $\underline{d}_1,\ldots,\underline{d}*$).

Example 1:

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

Explanation:

The DATA statement indicates that the variables A, B, and C are to be initialized to the values 5.0, 6,1, and 7.3 respectively. In addition, the statement specifies that the first 25 variables in the array D are to be initialized to the value 1.0.

Example 2:

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

Explanation:

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

An initially defined variable, or variable of an array, may not be in blank common. However, in a labeled common block, they may be initially defined only in a block data subprogram. (See the section, "SUBPROGRAMS.")

DOUBLE PRECISION Statement

General Form
DOUBLE PRECISION $\underline{a}(\underline{k}_1),\underline{b}(\underline{k}_2),\ldots,\underline{z}(\underline{k}_n)$ Where: $\underline{a},\underline{b},\ldots,\underline{z}$ represent variable, array, or function names (see the section, "SUBPROGRAMS") $(\underline{k}_1),(\underline{k}_2),\ldots,(\underline{k}_n)$ are optional. Each $\underline{k}$ is composed of 1 through 7 unsigned integer constants, separated by commas, that represent the maximum value of each subscript in the array.

The DOUBLE PRECISION statement explicitly specifies that the variables $\underline{a},\underline{b},\underline{c},\ldots$ are of type double precision. This statement overrides any specification of a variable made by either the predefined convention or the IMPLICIT statement. This specification is identical to that of type REAL*8.

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

DOUBLE PRECISION FUNCTION $\underline{name}$ $(\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n)$

The FORTRAN supplied subprograms are of either of two types: mathematical subprograms and service subprograms. The mathematical subprograms correspond to a FUNCTION subprogram; the service subprograms correspond to a SUBROUTINE subprogram. Appendix C lists the in-line and out-of-line mathematical FUNCTION subprograms. An in-line subprogram is inserted by the FORTRAN compiler at any point in the program where the function is referenced. An out-of-line subprogram is located on a library. A detailed description of out-of-line mathematical subprograms and service subprograms is given in the publication IBM System/360 Operating System: FORTRAN IV (E), Library Subprograms.

• Table 4. Mathematical Function Subprograms

Function	Entry Name	Definition	In-Line (I) Out-of-Line (O)	No. of Arg.	Type of Arguments	Type of Function Value
Exponential	EXP	e^{arg}	O	1	Real *4	Real *4
	DEXP	e^{arg}	O	1	Real *8	Real *8
	CEXP	e^{arg}	O	1	Complex *8	Complex *8
	CDEXP	e^{arg}	O	1	Complex *16	Complex *16
Natural Logarithm	ALOG	ln (Arg)	O	1	Real *4	Real *4
	DLOG	ln (Arg)	O	1	Real *8	Real *8
	CLOG	ln (Arg)	O	1	Complex *8	Complex *8
	CDLOG	ln (Arg)	O	1	Complex *16	Complex *16
Common Logarithm	ALOG10	$\log_{10}$(Arg)	O	1	Real *4	Real *4
	DLOG10	$\log_{10}$(Arg)	O	1	Real *8	Real *8
Arcsine	ARSIN	arcsin (Arg)	O	1	Real *4	Real *4
	DARSIN	arcsin (Arg)	O	1	Real *8	Real *8
Arccosine	ARCOS	arccos (Arg)	O	1	Real *4	Real *4
	DARCOS	arccos (Arg)	O	1	Real *8	Real *8
Arctangent	ATAN	arctan (Arg)	O	1	Real *4	Real *4
	ATAN2	arctan (Arg_1/Arg_2)	O	2	Real *4	Real *4
	DATAN	arctan (Arg)	O	1	Real *8	Real *8
	DATAN2	arctan (Arg_1/Arg_2)	O	2	Real *8	Real *8
Trigonometric Sine (Argument in radians)	SIN	sin(Arg)	O	1	Real *4	Real *4
	DSIN	sin(Arg)	O	1	Real *8	Real *8
	CSIN	sin(Arg)	O	1	Complex *8	Complex *8
	CDSIN	sin(Arg)	O	1	Complex *16	Complex *16
Trigonometric Cosine (Argument in radians)	COS	cos(Arg)	O	1	Real *4	Real *4
	DCOS	cos(Arg)	O	1	Real *8	Real *8
	CCOS	cos(Arg)	O	1	Complex *8	Complex *8
	CDCOS	cos(Arg)	O	1	Complex *16	Complex *16
Trigonometric Tangent (Argument in radians)	TAN	tan (Arg)	O	1	Real *4	Real *4
	DTAN	tan (Arg)	O	1	Real *8	Real *8
Trigonometric Cotangent (Argument in radians)	COTAN	cotan (Arg)	O	1	Real *4	Real *4
	DCOTAN	cotan (Arg)	O	1	Real *8	Real *8
Square Root	SQRT	$(Arg)^{1/2}$	O	1	Real *4	Real *4
	DSQRT	$(Arg)^{1/2}$	O	1	Real *8	Real *8
	CSQRT	$(Arg)^{1/2}$	O	1	Complex *8	Complex *8
	CDSQRT	$(Arg)^{1/2}$	O	1	Complex *16	Complex *16
Hyperbolic Tangent	TANH	tanh(Arg)	O	1	Real *4	Real *4
	DTANH	tanh(Arg)	O	1	Real *8	Real *8

(Continued)

• Table 4. Mathematical Function Subprograms (Continued)

Function	Entry Name	Definition	In-Line (I) Out-of-Line (O)	No. of Arg.	Type of Arguments	Type of Function Value
Largest value	AMAX0	Max $(Arg_1, Arg_2, \ldots)$	O	≥2	Integer *4	Real *4
	AMAX1		O	≥2	Real *4	Real *4
	MAX0		O	≥2	Integer *4	Integer *4
	MAX1		O	≥2	Real *4	Integer *4
	DMAX1		O	≥2	Real *8	Real *8
Smallest value	AMIN0	Min $(Arg_1, Arg_2, \ldots)$	O	≥2	Integer *4	Real *4
	AMIN1		O	≥2	Real *4	Real *4
	MIN0		O	≥2	Integer *4	Integer *4
	MIN1		O	≥2	Real *4	Integer *4
	DMIN1		O	≥2	Real *8	Real *8
Float	FLOAT	Convert from integer to real	I	1	Integer *4	Real *4
	DFLOAT		I	1	Integer *4	Real *8
Fix	IFIX	Convert from real to integer	I	1	Real *4	Integer *4
	HFIX		I	1	Real *4	Integer *2
Transfer of sign	SIGN	Sign of Arg_2 times $\lvert Arg_1 \rvert$	I	2	Real *4	Real *4
	ISIGN		I	2	Integer *4	Integer *4
	DSIGN		I	2	Real *8	Real *8
Positive difference	DIM	$Arg_1 - Min(Arg_1, Arg_2)$	I	2	Real *4	Real *4
	IDIM				Integer *4	Integer *4
Hyperbolic Sine	SINH	sinh (Arg)	O	1	Real *4	Real *4
	DSINH	sinh (Arg)	O	1	Real *8	Real *8
Hyperbolic Cosine	COSH	cosh (Arg)	O	1	Real *4	Real *4
	DCOSH	cosh (Arg)	O	1	Real *8	Real *8
Error Function	ERF	$\frac{2}{\pi}\int_0^x e^{-u^2}\,du$	O	1	Real *4	Real *4
	DERF		O	1	Real *8	Real *8
Complemented Error Function	ERFC	1-erf (x)	O	1	Real *4	Real *4
	DERFC		O	1	Real *8	Real *8
Gamma	GAMMA	$\int_0^\infty u^{x-1} e^{-u}\,du$	O	1	Real *4	Real *4
	DGAMMA		O	1	Real *8	Real *8
Log-gamma	ALGAMA	$\log_e \Gamma(x)$	O	1	Real *4	Real *4
	DLGAMA		O	1	Real *8	Real *8
Modular Arithmetic	MOD	Arg_1 (mod Arg_2)	I	2	Integer *4	Integer *4
	AMOD	$Arg_1 - \left[\frac{Arg_1}{Arg_2}\right] * Arg_2$	I	2	Real *4	Real*4
	DMOD	Where: [x] is the largest integer≤|x|	I	2	Real *8	Real *8
Absolute value	IABS	|Arg|	I	1	Integer *4	Integer *4
	ABS		I	1	Real *4	Real *4
	DABS		I	1	Real *8	Real *8
	CABS	(a^2+b^2) for a+bi	O	1	Complex *8	Real *4
	CDABS		O	1	Complex *16	Real *8
Truncation	INT	Sign of Arg times largest integer <|Arg|	I	1	Real *4	Integer *4
	AINT		I	1	Real *4	Real *4
	IDINT		I	1	Real *8	Integer *4

(Continued)

• Table 4. Mathematical Function Subprograms (Continued)

Function	Entry Name	Definition	In-Line (I) Out-of-Line (O)	No. of Arg.	Type of Arguments	Type of Function Value
Obtaining most significant part of a Real *8 argument	SNGL		I	1	Real *8	Real *4
Obtain real part of complex argument	REAL		I	1	Complex *8	Real *4
Obtain imaginary part of complex argument	AIMAG		I	1	Complex *8	Real *4
Express a Real *4 argument in Real *8 form	DBLE		I	1	Real *4	Real *8
Express two real arguments in complex form	CMPLX DCMPLX	$C=Arg_1,+iArg_2$	I I	2 2	Real *4 Real *8	Complex *8 Complex *16
Obtain conjugate of a complex argument	CONJG DCONJG	C=X-iY For Arg=X+iY	I I	1 1	Complex *8 Complex *16	Complex *8 Complex *16

SAMPLE PROGRAM 1

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

IBM FORTRAN Coding Form

PROGRAM: SAMPLE PROGRAM 1 — PROGRAMMER: — DATE: 6/66 — PUNCHING INSTRUCTIONS: GRAPHIC / PUNCH — PAGE 1 OF 1 — CARD ELECTRO NUMBER*

```
C     PRIME NUMBER PROBLEM
  100 WRITE (6,8)
    8 FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
     119X,1H1/19X,1H2/19X,1H3)
  101 I=5
    3 A=I
  102 A=SQRT(A)
  103 J=A
  104 DO 1 K=3,J,2
  105 L=I/K
  106 IF(L*K-I)1,2,4
    1 CONTINUE
  107 WRITE (6,5)I
    5 FORMAT (I20)
    2 I=I+2
  108 IF(1000-I)7,4,3
    4 WRITE (6,9)
    9 FORMAT (14H PROGRAM ERROR)
    7 WRITE (6,6)
    6 FORMAT (31H THIS IS THE END OF THE PROGRAM)
  109 STOP
      END
```

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

• Figure 2. Sample Program 1

SAMPLE PRCGRAM 2

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

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

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

$$\begin{array}{ll}
(1) & W_0a_0 + W_1a_1 + \ldots + W_ma_m = Z_0 \\
(2) & W_1a_0 + W_2a_1 + \ldots + W_{m+1}a_m = Z_1 \\
\cdot & \\
\cdot & \\
\cdot & \\
(m+1) & W_ma + W_{m+1}a_1 + \ldots + W_{2m}a_m = Z_m
\end{array}$$

where:

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

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

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

$$\vdots \qquad \vdots$$

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

$$\vdots$$

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

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

$$\begin{array}{ll}
(1) & W_0a_0 + W_1a_1 + W_2a_2 = Z_0 \\
(2) & W_1a_0 + W_2a_1 + W_3a_2 = Z_1 \\
(3) & W_2a_0 + W_3a_1 + W_4a_2 = Z_2
\end{array}$$

The forward solution is as follows:

1. Divide equation (1) by W_0
2. Multiply the equation resulting from step 1 by W_1 and subtract from equation (2).
3. Multiply the equation resulting from step 1 by W_2 and subtract from equation (3).

The resulting equations are:

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

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

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

where:

$$b_{12} = W_1/W_o, \quad b_{13} = W_2/W_o, \quad b_{14} = Z_o/W_o$$

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

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

Steps 1 and 2 are repeated using equations (5) and (6), with b_{22} and b_{32} instead of W_o 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:

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

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

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

Figure 3 is a possible FORTRAN program for carrying out the calculations for the case: $n = 100$, $m \leq 10$. W_o, W_1, W_2, ..., W_{2m} are stored in W(1), W(2), W(3), ..., W(2M+1), respectively. Z_o, Z_1, Z_2, ..., Z_m are stored in Z (1), Z (2), Z (3), ..., Z(M+1), respectively.

IBM FORTRAN Coding Form

PROGRAM: SAMPLE PROGRAM 2 — DATE: 6/66 — PAGE 1 OF 3

```
      REAL X(1ØØ),Y(1ØØ),W(21),Ƶ(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
      LƵ = M+1
      DO 5  J=2,LW
    5 N(J) = Ø.Ø
      W(1) = N
      DO 6  J=1,LƵ
    6 Ƶ(J) = Ø.Ø
      DO 16  I=1,N
      P = 1.Ø
      Ƶ(1) = Ƶ(1)+Y(I)
      DO 13  J=2,LƵ
      P = X(I)*P
      W(J) = W(J)+P
   13 Ƶ(J) = Ƶ(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 3. Sample Program 2

IBM FORTRAN Coding Form

PROGRAM: SAMPLE PROGRAM 2 — DATE: 6/66 — PAGE 2 OF 3

```
   16 W(J) = W(J)+P
   17 DO 2Ø  I=1,LƵ
      DO 2Ø  K=1,LƵ
      J = K+I
   2Ø B(K,I) = W(J-1)
      DO 22  K=1,LƵ
   22 B(K,LB) = Ƶ(K)
   23 DO 31  L=1,LƵ
      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,LƵ
      FMULTB = B(I,L)
      DO 31  J=L,LB
   31 B(I,J) = B(I,J)-B(L,J)*FMULTB
   33 A(LƵ) = B(LƵ,LB)
      I = LƵ
   35 SIGMA = Ø.Ø
      DO 37  J=I,LƵ
```

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

• Figure 3. Sample Program 2 (Continued)

IBM FORTRAN Coding Form

PROGRAM SAMPLE PROGRAM 2 PAGE 3 OF 3

PROGRAMMER DATE 6/66

```
   37 SIGMA = SIGMA+B(I-1,J)*A(J)
      I = I-1
      A(I) = B(I,LB)-SIGMA
   40 IF (I-1) 41,41,35
   41 WRITE (6,2) (A(I),I=1,LZ)
      STOP
      END
```

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

• Figure 3. Sample Program 2 (Continued)

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:

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

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

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

$$\vdots$$

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

$$\vdots$$

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

By the time control has passed to statement 23, the values of W_0, W_1, ..., W_{2m+1} have been placed in the storage locations corresponding to columns 1 through M + 1, rows 1 through M + 1, of the B array, and the values of Z , Z_1, ..., Z_m have been stored in the locations corresponding to the column of the B array. For example, for the illustrative problem (M = 2), columns 1 through 4, rows 1 through 3, of the B array would be set to the following computed values:

$$\begin{matrix} W_0 & W_1 & W_2 & Z_0 \\ W_1 & W_2 & W_3 & Z_1 \\ W_2 & W_3 & W_4 & Z_2 \end{matrix}$$

This matrix represents equations (1), (2), and (3), the normal equations for M = 2.

The forward solution, which results in equations (4), (7), and (8) in the illustrative problem, is carried out by statements 23 through 31. By the time control has passed to statement 33, the coefficients of the AI terms in the M + 1 equations which would be obtained in hand calculations have replaced the contents of the locations corresponding to columns 1 through M+1, rows 1 through M+1, of the B array, and the constants on the right-hand side of the equations have replaced the contents of the locations corresponding to column M+2, rows 1 through M+1, of the B array. For the illustrative problem, columns 1 through 4, rows 1 through 3, of the B array would be set to the following computed values:

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

This matrix represents equations (4), (7), and (8).

The backward solution, which results in equations (9), (10), and (11) in the illustrative problem, is carried out by statements 33 through 40. By the time control has passed to statement 41, which prints the values of the A9 terms, the values of the $(M+1)*A_I$ terms have been stored in the M + 1 locations for the A array. For the illustrative problem, the A array would contain 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.

A format code 62-63
Addition 16
   (see also arithmetic operators)
Additional input/output statements 69-70
   BACKSPACE 70
   END FILE 69
   REWIND 69
Adjustable dimensions 81-82
Alphabetic characters (table) 105
Alphameric characters 13,105
American Standards Association (ASA)
 FORTRAN 5
AND (see logical operators)
Arithmetic and logical assignment
 statement 7,27-28
Arithmetic expressions 16-20
   arithmetic operators 16-20
   mode 18
   order of computation 10
   use of parentheses 19-20
Arithmetic IF 32
Arithmetic operators 16-20
Arrangement of arrays in storage 25-26
Arrays 23-26
   arrangement in storage 25-26
   declaring size of 25
   subscripted variables 23
   subscripts 25
ASSIGN statement 31-32
Assigned GO TO statement 31-32
Associated variable 71

BACKSPACE statement 70
Basic input/output statements 41-50,73-75
   READ statements 41-48,73-74
   WRITE statements 48-50,74-75
Basic Operating System 5
Basic Programming Support 5
Blank common 84-86
Blank fields (see X format code)
Blank lines (see carriage control)
Blanks 8
BLOCK DATA subprogram 104
Boundary alignment 86-87,89

CALL statement 97-98
Coding form 8
Comments lines 8
COMMON statement 83-87
   blank common 83-86
   declaring size of an array 25
   labeled common 84-86
   programming considerations 86-87
Compiler 5
Complex constants 11
COMPLEX statement 80
   (see also FUNCTION subprograms)
Computed GO TO statement 30
Constants 8-13
   complex 11
   double precision 9-10
   hexadecimal 12-13
   integer 9
   literal 12
   logical 11
   real 9-10
Continuation lines 7
CONTINUE statement 37-38
Control statements 29-39
   arithmetic IF 32-33
   assigned GO TO 31-32
   computed GO TO 30
   CONTINUE 37-38
   DO 34-37
   END 39
   logical IF 33-34
   PAUSE 38-39
   STOP 39
   unconditional GO TO 29-30
Conversion codes (see format codes)

D decimal exponent 9-10,60
D format code 60
DATA initialization statement 107-108
Data set 40
Data set reference number 40
Decimal exponents 9-10,60
Declaring the size of an array 25
DEFINE FILE statement 70-72
Device (I/O) 40
Digit (see numeric characters)
DIMENSION statement 82
   adjustable dimensions 81-82
   declaring the size of an array 25
Direct access input/output statements
 70-76
Division 16
   (see also arithmetic operators)
DO statement 34-37
DO variable 34-37
Double precision constants 9-10
DOUBLE PRECISION statement 108

E decimal exponent 9-10,60
E format code 60
END FILE statement 69
END parameter in a READ statement 41
END statement 39,95-96
ENTRY statement 101-103
EQ (see relational operators)
EQUIVALENCE statement 88-89
ERR parameter in a READ statement 41,73
Explicit specification statement 15,79-82
Exponentiation 16,18-19
   (see also arithmetic operators)
Exponents (see decimal exponents)
Expressions 16-23
   arithmetic 16-20
   logical 20-23
EXTERNAL statement 103-104

F format code 59
FALSE 11
   (see also logical expressions)

Features of System/360 FORTRAN IV 5-6
FIND statement 75-76
Format codes 54-69
  A code 61-62
  carriage control 69
  D and E codes 60
  F code 59
  G code 54-57
  H code 65
  I code 58-59
  L code 61
  numeric codes 58-60
  scale factor-P 66-68
  T code 66
  X code 65
  Z code 60-61
FORMAT statement 40,50-69
  format codes 54-69
  FORTRAN record 54-56
  literal data 64
  reading FORMAT statements 47
FORTRAN
  American Standards Association 5
  Basic Operating System 5
  Basic Programming Support 5
  coding form 8
  compiler 5
  library 109-111
  Model 44 Programming System 5
  object program 5
  operating system 5
  operating system (E) 5
  record 40,54-56
  source program 5
  statements 7
  supplied subprograms 109-111
Functions 91-96
  definition 91
  FUNCTION subprograms 93-96
  reference to 91
  statement function subprograms 91-92

G format code 54-57
GE (see relational operators)
GO TO statements 29-32
  assigned 31-32
  computed 30
  unconditional 29-30
GT (see relational operators)

H format code 65
Hexadecimal constants 12-13
Hierarchy of operations
  in a logical expression 22-23
  in an arithmetic expression 19-20

I format code 58-59
I/O list
  in a NAMELIST 42-44
  in a READ 41,73
  in a WRITE 47,74
Imaginary number (see complex constants)
IMPLICIT specification statement 15,77-79
In-line 109
Indexing I/O lists 46
Indexing parameters in a DO loop 34-35
Input/output statements 7,40-76
  BACKSPACE 70
  direct access statements 70-76
  END FILE 69
  FIND 75-76
  READ 41-48,73-74
  REWIND 69
  sequential statements 40-70
  WRITE 48-50,74-75
Integer constants 9
Integer division 20
INTEGER statement 79-82
  (see also FUNCTION subprograms)

L format code 61
Labeled common 84-86
LE (see relational operators)
Length specification
  (see optional length specification,
  standard length specification)
Library 109-111
List (see I/O list)
Literal constants 12
Literal data in a FORMAT statement 64
Logical constants 11
Logical expressions 20-23
  logical operators 21-22
  order of computation 22-23
  relational operators 20-21
  use of parentheses 23
Logical IF statement 33-34
Logical operators 21-22
LOGICAL statement 79-82
  (see also FUNCTION subprograms)
Looping (see DO statement)
LT (see relational operators)

Mathematical function subprograms 109-111
Mixed-mode 5
  (see also expressions)
Mode of an arithmetic expression 18-19
Model 44 Programming System 5
Multiline listing 57
Multiple ENTRY into a subprogram 100-103
Multiplication 16
  (see also arithmetic operators)

Named common (see labeled common)
NAMELIST statement 42-44,48
NE (see relational operators)
Nest of DOs 35-37
NOT (see logical operators)
Numeric characters 105
Numeric format codes 58-60

Object program 5
Operands 17-18
Operating system 5
Operating system (E) 5
Operators
  arithmetic 16
  logical 21-22
  relational 20-21
Optional length specification for
  variables 14,77-82
OR (see logical operators)
Order of computation
  in a logical expression 22-23
  in an arithmetic expression 19-20
Out-of-line 109-111

P format code 66-68
Parentheses
in a FORMAT statement 51-53
in a logical expression 23
in an arithmetic expression 20
PAUSE statement 38-39
Predefined specification 14-15
PRINT statement 107
Programming considerations
in using COMMON blocks 86-87
in using DEFINE FILE statements 72
in using DO statements 36-37
in using EQUIVALENCE groups 89
PUNCH statement 106

Range of a DO statement 35-36
READ statements 41-48,73-74
direct access READ statement 73-74
READ (a) list 45
READ (a,b) list 44-45
READ (a'r,b) list 73-74
READ (a,x) 42-44
READ b,list 106
sequential read statements 41-48
Reading FORMAT statements 47
Real constants 9-10
REAL statement 79-82
(see also FUNCTION subprograms)
Referencing of arguments by name and by value 98-99
Relational operators 20-21
Repeat constant 43
RETURN statement
in a FUNCTION subprogram 95-96
in a SUBROUTINE subprogram 99-100
REWIND statement 69

Sample program 1 112
Sample program 2 113-117
Sequential input/output statements 40-70
Service subprograms 109
Slashes in a FORMAT statement 51-52
Source program 5
Special characters (table) 105
Specification statements 7,90-104
COMMON 83-87
DEFINE FILE 70-71
DIMENSION 82
EQUIVALENCE 87-89
explicit 15,79-82
EXTERNAL 103-104
FORMAT 50-69
IMPLICIT 15,77-79
NAMELIST 42-44
Standard length specification for variables 14-15,77-82

Statements 7
arithmetic and logical assignment 27-28
control 29-39
direct access I/O 70-76
sequential I/O 40-70
specification 77-89
subprogram 90-104
STOP statement 39
Subprograms
FORTRAN supplied 109-111
FUNCTION 93-96
statement functions 91-92
SUBROUTINE 96-97
Subscripted variable 23
Subscripts 25
Subtraction 16
(see also arithmetic operators)
Symbolic unit number
(see data set reference number)

T format code 66
TRUE 11
(see also logical expressions)
Type and length specification 14
Type declaration
explicit 15,79-82
IMPLICIT 15,78-79
predefined convention 14-15
Type specification of FUNCTION subprograms 94-95
Type statements
explicit 79-82
IMPLICIT 78-79

Unconditional GO TO statement 29-30

Variable FORMAT statements
(see reading FORMAT statements)
Variables 13-15
names 14
subscripted 23
type declaration 14-15
types and length specifications 14

WRITE statements 48-50,74-75
direct access WRITE statement 74-75
sequential WRITE statements 48-50
WRITE (a) list 49-50
WRITE (a,b) list 48-49
WRITE (a'r,b) list 74-75
WRITE (a,x) 48

X format code 65

Z format code 60-61

READER'S COMMENTS

Title: IBM System/360 FORTRAN IV Language

Form: C28-6515-4

Is the material:

	Yes	No
Easy to Read?	___	___
Well organized?	___	___
Complete?	___	___
Well illustrated?	___	___
Accurate?	___	___
Suitable for its intended audience?	___	___

How did you use this publication?

___ As an introduction to the subject
___ For additional knowledge
Other ___________________________

Please check the items that describe your position:

___ Customer personnel
___ IBM personnel
___ Manager
___ Systems Analyst
___ Operator
___ Programmer
___ Customer Engineer
___ Instructor
___ Sales Representative
___ Systems Engineer
___ Trainee
Other ___________

Please check specific criticism(s), give page number(s), and explain below:

___ Clarification on page(s)
___ Addition on page(s)
___ Deletion on page(s)
___ Error on page(s)

Explanation:

CUT ALONG LINE

Name ___________________________
Company ___________________________
Address ___________________________
City ___________________________
State ___________ Zip Code ______

FOLD ON TWO LINES, STAPLE AND MAIL
No Postage Necessary if Mailed in U.S.A.

staple staple

fold fold

FIRST CLASS
PERMIT NO. 81

POUGHKEEPSIE, N.Y.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

IBM CORPORATION
P.O. BOX 390
POUGHKEEPSIE, N. Y. 12602

ATTN: PROGRAMMING SYSTEMS PUBLICATIONS
DEPT. D58

fold fold

Printed in U.S.A. C28-6515-4

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

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

staple

C28-6515-4

Printed in U.S.A. C28-6515-4

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

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