File No. S360-25
Form C28-6629-0

OS
DOS
TOS
BPS

Systems Reference Library

IBM System/360

Basic FORTRAN IV Language

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

PREFACE

This publication is designed to support four implementations of the Basic FORTRAN IV language for the IBM System/360. The language described is implemented for the IBM System/360 Operating System, the IBM System/360 Disk Operating System, the IBM System/360 Tape Operating System, and the IBM System/360 Basic Programming Support Tape System. Differences among the language implementations are indicated by footnotes.

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

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

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

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

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

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

IBM System/360 Basic Programming Support FORTRAN IV (Tape) Programmer's Guide, Form C28-6583

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

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

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

First Edition (August 1966)

This publication, Form C28-6629-0, combines the specifications contained in the following publications and associated technical newsletters:

IBM Operating System/360: FORTRAN IV (E Level Subset), Form C28-6513

IBM System/360 Basic Operating System: Specifications FORTRAN IV (16K Disk/Tape), Form C24-5014 and Technical Newsletters N24-5041, N24-5069, N21-5018

IBM System/360 Basic Programming Support: FORTRAN IV (Tape) Specifications, Form C28-6504 and Technical Newsletters N28-2105 and N28-2168

Although the information contained in these publications and newsletters is not obsolete, the publications themselves are no longer being maintained. Significant changes or additions to the specifications of the Basic FORTRAN IV language will be reported in subsequent revisions or Technical Newsletters to this publication, C28-6629-0.

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 D39
1271 Avenue of the Americas, New York, New York 10020

CONTENTS

ILLUSTRATIONS

FIGURES

TABLES

The FORTRAN language is especially useful in writing programs for scientific and engineering applications that involve mathematical computations. 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.

The IBM System/360 Basic FORTRAN IV language is compatible with and encompasses the American Standards Association (ASA) Basic FORTRAN, including its mathematical subroutine provisions. Basic FORTRAN IV is a subset of FORTRAN IV, as described in the publication IBM System/360 FORTRAN IV Language.

The Basic FORTRAN IV language can be used with the following compilers:

IBM System/360 Operating System FORTRAN IV (E) Compiler

IBM System/360 Disk Operating System FORTRAN IV Compiler

IBM System/360 Tape Operating System FORTRAN IV Compiler

IBM System/360 Basic Programming Support Tape System FORTRAN IV Compiler

All of the features and facilities in Basic FORTRAN IV also exist in FORTRAN IV. Equivalent results from valid programs compiled by either Basic FORTRAN IV or FORTRAN IV are assured by:

1. Common data formats.

2. Common format code routines.

3. Common calling sequences.

4. Common libraries.

The following features of Basic FORTRAN IV 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. The A Format Code: The A format code allows reading and writing of character data.

5. Scale Factor: The scale factor allows modification of the internal or external representation of data.

6. 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. Explicitly type a variable as integer, real, or double precision.
   b. Specify the dimension of arrays.

7. Carriage Control: The first character of a record to be printed is used for carriage control.

8. DOUBLE PRECISION Data Type: A third type of data is available; it gives greater precision than real data.

9. Three Dimension Arrays: An array may have one, two or three dimensions.

10. Six Character Variable Names: The name of a variable may contain up to six characters.

11. Direct-Access Statements: Data records may be either read from or written on direct-access input/output devices in an order specified by the user.

12. Function Subprograms may return results via the argument 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 (see Table 1) 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 Statements*: Upon execution of an arithmetic statement, the result of calculations performed 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 categories 1, 2, and 3.

Table 1. Key Words in Basic FORTRAN IV

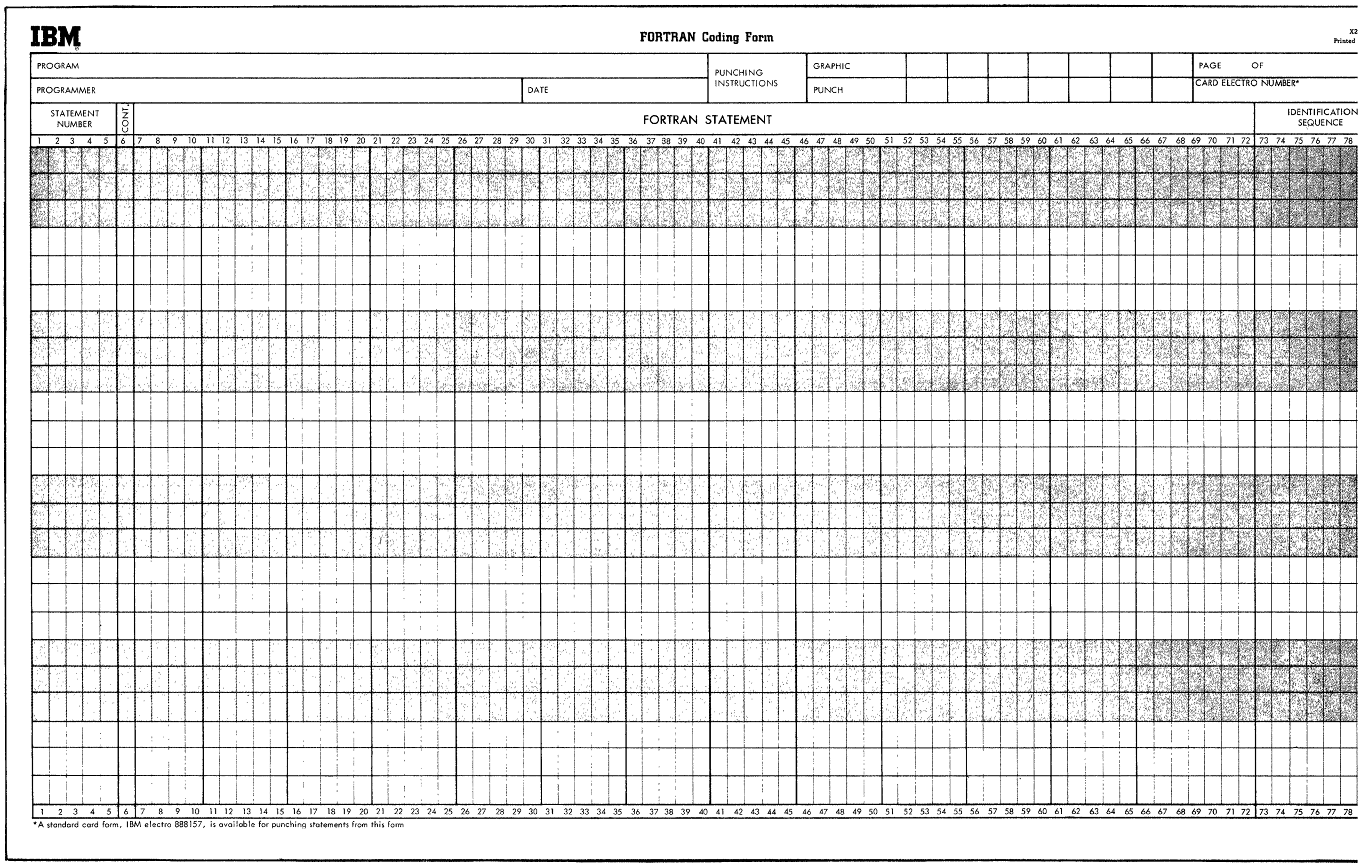
ABS	DFLOAT	FORMAT	READ
	DO	FUNCTION	REAL
BACKSPACE	DOUBLE		RETURN
	DSIGN	GO	REWIND
CALL		GOTO	
COMMON	END		
CONTINUE	ENDFILE	IABS	SIGN
	EQUIVALENCE	IDIM	SNGL
	EXIT	IF	STOP
DABS	EXTERNAL	IFIX	SUBROUTINE
DBLE		INTEGER	
DEFINE		ISIGN	WRITE
DIM	FIND		
DIMENSION	FLOAT	PAUSE	

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. Blanks may be inserted in statement numbers where desired.

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



IBM

FORTRAN Coding Form

X2
Printed

PROGRAM

PUNCHING INSTRUCTIONS

GRAPHIC

PAGE OF

PROGRAMMER

DATE

PUNCH

CARD ELECTRO NUMBER*

STATEMENT NUMBER

CONT.

FORTRAN STATEMENT

IDENTIFICATION SEQUENCE

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78

*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.

The format of all FORTRAN statements requires that every word of the statement or expression, every name (e.g., variable name or subprogram name), and every constant used in a statement must be terminated by one of the following delimiters:

```
- * .   , + / = ( ) '   Column 73
```

In a statement of more than one line, column 73 is only considered a delimiter in the last line of the statement. As many blanks as desired may be written with any delimiter to improve readability. In addition, blanks may be inserted in key words, names, and constants; the blanks are ignored by the compiler. However, blanks that are inserted in literal data either enclosed in apostrophes or used in the H format code are treated as blanks within the data. The following examples show how blanks are treated in Basic FORTRAN IV (where b represents a blank):

GbObbTbObb2b5 GObTOb25 GOTO25	treated as GO TO 25
3b74	treated as 374
0.b36	treated as 0.36
'TABLbE'	treated as TABLbE

CONSTANTS

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

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 consisting of from 1 through 7 significant decimal digits occupying four storage locations. A real constant optionally may be followed by a decimal exponent written as the letter E followed by a signed or unsigned, one- or two-digit integer constant. If the decimal point is not present, then an E exponent specifies that the constant is real.

Magnitude: If followed by an E decimal exponent: 0 or 16^{-63} through 16^{63} (i.e., approximately 10^{75}); otherwise, it may consist of from 1 through 7 decimal digits.

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

<u>Examples</u>:

<u>Valid</u> Real Constants:

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

<u>Invalid</u> Real Constants:

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 x 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 x $10^{97} > 16^{63}$)
-91437.143	(exceeds the number of significant decimal digits permitted)
1.0000000	(exceeds the number of significant decimal digits permitted)

DOUBLE PRECISION CONSTANTS

Definition

Double-Precision Constant - a number with a decimal point optionally followed by a decimal exponent. This exponent may be written as the letter D followed by a signed or unsigned, one- or two-digit integer constant. If the decimal point is not present, then a D exponent specifies that the constant is double precision. A double-precision constant may assume one of two forms (both forms occupy eight storage locations):

1. From 1 through 7 decimal digits followed by a D decimal exponent.

2. From 8 through 16 significant decimal digits optionally followed by a D decimal exponent.

Magnitude: (either form) 0 or 16^{-63} through 16^{63} (i.e., approximately 10^{75}).

A double-precision constant may be positive, zero, or negative (if unsigned, it is assumed to be positive) and must be of the allowable magnitude. It may not contain embedded commas. The decimal exponent D, similar to that for real constants (i.e., E), permits the expression of a double-precision constant as the product of a double-precision constant times 10 raised to a desired power. Note that a double-precision constant has more than twice the number of significant digits as that for a real constant, but its maximum magnitude remains the same.

Examples:

Valid Double-Precision Constants:

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

Invalid Double-Precision Constants

7.9E3	(should have decimal exponent D, not E)
7.9D	(missing a one- or two-digit integer constant following the D)
7.987143	(decimal exponent D must follow when less than 8 significant decimal digits are used)
21.3D90	(exceeds given magnitude, i.e., $21.3 \times 10^{90} > 16^{63}$)

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 other than the blank (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.

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

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.

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.

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

These statements are discussed in greater detail in the 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.

Basic FORTRAN IV provides only one kind of expression: the arithmetic expression. The value of an arithmetic expression is always a number whose type is integer, real, or double precision.

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. Double Precision

If the constant, variable, or subscripted variable is of the type integer, the expression is in the integer mode. If it is of the type real, the expression is in the real mode, etc.

Examples:

Expression	Type of Quantity	Mode of Expression
3	Integer Constant	Integer
I	Integer Variable	Integer
3.0	Real Constant	Real
A	Real Variable	Real
3.14D3	Double-precision Constant	Double precision
B(2*I)	Double-precision Subscripted Variable (Specified as Such in an Explicit Specification statement)	Double precision

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

More complicated arithmetic expressions containing two or more constants and/or variables may be formed by using arithmetic operators that express the 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 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 by the result. (For further uses of parentheses, see Rule 6.)

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

   Table 2. Determining the Mode of an Expression Containing Operands of Different Types

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

   From Table 2 it can be seen that there is a hierarchy of type that determines the mode of an expression. For example, double-precision data when combined with any other types of constants and variables results in double-precision.

   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
C,D	Double-precision variables

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

Expression	Mode of Expression
ROOT*5	Real
A+3	Integer
C+2.9D10	Double Precision
E/F+19	Real
C-18.7E05	Double Precision
A/I-D	Double Precision
C/D	Double Precision

4. The arithmetic operator denoting exponentiation (i.e.,**) may be used to combine any types of operands as shown in Table 3.

Table 3. Valid Combinations with Respect to the Arithmetic Operator, **

Base		Exponent
Integer	**	Integer
Real		Real
Double Precision		Double Precision

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, D	Double-Precision variables

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	(Double Precision**Integer)	(Double Precision)
E**C	(Real**Double Precision)	(Double Precision)
I**C	(Integer**Double Precision)	(Double Precision)
D**E	(Double Precision**Real)	(Double Precision)
D**C	(Double Precision**Double Precision)	(Double Precision)

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 2).

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       Call the result Y
c.  -Y         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 one integer is divided by another, the quotient is also an integer. If necessary, the result is truncated. For example:

```
5/2
```

gives a quotient of 2.

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 be assigned 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 subscripts, 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 subscripts. 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 4.)

Table 4. 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	1,000	ISIZE=1
36 - 40	IAGE=8	2,000	ISIZE=2
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 4, 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 through 20, "ISEX" from 1 through 2, and "ISIZE" from 1 through 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.

The actual size (in storage locations) of the array COST depends upon the type of elements in the array. If each element in the array is real, the array size is 1280 storage locations; if each element is double precision, the array size is 2560 storage locations.

SUBSCRIPTS

A subscript is a number used to refer to a particular variable within an array. A subscript may take one of several forms and there may be a maximum of three 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 number of subscripts used must be equal to the number of dimensions in the array.

General Form

<u>Subscripts</u> - may be one of seven forms:

```
v
c'
v+c'
v-c'
c*v
c*v+c'
c*v-c'
```

Where: *v* represents an unsigned, nonsubscripted, integer variable.

c and *c'* represent any unsigned integer constant.

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

<u>Examples:</u>

<u>Valid</u> Subscripted Variables:

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

<u>Invalid</u> Subscripted Variables

ARRAY (-I)	(the subscript I may not be signed)
COST(A+2)	(A is not an integer variable unless defined as such by an Explicit specification statement)
ARRAY(I+2.)	(the constant within a subscript must be an integer)
NEXT(-7*J)	(the constant within a subscript must be unsigned)
W(I(2))	(the subscript, I, may not be subscripted)
LOT (0)	(a subscript may never be nor assume a value of zero)

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

TEST (K*2) (if multiplication is indicated, the constant must precede the variable. Thus, TEST (2*K) is correct.)

TOTAL (2+K) (if addition is indicated, the variable must precede the constant. Thus, TOTAL (K+2) is correct.)

DECLARING THE SIZE OF AN ARRAY

The size of an array is determined by the number of subscripts 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 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 subscripts, 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 subscripts, 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.

ARITHMETIC STATEMENT

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

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

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

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

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

Statements	Description
A = B	The value of A is replaced by the current value of B.
W = B	The value of B is truncated to an integer value, and this value replaces the value of W.
A = I	The value of I is converted to a real value, and this result replaces the value of A.
I = I + 1	The value of I is replaced by the value of I + 1.
B = I**J+D	The value of I is raised to the power J and result is converted to a real value to which the value of D is added. This result then replaces the value of B.
A = B*D	The most significant part of the product of B and D replaces the value of A.
A = I+E	The value of I is converted to double precision and and added to E. The result of the addition is truncated from double precision to real and replaces the value of A.
A = F(5,4)	The value of F(5,4) replaces the value of A.
E = I	The value of I is converted to double precision, and this value replaces the value of E.
J = E	The value of E is truncated to an integer value, and this value replaces the value of J.
E = A	The value of A is converted to double precision, and this value replaces the value of E.

CONTROL STATEMENTS

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

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.

Unconditional GO TO Statement

General Form
GO TO <u>xxxxx</u> Where: <u>xxxxx</u> 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 ($\underline{x}_1$, $\underline{x}_2$, $\underline{x}_3$, ...,$\underline{x}_n$), $\underline{i}$

Where: $\underline{x}_1$,$\underline{x}_2$,...,$\underline{x}_n$, are executable statement numbers.

$\underline{i}$ is a nonsubscripted integer variable and is in the range: $1 \le \underline{i} \le n$

This statement causes control to be transferred to the statement numbered $\underline{x}_1$, $\underline{x}_2$, $\underline{x}_3$,..., or $\underline{x}_n$, depending on whether the current value of $\underline{i}$ is 1, 2, 3,..., or n, respectively. If the value of $\underline{i}$ 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.

ADDITIONAL CONTROL STATEMENTS

Arithmetic IF Statement

General Form

IF ($\underline{a}$) $\underline{x}_1$,$\underline{x}_2$,$\underline{x}_3$

Where: $\underline{a}$ is an arithmetic expression.

$\underline{x}_1$,$\underline{x}_2$,$\underline{x}_3$ 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.

DO Statement

General Form
DO $\overbrace{\underline{x}}^{\text{End of Range}}\quad \overbrace{\underline{i}}^{\text{DO Variable}} = \overbrace{\underline{m}_1,}^{\text{Initial Value}}\quad \overbrace{\underline{m}_2,}^{\text{Test Value}}\quad \overbrace{\underline{m}_3}^{\text{Increment}}$
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 $\underline{m}_3$. When, at the end of the iteration, $\underline{i}$ is equal to the highest value that does not exceed $\underline{m}_2$, control passes to the statement following the statement numbered $\underline{x}$. Thus, the number of times the statements in the range of the DO is executed is given by the expression:

$$\left[\frac{\underline{m}_2 - \underline{m}_1}{\underline{m}_3} \right] +1$$

where the brackets represent the largest integral value not exceeding the value of the expression. If $\underline{m}_2$ is less than $\underline{m}_1$, the statements in the range of the DO are executed once. Upon completion of the DO, the DO variable is undefined, and should not be used until 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.

Example 1:

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).

```
         .
         .
         .
 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 follows:

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

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

Programming Considerations in Using a DO Loop

1. The indexing parameters of a DO statement ($\underline{i}$, $\underline{m}_1$, $\underline{m}_2$, $\underline{m}_3$) 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        }
   ```

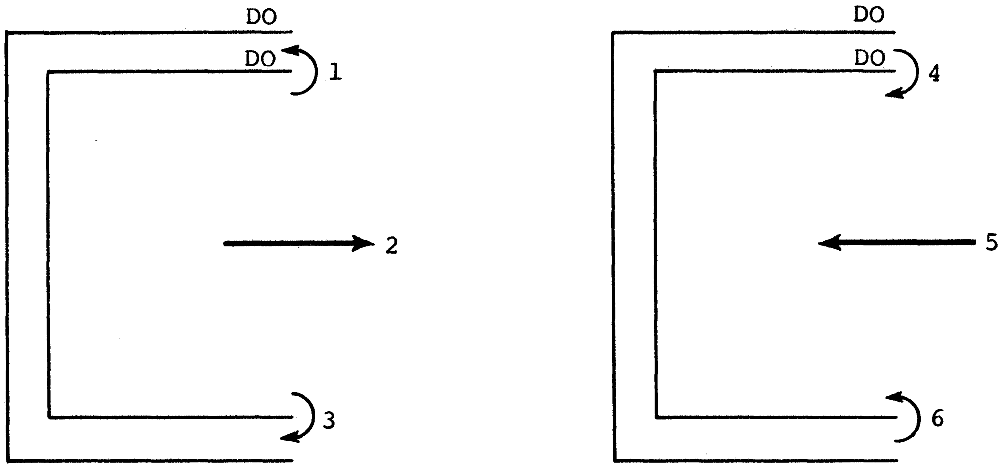
   Example 2:

   ```
      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 $(\underline{i},\underline{m}_1,\underline{m}_2,\underline{m}_3)$ 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
Where: n is an unsigned 1 through 5 digit integer 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.

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.

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.

There are two types of I/O statements: sequential I/O statements (available in all Basic FORTRAN IV systems) and direct access I/O statements (not available in Basic Programming Support FORTRAN IV). 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 statement, although it is not an I/O statement, is used with certain forms of the READ and WRITE statements. The FORMAT statement specifies the form in which the data is to be transmitted and allows 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) list

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

b is the statement number of the FORMAT statement that describes the data being read.

list is a series of variable or array names, which may be indexed and must be separated by commas. These names 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.

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

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

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 as specified by FORMAT 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 ranging from 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.

WRITE STATEMENT

General Form

WRITE (a, b) list

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

b is the statement number of the FORMAT statement that describes the data being written.

list is a series of variable or array names, which may be indexed and must be separated by commas. These names 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,b)list
        WRITE(a)list
```

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, as specified by FORMAT 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 ($c_1,c_2,\ldots,c_n/c_1',c_2',\ldots,c_n'/\ldots$)

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

$c_1,c_2,\ldots,c_n$ and $c_1',c_2',\ldots,c_n'$ 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:

I - to transfer integer data
F - to transfer real or double precision data that does not contain a decimal exponent
E or D - to transfer real or double precision data that contains an E or D decimal exponent, respectively
A - to transfer character 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, scale factor, or any literal, must be less than or equal to 255. The scale factor must be less than or equal to 127.

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 (----, ----, ----)

                    <-------------->
                           ↑
                           |
                           |
                           └---corresponds to 1
                               FORTRAN record
      ```

   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 (----/ ----/ ----)

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

   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:

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

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

When defining a FORTRAN record by a FORMAT statement it is important to consider the original source (input) or ultimate destination (output) of the record. For example, if a FORTRAN record is to be punched for output, the record should not be greater than 80 characters. For input, the FORMAT statement should not define a FORTRAN record longer[1] 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)
```

[1] In Basic Programming Support FORTRAN IV, Disk Operating System FORTRAN IV, and Tape Operating System FORTRAN IV, a maximum of 255 characters per record may be transmitted. In Operating System FORTRAN IV (E), the maximum depends upon the device (see the FORTRAN IV (E) Programmer's Guide).

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 one-level parenthetical expression is permitted to enable repetition of data fields according to certain format codes within a longer FORMAT statement. 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',I2,F3.1/'x',F10.8/('x',3F5.1))
```

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

```
I2,F3.1
F10.8
F5.1,F5.1,F5.1
F5.1,F5.1,F5.1
  .    .    .
  .    .    .
  .    .    .
```

As another example, consider the following statement:

```
FORMAT ('x',I2/2('x',I3,F6.1),F9.7)
```

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

```
I2
I3,F6.1,'x',I3,F6.1,F9.7
I3,F6.1,'x',I3,F6.1,F9.7
I3,F6.1
```

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.

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

aIw
aFw.d
aEw.d
aDw.d

Where: a 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.

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

d is an unsigned integer constant that specifies 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 quantity is negative, the position preceding the leftmost digit contains a minus sign. In this case, an additional position should be specified in w for the minus sign. If 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 number of characters to be transmitted is greater than w, on output, asterisks are given. The magnitude of the data to be transmitted must not exceed the maximum magnitude for an integer constant.

The following examples show the internal value for each of the quantities on the left, according to format code I3: (b represents a blank):

External Value	Internal Value
345	345
bb4	004
4b3	403
43b	430

The following examples show how each of the quantities on the left is printed according to the format code I3:

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. The magnitude of the data to be transmitted must not exceed the maximum magnitude for a real or double precision constant.

The integer portion of the number is handled in the same fashion as numbers transmitted by the I format code. If excessive positions are reserved by d, zeros are filled in on the right. If insufficient positions are reserved by d, on input the fractional portion is truncated from the right and on output the fractional portion is rounded; truncating and rounding occur at the dth position.

The following examples show the internal value for each of the quantities on the left, according to format code F5.2:

External Value	Internal Value
32.46	32.46
-8.4	-8.40
13.568	13.57

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	
7.3582	b7.36	
-1.	-1.00	
9.03	b9.03	
187.64	*****	(incorrect because of insufficient specification)

E and D Format Codes

The E and D format codes are used in conjunction with the transferral of real or double precision data that contains an E or D decimal exponent, respectively. For E and D 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.

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 the internal value for each of the quantities on the left, according to the format codes (D10.3/E10.3)

External Value	Internal Value
02.380+02	238.
-0.001E+03	-1.
-7.654D-06	-.000007654
1.E2	100.
4.673	4.673

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 either be marked by an E or D, 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.

A Format Code

General Form

aAw

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, including blanks.

The format code Aw is used to read or write character 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 storage to handle the characters being written (see the section "Specification Statements").

Example 1:

Assume that the array ALPHA consists of one subscript 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 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 (either with or without embedded blanks). 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
     1 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 (5,5)
```

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 must be less than or equal to 127 and is defined as follows:

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

INPUT

For input, when scale factors are used in a FORMAT statement, they have effect only on real or double precision data which does not contain an E or D format code, respectively. 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.

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 : 5.5365
```

For output, when scale factors are used, they have effect only on real or double precision data. However, this real or double precision data may contain an E or D decimal exponent, respectively. A positive scale factor used with real or double precision 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 are equivalent and each would 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.

BACKSPACE STATEMENT

General Form

BACKSPACE a

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

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

DIRECT ACCESS INPUT/OUTPUT STATEMENTS

There are four direct access I/O statements:[1] 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.

Because the DEFINE FILE statement is a specification statement, it must precede the first executable statement of the source program as well as all statement function definition statements. 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.

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)
```

[1] The direct access I/O statements are not available in Basic Programming Support FORTRAN IV.

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 unsigned integer constant that is the data set reference number.

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

r represents an unsigned 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.

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)
```

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

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

In this example, the program and the subprogram share the associated variable IUAR. An input/output operation that refers to data associated with data set reference number 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) list

Where: a is an unsigned integer constant or integer variable that represents a data set reference number; a must be followed by an apostrophe (').

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

b is optional and, if given, is the statement number of the FORMAT statement that describes the data being read.

list is a series of variable or array names, which may be indexed and must be separated by commas. These names 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 unsigned integer constant or integer variable that represents a data set reference number; a must be followed by an apostrophe (').

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

b is optional and, if given, is the statement number of the FORMAT statement that describes the data being written.

list is a series of variable or array names, which may be indexed and must be separated by commas. These names 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 unsigned integer constant or integer variable that represents a data set reference number; *a* must be followed by an apostrophe (').

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

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. If a WRITE statement refers to this record between the issuing of the FIND statement and the READ statement, the FIND operation is nullified.

General Examples -- Direct Access Operations

Example 1:

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

Explanation:

Example 1 illustrates the ability of direct access statements to gather and disperse data in an order designated by the user. The first DO loop in the example fills arrays A through F with data from the 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.

Example 2:

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

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

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

Explanation:

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

SPECIFICATION STATEMENTS

The specification statements provide the compiler with information about the nature of the data used in the source program. In addition, they supply the information required to allocate locations in storage for this data. Specification statements describing data must precede any statements which refer to that data. For example, if an element of an array is to be made equivalent to a variable, the specification statement that declares the size of the array (e.g., a DIMENSION statement) must precede the EQUIVALENCE statement.

All specification statements must precede the first executable statement of the source program.[1] They must also precede all Statement Function definition statements. Therefore, the source program layout is as follows:

1. Specification Statements.
2. Statement Function Definition Statements.
3. Executable Statements.

FORMAT statements may appear anywhere in the program. They do not affect the sequence of execution.

Explicit Specification Statements

General Form

Type $a(k_1),b(k_2),\ldots,z(k_n)$

Where: Type is INTEGER, REAL, or DOUBLE PRECISION.

$a,b,\ldots,z$ represent variable, array, or function names (see the section, "SUBPROGRAMS")

$(k_1),(k_2),\ldots,(k_n)$ are optional. Each k is composed of 1 through 3 unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

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

[1] In Operating System FORTRAN IV (E) and Basic Programming Support FORTRAN IV, EQUIVALENCE statements must follow any DIMENSION, COMMON, or explicit specification statements; they need not follow DEFINE FILE statements in FORTRAN IV (E).

Example 1:

```
INTEGER DEV, ARRAY, SMALL
```

Explanation:

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

Example 2:

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

Explanation:

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

Example 3:

```
DOUBLE PRECISION DOUB,TWIN
```

Explanation:

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

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 3 unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

The information necessary to allocate storage for arrays used in the source program may be provided by the DIMENSION statement. 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 $\underline{a}_1(\underline{k}_1),\underline{a}_2(\underline{k}_2),\underline{a}_3(\underline{k}_3),\ldots,\underline{a}_n(\underline{k}_n)$

Where: $\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$ are variable or array names.

$\underline{k}_1,\underline{k}_2,\underline{k}_3,\ldots,\underline{k}_n$ are optional and are each composed of 1 through 3 unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

The COMMON statement may be used to provide dimension information. However, this information must be given on the first appearance of an array. For example, if an array has appeared first in an Explicit specification statement, the array should be dimensioned in that statement rather than in the COMMON statement.

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 examples:

Example 1:

```
    Calling Program            Subprogram

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

Explanation:

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

```
Beginning of COMMON area ┌────────────────┐
                         │       A        │
                         ├────────────────┤
                         │       B        │
                         ├────────────────┤
                         │       C        │
                         ├────────────────┤
                         │       R(1)     │
                         │       R(2)     │
                         │       .        │
                         │       .        │
                         │       .        │
                         │       R(100)   │
                         └────────────────┘
                       4 storage locations
```

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

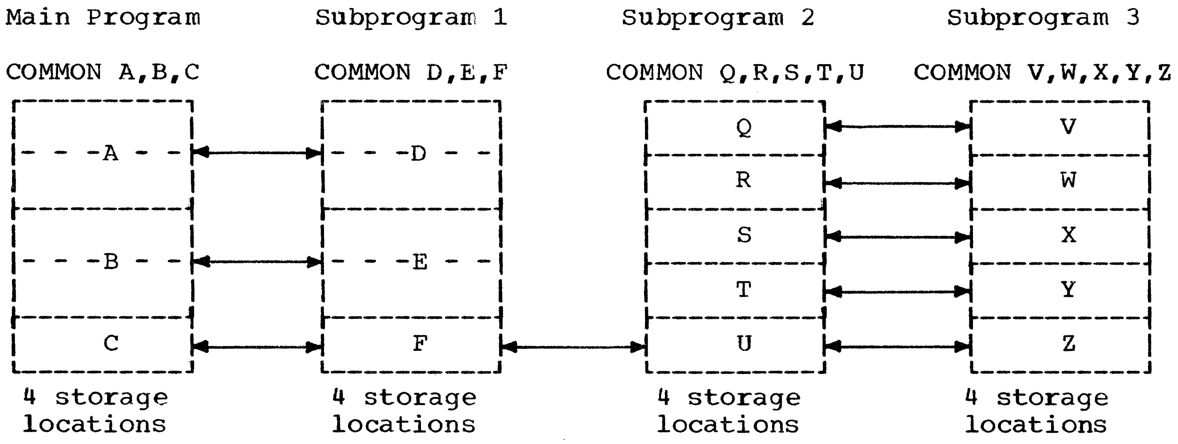
From the above example, it can be seen that COMMON statements may be used to serve an important function: namely, as a medium to implicitly transmit data from the calling program to the subprogram. That is, values for X, Y, Z, and S(1)...S(100), because they occupy the same storage locations as A, B, 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.")

Example 2:

Assume COMMON is defined in a main program and 3 subprograms as follows:

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

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



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

To insure proper boundary alignment, the user is advised to arrange the variables in COMMON in the following order:

Double Precision
Real or Integer

EQUIVALENCE Statement

General Form

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

Where: a, b, c, d, e, f,... are variables that may be subscripted. The subscripts may have two forms: If the variable is singly subscripted it refers to the position of the variable in the array (i.e., first variable, 25th variable, etc). If the variable is 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 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.

In storage established by EQUIVALENCE statements, double precision variables must precede real or integer variables.

Example 1:

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

Explanation:

This EQUIVALENCE statement indicates that the variables A, B(1), and C(5,3) are to share the same storage locations. All other corresponding variables in arrays B and C share the same storage locations; e.g., B(2) and C(6,3). In addition, it specifies that D(5,10,2) and E are to share 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 EQUIVALFNCE 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(3) with any other element of the array C, other than C(7,3), 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 share 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)
```

Two variables in COMMON may not be made equivalent. In addition, a user must observe the suggestion on boundary alignment mentioned in the description of the COMMON statement. Any double precision variable that is made equivalent to any variable in COMMON must be aligned properly. For example, the following statements will produce proper boundary alignment:

```
REAL A,B
DOUBLE PRECISION  DOUB,TWIN
INTEGER  I,J
COMMON  DOUB,A,I,B
EQUIVALENCE  (A,TWIN),(B,J)
```

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 be a variable name and may not contain special characters other than the blanks (see Appendix A). Blanks embedded in a subprogram name are ignored. The type of a subprogram can be indicated in the same manner as variables.

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

2. Type Declaration of FUNCTION Subprograms: The declaration can be made in one of two ways: by the predefined convention or by an explicit specification (see the section, "Type Specification of the FUNCTION Subprogram"). In addition, the type may appear in the FUNCTION definition statement.

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 of the name used for the reference must agree with the type 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.

A maximum of 15 variables appearing in the expression may be used as arguments of the function. 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)
```

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

Where: $\underline{name}$ is subprogram name (see the section "Naming Subprograms").

$\underline{a}_1, \underline{a}_2, \underline{a}_3, \ldots, \underline{a}_n$ 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 subprogram may contain any FORTRAN statement except a SUBROUTINE statement or another FUNCTION statement.

The arguments of the FUNCTION subprogram (i.e., $\underline{a}_1, \underline{a}_2, \underline{a}_3, \ldots, \underline{a}_n$) may be considered to be dummy variable names. These are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. The actual arguments may be any of the following:

1. any type of constant
2. any type of subscripted or nonsubscripted variable
3. an array name
4. an arithmetic expression
5. the name of another FUNCTION or SUBROUTINE subprogram.

The actual arguments must correspond in number, order, and type to the dummy arguments. The array size must also be the same. 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 implicitly declaring the type of a FUNCTION name by the predefined convention, there exists the option of explicitly specifying the type of a FUNCTION name within the FUNCTION statement.

However, the type of a FUNCTION name may not be declared in an Explicit specification statement in that subprogram.

General Form

Type FUNCTION name (a_1, a_2,...,a_n)

Where: Type is INTEGER, REAL, or DOUBLE PRECISION.

name is the name of the FUNCTION subprogram.

a_1,a_2,...,a_n are nonsubscripted variable or array names, or the dummy names of a SUBROUTINE or another FUNCTION subprogram. (There must be at least one argument.)

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 physical 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_ ($\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$)
.
.
.
RETURN

END

where: _name_ is the subprogram name (see the section "Naming Subprograms").

$\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$ are nonsubscripted variable or array names, or the dummy names of other SUBROUTINE or FUNCTION subprograms. (There need not be any arguments.)

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

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

The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. Any arguments so used must appear on the left side of an arithmetic statement 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, and type 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.

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
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

The arguments in a CALL statement must agree in number, order, and type with the corresponding arguments in the subroutine subprogram. The array sizes must also be the same in the subroutine and the calling programs. If an actual argument corresponds to a dummy argument that is defined or redefined in the subprogram, the actual argument must be a variable name, subscripted variable name, or array name.

A 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 (i.e., cannot appear to the left of an equal sign in an arithmetic statement) in the function DERIV because the actual argument list causes both A and B to be associated with more than one value.

RETURN Statement in a SUBROUTINE Subprogram

General Form
RETURN

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

EXTERNAL Statement

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

The EXTERNAL statement is a specification statement and must appear prior to any executable statement in the source program.

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 subprograms names in the following statements:

Example 1:

```
    Calling Program                   Subprogram

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

Explanation:

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

Example 2:

```
                .
                .
                .
    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.

Source programs may be coded in either BCDIC or EBCDIC character codes. Mixing of the two, however, is not allowed.

Alphabetic Characters	EBCDIC or BCDIC Card Punches	Numeric Characters	EBCDIC or BCDIC Card Punches
A	12-1	0	0
B	12-2	1	1
C	12-3	2	2
D	12-4	3	3
E	12-5	4	4
F	12-6	5	5
G	12-7	6	6
H	12-8	7	7
I	12-9	8	8
J	11-1	9	9
K	11-2		
L	11-3		
M	11-4		
N	11-5		
O	11-6		
P	11-7		
Q	11-8		
R	11-9		
S	0-2		
T	0-3		
U	0-4		
V	0-5		
W	0-6		
X	0-7		
Y	0-8		
Z	0-9		
$	11-8-3		

Special Characters	EBCDIC Card Punches	BCDIC Card Punches	Dual Characters
(blank)	(no punch)	(no punch)	
-	11	11	
*	11-8-4	11-8-4	
.	12-8-3	12-8-3	
, (comma)	0-8-3	0-8-3	
+	12-8-6	12	& (ampersand)
/	0-1	0-1	
=	8-6	8-3	#
(	12-8-5	0-8-4	%
)	11-8-5	12-8-4	⌑
' (apostrophe)	8-5	8-4	@

APPENDIX B: BASIC FORTRAN IV IMPLEMENTATION DIFFERENCES

The differences among the four implementations of the Basic FORTRAN IV language are minor except for the absence of the direct access input/output statements in Basic Programming Support FORTRAN IV. These differences are indicated in the body of this publication by footnotes and, in addition, are summarized in Table 5.

The following abbreviations are used in Table 5:

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

Table 5. Implementation Differences

	OS	DOS/TOS	BPS
Direct-access input/output statements	Direct-access I/O is available.	Direct-access I/O is available. A program with direct-access I/O may be compiled using either DOS or TOS, but it must be executed using DOS.	Direct-access I/O is not available.
Key word and blank restriction	Control card option may be used to either keep or remove the restriction. See the programmer's guide.	The restriction has been removed; there is no option to retain it.	The restriction has been removed; there is no option to retain it.
Maximum array size	131,068 storage locations	32,767 storage locations	32,767 storage locations
Maximum record size	Depends upon the input/output device in use; See the programmer's guide.	255 characters per record	255 characters per record
Order of Specification statements	Specification statements that describe data must precede any statements which refer to that data. EQUIVALENCE statements must follow the explicit specification, COMMON, and DIMENSION statements but do not have to follow the DEFINE FILE statement.	Specification statements that describe data must precede any statements which refer to that data. EQUIVALENCE statements must follow the explicit specification, COMMON, and DIMENSION statements but do not have to follow the DEFINE FILE statement.	The EQUIVALENCE statements must follow the explicit specification, COMMON, and DIMENSION statements.
Subprogram names	Any valid FORTRAN name may be used unless key word restriction is retained; key words may not then be used.	Any valid FORTRAN name may be used.	Any valid FORTRAN name may be used.

APPENDIX C: FORTRAN SUPPLIED SUBPROGRAMS

The FORTRAN supplied subprograms are either in-line or out-of-line. An in-line subprogram is inserted by the FORTRAN compiler at any point in the program where the subprogram is referenced. The in-line subprograms are mathematical function subprograms. These subprograms are listed in Table 6.

The out-of-line subprograms are located on a library. These subprograms are mathematical function subprograms and service subprograms. The out-of-line mathematical function subprograms are listed in Table 7; out-of-line service subprograms are listed in Table 8. A detailed description of all out-of-line subprograms is contained in the publication IBM System/360 Operating System: FORTRAN IV (E), Library Subprograms.

Note: Variables used as arguments of any mathematical function subprogram must be defined in accordance with the function in which they appear. This definition is accomplished either with the Explicit specification statement or with the predefined convention.

Table 6. In-Line Mathematical Function Subprograms

Function	Entry Name	Definition	No. of Arg.	Argument Type	Function Value Type
Absolute value	IABS ABS DABS	$\lvert$Arg$\rvert$	1 1 1	Integer Real Double Precision	Integer Real Double Precision
Float	FLOAT DFLOAT	Convert from integer to real	1 1	Integer Integer	Real Double Precision
Fix	IFIX	Convert from real to integer	1	Real	Integer
Transfer of sign	SIGN ISIGN DSIGN	Sign of Arg$_2$ times $\lvert$Arg$_1\rvert$	2 2 2	Real Integer Double Precision	Real Integer Double Precision
Positive difference	DIM IDIM	Arg$_1$-Min(Arg$_1$, Arg$_2$)	2 2	Real Integer	Real Integer
Obtaining most significant part of a Double Precision argument	SNGL		1	Double Precision	Real
Express a Real argument in Double Precision form	DBLE		1	Real	Double Precision

Table 7. Out-of-Line Mathematical Function Subprograms

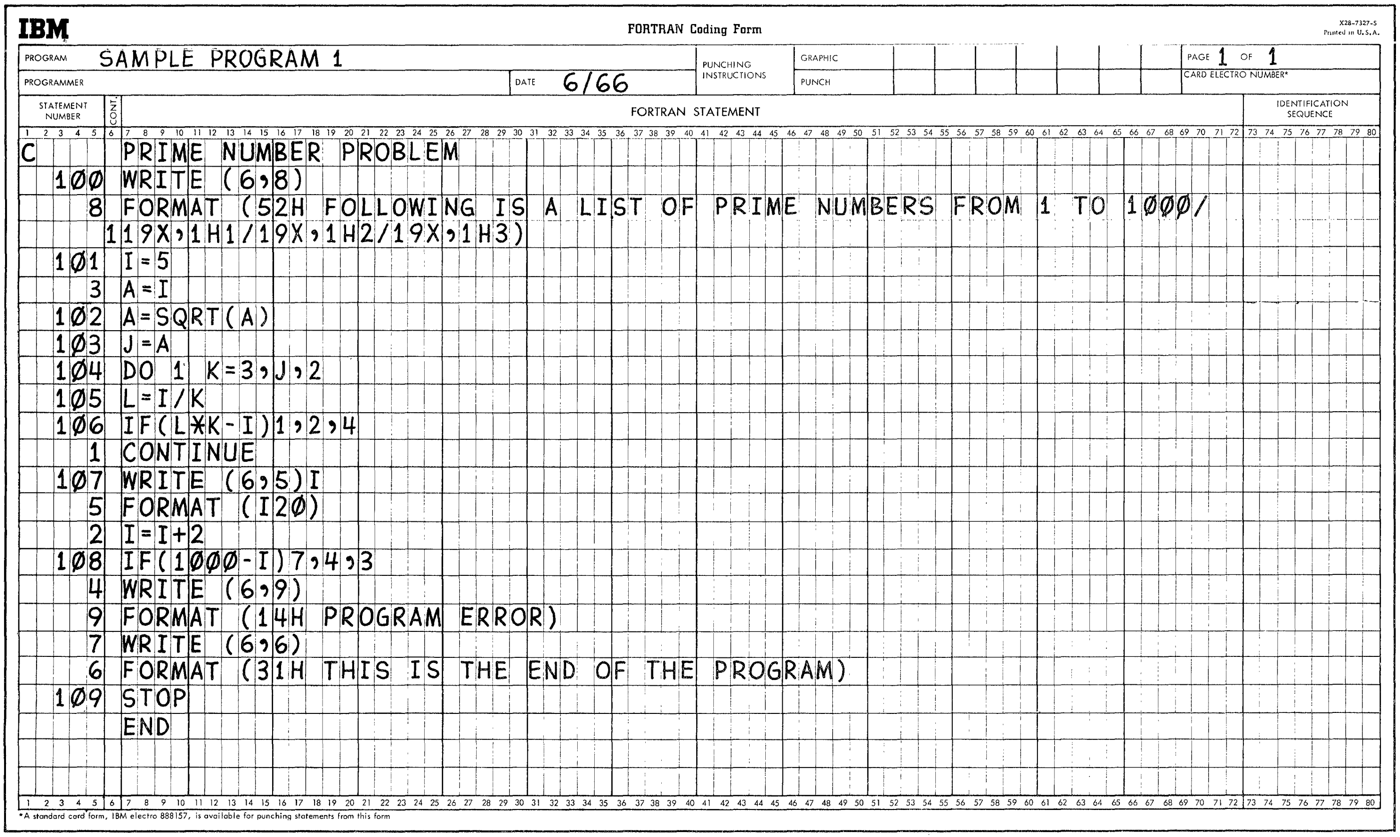
Function	Entry Name	Definition	No. of Arg.	Argument Type	Function Value Type
Exponential	EXP	e^{arg}	1	Real	Real
	DEXP	e^{arg}	1	Double Precision	Double Precision
Natural Logarithm	ALOG	Ln(Arg)	1	Real	Real
	DLOG	Ln(Arg)	1	Double Precision	Double Precision
Common Logarithm	ALOG10	Log_{10} (Arg)	1	Real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

Table 8. Out-of-Line Service Subprograms

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

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 X28-7327-5 Printed in U.S.A.

PROGRAM SAMPLE PROGRAM 1
PROGRAMMER
DATE 6/66
PUNCHING INSTRUCTIONS GRAPHIC PUNCH
PAGE 1 OF 1
CARD ELECTRO NUMBER*

```
C     PRIME NUMBER PROBLEM
  1ØØ WRITE (6,8)
    8 FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1ØØØ/
     119X,1H1/19X,1H2/19X,1H3)
  1Ø1 I=5
    3 A=I
  1Ø2 A=SQRT(A)
  1Ø3 J=A
  1Ø4 DO 1 K=3,J,2
  1Ø5 L=I/K
  1Ø6 IF(L*K-I)1,2,4
    1 CONTINUE
  1Ø7 WRITE (6,5)I
    5 FORMAT (I2Ø)
    2 I=I+2
  1Ø8 IF(1ØØØ-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)
  1Ø9 STOP
      END
```

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

Figure 2. Sample Program 1

SAMPLE PROGRAM 2

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

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

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

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

where:

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

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

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

$$\vdots \qquad \vdots$$

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

$$\vdots$$

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

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).

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

The forward solution is as follows:

1. Divide equation (1) by W_o
2. Multiply the equation resulting from step 1 by W_1 and subtract from equation (2).
3. Multiply the equation resulting from step 1 by W_2 and subtract from equation (3).

The resulting equations are:

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

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

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

where:

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

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

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

Steps 1 and 2 are repeated using equations (5) and (6), with b_{22} and b_{32} instead of W_0 and W_1. The resulting equations are:

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

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

where:

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

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

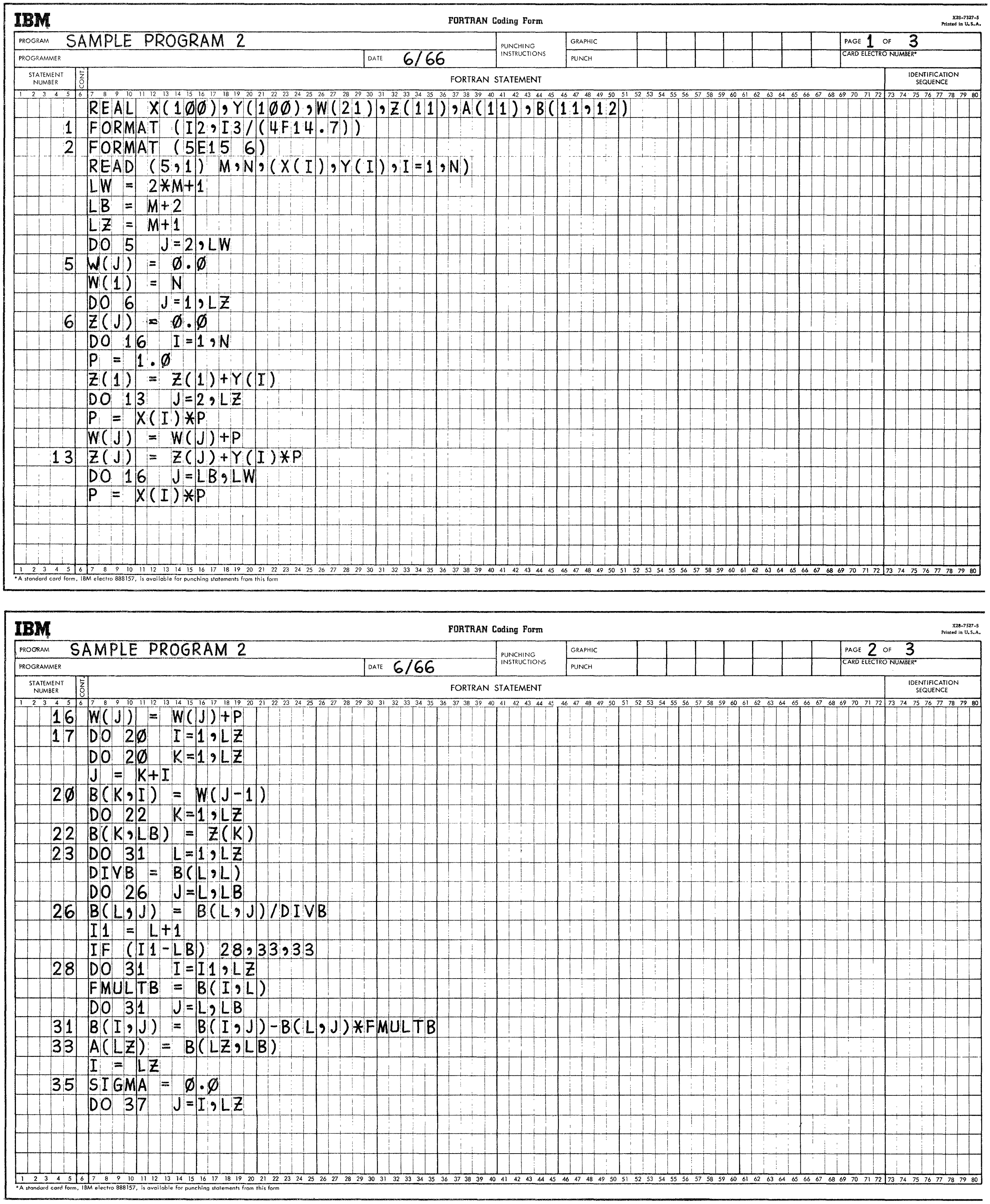
The backward solution is as follows:

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

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

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

Figure 3 is a possible FORTRAN program for carrying out the calculations for the case: $n = 100$, $m \le 10$. $W_0, W_1, W_2, \ldots, W_{2m}$ are stored in W(1), W(2), W(3), ..., W(2M+1), respectively. $Z_0, Z_1, Z_2, \ldots, Z_m$ are stored in Z (1), Z (2), Z (3), ..., Z(M+1), respectively.



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

PROGRAM SAMPLE PROGRAM 2 — PROGRAMMER — DATE 6/66 — PUNCHING INSTRUCTIONS — GRAPHIC — PUNCH — PAGE 1 OF 3 — CARD ELECTRO NUMBER*

STATEMENT NUMBER | CONT. | FORTRAN STATEMENT | IDENTIFICATION SEQUENCE

```
      REAL X(1ØØ),Y(1ØØ),W(21),Z(11),A(11),B(11,12)
    1 FORMAT (I2,I3/(4F14.7))
    2 FORMAT (5E15 6)
      READ (5,1) M,N,(X(I),Y(I),I=1,N)
      LW = 2*M+1
      LB = M+2
      LZ = M+1
      DO 5  J=2,LW
    5 W(J) = Ø.Ø
      W(1) = N
      DO 6  J=1,LZ
    6 Z(J) = Ø.Ø
      DO 16  I=1,N
      P = 1.Ø
      Z(1) = Z(1)+Y(I)
      DO 13  J=2,LZ
      P = X(I)*P
      W(J) = W(J)+P
   13 Z(J) = Z(J)+Y(I)*P
      DO 16  J=LB,LW
      P = X(I)*P
```

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

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

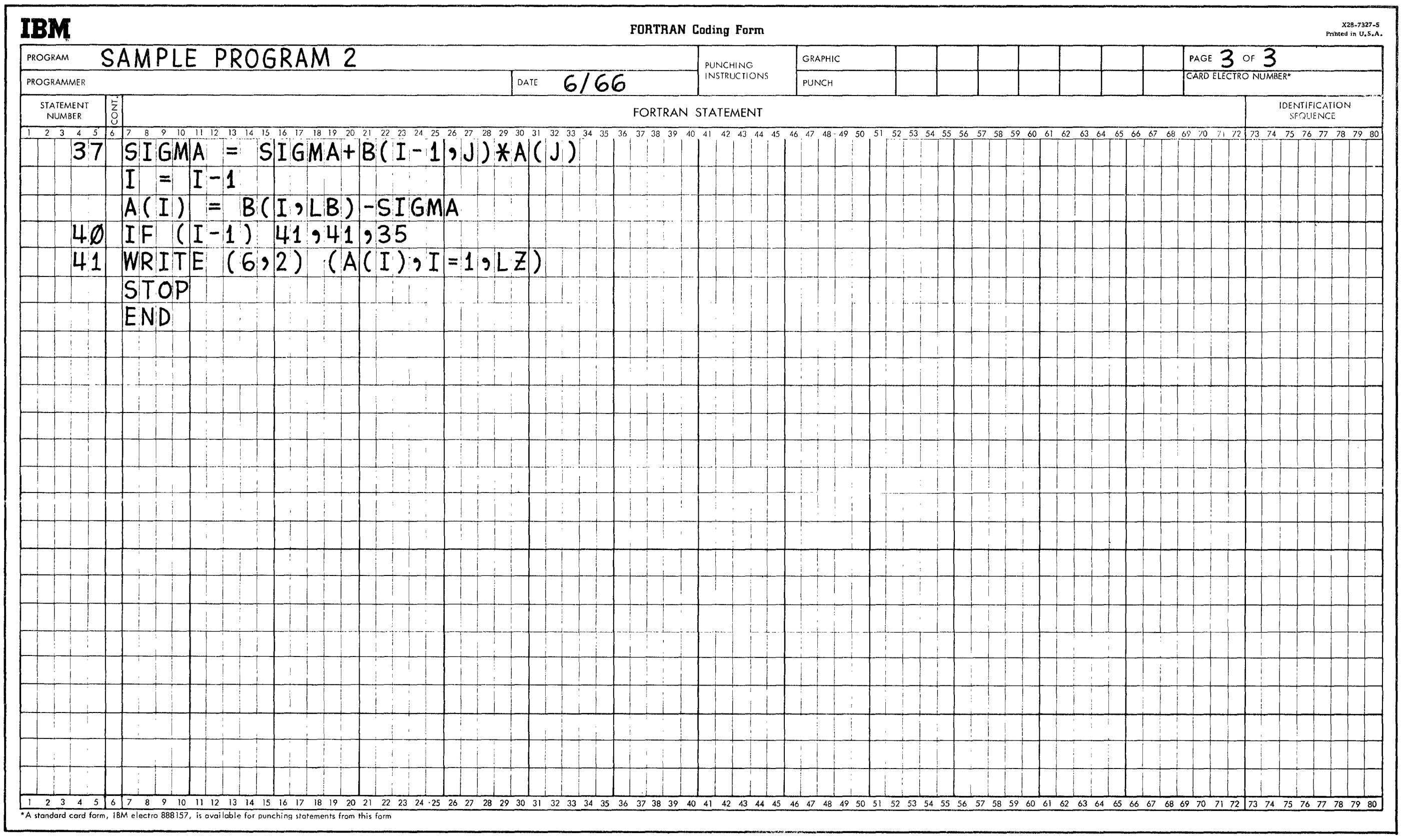
PROGRAM SAMPLE PROGRAM 2 — PROGRAMMER — DATE 6/66 — PUNCHING INSTRUCTIONS — GRAPHIC — PUNCH — PAGE 2 OF 3 — CARD ELECTRO NUMBER*

STATEMENT NUMBER | CONT. | FORTRAN STATEMENT | IDENTIFICATION SEQUENCE

```
   16 W(J) = W(J)+P
   17 DO 2Ø  I=1,LZ
      DO 2Ø  K=1,LZ
      J = K+I
   2Ø B(K,I) = W(J-1)
      DO 22  K=1,LZ
   22 B(K,LB) = Z(K)
   23 DO 31  L=1,LZ
      DIVB = B(L,L)
      DO 26  J=L,LB
   26 B(L,J) = B(L,J)/DIVB
      I1 = L+1
      IF (I1-LB) 28,33,33
   28 DO 31  I=I1,LZ
      FMULTB = B(I,L)
      DO 31  J=L,LB
   31 B(I,J) = B(I,J)-B(L,J)*FMULTB
   33 A(LZ) = B(LZ,LB)
      I = LZ
   35 SIGMA = Ø.Ø
      DO 37  J=I,LZ
```

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

Figure 3. Sample Program 2



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

PROGRAM SAMPLE PROGRAM 2

PROGRAMMER

DATE 6/66

PUNCHING INSTRUCTIONS GRAPHIC PUNCH

PAGE 3 OF 3

CARD ELECTRO NUMBER*

STATEMENT NUMBER | CONT. | FORTRAN STATEMENT | IDENTIFICATION SEQUENCE

```
   37 SIGMA = SIGMA+B(I-1,J)*A(J)
      I = I-1
      A(I) = B(I,LB)-SIGMA
   4Ø 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:

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

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

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

$$\vdots \qquad \vdots$$

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

$$\vdots$$

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

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:

$$\begin{array}{cccc} W_0 & W_1 & W_2 & Z_0 \\ W_1 & W_2 & W_3 & Z_1 \\ W_2 & W_3 & W_4 & Z_2 \end{array}$$

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:

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

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.

APPENDIX E: FORTRAN IV FEATURES NOT IN BASIC FORTRAN IV

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

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

READER'S COMMENTS

Title: IBM System/360
Basic FORTRAN IV Language

Form: C28-6629-0

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 ______________________________

fold

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

fold

FOLD ON TWO LINES,STAPLE AND MAIL
No Postage Necessary if Mailed in U.S.A.

staple **staple**

fold fold

FIRST CLASS
PERMIT 33504
NEW YORK, N.Y.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

IBM CORPORATION
1271 AVENUE OF THE AMERICAS
NEW YORK, NEW YORK 10020

ATTN: PROGRAMMING SYSTEMS PUBLICATIONS
DEPARTMENT D39

fold fold

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]

staple

C28-6629-0

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]