File No. 7040-25
Form C-28-6329-2

IBM Systems Reference Library

IBM 7040/7044 Operating System (16/32K)

FORTRAN IV Language

This publication describes the 7040/7044 FORTRAN IV language, the language accepted by the 16/32K FORTRAN IV compiler – 7040-FO-815. FORTRAN is a problem-oriented automatic coding system designed primarily for scientific and engineering computations, and it closely resembles the ordinary language of mathematics. The 7040/7044 FORTRAN IV compiler (IBFTC) operates under control of the 7040/7044 Processor Monitor (IBJOB), a component of the 7040/7044 Operating System.

Preface

This publication describes the language of the 7040/7044 FORTRAN IV compiler (IBFTC), which is a part of the 7040/7044 Processor. Before reading this publication, the reader should refer to the publication, *IBM 7040/7044 Operating System (16/32K): Programmer's Guide,* Form C28-6318, for a complete discussion of the use of the Processor.

7040/7044 FORTRAN IV, hereafter referred to as FORTRAN, is a component language of the Processor. It is assumed that the reader is thoroughly familiar with the basic concepts of FORTRAN, which are in the *FORTRAN General Information Manual,* Form F28-8074-1.

The minimum machine requirements for the use of the FORTRAN IV compiler are described in the *Programmer's Guide,* Form C28-6318. Operating instructions for the operator are in the publication, *IBM 7040/7044 Operating System (16/32K): Operator's Guide,* Form C28-6338.

MAJOR REVISION (December, 1964)
This publication, Form C28-6329-2, makes obsolete the previous edition, Form C28-6329-1. Changes have been made to incorporate material on an expansion of the previously existing DIMENSION statement; in addition, changes have been made to incorporate material on the DATA statement and on the BLOCK DATA subprogram. These changes are a part of Version 8 of the operating system.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Address comments concerning the contents of this publication to:
IBM Corporation, Programming Systems Publications, Dept. D39, 1271 Avenue of the Americas, New York, N. Y. 10020

Contents

A FORTRAN source program consists of a sequence of *source statements*, which are described in detail in the following chapters.

Punching a Source Program

Each statement of a FORTRAN source program is punched into a separate card (the standard FORTRAN card form is shown in Figure 1); however, if a statement is too long to fit on one card, it can be continued on as many as nine continuation cards. The order of the source statements is governed solely by the order of the statement cards.

Cards that contain a C in column 1 are not processed by the FORTRAN compiler. Such cards can be used for comments that appear when the source program deck is listed.

A number punched in columns 1-5 of the first card statement is the *statement number* of that statement. It permits cross-referencing within a source program. Statement numbers can be assigned in any order, since they do not affect the sequence of operation.

Column 6 of the first card of a statement must be blank or punched with a zero. However, in continuation cards (other than for comments), column 6 must be punched with some character other than zero. Continuation cards for comments need not be punched in column 6; only the C in column 1 is necessary.

The FORTRAN statements themselves are punched in columns 7-72, both on initial cards and on continuation cards. Thus, a statement can consist of not more than 660 characters (i.e., one initial card and nine continuation cards). A table of the admissible characters for FORTRAN is in Appendix A. Blank characters, except in column 6 and in alphameric fields (H fields of FORMAT statements or alphameric arguments of CALL statements), are ignored by FORTRAN and may be used freely to improve the readability of the source program listing.

Columns 73-80 are not processed by FORTRAN and may be punched with any desired identifying information.

The brief program shown in Figure 2 illustrates the general appearance and some of the properties of a FORTRAN program. It is shown as coded on a standard FORTRAN coding sheet.

The purpose of this program is to determine the largest value contained in a set of numbers, A_I [represented by the notation A (I)], and to write the numbers on logical unit 6. The numbers exist on an input medium, 12 to a record, each number occupying a field of 6 columns. The actual size of the set is given in the first record and is the only data in that record. The input medium is logical unit 5.

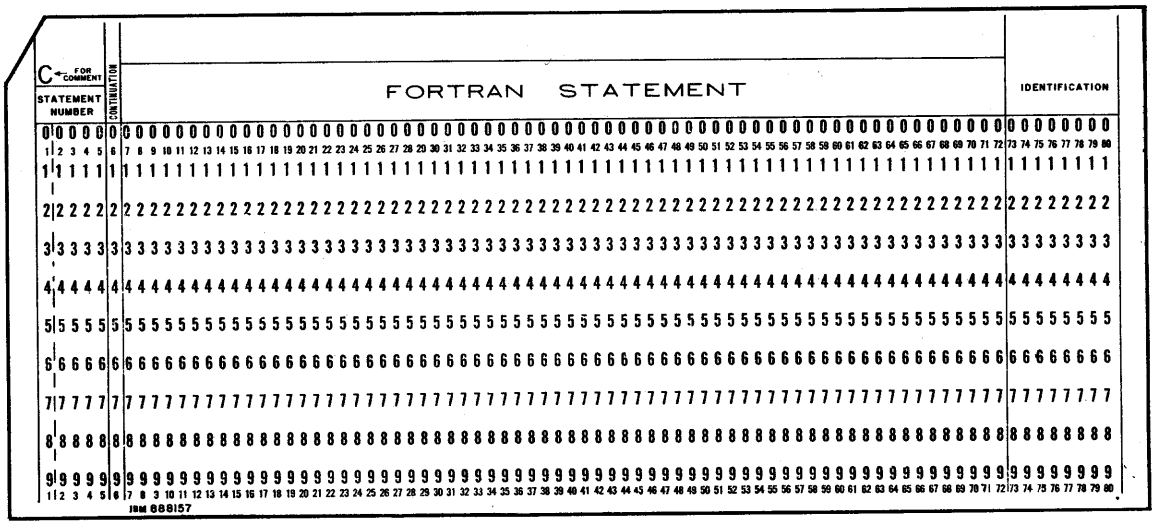
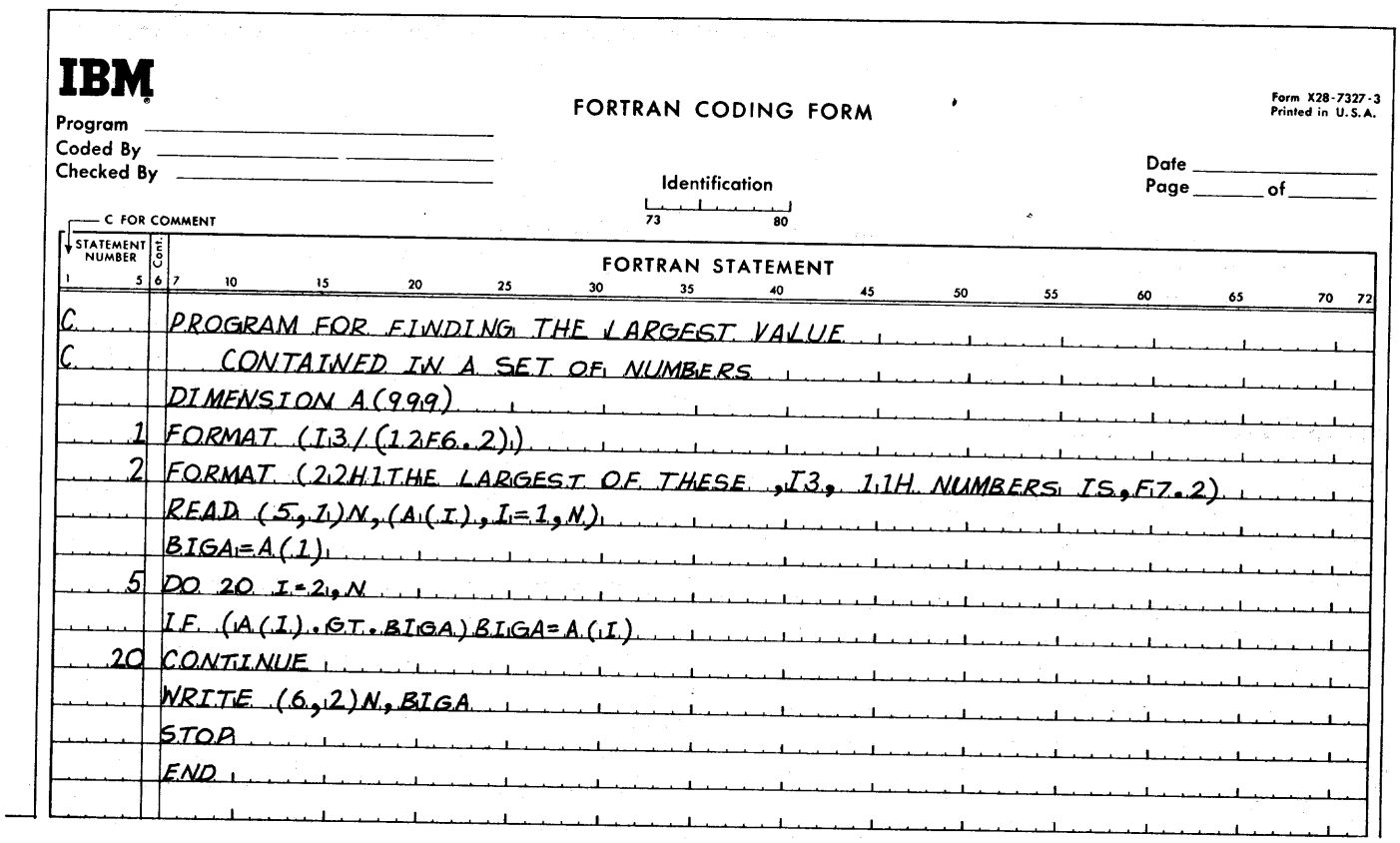


Figure 1. Standard FORTRAN Card



IBM

FORTRAN CODING FORM

Form X28-7327-3
Printed in U.S.A.

Program
Coded By
Checked By

Identification (73–80)

Date
Page of

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–72)
C		PROGRAM FOR FINDING THE LARGEST VALUE
C		CONTAINED IN A SET OF NUMBERS
		DIMENSION A(999)
1		FORMAT (I3/(12F6.2))
2		FORMAT (22H1THE LARGEST OF THESE ,I3, 11H NUMBERS IS ,F7.2)
		READ (5,1)N,(A(I),I=1,N)
		BIGA=A(1)
5		DO 20 I=2,N
		IF (A(I).GT.BIGA) BIGA=A(I)
20		CONTINUE
		WRITE (6,2)N,BIGA
		STOP
		END

Figure 2. Example of a FORTRAN Program

Types of FORTRAN Statements

The types of statements that can be used in a FORTRAN program are classified as follows:

Arithmetic or Logical Assignment Statements specify a numerical or logical computation. The symbols available for referring to constants, variables, and functions are discussed in the section "Constants, Variables, Subscripts, and Expressions;" the section "Arithmetic and Logical Assignment Statements" gives the rules for combining these into arithmetic and logical assignment statements.

Control Statements govern the flow of control in the program. These statements and the END statement are discussed in the section "Control Statements."

Input/Output Statements provide the necessary input and output functions. These statements are discussed in the section "Input/Output Statements."

Subprogram Statements enable the programmer to define and use subprograms. The use of subprograms is discussed in the section "Subroutines, Functions, and Subprogram Statements."

Declarative Statements provide information to the compiler regarding certain properties of names appearing in other statements, such as the type assumed by a variable or the dimensions of an array of numbers. These statements are discussed in the section "The Specification Statements."

FORTRAN provides a means of expressing numerical constants and variable quantities. A subscript notation is also provided to express one-, two-, or three-dimensional arrays of variables.

Constants

The following types of constants can be used in the FORTRAN source program language: integer (fixed point), real (floating point), double-precision, complex, and two special logical constants.

Integer Constants

An integer constant consists of one to eleven decimal digits written without a decimal point; the constant may be signed or unsigned.

EXAMPLES:

14
+3
−47
2898762403

An integer constant can be as large as $2^{35}-1$, except when used for the value of a subscript or as an index of a DO statement, in which case the value of the integer must be less than 2^{15}.

Real Constants

A real (floating point) constant consists of one to nine decimal digits, with a decimal point at the beginning, at the end, or between two digits.

A real constant can be followed by a decimal exponent that is written as the letter E followed by the exponent. The field following the letter E must not be blank; it may be zero. The decimal point may be omitted in a whole number written with an E exponent.

EXAMPLES:

15.4	
5.	
.0003	
5.0E3	(5.0×10^3; i.e., 5000)
5.0E−3	(5.0×10^{-3}; i.e., .005)
5E2	($5. \times 10^2$; i.e., 500; FORTRAN will treat this form as if the decimal point were punched between the 5 and the E)

A real constant may have up to nine significant digits that have a value not greater than $2^{27}-1$. The magnitude of the constant expressed must lie between the approximate limits of 10^{-38} and 10^{38}, or must be zero.

Double-Precision Constants

A double-precision constant consists of one to seventeen significant decimal digits written with a decimal point.

To specify a decimal exponent or a constant containing 10 or fewer digits, the programmer must use the letter D, followed by the exponent, after the number; however, when there are more than 10 digits, the D is not necessary, unless the programmer wishes to use an exponent.

The exponent is an integer constant. The field following the letter D must not be blank; it may be zero. The decimal point may be omitted in a whole number written with a D exponent.

EXAMPLES:

21.987538229	
.203D0	
5.0D3	(5.0×10^3; i.e., 5000)
5D2	($5. \times 10^2$; i.e., 500; FORTRAN will treat this form as if the decimal point were punched between the 5 and the D)

A double-precision constant is a floating-point quantity. It may have up to 17 significant decimal digits that have a value not greater than $2^{54}-1$. The magnitude of the constant expressed must lie between the approximate limits of 10^{-29} and 10^{38}, or must be zero. Numbers between 10^{-29} and 10^{-38} may also be used, but only eight digits are significant in this range.

Complex Constants

A complex constant consists of an ordered pair of real constants separated by a comma and enclosed in parentheses.

EXAMPLES:

(3.2, 1.86) represents 3.2+1.86i
(2.1, 0.0) represents 2.1+0.0i
(5.0E3, −2.12) represents 5000. −2.12i
Y=CLOG ((3.0,+1.33))
Z=(4.17, −1.0)+(18.28, 2.2)

The first real constant represents the real part of the complex number, and the second real constant represents the imaginary part of the complex number. The parentheses are required regardless of the context in which the complex constant appears. Each part of the complex constant may be preceded by a + (plus) or a − (minus) sign.

Logical Constants

A logical constant may be defined as either .TRUE. or .FALSE. and then used in a logical IF statement (see "Logical IF Statement"), as in this example:

```
A = .TRUE.
.
.
IF(A) GO TO 20
```

In the example, the logical constant .TRUE. has been stored in A. According to the logical IF statement, the program will go to statement number 20. However, if .FALSE. had been stored in A, the program would have continued on to the next sequential statement.

Variables

A variable is specified by its name and type. The following types of variables are permissible: integer, real, double-precision, complex, and logical. The rules for naming each type of variable are in the following text.

Variable Names

A variable name consists of one to six alphameric characters, the first of which is alphabetic. A variable name may not contain a special character (* $ @ etc.).

EXAMPLES:

L5
JOB1
BETATS
COST
K

Subroutines are named in the same manner as variables (the section "Naming Subroutines" has additional information).

The names of the built-in functions cannot be used as variable or subroutine names [the sections "Built-in (or Open) Functions" and "Appendix E," under "Built-in Features," have additional information].

The same name should not be used for more than one purpose in the same program.

Variable Type Specification

The type of a real or integer variable name or a real or integer function name may be specified in one of two ways: implicitly by name, or explicitly by a Type statement (the section "Type Statements" has additional information). Double-precision, complex, and logical variables must be specified by a Type statement.

Implicit Type Assignment

A variable or a function is considered *integer* if the first character of the name is I, J, K, L, M, or N; e.g., MAX, JOB, I, M2.

A variable or a function is considered *real* if the first character of the name is not I, J, K, L, M, or N; e.g., DELTA, BMAX, A, B7.

Subscripts

A variable can represent any element of a one-, two- or three-dimensional array of quantities if the user appends one, two, or three subscripts, respectively, to the variable name. The variable is then a *subscripted variable*. The value of the subscript determines the member of the array to which reference is made.

Forms of Subscripts

GENERAL FORM
Let v represent any unsigned, nonsubscripted integer variable and c (or c') any unsigned integer constant. Then, a subscript is an expression that may take any of the following forms: v c v+c or v−c c*v c*v+c' or c*v−c'

EXAMPLES:

IMAS
3
MU+2
5*MU−6
9+J (Invalid: for addition, the variable must precede the constant.)
K*2 (Invalid: for multiplication, the constant must precede the variable.)

Real quantities may not appear in subscripts; nor may constants be signed.

Subscripted Variables

A subscripted variable consists of a variable name followed by parentheses enclosing one, two, or three subscripts that are separated by commas.

EXAMPLES:

A (1)
K (3)
BETA (8*J+2, K−2,L)
MAX (I, J, K)

During execution, the subscript is evaluated so that the subscripted variable refers to a specific member of the array. The value of a subscript must be greater than zero but not greater than the corresponding array dimension. The size of each array must be specified before the first appearance of the subscripted variable. This is accomplished with a DIMENSION statement, a dimensioned COMMON statement, or a dimensioned Type statement.

Arrangement of Arrays in Storage

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

Example: The two-dimensional array $A_{m,n}$ is stored as follows, from lowest storage location to highest:

$A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, A_{2,2}, \ldots, A_{m,2}, \ldots, A_{m,n}$

Expressions

The FORTRAN language includes two kinds of expressions: arithmetic and logical. An expression is a sequence of constants, variables (subscripted or nonsubscripted), and operation symbols that indicates a quantity or a series of calculations. It must be formed according to the rules for constructing expressions. The expression may include commas and parentheses and may also include functions (to be discussed later).

Arithmetic Expressions

An arithmetic expression consists of certain sequences of constants, subscripted and nonsubscripted variables, and arithmetic function references separated by arithmetic operation symbols, commas, and parentheses.

The arithmetic operation symbols +, −, *, /, ** denote addition, subtraction, multiplication, division, and exponentiation, respectively. The rules for forming arithmetic expressions are:

1. Figures 3 and 4 indicate which constants, variables, and functions may be combined by the arithmetic operators to form arithmetic expressions. Figure 3 gives the valid combinations with respect to the following arithmetic operators: +, −, *, and /. Figure 4 gives the valid combinations with respect to the arithmetic operator **. In these figures, Y indicates a valid combination and N indicates an invalid combination.

+ − * /	Real	Integer	Complex	Double-Precision	Logical
Real	Y	N	Y	Y	N
Integer	N	Y	N	N	N
Complex	Y	N	Y	N	N
Double-Precision	Y	N	N	Y	N
Logical	N	N	N	N	N

Figure 3

Exponent

Base \ **	Real	Integer	Complex	Double-Precision	Logical
Real	Y	Y	N	Y	N
Integer	N	Y	N	N	N
Complex	N	Y	N	N	N
Double-Precision	Y	Y	N	Y	N
Logical	N	N	N	N	N

Figure 4

2. The simplest expression consists of a single constant, a single variable, or a subscripted variable. If the quantity is integer, the expression is said to be of the integer type. If the quantity is real, the expression is said to be of the real type.

A real constant, variable, or function name combined with a double-word quantity forms an expression of the same type as the double-word quantity; e.g., a real variable plus a complex variable forms a complex expression.

3. Quantities can be preceded by a + or a −, which does not affect the type of the expression. Also, expressions can be connected by any of the arithmetic operators (+, −, *, /) to form other expressions, provided:

   a. No two operators appear consecutively.
   b. All operators are explicitly expressed.

4. Any expression may be enclosed in parentheses. Parentheses can be used to specify the order in which the operations in the expression are to be computed. Where parentheses are omitted, the hierarchy of operations is:

   a. Function Reference
   b. Exponentiation **
   c. Multiplication and Division *and/
   d. Addition and Subtraction +and−

For example, the expression A+B/C+D**E*F−G will be taken to mean A + (B/C) + (D^{E*F}) −G.

(The expression A**B**C is not permitted; it must be written as either A**(B**C) or (A**B)**C, whichever is intended.)

Logical Expressions

The second type of expression is the logical expression. The logical operation symbols (where *a* and *b* are logical expressions) are:

SYMBOL	DEFINITION
.NOT.*a*	This has the value true only if *a* is false; it has the value false only if *a* is true.
a.AND.*b*	This has the value true only if both *a* and *b* are true; it has the value false if either *a* or *b* is false or both *a* and *b* are false.
a.OR.*b*	(Inclusive OR) This has the value true if either *a* or *b* is true or if both *a* and *b* are true; it has the value false only if both *a* and *b* are false.

The logical operators NOT, AND, and OR must always be preceded and followed by a period.

The relational operators are used to compare arithmetic quantities. The relational operation symbols (where *x* and *y* are arithmetic expressions) are:

SYMBOL	DEFINITION
x.GT.*y*	Greater than True if *x* is greater than *y*; false if *x* is less than or equal to *y*.

SYMBOL	DEFINITION
x.GE.*y*	Greater than or equal to True if *x* is greater than or equal to *y*; false if *x* is less than *y*.
x.LT.*y*	Less than True if *x* is less than *y*; false if *x* is greater than or equal to *y*.
x.LE.*y*	Less than or equal to True if *x* is less than or equal to *y*; false if *x* is greater than *y*.
x.EQ.*y*	Equal to True if *x* is equal to *y*; false if *x* is not equal to *y*.
x.NE.*y*	Not equal to True if *x* is not equal to *y*; false if *x* is equal to *y*.

The relational operators must always be preceded and followed by a period.

A logical expression consists of certain sequences of logical constants, references to logical functions, logical variables (which must be separated by logical operation symbols), and arithmetic expressions (which must be separated by relational operation symbols). A logical expression always has the value .TRUE. or .FALSE. .

Rules for constructing logical expressions are:

1. Figure 5 indicates which constants, variables, and functions may be combined by the relational operators to form a logical expression. In this figure, Y indicates a valid combination and N indicates an invalid combination.

2. A logical expression can consist of a single logical constant, a logical variable, or a reference to a logical function.

.GT. .GE. .LT. .LE. .EQ. .NE.	Real	Integer	Complex	Double-Precision	Logical
Real	Y	N	N	Y	N
Integer	N	Y	N	N	N
Complex	N	N	N	N	N
Double-Precision	Y	N	N	Y	N
Logical	N	N	N	N	N

Figure 5

3. The logical operator .NOT. must be followed by a logical expression, and the logical operators .AND. and .OR. must be preceded and followed by logical expressions to form more complex logical expressions.

4. Any logical expression can be enclosed in parentheses; however, the logical expression to which the .NOT. applies must be enclosed in parentheses if it contains two or more quantities as, for example, .NOT. (a.AND.b), where a and b are logical expressions or .NOT. (x.LE.y), where x and y are arithmetic expressions.

5. Parentheses can be used in logical expressions to specify the order in which the expression is to be computed. Where parentheses are omitted, the hierarchy of operation is:

a. Arithmetic Operations
b. .LT.,.LE.,.EQ.,.NE.,.GT.,.GE.
c. .AND.
d. .OR.

Since the logical operator .NOT. does not connect two operands, it does not appear in the hierarchy of operations.

Each arithmetic statement or logical statement defines a numerical or a logical calculation. These FORTRAN statements closely resemble conventional arithmetic formulas. However, the equal sign in a FORTRAN statement specifies replacement, or "assignment," rather than equality. Thus, A = 4.0 means "assign the quantity 4.0 to A."

GENERAL FORM
a=b where: 1. a is any type of variable (subscripted or nonsubscripted). 2. b is an expression.

EXAMPLES:

```
QI=K
A(I)=B(I) - SIN (C(I))
V=.TRUE.
E=C.GT.D.AND.F.LE.G
```

Figure 6 indicates which type of expression can be equated to which type of variable in an arithmetic or a logical statement. In this figure, Y indicates a valid statement and N indicates an invalid statement.

		Right Side of Equal Sign				
Left Side of Equal Sign	Expression Variable	Real	Integer	Complex	Double-Precision	Logical
	Real	Y	Y	N	Y	N
	Integer	Y	Y	N	Y	N
	Complex	Y	N	Y	N	N
	Double-Precision	Y	Y	N	Y	N
	Logical	N	N	N	N	Y

Figure 6

If the variable on the left and the expression on the right are the same type, the computation is done in that type.

The following rules hold for the manner in which the expression is evaluated and the result stored, if the type of the expression on the right and the type of the variable on the left are different.

1. If the variable on the left is integer and the expression on the right is real, the result is computed as real, truncated to the largest integer it contains, and converted to integer.

2. If the variable on the left is real and the expression on the right is integer, the result of the expression is computed as integer and is converted to real.

3. If the variable on the left is logical, the expression on the right must be logical.

4. If the variable on the left is double-precision and the expression on the right is real or integer, the expression is converted and the result computed as double-precision.

5. If the variable on the left is real or integer and the expression on the right is double-precision, the expression is evaluated in double-precision and the result is truncated.

6. If a relational expression includes both a real quantity and a double-precision quantity, the real quantity is converted to double-precision before the relation is evaluated.

In the following examples of arithmetic statements, I is an integer variable, A and B are real variables, DD is a double-precision variable, CP is a complex variable, and G, H, and P are logical variables.

```
A=B
```
Replace A by the current value of B.

```
I=B
```
Truncate B to an integer, convert it to an integer constant, and store it in I.

```
A=I
```
Convert I to a real variable and store it in A.

```
I=I+1
```
Add 1 to I and store it in I.

```
A=3*B
```
Not permitted. The expression is mixed for multiplication, i.e., it contains both a real variable and an integer constant.

```
DD=A+B
```
Convert A and B to double-precision, compute their sum, and store it as double-precision in DD.

```
A=DD+B
```
Convert B to double-precision, add B to DD, truncate the sum, and store it in A.

```
A=DD+SIN(X)
```
Convert the result of SIN(X) to double-precision, add DD, truncate the sum, and store it in A.

```
DD .GT. A
```
Convert A to double-precision and compare it with DD.

```
A .LE. CP
```
Not permitted. A complex quantity may not appear in a logical expression.

```
G=.TRUE.
```
Store the logical constant .TRUE. in G.

```
H=.NOT.G
```
If G is .TRUE., store the value .FALSE. in H; if G is .FALSE., store the value .TRUE. in H.

```
H=I.GE.A
```
Not permitted. An integer and a real variable may not be joined by a relational operator.

G=H. OR . . NOT. P

H	P	~P	Hv~P
T	T	F	T
T	F	T	T
F	T	F	F
F	F	T	T

where:
~ implies .NOT., and v implies .OR.

G=3 . . GT.B
G is .TRUE. if 3. is greater than B; G is .FALSE. otherwise.

The last two examples illustrate the following rules:

1. Two logical operators can appear in sequence only if the second logical operator is .NOT. .

2. Two decimal points may appear in succession only when the situation described in item 1 occurs or when one decimal point belongs to a constant and the other to a relational operator.

The second class of FORTRAN statements is a set of control statements. With this set the programmer can control the flow and termination of the program.

Unconditional GO TO Statement

This statement interrupts sequential execution and directs flow to the statement that is to be executed next.

GENERAL FORM
GO TO n where: n is a statement number.

EXAMPLE:

GO TO 3

In the example, control is transferred to statement number 3.

Computed GO TO Statement

This statement also interrupts sequential execution and directs flow to the statement that is to be executed next. It differs from the unconditional GO TO statement in that it allows different statements to be executed at various stages in the program.

GENERAL FORM
GO TO (n_1,n_2, . . . ,n_m),i where: 1. n_1, n_2, . . . , are statement numbers. 2. i is a nonsubscripted integer variable.

EXAMPLE:

GO TO (30, 42, 50, 9), I

Control is transferred to the statement numbered n_1, n_2, n_3, . . . , n_m, depending on whether the value of i at the time of execution is 1, 2, 3, . . . , m, respectively. Thus, in the example, if I is 3 at the time of execution, a transfer to the third statement of the list, namely statement 50, occurs.

The computed GO TO statement is used to obtain many computed choices from one statement.

Assigned GO TO Statement

The assigned GO TO is used to obtain one of a number of preset choices from one statement.

GENERAL FORM
GO TO i, (n_1, n_2, . . . ,n_m) where: 1. i is a nonsubscripted integer variable appearing in a previously executed ASSIGN statement. 2. n_1, n_2, . . . ,n_m are statement numbers.

EXAMPLE:

GO TO K, (17, 12, 19)

This statement causes transfer of control to the statement with a statement number equal to that value of i that was last assigned by an ASSIGN statement; n_1, n_2, . . . , n_m are the values that may be assigned to i. In the example, if K had been assigned 12 by a previous ASSIGN statement, a transfer to statement 12 would occur.

ASSIGN Statement

The ASSIGN statement is used in conjunction with the assigned GO TO statement.

GENERAL FORM
ASSIGN n to i where: 1. n is a statement number. 2. i is a nonsubscripted integer variable that appears in an assigned GO TO statement in the same program.

EXAMPLE:

ASSIGN 12 to K

This statement causes a subsequent GO TO K, (n_1, . . . , n_m) to transfer control to statement number 12.

Arithmetic IF Statement

This statement permits the programmer to change the sequence of statement execution, depending upon the value of an arithmetic expression.

GENERAL FORM
IF (a) n_1,n_2,n_3 where: 1. a is an arithmetic expression (it may not be of the complex type). 2. n_1, n_2, and n_3 are statement numbers of executable FORTRAN statements.

EXAMPLE:

IF (A(J,K)−B)10, 4, 30

Control is transferred to the statement numbered n_1, n_2, or n_3 if the value of a is less than, equal to, or greater than zero, respectively. In the example, if A(J,K) − B is less than zero, control will transfer to statement 10; if it is equal to zero, control will transfer to 4; or if it is greater than zero, control will transfer to statement 30.

Logical IF Statement

This statement permits a programmer to change the sequence of statement execution, depending on the value of a logical expression.

GENERAL FORM

IF (t)s
where:
1. t is a logical expression.
2. s is any executable FORTRAN statement except a DO statement or another logical IF statement.

EXAMPLES:

```
IF (A.AND.B) F=SIN(R)
IF (17.GT.L) GO TO 24
IF (D.OR.X.NE.Y) GO TO (18, 20), I
IF (Q) CALL SUB
```

If the logical expression t is true, statement s is executed and control passes to the next statement (unless statement s is an arithmetic IF-type or GO TO-type statement, in which case control is transferred as indicated by the statement). If t is false, control passes to the next sequential statement without statement s being executed.

If t is true and s is a CALL statement, control is transferred to the next sequential statement upon return from the subprogram.

DO Statement

The DO statement is used to cause repetitive execution of a series of statements, as many times as specified.

GENERAL FORM

DO n i=m_1,m_2,m_3
where:
1. n is a statement number.
2. i is a nonsubscripted integer variable.
3. m_1, m_2, and m_3 are each either an unsigned integer constant or an unsigned nonsubscripted integer variable; if m_3 is not stated, FORTRAN assumes it to be 1.

EXAMPLES:

```
DO 30 I=1, 10
DO 24 J=1, M, 2
```

The DO statement results in the repeated execution of the statements that follow the DO, up to and including the statement numbered n. The first time the statements are executed, $i=m_1$. (This takes place even if m_1 exceeds m_2.) For each succeeding execution, i is increased by m_3; i.e., the second time, $i=m_1+m_3$, the third time, $i=m_1+2m_3$, etc. This pattern of execution continues until i is equal to the highest value of this sequence that does not exceed m_2. Control then passes to the statement following the last statement in the range of the DO.

Consider, for example, the following program:

```
.
.
.
10 DO 20 I=1, 20, 2
20 A(I)=I*N(I)
.
.
.
```

This would cause the following computation to take place:

```
A(1)=1*N(1)
A(3)=3*N(3)
A(5)=5*N(5)
.
.
.
A(19)=19*N(19)
```

During each execution, the correct index is substituted for I, I*N(I) is computed, and the result is stored in A(I). When the DO is satisfied, control passes to the statement following statement 20.

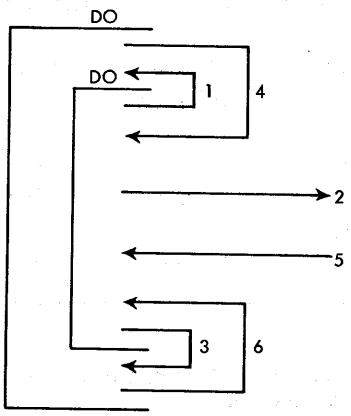
1. The *Range* of a DO is that set of statements that is executed repeatedly; i.e., it is the sequence of consecutive statements immediately following the DO statement, up to and including the statement numbered n. After the last execution of the range, the DO is said to be *satisfied*. In the previous example, the range is statement 20.

2. The *Index* of a DO is the integer variable i. Throughout the range of the DO, the index is available for computation, either as an ordinary integer variable or as the variable of a subscript. Upon exiting from a DO by satisying the DO, the index i must be redefined before it is used in computation. Upon exiting from a DO by transferring out of the range of the DO, the index i is available for computation and is equal to the last value it attained. In the previous example, the index is I.

3. *DOs within DOs.* Among the statements in the range of a DO may be other DO statements; such a configuration is called a *nest of DOs*. If the range of a DO includes another DO, then all the statements in the range of the latter must also be in the range of the former.

4. *Transfer of Control and DOs.* Control may not be transferred into the range of a DO from outside its range.

Thus, in the following configuration, 1, 2, and 3 are permitted transfers, but 4, 5, and 6 are not.



5. *Restrictions on Statements in the Range of a DO.* Any statement that redefines the index or any of the indexing parameters (m's) is not permitted in the range of a DO. In other words, the indexing of a DO loop must be completely set before the range is entered. When a CALL statement is executed in the range of a DO to reference a subprogram, care must be taken that the called subprogram or function does not alter the DO index or indexing parameters.

The statement that terminates the range of a DO must be an executable statement. The range of a DO cannot end with an arithmetic IF-type or GO TO-type statement. The range of a DO may end with a logical IF, in which case control is handled as follows: if the logical expression is false, the DO is reiterated; if the logical expression t is true, statement s is executed and then the DO is reiterated. However, if t is true and s is an arithmetic IF-type or GO TO-type statement, control is transferred as indicated.

CONTINUE Statement

The CONTINUE statement provides a means for inserting statement numbers in a source program without generating any instructions in the object program; that is, it is used as a point of reference.

GENERAL FORM
CONTINUE

CONTINUE may be used as the last statement in the range of a DO when the DO would otherwise end with an IF-or GO TO-type statement (neither of which is permitted).

PAUSE Statement

The PAUSE statement causes the machine to halt and (optionally) an octal number to be printed on the typewriter. Depressing the Start key causes the object program to resume execution with the next executable FORTRAN statement.

GENERAL FORM
PAUSE or PAUSE n where: n is an unsigned *octal* integer constant of one to five digits.

EXAMPLES:

```
PAUSE
PAUSE 77777
```

Note that when the PAUSE statement is executed, records processed by previous WRITE statements may not have been completely written or punched on the physical output unit.

STOP Statement

This statement terminates the execution of any program by returning control to the Monitor.

GENERAL FORM
STOP

Execution of a program can also be terminated by a RETURN statement or by a CALL to the EXIT and DUMP subroutines (the section, "EXIT, DUMP, and PDUMP" has additional information).

END Statement

The END statement terminates compilation of a program, a FUNCTION subprogram, or a SUBROUTINE subprogram.

GENERAL FORM
END

This statement must be the last statement of every program.

The FORTRAN statements that specify transmission of information to or from input/output devices may be grouped as follows:

FORMAT Statement: The nonexecutable statement FORMAT specifies the arrangement of the information in the external input/output medium specified by the general input/output statements.

General Input/Output Statements: The statements READ and WRITE cause the transmission of a specified list of quantities between core storage and an input/output device.

Auxiliary Input/Output Statements: The statements END FILE, REWIND, and BACKSPACE specify non-data actions of the input/output devices.

List Specifications

The statements that cause transmission of information require a list of quantities to be transmitted. This list must be in the same order that the words of information are (for input). For output, the list determines the order on the output medium.

The following example illustrates the formation and meaning of an input/output list:

```
A, B(3), (C(I), D(I,K), I=1, 10), ((E(I, J), I=1, 10,
  2), F(J, 3), J=1, K)
```

If this list is used with an output statement, the information will be written on the output medium in the following order:

```
A, B(3), C(1), D(1, K), C(2), D(2, K), . . . , C(10),
  D(10, K),
E(1, 1), E(3, 1), . . . , E(9, 1), F(1, 3),
E(1,2), E(3, 2), . . . , E(9, 2), F(2, 3), . . . , F(K, 3).
```

Similarly, if this list is used with an input statement, the successive words, as they are read from the external medium, are placed into the sequence of storage locations just given.

Thus, the list reads from left to right, with repetition for variables enclosed with parentheses. The list items are separated by commas. Only subscripted or non-subscripted variables or an implied DO may be listed. The execution is exactly that of a DO loop, as though each left parenthesis (except subscripting parentheses) were a DO, with indexing given immediately before the matching right parenthesis and with the DO range extending up to that indexing information. The order of the preceding list can thus be considered the equivalent of the following "program":

```
1. A
2. B(3)
3. DO 5 I=1, 10
4. C(I)                    (C(I), D(I, K), I=1, 10)
5. D(I,K)
6. DO 9 J=1, K
7. DO 8 I=1, 10, 2         ((E(I, J), I=1, 10, 2),
8. E(I, J)                   F(J, 3), J=1, K)
9. F(J, 3)
```

An implied DO is best defined by an example. In the preceding input/output list, the list item (C(I),D(I,K), I=1, 10) is an implied DO; it is evaluated as in the preceding program. The range of the implied DO must be clearly defined by parentheses. A constant may appear in an input/output list only as a subscript or as an indexing parameter. Indexing information, as in DOS, consists of three constants or integer variables, and the last of these may be omitted, in which case it is assumed to be 1.

For a list of the form K, A (K) or K, (A(I), I=1,K), where an index or indexing parameter itself appears earlier in the list of an input statement than its use, the indexing is carried out with the newly read-in value.

Any number of quantities may appear in a single list. During a read operation, the list controls the quantity of data read; if a record contains more quantities than are in the list, only the number of quantities specified in the list are transmitted, and any remaining quantities are ignored. Conversely, if the list contains more quantities than are given on one BCD input record (as defined by the FORMAT statement), more records are read; if a list contains more quantities than are given in one FORTRAN binary record, either job execution or reading is terminated and the remaining list items are filled with zeros.

Input/Output of Entire Arrays

When input/output of an entire matrix is desired, an abbreviated notation can be used in the list of the input/output statement; only the name of the array need be given, and the indexing information can be omitted.

Thus, if A has previously been listed in a DIMENSION statement or in a dimensioned COMMON or Type statement, the following statement is sufficient to read in all of the elements of the array, A:

```
READ (5, 10)A
```

If A has not appeared in a DIMENSION statement or a dimensioned COMMON or Type statement, only the first element is read in.

The elements read in by this notation are stored in accordance with the description of the arrangement of arrays in storage. Arrays are read or written in columnar fashion, with the first of their subscripts varying most rapidly and the last varying least rapidly.

Format

The BCD input/output statements require, in addition to a list of quantities to be transmitted, reference to a FORMAT statement that describes the type of conversion to be performed between the internal machine language and the external notation for each quantity in the list.

GENERAL FORM
FORMAT (S_1,S_2, . . . ,S_n/S'_1,S'_2, . . . ,S'_n/. . .) where: each field, S_i, is a format specification.

EXAMPLE:

FORMAT (I2/(E12.4, F10.2))

The FORMAT statement specifies the types of data conversion to be performed.

1. FORMAT statements are executed out-of-line; they can be placed anywhere in the source program. Each FORMAT statement must be given a statement number.

2. The FORMAT statement indicates, among other things, the maximum size of each record to be transmitted. In this connection, it must be remembered that the FORMAT statement is used with the list of some particular input/output statement, except when a FORMAT statement consists entirely of H and/or X fields. In all other cases, control in the object program switches back and forth between the list (which specifies whether data remains to be transmitted) and the FORMAT statement (which contains the specifications for transmission of that data).

3. Slashes are used to terminate records. In each case, the record length specified must be no longer than 132 characters. FORMAT (3F9.2, 2F10.4/8E14.5) specifies that the first, third, fifth, etc., records have the format (3F9.2, 2F10.4) and that the second, fourth, sixth, etc., records have the format (8E14.5).

4. During input/output of data, the object program interprets the FORMAT statement to which the relevant input/output statement refers. When a specification for an A-, D-, E-, F-, I-, L-, or O-type field is found, and list items remain to be transmitted, input/output takes place according to the specification, and interpretation of the FORMAT statement resumes. If no items remain, transmission ceases and execution of that particular input/output statement is terminated. Thus, a BCD input/output operation ends when there are no items remaining in the list.

Numerical Fields

Five forms of conversion for numerical data are available:

INTERNAL	TYPE	EXTERNAL
Real	D	Real, with D exponent
Real	E	Real, with exponent
Real	F	Real, without exponent
Integer	I	Integer
Octal integer	O	Octal integer

These types of conversion are specified in the forms:

Dw. d, Ew. d, Fw. d, Iw, and Ow

where:

D, E, F, I, and O

represent the type of conversion.

w

is an unsigned integer constant that represents the field width for converted data; this field width may be greater than required to provide spacing between numbers.

d

is an unsigned integer or a zero that represents the number of positions of the field that appear to the right of the decimal point, not including the D or E exponent field, if present.

The format of a numerical field is specified by giving, from left to right (beginning with the first character of the field):

1. The control character (D, E, F, I, or O) for the field.

2. The width (w) of the field. Leading zeros in the integer part of an output number are suppressed, and a blank or a minus sign is placed in front of the first integer digit. If the entire integer part of the number is zero, the digit is zero. The specified width can be greater than required to provide spacing between numbers.

3. The number of positions (d) of the decimal fraction that appears to the right of the decimal point for D-, E-, and F-type conversion. If d is greater than the maximum number of digits permitted by the machine (i.e., 16 for D-type conversion, and 8 for E- and F-type conversion), the maximum number of digits is carried and unused digits are truncated.

For example, the statement FORMAT (I2,E12.4, O8, F10.4, D25.16) might cause the following line to be printed:

```
I2 E12.4      O8          F10.4    D25.16
27b-0.9321Eb02577342760bbb-0.0076b-0.7878977909500672Db03
```

where:

b indicates a blank space.

Specifications for successive fields are separated by commas. A format specification that provides for more characters than the maximum input or output unit record size should not be given. Thus, a format for printed output should not provide for more characters per line (including blanks) than may be printed on one line by the printer.

D-, E-, F-, I-, AND O-CONVERSION

D-conversion results in the transmission of w characters containing a double-precision number of up to 16 decimal digits. For input, the number is stored such that the most significant and least significant parts are in adjacent core storage locations. For output, the two core storage words representing the double-precision quantity are treated as a single data item and are converted as such.

E-conversion results in the transmission of w characters containing a decimal mixed number and its exponent field, e.g., 5.02E2. The exponent, which must be used with E-conversion, is the power of 10 to which the number must be raised to obtain its true value. The exponent is written with an E, followed by a minus sign if the exponent is negative or a plus sign or a blank if the exponent is positive, and then followed by the two numbers that are the exponent. For example, the number .002 is equivalent to the number .2E-02.

F-conversion results in the transmission of w characters containing a decimal mixed number only.

I-conversion results in the transmission of w characters containing an integer number of up to 11 decimal digits. If an output number converted by D-, E-, F-, or I-conversion requires more spaces than are allowed by the field width w, the excess on the high-order side is lost and no rounding occurs. If the number requires fewer than w spaces, the leftmost spaces are filled with blanks. If the number is negative, the space preceding the leftmost digit contains a minus sign if sufficient spaces have been reserved.

O-conversion results in the transmission of w characters of octal information. If $w>12$, only the twelve rightmost characters are transmitted and $w-12$ blanks precede the field (output) or $w-12$ preceding characters are skipped (input). If $w\leqq12$, the rightmost w characters of the word are transmitted (output) or the next w characters are right-adjusted in the word and the word is filled out on the left with zeros (input).

The field width w for D-, E-, and F-conversion of output must include a space for the decimal point and a space for the sign. Thus, for D- and E-conversion, $w\geqq d+7$, and for F-conversion, $w\geqq d+3$.

Information to be transmitted with O-conversion may be given either a real name or an integer variable name; information to be transmitted with E- and F-conversion must have real names; information to be transmitted with D-conversion must have a double-precision name; and information to be transmitted with I-conversion must have an integer name. The names must be used as specified above; any other practice is invalid.

COMPLEX NUMBER FIELDS

Since a complex quantity consists of two separate and independent real numbers, a complex number is transmitted by two successive real number specifications or by one real number specification that is repeated.

The following is an example of a FORMAT statement that transmits an array consisting of six complex numbers:

```
FORMAT (2E.10, E8.3, 1PE9.4, E10.2, F8.4,
  3(E10.2, F8.2))
```

Two levels of parentheses, in addition to the parentheses required by the FORMAT statement, are permitted. The second level of parentheses facilitates the transmission of complex quantities.

Alphameric Fields

FORTRAN provides two specifications to transmit alphameric information: Aw and nH. Both result in storing the alphameric information internally in BCD form.

1. The specification Aw causes w characters to be read into, or written from, a variable or array name, without conversion.
2. The specification nH introduces alphameric information into a FORMAT statement.

The basic difference between A- and H-conversion is that information handled by A-conversion is given a variable name or an array name and hence can be referred to by this name for processing and modification, whereas information handled by H-conversion is not given a name and may not be referred to or manipulated in storage in any way.

A-CONVERSION

The variable name to be converted by A-conversion must conform to the normal rules for naming FORTRAN variables; the name may be any type.

Input: nAw means that the next n successive fields of w characters each are to be stored as BCD information. If w is greater than 6, only the 6 rightmost characters are significant. If w is less than 6, the characters are left-adjusted, and the word is filled out on the right with blanks.

Output: nAw means that the next n successive fields of w characters each are to be the result of transmission from storage without conversion. If w is greater than 6, only 6 characters of output are transmitted, preceded by w-6 blanks. If w is less than 6, the w leftmost characters of the word are transmitted.

H-CONVERSION

The specification nH is followed in the FORMAT statement by n alphameric characters, as for example, in the following:

31HbTHISbISbALPHAMERICbINFORMATION

Blanks are considered alphameric characters and must be included as part of the count n. The effect of nH depends on whether it is used with input or with output.

Input: The n characters are extracted from the input record and replace the n characters immediately following the character H in the FORMAT specification.

Output: The n characters following the character H in the specification, or the characters that replaced them, are written as part of the output record.

Example: The statement FORMAT (4HbXY=, F8.3,A8) might produce the following lines:

```
XY =.b-93.210bbbbbbbb
XY = 9999.999bbOVFLOW
XY = bb28.768bbbbbbbb
```

where:
b indicates a blank space.

This example assumes that there are steps in the source program that read the data OVFLOW, store this data in the word to be printed (as six BCD characters) in the format A8 when overflow occurs, and store six blanks in the word when overflow does not occur.

LOGICAL FIELDS

Logical variables may be read or written by means of the specification Lw.

Input: The first T or F encountered in the next w characters of the input record causes a value of TRUE or FALSE, respectively, to be assigned to the corresponding logical variable. If the field w consists entirely of blanks, a value of FALSE is assumed. Any character other than T, F, or blank encountered prior to a T, F, or blank causes an error exit.

Output: A T or an F is inserted in the output record for a corresponding logical variable with a value of TRUE or FALSE, respectively. The single character is preceded by w−1 blanks.

Blank Fields — X-Conversion

The specification nX affects an input or an output record as follows:

Input: nX causes n characters in the input record to be skipped.

Output: nX causes n blanks to be introduced into the output record.

Repetition of Field Format

It may be desired to read, write, punch, or print n successive fields within one record in the same format. The programmer may specify this by giving n, an unsigned integer constant, before A, D, E, F, I, L, or O. The following format field will then be repeated n times. Thus, the field specification 3E12.4 has the same effect as the specification E12.4, E12.4, E12.4.

Repetition of Groups

A limited parenthetical expression is permitted to enable repetition of data fields according to certain format specifications within a longer FORMAT statement specification. The first character in the expression specifies the number of repetitions and is known as the *group count.* Thus, FORMAT (2(F10.6, E10.2), I4) is equivalent to FORMAT (F10.6, E10.2, F10.6, E10.2, I4).

Scale Factors

To permit more general use of D-, E-, and F-conversion, a scale factor followed by the letter P can precede the specification. The magnitude of the scale factor must be between −8 and +8 inclusive.

The scale factor for input is defined as follows:

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

The scale factor for output is defined as follows:

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

For input, scale factors affect only F-conversion. 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 specification to effect this change is 2PF7.4. For output, scale factors can be used with D-, E-, and F-conversion.

For example, the statement FORMAT (I2, 3F11.3) would give the printed line:

27bbbb−93.209bbbbb−0.008bbbbbb0.554

The statement FORMAT (I2, 1P3F11.3), used with the same data, would give the line:

27bbb−932.094bbbbb−0.076bbbbbb5.536

On the other hand, the statement FORMAT (I2, −1P3F11.3) would give the line:

27bbbbb−9.321bbbbb−0.001bbbbbb0.055

A positive scale factor used for output with D- and E-conversion increases the number and decreases the exponent. Thus, with the same data, FORMAT (I2, 1P3E12.4) would produce the line:

27b−9.3209Eb01b−7.5804E−03bb5.5536E−01

The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it holds for all D-, E-, and F-conversions following the scale factor within the same FORMAT statement. This applies to both single-record formats and multiple-record formats. Once a scale factor has been given, a subsequent scale factor of zero in the same FORMAT statement must be specified by 0P. Scale factors have no effect on I- or O-conversion.

Multiple-Record Formats

So that it applies to more than one input or output record, a FORMAT specification can have several different one-record formats separated by a slash (/) to indicate the beginning of a new record.

Thus, FORMAT (3F9.2, 2F10.4/8E14.5) would specify a multi-record block of print in which records 1, 3, 5, . . . have the format (3F9.2, 2F10.4), and records 2, 4, 6, . . . have the format (8E14.5).

If a multiple-record format is desired in which the first two records are to be transmitted according to a special format and all succeeding records transmitted according to another format, the last record specification should be enclosed in a second pair of parentheses; e.g., FORMAT (I2, 3E12.4/2F10.3, 3F9.4/(10F12.4)). If data items remain to be transmitted after the format specification has been completely used, the format repeats from the last open parenthesis, using its preceding group count, if any. As these examples show, both the slash and the right parenthesis of the FORMAT statement indicate a termination of a record.

The programmer may introduce blank records into a multi-record FORMAT statement by listing consecutive slashes. When n consecutive slashes appear at the end of the FORMAT, they are treated as follows: for input, n records are skipped; for output, n blank records are written. When n consecutive slashes appear in the middle of the FORMAT, n-1 records are skipped for input and n-1 blank records are written for output.

Carriage Control

When any BCD output statement that causes records to be written is given, the records are either printed directly on-line or may be printed later by an off-line printer. To space the printed output records properly, the following carriage control characters may be used:

CHARACTER	EFFECT
Blank	Single space before printing.
0	Double space before printing.
1	Skip to a punch in Channel 1; i.e., eject.

To obtain such carriage control, the control character must appear as the first character of the first word of the BCD record. This may be done if the FORMAT specification for a BCD record is begun with 1H followed by the desired control character. Under program control, the control character is not printed.

FORMAT Statements Read in at Object Time

FORTRAN accepts a variable name in the place of a statement number when referencing a FORMAT statement. This provides the facility of specifying a FORMAT for an input/output list during execution of the compiled program.

EXAMPLES:

```
    DIMENSION FMT (12)
1   FORMAT (12A6)
    READ (5,1) (FMT (I), I=1, 12)
    READ (5, FMT) A, B, (C(I), I=1, 5)
```

Thus, A, B, and the array C would be converted and stored according to the FORMAT specification read into the array FMT at object time.

The format read in at object time must take the same form as a source program FORMAT statement, except that the word FORMAT is omitted; i.e., the variable format begins with a left parenthesis and ends with a right parenthesis.

FORMAT statements may not be subscripted; thus, an expression such as READ (5, FMT(I)) A, B would not be permitted.

Data Input to the Object Program

Data input to the object program is prepared according to the following specifications:

1. The data must correspond in order, type, and field to the field specifications in the FORMAT statement. Punching begins in card column 1.

2. Plus signs can be omitted, or they can be indicated by a blank or a 12-punch. Minus signs must be indicated by an 11-punch.

3. Blanks in numerical fields are regarded as zeros.

4. Numbers for D-, E-, and F-conversion can contain any number of digits, but only the high-order digits are retained. For D-conversion, the number is rounded to the 16 high-order digits of accuracy and, for E- and F-conversion, to the eight high-order digits of accuracy.

To permit economy in punching, certain relaxations in input data format are permitted:

1. Numbers for D- and E-conversion need not have four columns occupied by the exponent field. The start of the exponent field must be marked by the E or D or, if that is omitted, by a + (plus) or a − (minus), not a blank. Thus, E2, E+2, +2, +02, and D+02 are all permissible exponent fields.

2. Numbers for D-, E-, and F-conversion need not have their decimal point punched. If it is not punched, the FORMAT specification supplies the number of decimal places expected. For example, the number −09321+2 with the specification E12.4 is treated as though the decimal point had been punched between the 0 and the 9. If the decimal point is punched on the card, its position overrides the position indicated in the FORMAT statement.

The General Input/Output Statements

This set of statements specifies transmission of information between core storage and the input or output media.

Input

The READ statement designates input.

GENERAL FORM
Following are the two forms of the READ statement: READ (i, n) list READ (i) list where: 1. i is an unsigned integer constant or an integer variable that refers to an input device. 2. n is a FORMAT statement number or a variable FORMAT name.

EXAMPLES:

```
READ (5, 10) A, B, (D(J), J=1, 10)
READ (N, 10) K, DC (J)
READ (3) (A(J), J=1, 10)
READ (N) (A(J), J=1, 10)
```

1. The READ (i, n) list statement causes BCD information to be read from logical unit i.

2. The READ (i) list statement causes binary information to be read from logical unit i.

Under the form READ (i, n) list, successive records are read until the entire input/output list has been satisfied; i.e., all data items have been read, converted, and stored in the locations specified by the input/output list.

Under the form READ (i) list, a record is read completely only if the list specifies as many words as the record contains. The list cannot specify more words than the record contains. Binary records to be read in by a FORTRAN program should be written by a FORTRAN program or should be in the proper binary record format, as follows:

Consider a FORTRAN record to be any sequence of words written by one FORTRAN output statement. When written in binary, this FORTRAN record may be divided into several IOCS logical records. The IOCS record length is specified in the publication, *IBM 7040/7044 Operating System (16/32K): Systems Programmer's Guide*, Form C28-6339. The first word of each of these logical records is a *signal* word interpreted only by FORTRAN. The decrement of this signal word specifies the number of words in the logical records that follow; the address is zero for all but the last logical record of the FORTRAN record. The address of this last signal word contains the number of logical records in the entire FORTRAN record.

Output

The WRITE statement designates output.

GENERAL FORM
Following are the forms of the WRITE statement: WRITE (i, n) list WRITE (i) list where: 1. i is an unsigned integer constant or an integer variable that refers to an output device. 2. n is a FORMAT statement number or a variable FORMAT name.

EXAMPLES:

```
WRITE (6, 10) A, B, (C(J), J=1, 10)
WRITE (N, 11) K, D(J)
WRITE (2) (A(J), J=1, 10)
WRITE (M) A, B, C
```

1. The WRITE (i, n) list statement causes BCD information to be written on logical unit i.

2. The WRITE (i) list statement causes binary information to be written on logical unit i.

3. In the output statement in item 1, successive records are written in accordance with the FORMAT statement until the list has been satisfied. In the output statement in item 2, one FORTRAN record, consisting of all the words specified in the list, is written.

Figure 7 shows the correspondence between FORTRAN logical units and system units.

FORTRAN Logical Input/Output Unit	System Unit Assignment	System Unit Description
0	S.SU00	Utility 0
1	S.SU01	Utility 1
2	S.SU02	Utility 2
3	S.SU03	Utility 3
4	S.SU04	Utility 4
5	S.SIN1	System Input Unit
6	S.SOU1	System Output Unit
7	S.SPP1	System Peripheral Punch Unit
READ	S.SIN1	System Input Unit
PRINT	S.SOU1	System Output Unit
PUNCH	S.SPP1	System Peripheral Punch Unit

Figure 7. Correspondence Between FORTRAN Logical Units and System Units

The Auxiliary Input/Output Statements

The statements END FILE, REWIND, and BACKSPACE, cause, respectively, the object program to write an end-of-file mark on the tape unit specified, or a rewind or backspace of the symbolic or actual tape unit specified.

GENERAL FORM
Following is the form of the END FILE, REWIND, and BACKSPACE statements: END FILE i REWIND i BACKSPACE i where: i is an unsigned integer constant or integer variable that refers to an input/output device.

EXAMPLES:

```
END FILE 3
END FILE N
REWIND 3
REWIND N
BACKSPACE 3
```

1. The END FILE i statement causes the object program to close the file corresponding to logical unit i with an end-of-file procedure; there is no rewind procedure.

2. The REWIND i statement causes the object program to close the file corresponding to logical unit i with the end-of-file and the rewind procedure.

3. The BACKSPACE i statement causes device i to be backspaced one *physical record* if i refers to an input/output device in *BCD mode*, or it causes the device i to be backspaced one *FORTRAN record* if i refers to an input/output device in *binary mode*. The mode of a device is determined by the mode of the most recent input or output statement referring to that device. When backspacing occurs, one physical tape record is assumed to consist of only one IOCS logical record. Therefore, when a BACKSPACE statement refers to a logical unit of which the current mode is binary, the signal word address of the last physical record determines the total number of physical records to be backspaced.

4. A request to write an end of file or to rewind system files S.SIN1, S.SOU1, and S.SPP1, corresponding in the standard FORTRAN Input/Output Library to logical units 5, 6, and 7, causes job termination or is ignored. (The publication, *IBM 7040/7044 Operating System (16/32K): Programmer's Guide*, Form C28-6318, has additional information.)

5. Care should be taken to prevent writing an end of file, rewinding, and backspacing on units attached to unit record equipment. (The publication, *IBM 7040/7044 Operating System (16/32K): Input/Output Control System*, Form C28-6309, has additional information.)

There are four classes of subroutines in FORTRAN: arithmetic or logical statement functions, built-in functions, FUNCTION subprograms (including library functions), and SUBROUTINE subprograms. The major differences among the four classes of subroutines are:

1. The first three classes may be grouped as functions; they differ from the SUBROUTINE subprogram in the following respects:
   a. The functions are always single-valued (that is, they return a single result); the SUBROUTINE subprogram may return more than one value.
   b. A function is referred to by an arithmetic or logical expression containing its name; a SUBROUTINE subprogram is referred to by a CALL statement.
2. A built-in function is an open subroutine; i.e., a subroutine that is incorporated into the object program each time it is referred to in the source program. The three other FORTRAN subroutines are closed; i.e., they appear only once in the object program.

Naming Subroutines

All four classes of subroutines are named in the same manner as a FORTRAN variable (see the section "Variables").

1. A subroutine name consists of 1-6 alphameric characters, the first of which must be alphabetic.
2. The type of the function, which determines the type of the result, may be defined as follows:
   a. The type of an arithmetic or logical statement function may be indicated by the name of the function or by placing the name in a Type statement.
   b. The type of a FUNCTION subprogram may be indicated by the name of the function (if it is real or integer) or by writing the type as part of the function statement (REAL FUNCTION, INTEGER FUNCTION, COMPLEX FUNCTION, DOUBLE PRECISION FUNCTION, LOGICAL FUNCTION). In the latter case, the type, implied by name, is overridden.
   c. The type of a built-in function is indicated within the FORTRAN processor and need not appear in a Type statement.
3. The type of a SUBROUTINE subprogram is unimportant and need not be defined, since the type of results returned is dependent only on the type of the variable names in the dummy argument list.

Defining Subroutines

The method of defining each class of subroutines is discussed in the following text.

Arithmetic and Logical Statement Functions

Arithmetic and logical statement functions are defined by a single arithmetic or logical assignment statement in the source program and apply only to the particular program or subprogram in which the definition appears.

GENERAL FORM
a=b where: 1. a is a function name followed by parentheses enclosing its arguments, which must be distinct, nonsubscripted variables, separated by commas. 2. b is an expression that does not involve subscripted variables. Any arithmetic statement function appearing in b must have been defined previously.

EXAMPLES:

```
FIRST (X)=A*X+B
JOB (X, B)=C*X+B
THIRDF (D)=FIRST (E)/D
MAX (A, I)=A**I-B-C
LOGFCT (A, C)=A**2. GE. C/D
```

The type of the function and its arguments must be defined in the same manner as normal variables.

As many as desired of the variables appearing in b may be stated in a as the arguments of the function. Since the arguments are dummy variables, their names, which indicate the type of the variable, may be the same as names of the same type appearing elsewhere in the program.

Variables included in b that are not stated as arguments are the parameters of the function. They are ordinary variables.

The type of each argument must be defined preceding its use in the statement function definition.

All statement function definitions must precede the first appearance of the function name in an executable statement or another statement function definition.

Built-in (or Open) Functions

Built-in functions are predefined open subroutines that exist within the FORTRAN processor. They generate instructions that are compiled in-line every time the function is referred to.

The names of built-in functions cannot be used as subroutine or variable names.

Figure 8 lists all available built-in functions.

Library (or Closed) Functions

Library (or closed) functions are FUNCTION subprograms that are prewritten and exist on the library tape or in prepared card decks. These functions constitute closed subroutines; i.e., instead of appearing in the object program for every reference that has been made to them in the source program, they appear only once regardless of the number of references.

Library subroutines that perform mathematical functions are provided. Subroutines marked with an * (asterisk) are internal to FORTRAN and cannot be referred to by a call in the source program.

Function Definition	Function Name	No. of Args.	Type of	
			Function	Argument
Absolute value of the argument	ABS	1	Real	Real
	IABS	1	Integer	Integer
	DABS	1	Double-precision	Double-precision
Truncation, sign of argument times absolute value of the largest integer in argument	AINT	1	Real	Real
	INT	1	Integer	Real
	IDINT	1	Integer	Double-precision
Remaindering, Arg 1 − [Arg 1/Arg2]* Arg2, where [X] indicates the integral part of X	AMOD	2	Real	Real
	MOD		Integer	Integer
Choosing the largest value of the set of arguments	AMAX0	≧2	Real	Integer
	AMAX1	≧2	Real	Real
	MAX0	≧2	Integer	Integer
	MAX1	≧2	Integer	Real
	DMAX1	≧2	Double-precision	Double-precision
Choosing the smallest value of the set of arguments	AMIN0	≧2	Real	Integer
	AMIN1	≧2	Real	Real
	MIN0	≧2	Integer	Integer
	MIN1	≧2	Integer	Real
	DMIN1	≧2	Double-precision	Double-precision
Floating an integer	FLOAT	1	Real	Integer
Same as INT	IFIX	1	Integer	Real
Transfer of sign, the sign of Arg 2 times Arg 1	SIGN	2	Real	Real
	ISIGN	2	Integer	Integer
	DSIGN	2	Double-precision	Double-precision
Positive difference, Arg 1 − Min (Arg 1, Arg 2)	DIM	2	Real	Real
	IDIM	2	Integer	Integer
Obtaining the most significant part of a double-precision argument	SNGL	1	Real	Double-precision
Obtaining the real part of a complex argument	REAL	1	Real	Complex
Obtaining the imaginary part of a complex argument	AIMAG	1	Real	Complex
Expressing a single-precision argument in double-precision form	DBLE	1	Double-precision	Real
Expressing two real arguments in complex form C = Arg1 + iArg2	CMPLX	2	Complex	Real
Obtaining the conjugate of a complex argument; for Arg = X + iY, C = X − iY	CONJG	1	Complex	Complex

Figure 8. Built-in Functions

ENTRY POINT	DESCRIPTION	ROUTINE
.EXP1. Entry point for an integer exponent and base. The output is integer.	A call to this subroutine is compiled for a source program exponential term such as I**J, where I and J are integer. This subroutine computes the Jth power of I with accuracy up to 35 significant bits.	*XP1
.EXP2. Entry point for an exponential subroutine whose base is real and whose exponent is integer.	A call to this subroutine is compiled for a source program exponential term such as A**J, where A is real and J is integer. This subroutine computes the Jth power of A with accuracy up to eight significant digits.	*XP2
.EXP3. Entry point for an exponential subroutine whose base and exponent are real.	A call to this subroutine is compiled for a source program exponential term such as A**B, where A and B are real. This subroutine computes the Bth power of A with accuracy up to eight significant digits.	*XP3
EXP Entry point for a real natural antilogarithm subroutine.	A call to this subroutine is compiled for a source program statement such as Y=EXP(X), where X is real. This subroutine computes e^x, the natural antilogarithm of X, with an accuracy up to eight significant digits.	XPN
ALOG Entry point for the real natural logarithm subroutine.	A call to this subroutine is compiled for a source program statement such as Y=ALOG(X), where X is real. This subroutine computes $\text{Log}_e(X)$ with accuracy up to eight significant digits.	LOG
ALOG10 Entry point for the real common logarithm subroutine.	A call to this subroutine is compiled for a source program statement such as Y=ALOG10(X), where X is real. This subroutine computes $\text{Log}_{10}(X)$ with accuracy up to eight significant digits.	LOG
ATAN Entry point for the real arc tangent subroutine. This entry point has one argument.	A call to this subroutine is compiled for a source program statement such as Y=ATAN(X), where X is real. This subroutine computes Arc tan(X) in radians with accuracy up to eight significant digits.	ATN
ATAN2 Entry point for the real arc tangent subroutine. This entry point requires two arguments.	A call to this subroutine is compiled for a source program statement such as Y=ATAN2(X, Y), where X and Y are real. This subroutine computes Arc tan(X/Y) in radians with accuracy up to eight significant digits.	ATN
SIN Entry point for the real sine-cosine subroutine. The input is expressed in radians.	A call to this subroutine is compiled for a source program statement such as Y=SIN(X), where X is real. This subroutine computes Sin(X) with accuracy up to eight significant digits.	SCN
COS Entry point for the real sine-cosine subroutine. The input is expressed in radians.	A call to this subroutine is compiled for a source program statement such as Y=COS(X), where X is real. This subroutine computes Cos(X) with accuracy up to eight significant digits.	SCN
TANH Entry point for the hyperbolic tangent subroutine.	A call to this subroutine is compiled for a source program statement such as Y=TANH(X), where X is real. This subroutine computes Tanh(X) with accuracy up to eight significant digits.	TNH
SQRT Entry point for the real square root subroutine.	A call to this subroutine is compiled for a source program statement such as Y=SQRT(X), where X is real. This subroutine computes the square root of X with accuracy up to eight significant digits.	SQR
ARSIN Entry point for the real arc sine-arc cosine subroutine.	A call to this subroutine is compiled for a source program statement such as Y=ARSIN(X), where X is real. This subroutine computes Arc sin(X) in radians with accuracy up to eight significant digits.	ARSCN
ARCOS Entry point for the real arc sine-arc cosine subroutine.	A call to this subroutine is compiled for a source program statement such as Y=ARCOS(X), where X is real. This subroutine computes Arc cos(X) in radians with accuracy up to eight significant digits.	ARSCN
FCAOP. Entry point for arithmetic operations involving two complex numbers.	A call to this subroutine is compiled for souree program statement such as CA=CB+CC, CA=CB-CC, CA=	*FCA

ENTRY POINT	DESCRIPTION	ROUTINE
	CB*CC, or CA=CB/CC, where CA, CB, and CC are complex. This subroutine performs the complex operation required with accuracy up to eight significant digits for the real and imaginary parts of the result.	
CXP1. Entry point for a complex base and integer exponent.	A call to this subroutine is compiled for a source program exponential term such as CA**I, where CA is complex. This subroutine computes the Ith power of CA with accuracy up to eight significant digits.	*FDX1
DXP1. Entry point for a double-precision base and an integer exponent.	A call to this subroutine is compiled for a source program exponential term such as DA**I, where DA is double-precision. This subroutine computes the Ith power of DA with accuracy up to 16 significant digits.	*FDX1
DXP2. Entry point for a double-precision base and exponent.	A call to this subroutine is compiled for a source program exponential term such as A**B, where A is real and B is double-precision, or A is double-precision and B is real, or A and B are double-precision. If one term is real, it will be converted to double-precision before entry into the subroutine. This subroutine computes the Bth power of A with accuracy up to 16 significant digits.	*FDX2
DMOD Entry point for a double-precision remaindering function.	A call to this subroutine is compiled for a source program statement such as Y=DMOD(X, Z), where X and Z are double-precision. This subroutine computes the expression X−[X/Z]*Z, where [ ] (brackets) indicate the integral part of the division. The result is accurate up to 16 significant digits.	FDMD
DEXP Entry point for a double-precision natural antilogarithm subroutine.	A call to this subroutine is compiled for a source program statement such as Y=DEXP(X), where X is double-precision. This subroutine computes e^x, the natural antilogarithm of X, with accuracy up to 16 significant digits.	FDXP
DLOG Entry point for a double-precision natural logarithm subroutine.	A call to this subroutine is compiled for a source program statement such as Y=DLOG(X), where X is double-precision. This subroutine computes $Log_e(X)$ with accuracy up to 16 significant digits.	FDLG
DLOG10 Entry point for a double-precision common logarithm subroutine.	A call to this subroutine is compiled for a source program statement such as Y=DLOG10(X), where X is double-precision. This subroutine computes $Log_{10}(X)$ with accuracy up to 16 significant digits.	FDLG
DATAN Entry point for the double-precision arc tangent subroutine. This entry point has one argument.	A call to this subroutine is compiled for a source program statement such as Y=DATAN(X), where X is double-precision. This subroutine computes Arc tan (X) in radians with accuracy up to 16 significant digits.	FDAT
DATAN2 Entry point for the double-precision arc tangent subroutine. This entry point requires two arguments.	A call to this subroutine is compiled for a source program statement such as Y=DATAN2(X,Z), where X and Z are double-precision. This subroutine computes Arc tan(X/Z) with accuracy up to 16 significant digits.	FDAT
DSIN Entry point for the double-precision sine-cosine subroutine. The input is expressed in radians.	A call to this subroutine is compiled for a source program statement such as Y=DSIN(X), where X is double-precision. This subroutine computes Sin(X) with accuracy up to 16 significant digits.	FDSC
DCOS Entry point for the double-precision sine-cosine subroutine. The input is expressed in radians.	A call to this subroutine is compiled for a source program statement such as Y=DCOS(X), where X is double-precision. This subroutine computes Cos(X) with accuracy up to 16 significant digits.	FDSC
DSQRT Entry point for the double-precision square root subroutine.	A call to this subroutine is compiled for a source program statement such as Y=DSQRT(X), where X is double-precision. This subroutine computes the square root of X with accuracy up to 16 significant digits.	FDSQ

ENTRY POINT	DESCRIPTION	ROUTINE
CABS Entry point for the complex absolute value subroutine. (The argument is complex, but the function is real.)	A call to this subroutine is compiled for a source program statement such as Y=CABS(X), where X is complex. This subroutine computes $(X_1^2+X_2^2)^{1/2}$, where X_1 and X_2 are the real and imaginary parts of X, with accuracy up to eight significant digits for the real variable Y.	FCAB
CEXP Entry point for the complex natural exponential subroutine.	A call to this subroutine is compiled for a source program statement such as Y=CEXP(X), where X is complex. This subroutine computes e^x, the natural antilogarithm of X, with accuracy up to eight significant digits for the real and imaginary parts of the result.	FCXP
CLOG Entry point for the complex natural logarithm subroutine.	A call to this subroutine is compiled for a source program statement such as Y=CLOG(X), where X is complex. This subroutine computes $Log_e(X)$ with accuracy up to eight significant digits for the real and imaginary parts of the result.	FCLG
CSIN Entry point for the complex sine-cosine subroutine.	A call to this subroutine is compiled for a source program statement such as Y=CSIN(X), where X is complex. This subroutine computes Sin(X) with accuracy up to eight significant digits for the real and imaginary parts of the result.	FCSC
CCOS Entry point for the complex sine-cosine subroutine.	A call to this subroutine is compiled for a source program statement such as Y=CCOS(X), where X is complex. This subroutine computes Cos(X) with accuracy up to eight significant digits for the real and imaginary parts of the result.	FCSC
CSQRT Entry point for the complex square root subroutine.	A call to this subroutine is compiled for a source program statement such as Y=CSQRT(X), where X is complex. This subroutine computes the square root of X with accuracy up to eight significant digits for the real and imaginary parts of the result.	FCSQ

FUNCTION Subprogram

FUNCTION subprograms are subroutines that cannot be defined by only one arithmetic statement and that are not used often enough to warrant a place on the library tape. FUNCTION subprograms are defined as separate FORTRAN source language programs.

GENERAL FORM

FUNCTION name (a_1,a_2, . . . ,a_n)
where:
1. name is the symbolic name of a single-valued function.
2. The arguments a_1, a_2, . . . a_n, of which there must be at least one, are nonsubscripted variable names, names of SUBROUTINE subprograms, names of FORTRAN functions, or names of library functions.

EXAMPLES:

```
REAL FUNCTION ROOT (B, A, C)
FUNCTION INTRST (RATE, YEARS)
LOGICAL FUNCTION IFTRU (D, E, F)
INTEGER FUNCTION CONST (ING, SG)
DOUBLE PRECISION FUNCTION DUBL (DD, DF)
COMPLEX FUNCTION CPLEX (CA, CB)
```

Where it is desired to override the implicit or normal type of name in a FUNCTION statement, one must state the type (see "Type Declaration Statements"); the type must immediately precede the word FUNCTION.

The FUNCTION statement appears, therefore, in one of the following six ways:

```
FUNCTION
REAL FUNCTION
INTEGER FUNCTION
LOGICAL FUNCTION
DOUBLE PRECISION FUNCTION
COMPLEX FUNCTION
```

The FUNCTION statement must be the first statement of a FUNCTION subprogram. In a FUNCTION subprogram, the name of the function must appear at least once as the variable on the left side of an arithmetic or logical statement or in an input statement; for example, NAME appears in the following both as the name of the function and as the variable on the left of an arithmetic statement:

```
FUNCTION NAME (A, B)
.
.
.
NAME=Z+B
.
.
RETURN
```

This returns the output value of the function to the calling program.

A FUNCTION subprogram constitutes a complete compilation and need not be compiled with other programs or subprograms. It must be loaded with other programs to form a complete object program.

The arguments following the name in the FUNCTION statement may be considered dummy variable names. That is, during object program execution, other

actual arguments are substituted for them. Therefore, the arguments that follow the function reference in the calling program must agree with those in the FUNCTION statement in the subprogram in number, order, and type. Furthermore, when a dummy argument is an array name, the corresponding actual argument must also be an array name. Each of these array names with the same dimensions must appear in DIMENSION statements of their respective programs. None of the dummy variables may appear in EQUIVALENCE or in COMMON statements in the FUNCTION subprograms.

The FUNCTION subprogram must be logically terminated by a RETURN statement (the section "RETURN Statement" has additional information).

The FUNCTION subprogram can contain any FORTRAN statement except a SUBROUTINE statement or another FUNCTION statement. A FUNCTION subprogram is referred to when the programmer uses its name as an operand in an arithmetic expression.

RULES FOR CALLING FUNCTIONS

Functions are called in a program by their appearance within an arithmetic expression. For example:

```
Y = A-SIN(B-C)
X = NAME (Z(I), Q)+KOUNT
```

All names of library and FORTRAN functions are used in this way. Their appearance in the arithmetic expression serves to call the function; the value of the function is then computed, using the arguments that are supplied in the parentheses following the function name. The arguments must be the same type as those used in the function definitions. The arguments used can be constants, variables with or without subscripts, and/or arithmetic/logical expressions, including function usages. An argument of a statement function cannot be a function or subroutine name.

The arguments of a FUNCTION subprogram can be any of the following:

1. Any type of constant.
2. Any type of subscripted or nonsubscripted variable.
3. An arithmetic or logical expression.
4. The name of a FUNCTION or SUBROUTINE subprogram.

SUBROUTINE Subprogram

SUBROUTINE subprograms are separate FORTRAN source language programs.

GENERAL FORM
SUBROUTINE name (a_1, a_2, . . . a_n) where: 1. name is the symbolic name of a subprogram. 2. Each argument, if any, is a nonsubscripted variable name or the name of a SUBROUTINE or FUNCTION

EXAMPLES:

```
SUBROUTINE MATMPY (A, N, M, B, L, J, C, K, I)
SUBROUTINE NOPAR
```

The SUBROUTINE statement must be the first statement of a SUBROUTINE subprogram and defines it as such. A subprogram introduced by the SUBROUTINE statement must be a FORTRAN program and may contain any FORTRAN statement except a FUNCTION statement or another SUBROUTINE statement.

Unlike the FUNCTION subprogram, which returns only a single value, the SUBROUTINE subprogram may use one or more of its arguments to return output. The arguments may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement that refers to the SUBROUTINE subprogram. The actual arguments must agree with the dummy arguments in number, order, and type. When a dummy argument is an array name, it must appear in a DIMENSION statement in the SUBROUTINE subprogram; also, the corresponding actual argument in the CALL statement must be a dimensioned array name. None of the dummy arguments may appear in an EQUIVALENCE or COMMON statement in the SUBROUTINE subprogram.

The SUBROUTINE subprogram must be logically terminated by a RETURN statement.

RETURN Statement

This statement logically terminates SUBROUTINE and FUNCTION subprograms and returns control to the calling program.

GENERAL FORM
RETURN

The RETURN statement must be the last executed statement of the subprogram. It need not be physically last; it can be reached by program flow at any point. Any number of RETURN statements may be used.

A RETURN statement in a main program terminates execution in the same manner as a STOP statement.

CALL Statement

The CALL statement is used to transfer control to a SUBROUTINE subprogram.

GENERAL FORM
CALL subr (a_1, a_2, . . . a_n) or CALL subr where: 1. subr is the name of a SUBROUTINE subprogram 2. a_1, a_2, . . . a_n are the arguments.

EXAMPLES:

```
CALL MATMPY (X, 5, 10, Y, 10, 2, Z, 5, 2)
CALL QDRTIC (9, 732, Q/4. 536, R-S**2. 0, X1, X2)
```

The CALL statement transfers control to the subprogram and presents it with the actual arguments. Each argument may be one of the following:

1. Any type of constant.
2. Any type of subscripted or nonsubscripted variable.
3. An arithmetic or a logical expression.
4. The name of a FUNCTION or SUBROUTINE subprogram.
5. Alphameric characters. Such arguments must be preceded by nH, where n is the count of the characters included in the argument, e.g., 6HBSTART. Blank spaces and special characters are considered part of the character count when used in alphameric fields.

The arguments presented by the CALL statement must agree with the corresponding arguments in the SUBROUTINE statement of the called subprogram in number, order, type, and in array size.

Subprograms Provided by FORTRAN

FORTRAN includes several commonly used subroutines that are available to the programmer. The mathematical subroutines that are provided are defined as FUNCTION subprograms. In addition, FORTRAN includes the SUBROUTINE subprograms EXIT, DUMP, and PDUMP. EXIT terminates job execution, DUMP dumps core storage and then terminates job execution, and PDUMP dumps core storage and then continues execution.

Mathematical Subroutines

The mathematical subroutines are FUNCTION subprograms and are listed in the section "Library (or Closed) Functions."

EXIT, DUMP, and PDUMP

The subprograms EXIT, DUMP, and PDUMP are referred to with a CALL statement. The following table shows how they appear:

```
CALL EXIT
CALL DUMP (a1, b1, f1, . . . , an, bn, fn)
CALL PDUMP (a1, b1, f1, . . . , an, bn, fn)
where:
  1. a and b are variable names that indicate the limits
  of core storage to be dumped (either a or b may represent
  upper or lower limits).
  2. f is an integer that indicates the dump format
  desired, defined as follows:
     0=dump as octal
     1=dump as real
     2=dump as integer
     3=dump as octal with mnemonics
```

If for the DUMP or PDUMP subprogram the integer that indicates the dump format is omitted, FORTRAN assumes it to be 0 and the dump is in octal; for example, instead of FLD1,FLD2,0,FLD3,FLD4,0 the programmer may write FLD1,FLD2,,FLD3,FLD4 for his arguments. If no arguments are given, all of core storage is dumped in octal.

CALLS to the EXIT, DUMP, and PDUMP subprograms perform the following:

1. CALL to the EXIT subprogram: This terminates the execution of a program by returning control to the Monitor.
2. CALL to the DUMP subprogram: This causes the limits of core storage indicated by the arguments to be dumped and execution to be terminated by returning control to the Monitor.
3. CALL to the PDUMP subprogram: This causes the limits of core storage indicated by the arguments to be dumped and execution to be continued.

DIMENSION, DATA, COMMON, EQUIVALENCE, and the Type statements are nonexecutable statements that supply the compiler with necessary information about storage allocation for the constants and variables used in the program.

DIMENSION Statement

The DIMENSION statement specifies the number of dimensions of an array. A single DIMENSION statement may specify the dimensions of more than one array. The dimensions may be unsigned integer constants or integer variables. When integer constants are specified, the size of the array is defined. When integer variables are specified, location and size of an array are determined at execution time as explained in the section, "Adjustable Dimensions."

The DIMENSION statement must precede the first appearance of each subscripted variable in an executable statement or DATA statement.

GENERAL FORM

DIMENSION $v_1(i_1), v_2(i_2), \ldots, v_m(i_m)$
where:
1. v is a variable.
2. i is composed of one, two, or three unsigned integer constants and/or integer variables, separated by commas (i may be composed of variables only when the DIMENSION statement appears in a FUNCTION or SUBROUTINE subprogram).

EXAMPLES:

```
DIMENSION A(10), B(5, 15), CVAL(3, 4, 5)
DIMENSION NEXT(I, J, K)
DIMENSION A(I, 2)
```

Integer Dimensions

The size of an array is defined when all its dimensions are integer constants in a DIMENSION statement. This DIMENSION statement provides the information necessary to allocate storage for an array.

One, two, or three integer constants may also be specified as the dimensions of an array in a COMMON statement or any of the Type statements.

Adjustable Dimensions

The name of an array and the constants that are its dimensions may be presented as arguments in a call to a FUNCTION or SUBROUTINE subprogram. In the subprogram the dimensions of the array are given as variables, and the absolute dimensions are substituted when the subprogram is entered. Thus, the same subprogram may be used for arrays of varying sizes (but with the same number of dimensions).

A subprogram array that has adjustable dimensions must be defined in a calling program. Only a DIMENSION statement or one of the Type statements may be used to define the array. The actual dimensions are presented to a FUNCTION subprogram through an arithmetic expression and to a SUBROUTINE subprogram through a CALL statement.

In the subprogram the SUBROUTINE or FUNCTION statement argument list must contain the dummy array name and all the variables used as dimensions. The DIMENSION statement in the subprogram must show the variables as dimensions of the dummy array. These dimensions may not be altered within the subprogram.

The following example illustrates the use of adjustable dimensions:

```
CALLING PROGRAM                 SUBPROGRAM
                                SUBROUTINE MAYMY
                                (R, L, M)
      .                           .
      .                           .
      .                           .
DIMENSION B(2, 3)                 .
CALL MAYMY(B, 2, 3)             DIMENSION R(L, M)
      .                           .
      .                           .
      .                           .
DIMENSION C(4, 5)               DO 100 I=1, L
CALL MAYMY(C, 4, 5)               .
      .                           .
      .                           .
```

The first time the subprogram MAYMY is entered, L and M, the variables used as dimensions of the array R, are replaced by 2 and 3, respectively. The array name B is substituted for the dummy array name R. The second time the subprogram is entered, the variables L and M are replaced by 4 and 5, respectively, and the array name C is substituted for the dummy array name R.

Following is an example of the use of adjustable dimensions in a FUNCTION subprogram:

```
MAIN PROGRAM                    SUBPROGRAM
DIMENSION A(3)                  FUNCTION JOB (R, I)
X=JOB(A, 3)                     DIMENSION R(I)
.                                 .
.                                 .
.                               JOB=R(1) + R(2) + R(3)
.                                 .
.                                 .
```

DATA Statement

Data variables may be defined in the source program by means of the DATA statement. The DATA statement provides for data to be placed into the object program during compilation of the program. If the data variables are redefined during program execution, they will assume their new values regardless of the value given in the DATA statement.

GENERAL FORM
DATA list/d_1, d_2, . . . d_n/, list/d_1, d_2, k*d_3, . . . d_m/, . . . where: 1. list contains the names of the variables being defined, separated by commas (the elements of the list may have integer constant subscripts). 2. d is an element of data. 3. k is an integer constant that appears before the d field to indicate that the d field is to be repeated k times [an asterisk (*) must appear after the k].

The element, d, may be any of the following:

1. *An integer, real, double precision, or complex constant.* When double precision and/or complex constants are specified, the corresponding variables must have appeared in a preceding DOUBLE PRECISION statement and/or COMPLEX statement.

2. *One or more alphameric characters.* This field is written as:

nH$a_1a_2a_3$ . . . a_n

where:

n is the number of alphameric characters following H.

a is an alphameric character or blank.

Each group of six alphameric characters or blanks forms a word. If n is not a multiple of six, the characters are left-justified and the remaining character locations in the last core storage word are filled with blanks.

When more than 6 alphameric characters are required, the DATA statement must be preceded by a DIMENSION statement indicating the number of core storage words needed by the alphameric field. For example, if the programmer wishes to define 15 alphameric characters starting at G(1), he must use a DIMENSION statement defining G as at least three words in length.

EXAMPLE:

```
DIMENSION G(3)
DATA G/15HDATAbTObBEbREAD/
```

3. *A series of octal digits.* An octal field is written as a series of octal digits preceded by the letter O. The field following the letter O may include from one to twelve signed or unsigned octal digits.

4. *A logical constant.* A logical field may be written in one of the following forms:

```
.TRUE.
.FALSE.
T
F
```

The logical constants T and F may be used only in the DATA statement. Furthermore, if a logical constant is specified in a DATA statement, a LOGICAL statement must precede it. The LOGICAL statement must contain the variables that will be replaced by logical constants.

Uses of the DATA statement are illustrated in the following examples:

```
      .
      .
      .
DATA R, Q/14.2, 3HEND/, Z/O777777000001
      .
      .
      .
LOGICAL LA, LB, LC, LD
DATA LA, LB, LC, LD/F, .TRUE., .FALSE., T/
      .
      .
      .
DIMENSION E(5),A(4),B(3,3,4)
COMPLEX D
DATA E, A(3), B(2,1,4), D/5*0.0,1.0,2.0,(3.1,4.5)/
```

There must be a direct relationship between the list and the data. Each element of data should correspond to one nondimensioned variable or an element of an array. When data is to be placed into an entire array, the name of the array is placed in the list without subscripts. The number of data elements must be equal to the size of the array. When data is to be placed into a complex variable field, the data should be a complex constant.

The following two statements illustrate the one-for-one correspondence between list items and data items.

```
DIMENSION B(25)
DATA A,B,C/24*4.0,3.0,2.0,1.0/
```

The DATA statement places 4.0 into A, B(1), . . . B(23), and 3.0, 2.0, and 1.0 into B(24), B(25), and C, respectively.

BLOCK DATA Subprogram

The BLOCK DATA subprogram causes data to be entered into a labeled COMMON block. This subprogram may contain only the DATA, COMMON, DIMENSION, and Type statements associated with the data being defined.

GENERAL FORM
BLOCK DATA

The first statement of this subprogram must be the BLOCK DATA statement. The subprogram may not contain any executable statements.

One or more labeled COMMON blocks may be specified in a COMMON statement, and data can be entered into as many labeled COMMON blocks as are specified.

The COMMON statement must include all elements of the labeled COMMON blocks, although these elements need not appear in the DATA statement. The DATA statement, however, must define only the elements listed in the COMMON statement.

Following is an example of a BLOCK DATA subprogram. Note that the element A of the COMMON block ELN and the element Y of the COMMON block RMG are listed in the COMMON statement but are not defined in the DATA statement.

```
BLOCK DATA
COMMON/ELN/C,A,B/RMG/Z,Y
DIMENSION B(4), Z(3)
DOUBLE PRECISION Z
COMPLEX C
DATA B(1)/1.1/,C/(2.4,3.769)/,Z(1)/7.6498085D0/
END
```

COMMON Statement

GENERAL FORM
COMMON a, b, c, . . . where: a, b, c, . . . are variables that may be dimensioned.

EXAMPLES:

```
COMMON A, B, C
COMMON A, B(5, 3), C
```

Variables, including array names, appearing in a COMMON statement are assigned locations relative to the beginning of COMMON. This COMMON area may be shared by a program and its subprograms. The locations in the COMMON area are assigned in the sequence in which the variables appear in the COMMON statement, beginning with the first COMMON statement of the program.

1. Two variables in COMMON may not be made equivalent to each other.

2. Variables brought into a COMMON block through EQUIVALENCE statements may increase the size of the block (the section "The EQUIVALENCE Statement" has additional information).

3. If the variables appearing in a COMMON statement contain dimension information, they must not appear in a DIMENSION statement.

4. Elements placed in COMMON may be placed in separate blocks. These separate blocks may share space in core storage at object time. Blocks are given names, and those with the same name occupy the same space.

5. The symbolic name of a COMMON block contains one to six alphameric characters, the first of which is alphabetic. The symbolic name precedes the variable names comprising the block. The block name is always embedded in slashes (e.g., /BB/). It must not be used for any other purpose in the same Processor application.

There are two types of COMMON blocks: blank and labeled.

a. Blank COMMON is indicated either by omitting the block name if it appears at the beginning of the COMMON statement, or by preceding the blank COMMON variable by two consecutive slashes.

b. Labeled COMMON is indicated by preceding the labeled COMMON variables with the block name embedded in slashes.

6. The field of entries pertaining to a block name ends with a new block name, the end of the COMMON statement, or a blank COMMON designation.

7. Block name entries are cumulative throughout the program. For example, the first two COMMON statements have the same effect as the third:

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

8. Blank COMMON may be any length. Labeled COMMON must conform to the following size requirement: all COMMON blocks of a given name must have the same length in all programs that are executed together.

9. Arrays with adjustable dimensions may not be specified in a COMMON statement (see "Adjustable Dimensions" under "DIMENSION Statement").

COMMON and DIMENSION

Array names appearing in COMMON must also appear in a DIMENSION statement in the same program, or the dimension information must be included in the COMMON statement, as in the following example:

```
COMMON A, B (10, 15), C
```

The COMMON statement may appear anywhere in the program, unless it contains dimension information. In this case, it must precede the first appearance of the dimensioned variables in any executable statement.

EQUIVALENCE Statement

This statement permits data storage to be shared within a single program in a way analogous to that in which the COMMON statement causes the data storage area to be shared between programs.

GENERAL FORM
EQUIVALENCE (a, b, c, . . .), (d, e, f, . . .) where: a, b, c, d, e, f, . . . are variables that may be subscripted; these subscripts must be integer constants. The number of subscripts appended to a variable must be equal to the number of dimensions of the variable.

EXAMPLE:

```
DIMENSION B(5), C(10, 10), D(5, 10, 15)
EQUIVALENCE (A, B(4), C(5, 4)), (D(1, 4, 3), E)
```

Each major pair of parentheses in the EQUIVALENCE statement list encloses the names of two or more variables that are to be stored in the same location during execution of the object program; any number of equivalences (i.e., sets of parentheses) may be given.

In an EQUIVALENCE statement, B(4) is the third storage location following the one that contains B(1). In general, B(p) is defined for p>0 to mean the (p−1)th location after the beginning of the B array (i.e., the pth location in the array). If p is not specified, it is taken to be 1. Thus, in the preceding example, the EQUIVALENCE statement indicates that A, B(4), and C(5, 4) occupy the same location. It also specifies that D(1, 4, 3) and E are to share the same location.

Locations can be shared only by variables, not by constants. The sharing of storage locations requires the knowledge of which FORTRAN statements will cause a new value to be stored in a location. There are four such statements:

1. Execution of an arithmetic or logical statement stores a new value in the variable on the left side of the equal sign.

2. Execution of a DO statement stores a new indexing value.

3. Execution of a READ statement stores new values in the variables in the input list.

4. Execution of a CALL statement stores new values in arguments used to return values.

NOTE: The EQUIVALENCE statement is designed to cause only the sharing of data storage. It does not cause the sharing of characteristics of two variable names; for example:

1. A Type declaration made for one variable name will not be applied to the other.

2. The fact that a variable controls indexing will not be applied to a related variable. Although the second variable changes, it may not be reflected in the indexing generation.

EQUIVALENCE and COMMON

The sequence in which data is listed in COMMON is the sequence in which data is stored. However, variables brought into a COMMON block through EQUIVALENCE statements may increase the size of the block indicated by the COMMON statements, as in the following example:

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

The layout of storage is:

```
A
B
C    D(1)
     D(2)
     D(3)
```

In the above example, a statement such as EQUIVALENCE (A,D(3)) which would extend COMMON in the reverse direction, is invalid.

Because the programmer has complete control over the sequence of location of the variables in COMMON, they may be used as the medium for transmitting arguments from the calling program to the FORTRAN FUNCTION or SUBROUTINE subprogram being called. In this way, they are transmitted implicitly, as if specified in the argument list following the subroutine name.

To obtain implicit arguments, it is necessary only to have the corresponding variables in the two programs occupy the same location. This can be done by having them occupy corresponding positions in COMMON statements of the two programs or by having them appear in COMMON blocks with the same label.

A double-word variable in COMMON must start the COMMON block or be an even number of words away from the beginning of each COMMON block.

In EQUIVALENCE statements, the effect of the EQUIVALENCE must place double-word variables at the beginning of the EQUIVALENCE chain or an even number of words away from the beginning of the EQUIVALENCE chain. For example, the following statement is valid:

```
EQUIVALENCE (A, C(3))
```

The following statement is invalid:

```
EQUIVALENCE (A, C(2))
```

In these examples, A is a double-word variable, and C is a dimensioned single-word variable.

Type Declaration Statements

There are six Type statements: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, EXTERNAL. These statements determine the type of variable associated with each variable name in the statement. This Type declaration is in effect throughout the program.

GENERAL FORM

```
INTEGER a(i1), b(i2), c(i3), . . .
REAL a(i1), b(i2), c(i3), . . .
DOUBLE PRECISION a(i1), b(i2), c(i3), . . .
COMPLEX a(i1), b(i2), c(i3), . . .
LOGICAL a(i1), b(i2), c(i3), . . .
EXTERNAL a,b,c, . . .
```

where:

1. a,b,c, . . . are variable names or function names appearing within the program.
2. i is an optional subscript composed of 1, 2, or 3 integer constants that may be used to specify dimensions for each variable. Subscripts may only be appended to variable names appearing within the program, not to function names.

EXAMPLES:

```
INTEGER BIGF, X, QF, LSL, A(10, 10)
REAL IMIN, LOG, GRN, KLW
DOUBLE PRECISION Q, J, DSIN
EXTERNAL SIN, MATMPY, INVTRY
LOGICAL F, G, L(2, 5)
COMPLEX C(4, 5, 3), D
```

RULES FOR TYPE STATEMENTS

1. A variable or function declared to be of a given type remains of that type throughout the program. The type may not be changed.

2. A variable may appear in a maximum of two Type statements only if one of them is the EXTERNAL type.

3. INTEGER indicates that the variables listed are integer (fixed point) and overrides the alphabetic naming convention.

4. REAL indicates that the variables listed are real (single-precision floating-point) variables and overrides the alphabetic naming convention.

5. LOGICAL indicates that the variables are logical variables and assume only the value TRUE or FALSE.

6. EXTERNAL indicates that the names listed after EXTERNAL are subprogram names to be used as arguments in a subroutine or function call; for example:

```
MAIN PROGRAM                SUBPROGRAM
  .                         SUBROUTINE SUBR (F, X, Y)
  .                         .
  .                         .
```

```
MAIN PROGRAM                SUBPROGRAM
  EXTERNAL INVTRY           .
  EXTERNAL AMNT             Y=F(X)
  .                         .
  .                         RETURN
  CALL SUBR (INVTRY,A,B)    .
  .
  .
  CALL SUBR (AMNT,C,D)
  .
  .
```

In the first CALL, F becomes INVTRY, X becomes A, and Y becomes B; that is, the CALL means B=INVTRY(A). The second CALL means D=AMNT(C).

The section "The CALL Statement" contains a discussion of arguments.

7. DOUBLE PRECISION indicates that the variables listed are double-precision variables.

8. COMPLEX indicates that the variables listed are complex variables.

9. All Type statements pertaining to any given variable must precede the first appearance of the variable within an executable statement of the program.

10. The EXTERNAL statement may not specify the dimension of variables.

11. Any variable having its dimension specified by a Type statement may not have its dimension specified elsewhere.

Appendix A. Table of Source Program Characters

Character	Card	BCD Tape	Storage	Character	Card	BCD Tape	Storage	Character	Card	BCD Tape	Storage	Character	Card	BCD Tape	Storage
1	1	01	01	A	12 1	61	21	J	11 1	41	41	/	0 1	21	61
2	2	02	02	B	12 2	62	22	K	11 2	42	42	S	0 2	22	62
3	3	03	03	C	12 3	63	23	L	11 3	43	43	T	0 3	23	63
4	4	04	04	D	12 4	64	24	M	11 4	44	44	U	0 4	24	64
5	5	05	05	E	12 5	65	25	N	11 5	45	45	V	0 5	25	65
6	6	06	06	F	12 6	66	26	O	11 6	46	46	W	0 6	26	66
7	7	07	07	G	12 7	67	27	P	11 7	47	47	X	0 7	27	67
8	8	10	10	H	12 8	70	30	Q	11 8	50	50	Y	0 8	30	70
9	9	11	11	I	12 9	71	31	R	11 9	51	51	Z	0 9	31	71
b	b	20	60	+	12	60	20	-	11	40	40	0	0	12	00
=	8-3	13	13	.	12 8-3	73	33	$	11 8-3	53	53	,	0 8-3	33	73
'	8-4	14	14	)	12 8-4	74	34	*	11 8-4	54	54	(	0 8-4	34	74

NOTE: The characters $ and ' can be used in FORTRAN only as alphameric text in a FORMAT statement or as alphameric arguments.

Appendix B. Source Program Statements and Sequencing

The following is a complete list of the 7040/7044 FORTRAN IV source program statements, their sequence of execution, and their ordering in the source program.

STATEMENT	NORMAL SEQUENCING	EXECUTABLE OR NONEXECUTABLE	ORDERING IN THE SOURCE PROGRAM
a=b	Next statement	Executable	May be placed anywhere.
ASSIGN n to i	Next statement	Executable	May be placed anywhere.
BACKSPACE i	Next statement	Executable	May be placed anywhere.
BLOCK DATA	Next statement	Nonexecutable	Must be the first statement of the subprogram.**
CALL	First statement of called subprogram	Executable	May be placed anywhere.
COMMON	Next statement	Nonexecutable	May be placed anywhere in the program unless it contains dimension information, in which case it must precede the first appearance of the dimensioned variables in any executable statement.*
COMPLEX	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers in any executable statement of the program.*
CONTINUE	Next statement	Executable	May be placed anywhere but it is most often used as the last statement in the range of a DO.
DATA	Next statement	Nonexecutable	May be placed anywhere.
DIMENSION	Next statement	Nonexecutable	Must precede the first appearance of each subscripted variable in any executable statement for which it specifies the size.*
DO	Normal DO sequencing, then the next statement	Executable	May be placed anywhere.*

STATEMENT	NORMAL SEQUENCING	EXECUTABLE OR NONEXECUTABLE	ORDERING IN THE SOURCE PROGRAM
DOUBLE PRECISION	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers within an executable statement of the program.*
END	Terminates compilation of program	Nonexecutable	Must be the physically last statement of the program.
END FILE	Next statement	Executable	May be placed anywhere.
EQUIVALENCE	Next statement	Nonexecutable	May be placed anywhere.*
EXTERNAL	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers within an executable statement of the program.*
FORMAT	Next statement	Nonexecutable	May be placed anywhere.*
FUNCTION	Next statement	Nonexecutable	Must be the first statement of a FUNCTION subprogram.**
GO TO n	Statement n	Executable	May be placed anywhere.*
GO TO i, (n_1, n_2, . . . , n_m)	Statement last assigned to i	Executable	May be placed anywhere.*
GO TO (n_1, n_2, . . . , n_m), i	Statement n_i	Executable	May be placed anywhere.*
IF (t)s	Statement s or next statement if relation is true or false, respectively	Executable	May be placed anywhere.
INTEGER	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers within an executable statement of the program.*
LOGICAL	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers within an executable statement of the program.*
PAUSE	Next statement	Executable	Should be placed where a temporary halt is desired.
READ	Next statement	Executable	May be placed anywhere.
REAL	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers within an executable statement of the program.*
RETURN	The first statement, or part of a statement, following the reference to the subprogram	Executable	Must be the last executed statement of a subprogram.
REWIND	Next statement	Executable	May be placed anywhere.
STOP	Terminates the execution of the program	Executable	Should be placed where the termination of the program is desired.
SUBROUTINE	Next statement	Nonexecutable	Must be the first statement of a SUBROUTINE subprogram.**
WRITE	Next statement	Executable	May be placed anywhere.

*This statement may not end the range of a DO.
**This statement may not appear anywhere except as the first statement of a subprogram.

Appendix C. Differences Between FORTRAN II and FORTRAN IV

1. Function Naming
   a. Incompatibilities may arise where the initial character of a function name is used to denote the type as floating point (real) or fixed point (integer) in FORTRAN II. In FORTRAN IV, this difficulty is handled by the Type statements REAL and INTEGER, which define a variable or function name as floating or fixed point, respectively (the section "The Type Declaration Statements" has additional information).
   b. An open, closed, or arithmetic statement function name in FORTRAN II contains 4-7 characters, ending in F; in FORTRAN IV, the number of characters is 1 through 6 and the final F has no meaning. In both cases, the first character of the function name must be alphabetic.
   c. Built-in and arithmetic statement functions are not identified by a terminal F in FORTRAN IV; they are named as described in item b. The FORTRAN II library function is a FORTRAN IV FUNCTION subprogram that is internal to the Processor.

2. COMMON and EQUIVALENCE
   a. In FORTRAN IV, EQUIVALENCE does not affect the ordering within COMMON, and it does not create a gap in COMMON storage; the only effect it can have on a COMMON block is to make its size greater than that indicated by the COMMON statements of the program.
   b. The FORTRAN IV COMMON statements may contain dimension information.

3. In FORTRAN IV, if an explicit type is given to a variable name that is used throughout the program as an ordinary variable and also as an argument of an arithmetic statement function, the explicit type applies in both contexts.

4. Implicit multiplication, which occurs in FORTRAN II as a by-product of the arithmetic translator techniques, is not permitted in FORTRAN IV. Thus, the following combinations are not permitted in FORTRAN IV:

```
K( )
( )V
( )K
```

where:

V
is a variable.

K
is a constant.

( )
is any arithmetic expression within parentheses.

5. The FORTRAN II statements in column 1 are changed to the FORTRAN IV statements in column 2.

FORTRAN II STATEMENT	FORTRAN IV STATEMENT
IF ACCUMULATOR OVERFLOW n_1, n_2	CALL OVERFL (J)
IF QUOTIENT OVERFLOW n_1, n_2	CALL OVERFL (J)
IF DIVIDE CHECK n_1, n_2	CALL DVCHK (J)
IF (SENSE SWITCH i) n_1, n_2	CALL SSWTCH (I,J)
SENSE LIGHT i	CALL SLITE (I)
IF (SENSE LIGHT i) n_1, n_2	CALL SLITET (I,J)
READ TAPE i, list	READ (i) list Binary record
READ INPUT TAPE i, n, list	READ (i, n) list BCD record
WRITE TAPE i, list	WRITE (i) list Binary record
WRITE OUTPUT TAPE i, n, list	WRITE (i, n) list BCD record

The FREQUENCY, READ DRUM, and WRITE DRUM statements of FORTRAN II are not part of the FORTRAN IV language.

6. LOGICAL, an additional Type statement that defines variables to be used in logical computation, has been added to FORTRAN IV.

7. DOUBLE PRECISION and COMPLEX, additional Type statements that define variables to be used in arithmetic computation, have been added to FORTRAN IV.

8. The computational results of an object program produced from a source program by the 7040/7044 FORTRAN IV compiler and the computational results of an object program produced from an equivalent source program by the 7090/7094 FORTRAN II compiler may not be identical. Differences that occur are due to the following dissimilarities between the 7040/7044 FORTRAN IV compiler and the 7090/7094 FORTRAN II compiler:

   a. The logarithm subroutine of FORTRAN IV employs a new algorithm that yields more accurate results for most arguments than does the logarithm subroutine of FORTRAN II.
   b. Real constants written into the source program are converted by FORTRAN IV by a somewhat different algorithm than that used by FORTRAN II. The result is that more significance tends to be preserved and a more accurate conversion is achieved by FORTRAN IV than by its predecessor.
   c. The mathematical subroutines in FORTRAN IV are assembled by the Macro Assembly Program (IBMAP), and those in FORTRAN II are assembled by FAP. The conversion routines in IBMAP provide more precise conversions for constants than do those in FAP. As a consequence, FORTRAN IV tends to produce more precise results than FORTRAN II for subroutines that use the same algorithm (and its associated constants). The SIN/COS subroutine is a very good example of this effect.
   d. The order in which a sequence of multiplications (or of multiplications and divisions) is executed by the object program in FORTRAN IV may be different from that in FORTRAN II. If such a difference in ordering should occur, neither method may be considered superior to the other from the standpoint of computational accuracy.
   e. In FORTRAN II, the maximum size of an integer is 2^{17}; in FORTRAN IV it is 2^{35}.

Appendix D. Additional Statements Accepted by the Compiler

The following source program statements are not part of the FORTRAN IV language. They are currently being processed by the FORTRAN IV compiler for compatibility with FORTRAN II.

Input/Output Statements

The input/output statements READ n, PUNCH n, and PRINT n are processed by the 7040/7044 compiler. The form of the statements is given as follows, where n is a FORMAT statement number or a variable FORMAT name.

INPUT

GENERAL FORM
READ n, list

EXAMPLE:

READ 10, (A(I), I=1, 5)

This statement causes records to be read from the systems input unit. Successive records are read in accordance with the FORMAT statement until the list has been satisfied; i.e., all data items have been read, converted, and stored in the locations specified by the list.

OUTPUT

GENERAL FORM
PUNCH n, list PRINT n, list

EXAMPLES:

PUNCH 20, (A(J), J=1, 6)
PRINT 2, (A(J), J=1, 6)

The PUNCH statement causes records to be transmitted to the systems punch unit. The PRINT statement causes the object program to transmit data to the systems output unit. In both of these statements, successive records are written in accordance with the FORMAT statement until the list has been satisfied.

Appendix E. Machine-Dependent Features

Built-in Features

The built-in functions shown in Figure 9 are included only to allow the user of FORTRAN to utilize the special logical operations of the 7040/7044 Data Processing Systems. They do not form a part of the standard FORTRAN language, since their function cannot be exactly duplicated on other machines.

Function Definition	Function Name	No. of Args.	Type of	
			Function	Arguments
Logical intersection of two or more 36-bit arguments	AND	≧2	Real	Real or Integer
Logical union of the two or more 36-bit arguments	OR	≧2	Real	Real or Integer
Logical 1's complement of the 36-bit argument	COMPL	1	Real	Real or Integer

Figure 9. Built-in Functions

Machine Indicator Tests

The following SUBROUTINE subprograms are available in the library so that the FORTRAN user may test the status of machine indicators and traps. They are called by a CALL statement. (The parameter J must be an integer variable and I must be an integer expression.)

OVERFL(J)

J is set to: 1 if a floating-point overflow condition exists, 3 if a floating-point underflow condition exists, or 2 if neither condition exists.

DVCHK(J)

J is set to: 1 if the Divide Check indicator is on, or 2 if it is off. The indicator is left in the off condition.

SSWTCH(I,J)

Sense Switch I (where I is equal to 1, 2, 3, 4, 5, or 6 at execution) is tested, and J is set to: 1 if the sense switch is on, or 2 if it is off.

SLITE(I)

If I=0, all FORTRAN sense lights are turned off. If I=1, 2, 3, or 4, the corresponding FORTRAN sense light is turned on. Because there are no physical sense lights on the 7040, the status of the sense lights is kept as a logical variable internal to the sense light routines.

SLITET(I,J)

Sense Light I is tested and turned off. The variable J is set to: 1 if the sense light is on, or 2 if the sense light is off.

Appendix F: Compilation Output

Depending on control card options, a complete output listing of a FORTRAN compilation may consist of five parts:

1. FORTRAN source listing
2. MAP Control Dictionary
3. MAP assembly listing (3 options in $IBFTC card: FULIST, LIST, NOLIST)
4. MAP Cross-Reference Dictionary (2 options in $IBFTC card: REF, NOREF)
5. Error Message (if there are errors)

FORTRAN Source Statement Listing

Figure 10 illustrates a typical list.

1. The first line of the list contains text from the $JOB card, "FORTRAN SOURCE LIST," the deck name (card columns 8-13) from the $IBFTC card, date, and page number.

2. Line two consists of the column headings "ISN" (Internal Serial Number) and "SOURCE STATEMENT."

3. Each source statement that appears on the listing is assigned an ISN, and that number is shown in the ISN column next to the card image of the statement (note that the number set is octal). Numbers are also assigned internally; these numbers do not appear on the listing; they cause gaps in the number sequence.

4. The remainder of the listing is the FORTRAN source input.

```
FORTRAN ILLUSTRATION                    FORTRAN SOURCE LIST              08/24/64      PAGE  1
      ISN       SOURCE STATEMENT

        0 $IBFTC ILLUS   LIST,REF,NODECK                                    ILL00010
        1        DIMENSION Y(10),X(10,10)                                   ILL00020
        2        DO 1 I=1,10                                                ILL00030
        3 1      X(I,J)=Y(I)+1.                                             ILL00040
        5        IF (I.EQ. 1) A=A+1.                                        ILL00045
       10        IF (I.EQ.1) A=A+I                                          ILL00050
       13        READ     (0)  A,((X(I,J),I=1,10),B,Y(J),J=1,10)            ILL00060
       24        GO TO 8
       25        STOP                                                       ILL00080
       26        END                                                        ILL00090
```

Figure 10. FORTRAN Source Statement Listing

MAP Control Dictionary Listing

Figure 11 illustrates a typical Control Dictionary listing. The first column on the left contains the number of the Control Dictionary entry. The second column contains the octal representation of the binary entry. The remaining three columns are generated by the assembler for documentation: the third column is the external identification for the entry, the fourth column indicates the type of entry, and the last column gives additional miscellaneous information.

Assembled Text Listing

A MAP listing will be produced if either the LIST or the FULIST option is on the $IBFTC card. The full listing (FULIST option) is illustrated and described in the manual, *IBM 7040/7044 Operating System: Macro Assembly Program (MAP) Language*, Form C28-6336-1, Appendix C. The LIST option, which provides a condensed listing, is shown in Figure 12. For each instruction generated by the compiler, the listing contains the MAP statement number, the error flag (if any), the relative location assigned to each instruction, the symbolic operation code, and the symbolic variable field.

The listing is read across one line at a time; there is a maximum of three sequential instructions on each line. The MAP statement number appears only once per line, and it shows the number of the first instruction on the line.

There is a correspondence between the executable FORTRAN statements and the generated MAP instructions. The MAP instructions follow the statement USE PRGCT. (MAP statement number 19) and are assigned relative location 00000.

Each numbered FORTRAN statement has a corresponding name with an "S" appended to it on the MAP listing; e.g., ISN-3 is statement number 1, and MAP statement 31 has the name 1S.

All other executable FORTRAN statements have related names of the form P.nnnn on the MAP listing, where nnnn is the ISN; e.g., ISN-2 corresponds to MAP statement 21, which has the name P.0002.

Cross-Reference Dictionary Listing

The Cross-Reference Dictionary, shown in Figure 13, contains three types of references: to defined symbols, to location counters, and to multiply-defined symbols.

The listing of references to defined and multiply-defined symbols gives the symbol, its value, and the MAP statement numbers of those instructions that refer to it.

The P.xxxx symbols are generated by the compiler for literals, for names on generated coding, and with the LORG statement.

Error Messages

Figure 14 illustrates the Error Message listing, which contains the following information:

1. The severity code identifies the action taken because of the error. Following are the codes:

0	Warning message only.
1	Mild error. The object program will be loaded and executed as specified on the $IBJOB card.
2	Definite error. The object program will be loaded if the GO, MAP, or LOGIC options have been specified on the control card; then specified MAP or LOGIC options will be performed. However, the program will not be executed.
4	Serious error. GO, MAP, and LOGIC options will be deleted, and the binary deck will be incomplete.
7	Extreme error. GO, MAP, and LOGIC options will be deleted. Output will be incomplete.

2. The listing contains the MAP statement number that *may* be related to the Error Message. It will have no significance if there is an ISN number (see 4 below).

3. It contains the Error Message number.

4. When present, the ISN number will relate the message to a corresponding statement on the FORTRAN Source List. If the ISN information is present, the MAP statement number will have no significance. In referring to the FORTRAN Source List, if the number is skipped on the listing, the programmer should refer to the next lower ISN and its corresponding statement (this statement has more than one ISN assigned to it).

5. The Error Message listing contains the message text.

```
FORTRAN ILLUSTRATION                    IBMAP ASSEMBLY ILLUS                    08/24/64      PAGE  2
        CONTROL DICTIONARY

                                $IBLDR ILLUS   08/24/64
                                $CDICT ILLUS

BINARY CARD 00000
 000    000266000000                    PREFACE MODE=REL,PROGRAM-START=00000
        000000000004
 001    314343646260          ILLUS     DKNAME  DECK-START=00000,DECK-END=00266
        000267000000
 002    263143000033          FIL00.    EXTERN
        300000000000
 003    224543314633          BNLIO.    EXTERN
        300000000000
 004    636222314633          TSBIO.    EXTERN
        300000000000
 005    514351314633          RLRIO.    EXTERN
        300000000000
 006    623341673163          S.JXIT    EXTERN
        300000000000
 007    622563264733          SETFP.    EXTERN
        300000000000
 010    000000000000                    EVEN    AT  00101
        000000000101

                                $TEXT  ILLUS                                          ILLU0001
```

Figure 11. Control Dictionary Listing

```
FORTRAN ILLUSTRATION                    IBMAP ASSEMBLY ILLUS                    08/24/64      PAGE  3

1                       BEGIN   PLGCT.,*
4           00073       USE     STRCT.
7           00077       OCT     201400000000
10          00101 LTCR2. OCT    000000000000,000000000000
11          00103       OCT     201400000000,146000000000
13          00106 X     BSS     100
16                      EXTERN  FIL00.
19        E 00000       USE     PRGCT.
22          00002       STQ     I
25          00005       VLM     =0000001200000,0,15
27          00007       PAC     ,1
30          00011       FAD     LITCT.+1
33          00014       TXI     *+1,2,-1
36          00017       SUB     LITCT.+2
39          00022       FAD     LITCT.+1
42          00024 P.0010 CLA    I
45          00027 P.0012 BSS
48          00030       PZE     FIL00.
51          00033 P.0014 LDQ    =0000000000001
54          00036       SXA     S.0033,4
57          00040       STQ     I
59          00042       VLM     =0000001200000,0,15
61          00044       PAC     ,1
64          00046       STO     X-11,1
67          00051       TXI     *+1,4,1
70          00054 P.0021 TSL    BNLIO.
73          00057       LXA     S.0033,1
76          00061 S.0033 AXT    **,4
79          00064       PXA     ,4
82          00067       TXH     S.0034,4,-11
85          00072 P.0025 TRA    S.JXIT
88          00000       USE     PLGCT.
91                      EXTERN  RLRIO.
94                      BEGIN   PRGCT.,*
97          00073       USE     SFLCT.

103         00074       OCT     000001200000

                $DKEND ILLUS

            00000       USE     PRGCT.
            00073       LORG
            00100       OCT     000000000001

            00252 B     BSS     1
            00254 A     BSS     1
            00000       TSL     SETFP.
            00003       AXT     -1,2

            00010 S.0032 BSS
            00012       STO     X-11,1
            00015       TXH     S.0032,2,-11
            00020       TNZ     P.0006
            00023       STO     A
            00025       SUB     LITCT.+2
            00027 P.0011 BSS
            00031       TSL     BNLIO.
            00034       STO     J
            00037 S.0034 BSS
            00041       LDQ     J

            00045 S.0035 BSS
            00047 P.0020 TXI     *+1,1,-1
            00052       SXA     I,4
            00055       STO     B
            00060       STO     Y-1,1
            00062       TXI     *+1,4,-1
            00065       SUB     =0000000100000
            00070       TSX     RLRIO.,4
            00073 P.0026 BSS
                        EXTERN  BNLIO.
                        EXTERN  S.JXIT
            00073       USE     PRGCT.
                        BEGIN   STRCT.,*
                *        GENERATED LITERALS
            00075       OCT     000000100000
                        END

            00073       USE     SFLCT.
            00076 LITCT. OCT    000000000000
                        EVEN

            00105 J     BSS     1
            00253 I     BSS     1
            00255 Y     BSS     10
            00001 P.0002 LDQ    =0000000000001
            00004       LDQ     J
            00006       ADD     I
            00010 1S    CLA     Y-1,2
            00013 P.0004 TXI    *+1,1,-1
            00016 P.0005 CLA    I
            00021 P.0007 CLA    A
            00024 P.0006 BSS
            00026       TNZ     P.0011
            00027 P.0013 TSX    TSBIO.,4
            00032       STO     A
            00035       AXT     -1,4
            00037 P.0016 LDQ    =0000000000001

            00043       ADD     I
            00045 P.0017 TSL    BNLIO.
            00050       LXA     I,4
            00053       TXL     S.0035,4,10
            00056       TSL     BNLIO.
            00061 P.0022 BSS
            00063       SXA     S.0033,4
            00066       SLW     J
          E 00071 P.0024 TRA    8S
                *        START OF PROLOGUE
                        EXTERN  TSBIO.
                        EXTERN  SETFP.
                        BEGIN   SFLCT.,*
            00073       OCT     000000000001
```

Figure 12. Condensed Listing

```
FORTRAN ILLUSTRATION                    IBMAP ASSEMBLY ILLUS          08/24/64      PAGE  4
        CROSS-REFERENCE DICTIONARY


REFERENCES TO DEFINED SYMBOLS
 VALUE    NAME      STATEMENT NUMBERS

 00010    1S
 00254    A         38,40,50
 00252    B         71
 VIRTUAL  BNLIO.    49,63,70,72
 VIRTUAL  FILOO.    48
 00253    I         22,26,35,42,57,60,66,68
 00105    J         24,52,58,81
 00076    LITCT.    30,36,39,43
 00101    LTCR2.
 VIRTUAL  RLRIO.    83
 VIRTUAL  SETFP.    20
 VIRTUAL  S.JXIT    85
 VIRTUAL  TSBIO.    47
 00106    X         31,64
 00255    Y         29,74
 00024    P.0006    37
 00027    P.0011    44
 00010    S.0032    34
 00061    S.0033    54,73,78
 00037    S.0034    82
 00045    S.0035    69
 00073    P.0036    100
 00076    P.0037    5
 00073    P.0040    106
 00073    P.0041    21,51,56
 00074    P.0042    25,59
 00075    P.0043    80

REFERENCES TO LOCATICN COUNTERS
LC START NAME      STARTING AND ENDING STATEMENT NUMBERS

 00000              1-1
 00000    PLGCT.    88-94
 00000    PRGCT.    2-2,19-87,95-96
 00073    SFLCT.    3-3,97-107
 00073    STRCT.    4-18

REFERENCES TO UNDEFINED SYMBOLS
          NAME      STATEMENT NUMBERS

          8S        84
```

Figure 13. Cross-Reference Dictionary Listing

```
FORTRAN ILLUSTRATION                    IBMAP ASSEMBLY ILLUS          08/24/64      PAGE  5
        ERROR MESSAGES


2 STATEMENT 19          ERROR 1323  ISN-12    ILLEGAL MIXED MODE ON RIGHT HAND SIDE OF ARITHMETIC STATEMENT.
0 STATEMENT 19          ERROR 1327  ISN-25    SHOULD HAVE A STATEMENT NUMBER BECAUSE OF  THE PRECEDING GO TO OR
                                    ARITHMETIC IF STATEMENT.
2 STATEMENT 84          ERROR 26    8S IS AN UNDEFINED SYMBOL

HIGHEST SEVERITY WAS 2.  EXECUTION DELETED.
```

Figure 14. Error Message Listing

Index

A page number in *italics* indicates that the designated reference is of particular importance.

Printed in U.S.A. C28-6329-2

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601