File No. 7090-25
Form C28-6390-3

IBM Systems Reference Library

IBM 7090/7094 IBSYS Operating System

Version 13

FORTRAN IV Language

This publication describes the IBM 7090/7094 FORTRAN IV language that is processed by the FORTRAN IV Compiler (IBFTC), #7090-FO-805, a component of the IBM 7090/7094 IBJOB Processor, which, in turn, is a subset of the IBSYS Operating System (Version 13). FORTRAN IV is a problem-oriented programming language designed primarily for scientific computations, and it closely resembles the language of mathematics. It includes various types of arithmetic, control, input/output, and specification statements.

PREFACE

This publication describes the FORTRAN IV language for the IBM 7090/7094 FORTRAN IV Compiler (IBFTC), a component of the IBM 7090/7094 IBJOB Processor. The IBJOB Processor is described in the publication IBM 7090/7094 IBSYS Operating System, IBJOB Processor, Form C28-6389. Minimum machine requirements for FORTRAN IV are also described in the IBJOB Processor publication.

The basic concepts of the FORTRAN language are described in the publication General Information Manual, FORTRAN, Form F28-8074, but the presentation of material in this publication is such that no previous knowledge of the FORTRAN language is required.

The information in Appendixes F and G permits a FORTRAN programmer to write, punch, and run a simple FORTRAN IV program without referring to another publication for source program format, control cards, or deck setup.

MINOR REVISION (APRIL 1966)
This edition, Form C28-6390-3, is a reprint of Form C28-6390-2, incorporating changes released in the following Technical Newsletter:

Form No.	Pages	Dated
N28-0179-0	7, 8, 9, 17, 23, 25, 31, 34, 36	March 1, 1966

Form C28-6903-2 and the Technical Newsletter are not obsolete.

CONTENTS

The FORTRAN IV language is a set of statements, expressions, and operators, which are used to write a source program. The FORTRAN IV Compiler converts this source program into an object program in machine language, ready to be run on the computer.

FORTRAN IV Language

The FORTRAN IV language is a programming code designed primarily for mathematically oriented computer applications. The language's close resemblance to algebraic notation facilitates the writing and documentation of programs involving mathematical formulas. It provides users with an easy method for writing efficient programs that will perform scientific calculations and data handling.

The statements in the FORTRAN IV language may be classified as follows:

1. The arithmetic statement specifies a numerical or logical calculation.
2. The control statements govern the flow of control in the program.
3. The input/output statements provide the necessary input/output routines and the input/output format.
4. The subroutine statements enable the programmer to define and use subprograms.
5. The specification statements provide information about the constants and variables used in the program and about storage allocation.

7090/7094 FORTRAN IV Compiler (IBFTC)

The IBM 7090/7094 FORTRAN IV Compiler is a component of the IBM 7090/7094 IBJOB Processor, which in turn is a component of the IBM 7090/7094 IBSYS Operating System. The compiler accepts source programs written in the FORTRAN IV language, analyzes the source program statements, and transforms them into Loader text (relocatable binary).

The compiler operates in two passes: an instruction compilation pass and an assembly pass. The output of the second pass is input to the Loader (IBLDR) of the IBJOB Processor. Because assembly is performed by the compiler, the Macro Assembly Program (IBMAP) is not used.

The compiler uses a technique called "phasing," which is designed to save considerable time for users, especially those with all-tape systems. With this technique, the system is read in only once for an entire job. Phasing is described in detail in the publication IBM 7090/7094 IBSYS Operating System, IBJOB Processor, Form C28-6389. Because the compiler uses this technique, programmers are encouraged to code subroutines for their programs where the same calculation must be performed more than once with different data for each calculation. Subroutines cost little in compile-time, they facilitate debugging, and, in FORTRAN programming shops, they facilitate the breaking up of assignments.

FORTRAN provides a means of expressing constants and variables, and provides subscript notations for expressing arrays of variables with one to seven dimensions.

CONSTANTS

Five types of constants are permitted in a FORTRAN source program: integer, real or single-precision, double-precision, complex, and logical.

Integer Constants

General Form
An integer constant consists of 1-11 decimal digits written without a decimal point.

Examples:

3
528
8085

An integer constant may be as large as $2^{35}-1$, except when used for the value of a subscript or as an index of a DO, or a DO parameter, in which case the value of the integer is computed modulo 2^{15}.

Real Constants

General Form
A real constant consists of one of the following: 1. One to nine significant decimal digits written with a decimal point, but not followed by a decimal exponent. 2. A sequence of decimal digits written with a decimal point, followed by a decimal exponent, which is written as the letter E followed by a signed or unsigned integer constant.

Examples:

21.
.203
8.0067
5.0 E3 (means 5.0×10^3, i.e., 5000.)
5.0 E-3 (means 5.0×10^{-3}, i.e., .005)

1. The magnitude of a real constant must be between the approximate limits of 10^{38} and 10^{-38}, or must be zero.
2. A real constant has precision to eight digits.

Double-Precision Constants

General Form
A double-precision constant consists of one of the following: 1. Ten or more significant decimal digits written with a decimal point, but not followed by a decimal exponent. 2. A sequence of decimal digits written with a decimal point, followed by a decimal exponent, which is written as the letter D followed by a signed or unsigned integer constant.

Examples:

21.987538294
21.9D0
.203D0
5.0D3 (means 5.0×10^3, i.e., 5000.)
5.0 D-3 (means 5.0×10^{-3}, i.e., .005)

1. The magnitude of a double-precision constant must lie between the approximate limits of 10^{-29} and 10^{38}, or must be zero. Numbers between 10^{-29} and 10^{-38} may be used, but only eight digits are significant in this range.
2. Double-precision constants are floating-point quantities that have precision to 16 digits.

Complex Constants

General Form
A complex constant consists of an ordered pair of signed or unsigned real constants separated by a comma and enclosed in parentheses.

Examples:

(3.2, 1.86) is equal to 3.2 + 1.86i.
(2.1, 0.0) is equal to 2.1 + 0.0i.
(5.0E3, 2.12) is equal to 5000. +2.12i.

where i is the square root of -1.

1. The first real constant represents the real part of the complex number; the second real constant represents the imaginary part of the complex number.
2. The parentheses are required regardless of the context in which the complex constant appears.
3. Each part of the complex constant may be preceded by a plus sign or a minus sign, or it may be unsigned.

Logical Constants

General Form
A logical constant may take either of the following forms: .TRUE. .FALSE.

VARIABLES

A variable is specified by its name and its type. There are five types of variables: integer, real, double-precision, complex, and logical.

Variable Names

General Form
A variable name consists of one to six alphameric characters, the first of which must be alphabetic.

Examples:

L5
JOB1
BETATS
COST
K

Subroutines are named in the same manner as variables (see "Naming Subroutines").

Variable Type Specification

The type of a real variable name or function name or an integer variable name or function name may be specified in one of two ways: implicitly by name, or explicitly by a Type statement (see the sections "Type Statements" and "Naming Subroutines"). All other variables must have their type specified by a Type statement. The type of a function name that appears in a FUNCTION statement is specified either implicitly or by that FUNCTION statement.

Implicit Type Assignment

Implicit type assignment pertains only to integer variable names and function names and real variable names and function names:

1. If the first character of the symbol is I, J, K, L, M, or N, it is an integer name; e.g., MAX, JOB, IDIST, LESL.
2. If the first character of the symbol is <u>not</u> I, J, K, L, M, or N, it is a real name; e.g., ALPHA, BMAX, Q, WHIT.

Exception: If the symbol is used as a function reference and is the same name as that of a built-in or standard library function, it is implicitly typed as shown in column 6 of Figures 8, 9, and 16.

SUBSCRIPTS

A variable may be made to represent any element of a one-, two-, ..., or seven-dimensional array of quantities by appending one, two, ..., or seven subscripts, respectively, to the variable name. The variable is then a subscripted variable. The subscripts are expressions of a special form whose value determines the member of the array to which reference is made.

Form of Subscripts

General Form

A subscript may take <u>only</u> one of the following forms, where v represents any unsigned, nonsubscripted integer variable, and c and c' represent any unsigned integer constant having a positive value:

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

Examples:

```
IMAS
J9
K2
N+3
8*IQUAN
5*L+7
4*M-3
7+2*K invalid
9+J   invalid
```

Note: The value of a subscript expression must be greater than zero and not greater than the corresponding array dimension. The value of an integer variable in a subscript expression must not be less than zero.

Subscripted Variables

General Form
A subscripted variable consists of a variable name followed by parentheses enclosing one to seven subscripts that are separated by commas.

Examples:

```
A(I)
K(3)
BETA (8*J + 2, K-2, L)
MAX (I, J, K, L, M, N)
```

1. During the execution, the subscript is evaluated so that the subscripted variable refers to a specific member of the array.
2. Each variable that appears in subscripted form must have the size of the array specified preceding the first appearance of the subscripted variable in any executable, NAMELIST, or DATA statements. This must be done by a DIMENSION statement or by a COMMON or Type statement (EXCEPT EXTERNAL) that contains dimension information.

Arrangement of Arrays in Storage

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

For example, the two-dimensional array A(m,n) is stored as follows, from the lowest core storage location to the 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.

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 following arithmetic operation symbols denote addition, subtraction, multiplication, division, and exponentiations, respectively:

+ - * / **

The following are the rules for constructing arithmetic expressions:

1. Figures 1 and 2 indicate which constants, variables, and functions may be combined by the arithmetic operators to form arithmetic expressions. Figure 1 gives the valid combinations with respect to the arithmetic operators +, -, *, and /. Figure 2 gives the valid combinations with respect to the arithmetic operator**. In these figures, Y indicates a valid combination and N indicates an invalid combination.

2. A real constant, variable, or function name combined with a double-word quantity results in an expression with the type of the double-word quantity; e.g., a real variable plus a complex variable forms a complex expression. A real quantity combined with a real quantity results in a double-precision quantity if this result is in turn combined with, or substituted for, a double-precision quantity.

3. Any expression may be enclosed in parentheses.

4. Expressions may be connected by the arithmetic operation symbols to form other expressions, provided that:

   a. No two operators appear in sequence.

   b. No operation symbol is assumed to be present. For example, (X)(Y) is invalid.

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

5. Preceding an expression by a plus or minus sign does not affect the type of the expression.

+, -, *, /	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 1

	**	Exponent: Real	Integer	Complex	Double-Precision	Logical
Base	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 2

6. In the hierarchy of operations, parentheses may be used in arithmetic expressions to specify the order in which operations are to be computed. Where parentheses are omitted, the order is understood to be as follows (from innermost operations to outermost operations):

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

(Even if operators are on the same level, parentheses may be used if a particular order of computations is required by the program.)

Logical Expressions

A logical expression consists of certain sequences of logical constants, logical variables, references to logical functions, and arithmetic expressions (except complex expressions) separated by logical operation symbols or relational operation symbols. A logical expression always has the value:

TRUE. or .FALSE.

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 a and b are both .TRUE.; it has the value .FALSE. if either a or b is .FALSE.
a. OR. b	(Inclusive OR) This has the value .TRUE. if either a or b is .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 operation symbols are:

Symbol	Definition
.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to

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

The following are the rules for constructing logical expressions:

1. Figure 3 indicates which constants, variables, functions, and arithmetic expressions may be combined by the relational operators to form a logical expression. In Figure 3, Y indicates a valid combination and N indicates an invalid combination.

.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 3

The logical expression will have the value .TRUE. if the condition expressed by the relational operator is met; otherwise, the logical expression will have the value .FALSE. .

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

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 may be enclosed in parentheses.

5. In the hierarchy of operations, parentheses may be used in logical expressions to specify the order in which operations are to be computed. Where parentheses are omitted, the order is understood to be as follows (from innermost operation to outermost operation):

a. Function Reference
b. ** Exponentiation
c. * and / Multiplication and Division
d. + and - Addition and Subtraction
e. .LT.,.LE.,.EQ.,.NE.,.GT.,.GE.
f. .NOT.
g. .AND.
h. .OR.

The arithmetic statement defines a numerical or logical calculation. A FORTRAN arithmetic statement closely resembles a conventional arithmetic formula; however, the equal sign of the FORTRAN statement specifies replacement rather than equivalence.

General Form
a=b where: 1. a is a subscripted or nonsubscripted variable, and 2. b is an expression.

Examples:

```
Q1 = K
A(I) = B(I) + ASIN(C(I))
V = .TRUE.
E = C.GT.D.AND.F.LE.G
```

Figure 4 indicates which type expressions may be equated to which type of variable in an arithmetic statement. In Figure 4, Y indicates a valid statement and N indicates an invalid statement.

Right side of equal sign

	expression / variable	Real	Integer	Complex	Double-Precision	Logical
Left side of equal sign	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 4

In the following examples of arithmetic statements, I is an integer variable, A and B are real variables, C and D are double-precision variables, E and F are complex variables, 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.
A = B*C	Multiply B by C using double-precision arithmetic, and store the most significant part of the result as a real number in A.
E = F*(3.7, 2.0)	Multiply F by 3.7 + 2.0i using complex arithmetic, and store the result in E as a complex number.
F = B	Replace the real part of F by the current value of B, and set the imaginary part of F to zero.
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	(see table below)

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.

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

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

The last two examples illustrate the following rule:

Two decimal points may appear in succession if (1) two logical operators appear in sequence (the second one must be .NOT.) or (2) a constant with a decimal point precedes a relational operator.

The control statements enable the programmer to control and terminate the flow of his program.

Unconditional GO TO Statement

General Form
GO TO n where: n is a statement number.

Example:

GO TO 25

This statement causes control to be transferred to the statement numbered n.

Computed GO TO Statement

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

Example:

GO TO (30,45,50,9),K

This statement causes control to be transferred to the statement numbered $n_1, n_2, \ldots, n_m$ depending on whether the value of i is 1, 2, 3,..., m, respectively, at the time of execution. Thus, in the example, if K is 3 at the time of execution, a transfer to the third statement in the list, i.e., statement 50, will occur.

Assigned GO TO 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, and 2. $n_1, n_2, \ldots, n_m$ are statement numbers.

Example:

GO TO J, (17,12,19)

This statement causes control to be transferred to the statement number last assigned to i by an ASSIGN statement; $n_1, n_2, \ldots, n_m$ is a list of the m values that i may assume.

ASSIGN Statement

General Form
ASSIGN n TO i where: 1. n is a statement number, and 2. i is a nonsubscripted integer variable that appears in an assigned GO TO statement.

Examples:

ASSIGN 12 TO K

ASSIGN 37 TO JA

This statement causes a subsequent GO TO i, (n_1, n_2, ..., n_m) to transfer control to the statement numbered n, where n is one of the statement numbers included in the series $n_1, n_2, \ldots, n_m$.

Arithmetic IF Statement

General Form
IF (a) n_1, n_2, n_3 where: 1. a is an arithmetic expression (not complex), and 2. n_1, n_2, n_3 are statement numbers.

Examples:

IF (A(J,K) - B) 10,4,30

IF (D*E+BRN) 9,9,15

This statement causes control to be 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.

Logical IF Statement

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

Examples:

IF (A.AND.B) F = SIN (R)

IF (16.GT.L) GO TO 24

IF (D.OR.X.LE.Y) GO TO (18,20),I

IF (Q) CALL SUB

1. If the logical expression t is true. statement s is executed. Control is then transferred to the next

sequential statement unless s is a transfer statement, in which case, control is transferred as indicated.

2. If t is false, control is transferred to the next sequential statement.

3. If t is true and s is a CALL statement that does not have any nonstandard returns, control is transferred to the next sequential statement upon return from the subprogram.

DO Statement

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, and 3. m_1, m_2, m_3 are each either an unsigned integer constant or a nonsubscripted integer variable; if m_3 is not stated, is taken to be 1.

Examples:

DO 30 I = I, M, 2

DO 24 I = 1, 10

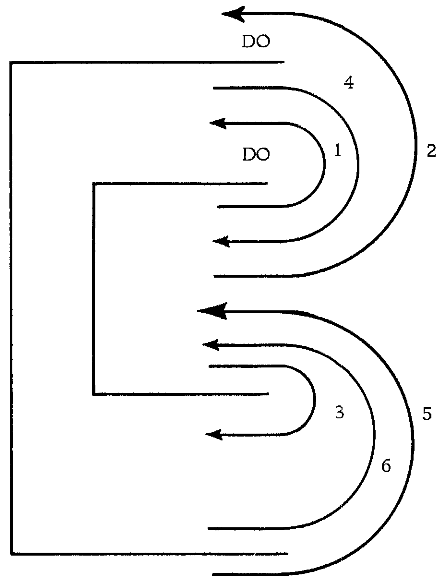
The DO statement is a command to execute repeatedly the statements that follow, up to and including the statement numbered n. The statements in the range of the DO are executed repeatedly with i equal to m_1, then i equal to $m_1 + m_3$, then i equal to $m_1 + 2m_3$, etc., until i is equal to the highest value in this sequence that does not exceed m_2. The statements in the range of the DO will be executed at least once. The value of m_1, m_2, and m_3 must be greater than zero when the DO statement is executed.

1. The range of a DO is that set of statements that will be 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.

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 satisfying 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.

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

4. Transfer of Control and DO Statements. Control may not be transferred into the range of a DO from outside its range. Thus, in the configuration following, 1, 2, and 3 are permitted transfers, but 4, 5, and 6 are not.



5. Restrictions on Statements in the Range of a DO.

a. Any statement that redefines the index or any of the indexing parameters (m's) is not permitted in the range of a DO.

b. The range of a DO cannot end with an arithmetic IF or GO TO-type statement, with a nonexecutable statement, or with a RETURN or STOP statement. The range of a DO may end with a logical IF, in which case, control is handled as follows: if the logical expression t 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 or transfer type statement, control is transferred as indicated.

6. When a reference to a subprogram is executed in the range of a DO, care must be taken that the called subprogram does not alter the DO index or the indexing parameters.

CONTINUE Statement

General Form
CONTINUE

CONTINUE is a dummy statement that gives rise to no instructions in the object program. It is most frequently used as the last statement in the range of a DO to provide a transfer address for IF and GO TO statements that are intended to begin another repetition of the DO range.

PAUSE Statement

General Form
PAUSE or PAUSE n where: n is an unsigned octal integer constant of one to five digits.

Examples:

PAUSE

PAUSE 77777

The machine will halt with the octal number n in the address field of the Storage Register. If n is not specified, it is understood to be zero. Depressing the START key causes the program to resume execution of the object program with the next executable FORTRAN statement.

END Statement

General Form
END

1. The END statement terminates compilation of a program.
2. The END statement must be the physically last statement of the program.

STOP Statement

General Form
STOP or STOP n where: n is an unsigned octal integer constant of one to five digits.

Examples:

STOP

STOP 77777

The STOP and STOP n statements terminate the execution of any program by returning control to the Monitor.

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

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. The statements PUNCH and PRINT cause information to be transmitted from core storage to the card punch and on-line printer, respectively.

Manipulative Input/Output Statements: Statements END FILE, REWIND, and BACKSPACE manipulate input/output devices.

Nonexecutable Statements: Either of two nonexecutable statements (the FORMAT statement or the NAMELIST statement) may be used with the general input/output statements. The FORMAT statement, which can be used with any general input/output statement, specifies the arrangement of data in the external input/output medium. If the FORMAT statement is referred to by a READ statement, the input data must meet the specifications described in the section "Data Input Referring to a FORMAT Statement." The NAMELIST statement specifies an input/output list of variables and arrays. Input/output of the values associated with the list is effected by reference to the list in a READ or WRITE statement. If the NAMELIST statement is referred to by a READ statement, the input data must meet the specifications described in the section "Data Input Referring to a NAMELIST Statement."

LIST SPECIFICATIONS

If arrays or variables are transmitted by using a FORMAT statement, an ordered list of the quantities to be transmitted must be included in the general input/output statement. The order of the input/output list must be the same as the order in which the information exists in the input/output medium.

The following notes on the formation and meaning of an input/output list are most clearly understood by considering the following 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)
```

This list implies that the information in the external input/output medium is arranged as follows:

```
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)
```

1. An input/output list is a string of list items separated by commas. A list item may be:
   a. A subscripted or nonsubscripted variable:
   b. An implied DO.

An input/output list reads from left to right with repetition of variables enclosed in parentheses.

2. A constant may appear in an input/output list only as a subscript or as an indexing parameter.

3. The execution of an input/output list 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 input/output list above may be considered equivalent to the following program statements:

```
      A
      B(3)
      DO 5 I=1,10      }
      C(I)             }   (C(I),D(I,K),I=1,10)
  5   D(I,K)           }
      DO 9 J=1,K       }
      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)           }
```

4. An implied DO is best defined by an example. In the input/output list above, the list item (C(I), D(I,K), I=1, 10) is an implied DO; it is evaluated as in the above program.

The range of an implied DO must be clearly defined by parentheses.

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

6. Any number of quantites may appear in a single list. However, each quantity must have the correct format as specified in a corresponding FORMAT statement. Essentially, it is the list that controls the quantity of data read. If more quantities are to be transmitted than are in the list, only the number of quantities specified in the list are transmitted, and remaining quantities are ignored. Conversely, if a list contains more quantities than are given on one BCD input record, more records are read; if a list contains more quantities than are given in one binary record, reading is terminated as an object program error and control is transferred to FXEM (an error routine in the Subroutine Library).

INPUT/OUTPUT OF ENTIRE ARRAYS

By referring to the NAMELIST statement, an entire array can be designated for transmission between core storage and an input/output medium. Input of an entire array using the NAMELIST statement is described in the section "Data Input Referring to NAMELIST Statement"; output of an entire array using the NAMELIST statement is described in the section "Output." If the FORMAT statement is referred to and input/output of an entire array is desired, an abbreviated notation may be used in the list of the general input/output statement. Only the name of the array need be given and the indexing information may be omitted.

1. If A has previously been listed in a statement containing dimension information, the following statement is sufficient to read in all of the elements of the array A (see the section "Input"):

   READ (5,10)A

2. The elements read in by this notation are stored in accordance with the description of the arrangement of arrays in storage (see the section "Arrangement of Arrays in Storage").

3. If A has not been previously dimensioned, only the first element will be read in.

FORMAT STATEMENT

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 $(S_1, S_2, \ldots, S_n/S'_1, S'_2, \ldots, S'_n/\ldots)$ where: each field, S_i, is a format specification.

Example:

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

1. FORMAT statements are not executed; they may 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 in conjunction with the list of some particular input/output statement, except when a FORMAT statement consists entirely of alphameric 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 gives the specifications for transmission of that data.

3. Slashes are used to specify unit records, which must be one of the following:

   a. A tape record with a maximum length corresponding to the printed line of the off-line printer.

   b. A punched card, to be read on-line, with a maximum of 72 characters; a punched card, to be read off-line, with a maximum of 80 characters.

   c. A line to be printed on-line, with a maximum of 120 characters.

Thus, FORMAT (3F9.2, 2F10.4/8E14.5) would specify records in which the first, third, fifth, etc., have the format (3F9.2, 2F10.4), and the second, fourth, sixth, etc., have the format (8E14.5).

4. During input/output of data, the object program scans the FORMAT statement to which the relevant input/output statement refers. When a specification for a numerical field is found and list items remain to be transmitted, input/output takes place according to the specification, and scanning of the FORMAT statement resumes. If no items remain, transmission ceases and execution of that particular input/output statement is terminated. Thus, a decimal input/output operation is brought to an end when there are no items remaining in the list.

Numeric Fields

Five types of conversion are available for numeric data:

Internal	Conversion Code	External
Floating point (double-precision)	D	Real with D exponent
Floating point	E	Real with E exponent
Floating point	F	Real without exponent
Integer	I	Decimal Integer
Integer	O	Octal Integer

These types of conversion are specified in the forms Dw.d, Ew.d, Fw.d, Iw, Ow, where:

1. D,E,F,I, and O represent the type of conversion.

2. 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.

3. d is an unsigned integer or zero that represents the number of positions of the field that appear

to the right of the decimal point. For E- and F-conversion, d is calculated modulo 10.

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

```
I2E12.4        O8        F10.4        D25.16

27b-0.9321Eb0257734276bbb-0.0076bb-0.7878977909500672Db03
```

where b indicates a blank space.

The following are notes on D-, E-, F-, I-, and O-conversion.

1. Specifications for successive fields are separated by commas and/or slashes. (See the section "Multiple-Record Formats.")

2. No format specification should be given that provides for more characters than permitted for a relevant input/output record. Thus, a format for a BCD record to be printed off-line should not provide for more characters (including blanks) than the capabilities of the printer.

3. Information to be transmitted with O-conversion may be real or integer names; information to be transmitted with E- and F-conversion must have real names; information to be transmitted with I-conversion must have integer names; information to be transmitted with D-conversion must have double-precision names.

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

5. The exponent, which may be used with D- and 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 (for E-conversion) or D (for D-conversion) 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 two numbers that are the exponent. For example, the number .002 is equivalent to the number .2E-02.

6. For input under D-conversion, up to 17 decimal digits are converted and the result is stored so that the most significant part and the least significant part are in adjacent core storage locations.

For output under D-conversion, the two core storage words representing the double-precision quantity are considered one piece of data and converted as such.

E- or F-conversion may be used for floating point numbers whose absolute value is less than 2^{27}. E-conversion must be used for numbers whose absolute value is greater than or equal to 2^{27}.

7. If a number converted by I-conversion requires more spaces than are allowed by the field width w, the excess on the high-order side is lost. 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 will contain a minus sign if sufficient spaces have been reserved.

Note: If the optional 7094 library conversion routine is used, output will differ from the standard described above.

If an output number that is converted by D-, E-, F-, or I-conversion requires more spaces than are allowed by the field width w, the number is disregarded and the field is filled with asterisks. If the number requires fewer than w spaces, the leftmost spaces are filled with blanks.

Complex Number Fields

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

Figure 5 is an example of a FORMAT statement that transmits an array consisting of six complex numbers.

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

Figure 5

Alphameric Fields

FORTRAN provides two ways by which alphameric information may be transmitted; both specifications result in storing the alphameric information internally in BCD.

1. The specification Aw causes w characters to be read into, or written from, a variable or array name.

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 array name that can be referred to 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; it may be real or integer.

1. On input, nAw will be interpreted to mean 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 will be significant. If w is less than 6, the characters will be left-adjusted, and the word filled out with blanks.

2. On output, nAw will be interpreted to mean that the next n successive fields of w characters each are to be the result of transmission from storage without conversion. If w exceeds 6, only 6 characters of output will be transmitted, preceded by w-6 blanks. If w is less than 6, the w leftmost characters of the word will be transmitted.

H-Conversion

The specification nH is followed in the FORMAT statement by n alphameric characters. For example:
31HbTHISbISbALPHAMERICbINFORMATION
Note that 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 output.

1. On input, n characters are extracted from the input record and replace the n characters included with the source program FORMAT specification.

2. On output, the n characters following the specification, or the characters that replaced them, are written as part of the output record.

Figure 6 is an example of A- and H-conversion in a FORMAT statement.

The statement FORMAT (4HbXY=, F8.3, A8) might produce the following lines, where b indicates a blank character:

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

Figure 6

Figure 6 assumes that there are steps in the source program that read the data OVFLOW, store this data in the word to be printed 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, where L represents the logical type of conversion and w is an integer constant that represents the data field width.

1. On input, a value of either true or false will be stored if the first nonblank character in the field of w characters is a T or an F, respectively. If all the w characters are blank, a value of false will be stored.

2. On output, a value of true or false in storage will cause w minus 1 blanks, followed by a T or an F, respectively, to be written out.

Blank Fields -- X-Conversion

The specification nX introduces n blank characters into an input/output record where $0 < n \leq 132$.

1. On input, nX causes n characters in the input record to be skipped, regardless of what they are.

2. On output, nX causes n blanks to be introduced into the output record.

Repetition of Field Format

It may be desired to print or read n successive fields in the same format within one record. This may be specified by giving n, an unsigned integer, before D, E, F, I, L, O, or A. Thus, the field specification 3E12.4 is the same as writing 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. Thus, FORMAT (2(F10.6, E10.2), I4) is equivalent to FORMAT (F10.6, E10.2, F10.6, E10.2, I4). (See the section "Multiple-Record Formats.") 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.

Scale Factors

To permit more general use of D-, E-, and F-conversion, a scale factor followed by the letter P may precede the specification. The magnitude of the scale factor must be between -8 and +8, inclusive. The scale factor is defined for input as follows:

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

The scale factor is defined for output as follows:

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

For input, scale factors have effect only on 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 may be used with D-, E-, and F-conversion.

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

27bbbb-93.209bbbbb-0.008bbbbbb0.554

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

27bbb-932.094bbbbb-0.076bbbbbb5.536.

Whereas, the statement FORMAT (I2, -1P3F11.3) would give the following 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 following 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 will hold 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. (See the next section.) Once the scale factor has been given, a subsequent scale factor of zero in the same FORMAT statement must be specified by 0P. For F-type conversion, output may not include numbers whose absolute value is greater than or equal to 2^{27} after scaling. Scale factors have no effect on I- and O-conversion.

Multiple-Record Formats

To deal with a block of more than one line of print, a FORMAT specification may have several different one-line formats separated by a slash (/) to indicate the beginning of a new blank line. Thus, FORMAT (3F9.2,2F10.4/8E14.5) would specify a multiline block of print in which lines 1, 3, 5, ... have format (3F9.2,2F10.4), and lines 2, 4, 6, ... have format (8E14.5).

If a multiple-line format is desired in which the first two lines are to be printed according to a special format and all remaining lines according to another format, the last line-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 previous parenthesis, which is a zero or a first level parenthesis. For example, consider the FORMAT statement:

```
FORMAT (3E10.3, (I2,2 (F12.4, F10.3)), D28.17)
       0        1   2              21        0
```

The parentheses labeled 0 are 0 level parentheses; those labeled 1 are first level parentheses; and, those labeled 2 are second level parentheses. If more items in the list are to be transmitted after the format statement has been completely used, the FORMAT repeats from the last first-level left parenthesis; i.e., the parenthesis preceding I2.

As these examples show, both the slash and the final right parenthesis of the FORMAT statement indicate a termination of a record.

Blank lines may be introduced into a multiline FORMAT statement by listing consecutive slashes. When n + 1 consecutive slashes appear at the end of the FORMAT, they are treated as follows: for input, n + 1 records are skipped; for output, n blank lines are written. When n + 1 consecutive slashes appear in the middle of the FORMAT, n records will be skipped for both input and output.

Carriage Control

The WRITE (i,n) list statement prepares a BCD tape that can be used to obtain off-line printed output. The PRINT n, list statement prints on-line during execution. The off-line printer may be set manually to operate in one of three modes: single space, double space, or program control. Under program control, the first character of each BCD record, the control character, controls spacing of the printer; this first character is not printed. Control characters that produce standard effects, both on-line and off-line, are:

Character	Effect	Off-line	On-line
Blank	Single space	Before printing	After printing
0	Double space	Before printing	After printing
1	Eject	Before printing	Before printing followed by a single space after printing

Program control is usually obtained by beginning a FORMAT specification, for a BCD record, with 1H followed by the desired control character.

FORMAT Statements Read In at Object Time

FORTRAN accepts a variable FORMAT address. This permits specifying a FORMAT for an input/output list at object time.

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

Figure 7

In Figure 7, A,B, and the array C are converted and stored according to the FORMAT specifications read into the array FMT at object time.

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

2. 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.

Data Input Referring to a FORMAT Statement

Data input to the object program is punched into cards according to the following specifications:

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

2. Plus signs may be omitted or indicated by a +. Minus signs are indicated by an 11-punch.

3. A blank in a numeric field is treated as a zero. A numeric field containing all blanks is converted to minus zero.

4. Numbers for E- and F-conversion may contain any number of digits, but only the high-order 8 digits of precision will be retained. For D-conversion, the high-order 16 digits of precision will be retained. In both cases, the number is rounded to 8 or 16 digits of accuracy, as applicable.

5. For input, numeric data must be situated at the extreme right of its field (right justified).

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 devoted to the exponent field. The start of the exponent field must be marked by a D or an E or, if that is omitted, by a plus or minus sign (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; the format specification will supply it. For example, the number -09321+2 with the specification E12.4 will be treated as though the decimal point had been punched between the 0 and the 9. If the decimal point is punched in the card, its position overrides the position indicated in the FORMAT specification.

NAMELIST STATEMENT

The NAMELIST statement and modified forms of the READ and WRITE statements provide for reading, writing, and converting data without the use of an input/output list in the input/output statement and without a reference to a FORMAT statement.

General Form
NAMELIST /X/A, B, ... C/Y/D, E, ... , F/Z/G, H, ... , I where: X, Y, Z, ... are NAMELIST names, and A, B, C, D, ... are variable or array names.

Examples:

DIMENSION A(10), I(5,5), L(10)

NAME LIST /NAM1/ A,B,I,J,L/NAM2/A,C,J,K

In the preceding examples, the arrays A, I, and L and the variables B and J belong to the NAMELIST name, NAM1, and the array A and the variables C, J, and K belong to the NAMELIST name, NAM2.

Each list that is mentioned in the NAMELIST statement is given a NAMELIST name. Only the NAMELIST name is needed in an input/output statement to refer to that list thereafter in the program. The following rules apply to assigning and using a NAMELIST name:

1. A NAMELIST name consists of one to six alphameric characters; the first character must be alphabetic.

2. A NAMELIST name is enclosed in slashes. The field of entries belonging to a NAMELIST name ends either with a new NAMELIST name enclosed in slashes or with the end of the NAMELIST statement.

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

4. A NAMELIST name must not be the same as any other name in the program.

5. A NAMELIST name may be defined only once by its appearance in a NAMELIST statement. After it has been defined in the NAMELIST statement, the NAMELIST name may appear only in READ or WRITE statements thereafter in the program.

6. A NAMELIST statement defining a NAMELIST name must precede any appearance of the name in the program.

7. A dummy argument which appears in a FUNCTION, SUBROUTINE, or ENTRY statement cannot be used as a variable in a NAMELIST statement.

8. If a NAMELIST statement contains a dimensioned variable, the statement that contains the dimension information defining the variable must precede the NAMELIST statement.

Data Input Referring to a NAMELIST Statement

When a READ statement refers to a NAMELIST name, the designated input device is prepared and input of data is begun. The first character on all input data records is always ignored. The first input data record is searched for a $ as the second character, immediately followed by the NAMELIST name, immediately followed by one or more blank characters. If the search fails, additional records are examined consecutively until there is a successful match. When a successful match is made of the NAMELIST name on a data record and the NAMELIST name referred to in a READ statement, data items are converted and placed in storage.

Any combination of three types of data items, described in the following text, may be used in a data record. The data items must be separated by commas; however, use of a comma following the last item is optional. If more than one record is needed for input data, the last item of each record, except the last, must be a constant followed by a comma. The end of a group of data is signaled by a $ either in the same data record as the NAMELIST name or anywhere in any succeeding records except in the first character position.

Serialization should not be used on data cards since all items on a data card may be scanned.

The form that data items may take is:

1. Variable name = constant

where variable name may be an array element name or a simple variable name. Subscripts must be integer constants.

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

where k*constant may be included to represent k constants (k must be an unsigned interger). The number of constants must be equal to the number of elements in the array.

3. Subscripted variable = set of constants (separated by commas)

where k*constant may be included to represent k constants (k must be an unsigned integer). A data item of this form results in the set of constants being placed in consecutive array elements, starting with the element designated by the subscripted variable. The number of constants given cannot exceed the number of elements in the array that are included between the given element and the last element in the array, inclusive.

Constants used in the data items may take any of the following forms:

a. integers
b. real numbers
c. double-precision numbers
d. complex numbers, which must be written in the usual form, (C1, C2), where C1 and C2 are real numbers
e. logical constants, which must be written as T or .TRUE., and F or .FALSE.

Logical and complex constants may be associated only with logical and complex variables, respectively. The other types of constants may be associated with integer, real, or double-precision variables and are converted in accordance with the type of variable. Blanks must not be embedded in a constant or repeat constant field, but may be used freely elsewhere within a data record.

Any selected set of variable or array names belonging to the NAMELIST name that is referred to by the READ statement may be used as specified in the preceding description of data items. Names that are made equivalent to these names may not be used unless they also belong to the NAMELIST name.

Example:

```
                   Col
                   2
First Data Card    $NAM1 I(2,3) =5, J=4.2, B=4,
Second Data Card   A(3) = 7, 6.4, L = 2,3,8*4.3$
```

If this data is input to be used with the NAMELIST statement previously illustrated and with a READ statement, the following actions take place. The input unit designated in the READ statement is prepared and the first record is read. The record is searched for a $ in column 2, immediately followed by the NAMELIST name, NAM1. Since the search is successful, data items are converted and placed in core storage.

The integer constant 5 is placed in I(2,3), the real constant 4.2 is converted to an integer and placed in J, and the integer constant 4 is converted to real and placed in B. Since no data items remain in the record, the next input record is read. The integer constant 7 is converted to real and placed in A(3), and the real constant 6.4 is placed in the next consecutive location of the array, A(4). Since L is an array name not followed by a subscript, the entire array is filled with the succeeding constants. Therefore, the integer constants 2 and 3 are placed in L(1) and L(2), respectively, and the real constant 4.3 is converted to an integer and placed in L(3), L(4), ..., L(10). The $ signals termination of the input for the READ operation.

THE GENERAL INPUT/OUTPUT STATEMENTS

Input

The READ statement designates input. The following table gives the forms of the READ statement, where i, an unsigned interger constant or an unsubscripted integer variable, is a reference to an input device, n is a FORMAT statement number, and x is a NAMELIST name.

Type of Input	General Form
Cards on-line	READ n, list
BCD record	READ (i, n) list
Binary record	READ (i) list
BCD records	READ (i, x)

Examples:

```
READ 10, (A(I), I=1,5)
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)
READ (5,NAM1)
```

1. The READ n, list statement causes cards to be read from the card reader.

2. The READ (i,n) list statement causes BCD information to be read from symbolic input device i (except the card reader).

3. The READ (i) list statement causes binary information to be read from symbolic input device i (except the card reader).

4. The READ (i, x) statement causes BCD information relating to variables and arrays associated with the NAMELIST name x to be read from symbolic input device i (except the card reader).

5. Under the first two forms of the READ statement, 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.

Binary conversion of input numbers is identical, whether the numbers are compiled into the program, appear in a DATA statement, or are read in at object time.

For information on the binary record format used with the READ (i) list statement, see Appendix E.

Output

The PRINT and PUNCH statements designate on-line printing and punching of data and require both a reference to a FORMAT statement and an output list as part of the statement. All other output is designated by a WRITE statement, which can refer to either a FORMAT statement or a NAMELIST statement.

The following table gives the forms of the output statement, where i, an unsigned integer constant or an integer variable, is a reference to an output device, n is a FORMAT statement number, and x is a NAMELIST name.

Type of Output	General Form
Cards on-line	PUNCH n, list
Print on-line	PRINT n, list
BCD Record	WRITE (i, n) list
BCD Record	WRITE (i, x)
Binary Record	WRITE (i) list

Examples:

```
PUNCH 20, (A(J), J=1,6)
PRINT 2, (A(J), J=1,6)
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
WRITE (6,NAM1)
```

1. The PUNCH n, list statement causes alphameric cards to be punched on-line.

2. The PRINT n, list statement causes data to be output on the on-line printer.

3. The WRITE (i,n) list statement causes BCD information to be written on symbolic output device i according to the format specified in statement n.

4. The WRITE (i,x) statement causes all variable and array names (as well as their values) that belong to NAMELIST name x to be written on symbolic output device i.

5. The WRITE (i) list statement causes binary information to be written on symbolic output device i.

The PUNCH, PRINT, and WRITE (i,n) statements cause successive records to be written in accordance with the FORMAT statement until the list has been satisfied. The WRITE (i) list statement causes the writing of one logical record consisting of all the words specified in the list.

When a WRITE statement refers to a NAMELIST name, the values and names of all variables and arrays belonging to the NAMELIST name are written, each according to its type. A complete array is written out by columns. The output data is written such that:

1. The fields for the data are large enough to contain all the significant digits.

2. The output can be read by an input statement referring to the NAMELIST name.

THE MANIPULATIVE INPUT/OUTPUT STATEMENTS

The statements END FILE, REWIND, and BACKSPACE manipulate input/output devices. In the following table, i, an unsigned integer constant or integer variable, is a reference to a symbolic input/output device.

General Form
END FILE i REWIND i BACKSPACE i

Examples:

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

1. The END FILE i statement causes an end-of-file mark to be written on symbolic tape i.

2. The REWIND i statement causes symbolic tape unit i to be rewound.

3. A request to write an end of file or rewind the following system files will be ignored: SYSIN1,

SYSOU1, and SYSPP1, corresponding in the standard FORTRAN I/O package to symbolic units 5, 6, and 7.

4. The BACKSPACE i statement causes tape i to be backspaced one physical record if i refers to an input/output device in the BCD mode, or it causes tape i to be backspaced one logical record if i refers to an input/output device in the binary mode. If the BACKSPACE statement is preceded by an ENDFILE statement, the end-of-file mark is ignored and execution of the BACKSPACE proceeds as above. However, if two consecutive ENDFILE statements precede the BACKSPACE, execution will differ depending upon which IOCS package was used. These can include the standard FORTRAN input/output package (FIOCS) or the FORTRAN IV alternate input/output package (ALTIO). (See "Symbolic Input/Output Designation" below.) Under FIOCS the tape will be positioned immediately before the first end-of-file mark and not before the record preceding it. Under ALTIO the tape will always be positioned before the last physical record (for BCD files) or logical record (for binary files) of the preceding data files, regardless of the number of consecutive ENDFILE statements that precede the BACKSPACE statement. A request to backspace SYSOU1 corresponding in the standard FORTRAN input/output library to symbolic unit 6 will be ignored.

5. If a REWIND statement is preceded by either a BACKSPACE or an ENDFILE statement, the tape redundancy history message will always treat the tape as an input file (see the publication IBM 7090/7094 IBSYS Operating System, Version 13: Input/Output Control System, Form C28-6345). The tape redundancy history message is printed for a rewound tape that contained either retries while reading or erases while writing, when neither FIOCS nor ALTIO was specified or assumed on the $IBJOB card (see appendix G).

SYMBOLIC INPUT/OUTPUT DESIGNATION

Input/output devices are always referred to symbolically in FORTRAN input/output statements.

1. Object program input/output operates through either the standard FORTRAN input/output package (FIOCS) or the FORTRAN IV Alternate Input/Output Package (ALTIO). The correspondence between the symbolic unit reference and the actual physical unit is established in the initialization of IOCS.

2. Both FIOCS and ALTIO allow for symbolic tape units 1 through 8. The normal unit designation for BCD input statements is 5; for BCD output statements it is 6.

The symbolic unit references may be changed by each installation in accordance with its own needs. See the publications IBM 7090/7094 IBSYS Operating System, System Monitor (IBSYS), Form C28-6248, and IBM 7090/7094 IBSYS Operating System, IBJOB Processor, Form C28-6389.

FORTRAN Tape Units	Mode		Function
	FIOCS	ALTIO	
1	Binary	Mixed	Input or Output
2	Binary	Mixed	Input or Output
3	Binary	Mixed	Input or Output
4	Binary	Mixed	Input or Output
5	BCD	Mixed	Input
6	BCD	Mixed	Output
7	Binary	Mixed	Output
8	BCD	Mixed	Input or Output

There are four classes of subroutines in FORTRAN: arithmetic statement functions, built-in functions, FUNCTION subprograms, and SUBROUTINE subprograms. The major differences among the four classes of subroutines are as follows:

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 only a single result); the SUBROUTINE subprogram may return more than one value.
   b. A function is referred to by an arithmetic expression containing its name; a SUBROUTINE subprogram is referred to by a CALL statement.
2. The 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

In the following text, the terms calling program and called program are used. The calling program is the program in which a subroutine is referred to or called. The called program is the subroutine that is referred to or called by the calling program.

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 one to six 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 statement function may be indicated by the name (if it is real or integer) 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 (REAL, INTEGER, COMPLEX, DOUBLE PRECISION, LOGICAL) preceding the word FUNCTION. In the latter case, the type, implied by name is overridden. The type of a reference to a FUNCTION subprogram in the Subroutine Library (the mathematics subroutines) is automatically defined as shown in column 6 of Figure 9. Therefore, the subprogram need not be typed in the calling program.
   c. The type of a built-in function is indicated within the FORTRAN Processor and need not appear in a Type statement (see column 6 of Figure 8).
3. The name of a SUBROUTINE subprogram has no type and should 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 subroutine is discussed below.

Arithmetic Statement Functions

Arithmetic statement functions are defined by a single arithmetic statement and apply only to the source program containing the definition.

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 previously defined.

Examples:

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

1. 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 appearing elsewhere in the program of the same type.
2. Those variables included in b that are not stated as arguments are the parameters of the function. They are ordinary variables.
3. All arithmetic statement function definitions must precede the first executable statement of the source program.
4. The type of any arithmetic statement function name or argument that differs from its implicit type must be defined preceding its use in the arithmetic statement function definition.

Function	Definition	Number of Arguments	Name	Type of Argument	Type of Function
Absolute value	|Arg|	1	ABS IABS	Real Integer	Real Integer
Truncation	Sign of Arg times largest integer ≤ |Arg|	1	AINT INT	Real Real	Real Integer
Remaindering (see note below)	Arg_1 (mod Arg_2)	2	AMOD MOD	Real Integer	Real Integer
Choosing largest value	Max(Arg_1, Arg_2, ...)	≥ 2	AMAX0 AMAX1 MAX0 MAX1	Integer Real Integer Real	Real Real Integer Integer
Choosing smallest value	Min(Arg_1, Arg_2, ...)	≥2	AMIN0 AMIN1 MIN0 MIN1	Integer Real Integer Real	Real Real Integer Integer
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conv fm real to integer with truncation	1	IFIX	Real	Integer
Transfer of sign	Sign of Arg_2 times |Arg_1|	2	SIGN ISIGN	Real Integer	Real Integer
Positive difference	Arg_1 - Min (Arg_1, Arg_2)	2	DIM IDIM	Real Integer	Real Integer
Obtain most significant part of double-precision argument		1	SNGL	Double	Real
Obtain real part of complex argument		1	REAL	Complex	Real
Obtain imaginary part of complex argument		1	AIMAG	Complex	Real
Absolute value Truncation	|Arg| Sign of Arg times largest integer ≤ |Arg|	1 1	DABS IDINT	Double Double	Double Integer
Choosing largest value	Max (Arg_1, Arg_2, ...)	≥2	DMAX1	Double	Double
Choosing smallest value	Min (Arg_1, Arg_2, ...)	≥2	DMIN1	Double	Double
Transfer of sign	Sign of Arg_2 times |Arg_1|	2	DSIGN	Double	Double
Express single-precision argument in double-precision form	D=(Arg, 0)	1	DBLE	Real	Double
Express two real arguments in complex form	C=Arg_1+iArg_2	2	CMPLX	Real	Complex
Obtain conjugate of a complex argument	For Arg=X+iY, C=X-iY	1	CONJG	Complex	Complex

Note: The function MOD (Arg_1, Arg_2) is defined as Arg_1 - [Arg_1/Arg_2] Arg_2, where [Arg_1/Arg_2] is the truncated value of that quotient.

Figure 8. Built-In Functions

Built-In Functions

Built-in functions are pre-defined, open subroutines that exist within the FORTRAN Processor. A list of all the available built-in functions is given in Figure 8. Appendix H lists the machine-dependent built-in functions.

FUNCTION Subprogram

FUNCTION subprograms are defined by a special FORTRAN source language program.

General Form
FUNCTION name ($a_1, a_2, \ldots, a_n$) REAL FUNCTION name ($a_1, a_2, \ldots, a_n$) INTEGER FUNCTION name ($a_1, a_2, \ldots, a_n$) DOUBLE PRECISION FUNCTION name ($a_1, a_2, \ldots, a_n$) COMPLEX FUNCTION name ($a_1, a_2, \ldots, a_n$) LOGICAL FUNCTION name ($a_1, a_2, \ldots, a_n$) where: 1. name is the symbolic name of a single-valued function, 2. the arguments $a_1, a_2, \ldots, a_n$, of which there must be at least one, are nonsubscripted variable names or the dummy name of a SUBROUTINE or FUNCTION subprogram, and 3. the type of the function may be explicitly stated preceding the word FUNCTION.

Examples:

```
FUNCTION ARCSIN(RADIAN)
REAL FUNCTION ROOT (A, B, C)
INTEGER FUNCTION CONST(ING, SG)
DOUBLE PRECISION FUNCTION DBLPRE(R, S, T)
COMPLEX FUNCTION CCOT(ABI)
LOGICAL FUNCTION IFTRU (D, E, F)
```

1. The FUNCTION statement must be the first statement of a FUNCTION subprogram.

2. The name of the function must appear at least once as a variable on the left side of an arithmetic statement or in an input statement. This name cannot be used in a NAMELIST statement.
For example:

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

By this means the output value of the function is returned to the calling program.

3. The arguments may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. The actual arguments must correspond in number, order, and type with the dummy arguments.

4. When a dummy argument is an array name, a statement with dimension information must appear in the FUNCTION subprogram; also, the corresponding actual argument must be a dimensioned array name.

5. None of the dummy arguments may appear in an EQUIVALENCE statement in the FUNCTION subprogram.

6. The FUNCTION subprogram must be logically terminated by a RETURN statement (see the section "Normal Returns from Subprograms").

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

8. The actual arguments of a FUNCTION subprogram may be any of the following:
   a. Any type of constant.
   b. Any type of subscripted or nonsubscripted variable.
   c. An arithmetic or a logical expression.
   d. The name of a FUNCTION or SUBROUTINE subprogram.

9. A FUNCTION subprogram is referred to by using its name as an operand in an arithmetic expression.

10. If the type of a FUNCTION subprogram is other than its implicit type, the type must be defined in the calling program as well as in the called FUNCTION subprogram.

11. If the name of a FUNCTION is the same as one of the built-in functions listed in Figures 8 and 16, it must be explicitly typed as EXTERNAL in the calling program in order to prevent use of the built-in function.

Those FUNCTION subprograms that are supplied with FORTRAN are given in Figure 9.

SUBROUTINE Subprogram

SUBROUTINE subprograms are defined by a special FORTRAN source language program.

General Form
SUBROUTINE name (a_1, $a_2, \ldots, a_n$) or SUBROUTINE name where: 1. name is the symbolic name of a subprogram; and 2. each argument, a, if any, is a nonsubscripted variable name or the dummy name of a SUBROUTINE or FUNCTION subprogram.

3. 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, which refers to the SUBROUTINE subprogram. The actual arguments must correspond in number, order, and type with the dummy arguments.

4. When a dummy argument is an array name, a statement containing dimension information must appear in the SUBROUTINE subprogram; also, the corresponding actual argument in the CALL statement be a dimensioned array name.

5. None of the dummy arguments may appear in an EQUIVALENCE statement in the SUBROUTINE subprogram.

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

7. The SUBROUTINE subprogram may contain any FORTRAN statements except FUNCTION, another SUBROUTINE statement, or BLOCK DATA.

Normal Returns from Subprograms

The normal exit from any subprogram is the RETURN statement, which returns control to the calling program. The RETURN statement is the logical end of the program; there may be any number of RETURN statements in the program.

General Form
RETURN

Nonstandard Returns from SUBROUTINE Subprograms

The normal sequence of execution following the RETURN statement of a SUBROUTINE subprogram is to the next executable statement following the CALL statement in the calling program. It is also possible to return to any numbered executable statement in the calling program by using a special return from the called subprogram. This return may not violate the transfer rules for DO loops.

The following text describes the form of the FORTRAN statements that is required to return from the subroutine to a statement other than the next executable statement following the CALL.

The general form of the CALL statement in the calling program is:

General Form

CALL subr $(a_1, a_2, a_3, \ldots, a_n)$

where:

1. subr is the name of the SUBROUTINE subprogram being called, and
2. a_i is a dummy argument of the form described in the section "CALL Statement," or is of the form:

   $n

   where n is a statement number $ is the character $.

The general form of the SUBROUTINE statement in the called program is:

General Form

SUBROUTINE subr $(a_1, a_2, a_3, \ldots, a_n)$

where:

1. subr is the name of the subprogram, and
2. a_i is a dummy argument of the form described in the section "SUBROUTINE Subprogram," or is of the form:

   *

   where * is the character asterisk (*) and denotes a nonstandard return.

The general form of the RETURN statement in the called program is:

General Form

RETURN i

where:

i is an integer constant or variable which denotes the ith nonstandard return in the argument list, reading from left to right.

Example:

```
Calling Program                  Called Program
        .                        SUBROUTINE SUB(X, Y, Z, *, *)
        .                                .
        .                                .
10  CALL SUB (A, B, C, $30, $40)         .
20      ---                      100  IF (R) 200, 300, 400
        .                        200  RETURN
        .                        300  RETURN 1
        .                        400  RETURN 2
30      ---                               END
        .
        .
        .
40      ---
        .
        .
        .
        END
```

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

Nonstandard returns may be best understood by considering that a CALL statement that uses the nonstandard return is equivalent to a CALL and a computed GO TO statement in sequence. For example,

CALL NAME (P, $20, Q, $35, R, $22)

is equivalent to

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

where I is set to the value of the integer in the RETURN statement executed in the called subprogram. If the RETURN is blank or zero, a normal (rather than nonstandard) return is made to the statement immediately following the GO TO.

Similarly, the arguments in the associated SUBROUTINE statement correspond to the arguments in the CALL statement as follows:

SUBROUTINE NAME (S, *, T, *, U, *)

Multiple Entry Points Into a Subprogram

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

It is also possible to enter a subprogram by a CALL statement or a function reference that refers to an ENTRY statement in the subprogram. Entry is made at the first executable statement following the ENTRY statement.

ENTRY statements are nonexecutable and, therefore, do not affect control sequencing during normal execution of a subprogram. The order, type, and number of arguments need not agree between the SUBROUTINE or FUNCTION statement and the ENTRY statements, nor do the ENTRY statements have to agree among themselves in these respects. Each CALL or function reference, however, must agree in order, type, and number with the SUBROUTINE, FUNCTION, or ENTRY statement that it refers to. No subprogram may refer to itself directly or through any of its entry points, nor may it refer to any other subprogram whose RETURN statement has not been satisfied.

The general form of the ENTRY statement in the called subprogram is:

General Form
ENTRY name ($b_1, b_2, \ldots b_n$) where: 1. name is the symbolic name of an entry point, and 2. each b_i is a dummy argument corresponding to an actual argument in a CALL statement or in a function reference.

Example:

```
Calling Program                Called Program
      .                          SUBROUTINE SUB1(U, V, W, X, Y, Z)
      .                                  .
      .                                  .
1 CALL SUB1 (A, B, C, D, E, F)           .
      .                        10 U = V
      .                                  .
      .                                  .
2 CALL SUB2 (G, H, P)                    .
      .                          ENTRY SUB2 (T, U, V)
      .                          GO TO 10
      .                                  .
3 CALL SUB3                              .
      .                                  .
      .                          ENTRY SUB3
      .                                  .
    END                                  .
                                         .
                                        END
```

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

Additional Rules for Entry Points

The following rules also apply to entry points:

1. If an adjustable array name or any of its adjustable dimensions appears in an argument list for a FUNCTION, SUBROUTINE, or ENTRY statement, that array name and all its adjustable dimensions must appear in that argument list.

2. A dummy argument may not appear in any statement unless it previously appeared in an argument list of a FUNCTION, SUBROUTINE, or ENTRY statement.

3. In a FUNCTION subprogram, only the FUNCTION name may be used as the variable to carry a result back to the calling program. The ENTRY name may not be used for this purpose.

4. An ENTRY name may appear in an EXTERNAL statement in the same manner as a FUNCTION or SUBROUTINE name.

5. Entry into a subprogram initializes all references in the entire called subprogram from items in the argument list of the CALL or function reference. (For instance, if, in the example that appeared in the section "Multiple Entry Points Into a Subprogram," entry is made at SUB2, the variables in statement 10 will refer to the argument list of SUB2.)

6. ENTRY statements may appear only in subprograms.

7. The appearance of an ENTRY statement does not alter the rules regarding the placement of arithmetic statement functions in subroutines. Arithmetic statement functions may follow an ENTRY statement only if they precede the first executable statement following the SUBROUTINE or FUNCTION statement.

Subprogram Names as Arguments

FUNCTION and SUBROUTINE subprogram names may be the actual arguments of subprograms. To distinguish these subprogram names from ordinary variables when they appear in an argument list, they must appear in an EXTERNAL statement.

```
EXTERNAL SIN
CALL SUBR (A, SIN, B)
```

CALL STATEMENT

The CALL statement is used to refer to a SUBROUTINE Subprogram.

General Form
CALL Subr $(a_1, a_2, \ldots, a_n)$ where: 1. Subr is the name of a SUBROUTINE subprogram, and 2. $a_1, a_2, \ldots, a_n$ are the n arguments.

Examples:

```
CALL MATMPY (X, 5, 10, Y, 7, 2)
CALL QDRTIC (9. 732, Q/4. 536, R-S**2. 0, X1, X2)
CALL OUTPUT
```

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

1. Any type of constant.
2. Any type of subscripted or nonsubscripted variable.
3. An arithmetic or a logical expression.
4. Alphameric characters. Such arguments must be preceded by nH where n is the count of characters included in the argument, e. g., 9HEND POINT. Note that blank spaces and special characters are considered in the character count when used in alphameric fields.
5. The name of a FUNCTION or SUBROUTINE subprogram.

The arguments presented by the CALL statement must agree in number, order, type, and array size (except as explained under the DIMENSION statement) with the corresponding arguments in the SUBROUTINE or ENTRY statement of the called subprogram.

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; the subroutines provided to test the status of the machine indicators (the sense switches and the sense lights) are defined as SUBROUTINE 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; PDUMP dumps core storage and then continues execution.

Mathematical Subroutines

FORTRAN provides various commonly used mathematical subroutines, defined as FUNCTION subprograms. The names of all of these subprograms are automatically typed by the FORTRAN IV Compiler; therefore, they need not appear in Type statements.

Variables used as arguments of mathematical subroutines must be typed, either explicitly or implicitly, in accordance with the function in which they appear. The mathematical subroutines are listed in Figure 9.

Machine Indicator Tests

In the following list of machine indicator test subroutines, assume that i is an integer expression and that j is an integer variable. These subroutines are referred to by CALL statements.

SLITE (i): If i = 0, all sense lights will be turned off. If i = 1, 2, 3, or 4, the corresponding sense light will be turned on.

SLITET (i, j): Sense light i will be tested and turned off. The variable j will be set to 1 if i was on, or j will be set to 2 if i was off.

SSWTCH (i, j): Sense switch i is tested and j is set to 1 if i was down, or j is set to 2 if i was up.

OVERFL (j): j is set to 1 if a floating point overflow condition exists, or j is set to 2 if no overflow condition exists. The machine is left in a no overflow condition.

DVCHK (j): If the divide check indicator is on, j is set to 1 and the divide check indicator is turned off; if the divide check indicator is off, j is set to 2.

Function	Definition	Number of Arguments	Name	Type of Argument	Type of Function
Exponential	e^{Arg}	1	EXP	Real	Real
Natural logarithm	$\log_e(Arg)$	1	ALOG	Real	Real
Common logarithm	$\log_{10}(Arg)$	1	ALOG10	Real	Real
Trignometric sine	sin(Arg)	1	SIN	Real	Real
Trigonometric cosine	cos(Arg)	1	COS	Real	Real
Trigonometric tangent	tan(Arg)	1	TAN	Real	Real
Trigonometric cotangent	cot(Arg)	1	COTAN	Real	Real
Arctangent	arctan(Arg)	1	ATAN	Real	Real
	$\arctan(Arg_1/Arg_2)$	2	ATAN2	Real	Real
Arcsine	arcsine(Arg)	1	ARSIN	Real	Real
Arccosine	arccosine(Arg)	1	ARCOS	Real	Real
Hyperbolic sine	sinh(Arg)	1	SINH	Real	Real
Hyperbolic cosine	cosh(Arg)	1	COSH	Real	Real
Hyperbolic tangent	tanh(Arg)	1	TANH	Real	Real
Square root	$(Arg)^{1/2}$	1	SQRT	Real	Real
Error Function	error function(Arg)	1	ERF	Real	Real
Gamma	gamma(Arg)	1	GAMMA	Real	Real
Log gamma	log gamma(Arg)	1	ALGAMA	Real	Real
Natural logarithm	$\log_e(Arg)$	1	DLOG	Double	Double
Common logarithm	$\log_{10}(Arg)$	1	DLOG10	Double	Double
Trigonometric sine	sin(Arg)	1	DSIN	Double	Double
Trigonometric cosine	cos(Arg)	1	DCOS	Double	Double
Remaindering	$Arg_1(\text{mod } Arg_2)$	2	DMOD	Double	Double
Exponential	e^{Arg}	1	DEXP	Double	Double
Arctangent	arctan(Arg)	1	DATAN	Double	Double
	$\arctan(Arg_1/Arg_2)$	2	DATAN2	Double	Double
Square root	$(Arg)^{1/2}$	1	DSQRT	Double	Double
Exponential	e^{Arg}	1	CEXP	Complex	Complex
Natural logarithm	$\log_e(Arg)$	1	CLOG	Complex	Complex
Trigonometric sine	sin(Arg)	1	CSIN	Complex	Complex
Trigonometric cosine	cos(Arg)	1	CCOS	Complex	Complex
Absolute value	$\lvert Arg \rvert$	1	CABS	Complex	Real
Square root	$(Arg)^{1/2}$	1	CSQRT	Complex	Complex

Figure 9. Mathematical subroutines

Note: The above mathematical subroutines are described in the publication IBM 7090/7094 IBSYS Operating System IBJOB Processor, Form C28-6389-0.

EXIT, DUMP, and PDUMP

EXIT

A CALL to the EXIT subprogram terminates the execution of any program by returning control to the Monitor.

DUMP

A CALL to the DUMP subprogram by the statement

CALL DUMP (A_1, B_1, F_1, . . . , A_n, B_n, F_n)

causes the indicated limits of core storage to be dumped and execution to be terminated by returning control to the Monitor.

1. A and B are variable data names that indicate the limits of core storage to be dumped; either A or B may represent upper or lower limits.
2. F_i is an integer indicating the dump format desired:

   F = 0 dump in octal
   1 dump as real
   2 dump as integer
   3 dump in octal with mnemonics
3. If no arguments are given, all of core storage is dumped in octal.
4. If the last argument F_n is omitted, it is assumed to be equal to 0 and the dump will be octal.

PDUMP

A CALL to the PDUMP subprogram by the statement

CALL PDUMP (A_1, B_1, F_1, . . . , A_n, B_n, F_n)

causes the indicated limits of core storage to be dumped and execution to be continued. The PDUMP arguments are the same as the DUMP arguments. The DUMP and PDUMP subprograms use SYSUT4 as an intermediate unit (corresponding to symbolic unit 4 in the standard FORTRAN Input/Output package). Since the unit is restored to its prior position before execution is resumed, information stored on SYSUT4 can be destroyed by these subprograms after the core storage dump.

BLOCK DATA SUBPROGRAM

The only way to enter data into a labeled COMMON block during compilation is by using a BLOCK DATA subprogram. (Data may not be entered into blank COMMON by the use of a DATA statement in any program or subprogram.) This subprogram may contain only the DATA, COMMON, DIMENSION, and Type statements associated with the data being defined.

General Form
BLOCK DATA

1. The BLOCK DATA subprogram may not contain any executable statements.
2. The first statement of this subprogram must be the BLOCK DATA statement.
3. All elements of a COMMON block must be listed in the COMMON statement even though they do not all appear in the DATA statement; for example, the variable A in the COMMON statement in Figure 10 does not appear in the DATA statement.
4. Data may be entered into more than one COMMON block in a single BLOCK DATA subprogram.
5. If two or more BLOCK DATA subprograms occur for the same application, the data specified by each of them is entered into the appropriate COMMON blocks. Data may not be entered into a COMMON block which is initialized by another DATA statement.

```
BLOCK DATA
COMMON  /ELN/C,A,B/RMG/Z,Y
DIMENSION B(4),Z(3)
DOUBLE PRECISION Z
COMPLEX C
DATA (B(I),I=1,4)/1.1,1.2,2*1.3/,C/(2.4,3.769)/,
     Z(1)/7.6498085D0/
END
```

Figure 10

The specification statements provide information about storage allocation and about the constants and variables used in the program.

DIMENSION Statement

General Form
DIMENSION $v_1(i_1), v_2(i_2), \ldots$ where: 1. each v_n is an array variable, and 2. each i_n is composed of from one to seven unsigned integer constants and/or integer variables, separated by commas. (Integer variables may be a component of i_n only when the DIMENSION statement appears in a subprogram.)

Examples:

```
DIMENSION A (1,2,3,4), B(10)
DIMENSION C(2,2,3,3,4,4,5)
```

In the preceding examples, A, B, and C are declared to be array variables with 4, 1, and 7 dimensions, respectively.

The DIMENSION statement provides the information necessary to allocate storage for arrays in the object program. The DIMENSION statement defines the maximum size of arrays. An array may be declared to have from one to seven dimensions by placing it in a DIMENSION statement with the appropriate number of subscripts appended to the variable.

1. The DIMENSION statement must precede the first appearance of the variable(s) to which it refers in any executable, NAMELIST, or DATA statement in the program. For exceptions, see "Appendix A."
2. A single DIMENSION statement may specify the dimensions of any number of arrays.
3. If a variable is dimensioned in a DIMENSION statement, it must not be dimensioned elsewhere.
4. Dimensions may also be declared in a COMMON or a Type statement. If this is done, then these statements are subject to all the rules for the DIMENSION statement.

Adjustable Dimensions

The name of an array and the constants that are its dimensions may be passed as arguments in a subprogram call. In this way, a subprogram may perform calculations on arrays whose sizes are not determined until the subprogram is called. Figure 11 illustrates the use of adjustable dimensions.

1. Variables may be used as dimensions of an array only in the DIMENSION statement of a FUNCTION or SUBROUTINE subprogram. For any such array, the array name and all the variables used as dimensions must appear as arguments in the FUNCTION, SUBROUTINE, or ENTRY statement.

```
SUBROUTINE MAYMY (...,R,L,M,...)
          .
          .
DIMENSION...,R(L,M),...
          .
          .
          .
DO 100 I=1,L
```

Figure 11

2. A FUNCTION, SUBROUTINE, or ENTRY argument must be explicitly declared as INTEGER prior to its appearance as a DIMENSION variable, unless it is implicitly of integer type.
3. The adjustable dimensions may not be altered within the subprogram.
4. The absolute dimensions must be specified in a DIMENSION statement of the calling program.
5. The calling program passes the specific dimensions to the subprogram. These specific dimensions are those that appear in the DIMENSION statement of the calling program. Variable dimension size may be passed through more than one level of subprogram.

COMMON Statement

General Form
COMMON a, b, c, .../r/d, e, f, .../s/g, h, ... where: 1. a, b, ... are variables that may be dimensioned, and 2. /r/ , /s/, ... are variables that are block names.

Examples:

```
COMMON A, B, C/X/Q, R/YY/M, P
COMMON /Z/G, H, J//D, F
```

Variables, including array names, appearing in a COMMON statement are assigned locations relative to the beginning of a particular common block. This COMMON area may be shared by a program and its subprograms.

1. The COMMON statement must precede any executable, any NAMELIST, and any DATA statements in the program. (For exceptions, see "Appendix A".) If the variables appearing in a COMMON statement contain dimension information, they must not be dimensioned elsewhere.

2. 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.

3. 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.

4. COMMON Block Names. The symbolic name of a block, which is one to six alphameric characters the first of which is alphabetic, precedes the variable names belonging to the block. The block name is always embedded in slashes, e.g., /BB/. It must not be the same as the name of any other subroutine that is part of the same job. 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 by the block name embedded in slashes.

5. 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.

6. Block name entries are cumulative throughout the program. For example, the COMMON statements

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

have the same effect as the statement

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

7. 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 the programs that are executed together.

8. Variables brought into a COMMON block through EQUIVALENCE statements may increase the size of the block.

9. Two variables in COMMON may not be made equivalent to each other, directly or indirectly.

10. A double-word variable in COMMON must be placed such that its high-order part is an even number of words away from the first element in COMMON.

EQUIVALENCE Statement

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 either equal to the number of dimensions of the variable or must be equal to one.

Examples:

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

The EQUIVALENCE statement controls the allocation of data storage by causing two or more variables to share the same core storage location.

1. An EQUIVALENCE statement must precede any executable, any NAMELIST, and any DATA statements in the program. (For exceptions, see "Appendix C." Each pair of parentheses in the 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 (sets of parentheses) may be given.

2. In an EQUIVALENCE statement, the meaning of D(5) is "the fourth array element following the first array element, (D(1, 1, 1))." In general, D(p) is defined for p > 0 to mean the (p-1)th element following the first element of the D array, i.e., the pth element of the array.

In the preceding example, the EQUIVALENCE statement indicates that A and the B and C arrays are to be assigned storage locations so that the elements A, B(1), and C(5, 4) are to occupy the same location. In addition, it also specifies that D(1, 4, 3) and E are to share the same location.

3. Quantities or arrays that are not mentioned in an EQUIVALENCE statement will be assigned unique locations.

4. Locations can be shared only among variables, not among constants.

5. The sharing of storage locations requires a knowledge of which FORTRAN statements will cause a new value to be stored in a location. There are four such statements:

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

b. Execution of an ASSIGN i TO n statement stores a new value in n.

c. Execution of a DO statement or an implied DO in an input/output list sometimes stores a new indexing value.

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

6. 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 (B, D(1))
```

The layout of core storage indicated by this example (extending from the lowest location of the block to the highest location of the block) is:

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

7. Since arrays must be stored in consecutive forward locations, a variable may not be made equivalent to an element of an array in such a way as to cause the array to extend beyond the beginning of the COMMON block. For example, the following coding is invalid:

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

The previous example would force D(1) to precede A, as follows:

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

8. The rule for making double-word variables equivalent to single-word variables is:

In COMMON, the effect of the EQUIVALENCE statements must be such that the high-order word of any double-word variable is an even number of locations away from the start of the COMMON block.

In non-COMMON, the effect of the EQUIVALENCE statements must be such that the high-order word of any double-word variable is an even number of words away from the start of any other double-word variable linked to it through EQUIVALENCE statements.

9. Two variables in one COMMON block or in two different COMMON blocks may not be made equivalent.

10. The EQUIVALENCE statement should not be used to make two or more elements mathematically equivalent.

Type Statements

The type of a variable or function may be specified by means of one of the six Type statements:

General Form
INTEGER $a(i_1)$, $b(i_2)$, $c(i_3)$, . . . REAL $a(i_1)$, $b(i_2)$, $c(i_3)$, . . . DOUBLE PRECISION $a(i_1)$, $b(i_2)$, $c(i_3)$, . . . COMPLEX $a(i_1)$, $b(i_2)$, $c(i_3)$, . . . LOGICAL $a(i_1)$, $b(i_2)$, $c(i_3)$, . . . EXTERNAL x, y, z, . . . where: 1. a, b, c, . . . are variable or function names appearing within the program. 2. x, y, z, . . . are function names appearing within the program. 3. each i_n is composed of from one to seven integer constants and/or integer variables. Subscripts may be appended only to variable names appearing within the program, not to function names.

Examples:

```
INTEGER BIXF, X, QF, LSL
REAL IMIN, LOG, GRN, KLW
DOUBLE PRECISION Q, J, DSIN
EXTERNAL SIN, MATMPY, INVTRY
INTEGER A(10,10), B
COMPLEX C(4,5,3), D
```

The variable or function names following the type (INTEGER, REAL, etc.) in the Type statement are defined to be of that type, and remain so throughout the program; the type may not be changed.

Note that LSL and GRN need not appear in their respective Type statements since their type is implied by their first characters. Note also that DSIN need not appear in its statement if it is used as a function in the program, since mathematical subroutines are automatically typed by the FORTRAN IV Compiler.

1. The appearance of a name in any Type statement, except EXTERNAL, overrides the implicit type assignment.

2. Variables that appear in EXTERNAL statements are subprogram names. Subprogram names must appear in an EXTERNAL statement if they are the

arguments of other subprograms or if they are the name of a built-in function that is used as the name of a FUNCTION or SUBROUTINE subprogram.

3. A name may appear in two Type statements only if one of the statements is EXTERNAL.

4. The type statements (except EXTERNAL) must precede the first appearance of the variable(s) to which they refer in any executable, NAMELIST, or DATA statements in the program. The EXTERNAL statement must precede the first appearance of the subprogram or ENTRY name to which it refers in any executable statement in the program. For exceptions, see "Appendix A: Source Program Statements and Sequencing."

5. A name declared to be of a given type may assume only the values of a constant of the same type.

6. The arguments of an ENTRY statement cannot appear in a Type statement that appears earlier in the program, unless they are also arguments in a SUBROUTINE or FUNCTION statement.

7. Any variable that is dimensioned by a Type statement may not be dimensioned elsewhere, i.e., it may not appear in a DIMENSION statement or in a COMMON statement that contains dimension information.

DATA Statement

Data may be compiled into the object program by means of the DATA statement.

General Form
DATA list/$d_1, d_2, \ldots, d_n$/, list/$d_1, d_2, k*d_3, \ldots, d_m$/, ... where: 1. list contains the names of the variables being defined, 2. d is the data literal, and 3. k is an integer constant.

Examples:

```
DATA R, Q/14.2,3HEND/, Z/O777777700001/
DATA (B(I), C(I), I=1,40,2)/2.0,3.0,38*100.0/
LOGICAL LA, LB, LC, LD
DATA LA, LB, LC, LD/F, .TRUE., .FALSE., T/
```

1. List. Subscripted variables may appear in the list. Where a subscript symbol is used, it must be under control of DO-implying parentheses and associated parameters. Subscripts not so controlled must be integer constants. The DO-defining parameters must be integer constants.

2. k. The letter k may appear before a d-field to indicate that the field is to be repeated k times. An asterisk (*) must follow the letter k to separate it from the field to be repeated.

3. d. The data literals may take any of the four following forms:

a. Integer, real, double-precision, and complex constants. They may be signed or unsigned.

b. Alphameric characters. The alphameric field is written as nH followed by n alphameric characters. Each group of six alphameric characters forms a word. If n is not a multiple of six, the remaining characters are left justified in the word, and the word is filled out with BCD blanks. Blanks are significant in alphameric fields.

c. Octal digits. The octal field is written as O, followed by 1-12 signed or unsigned octal digits.

d. Logical constants. The logical field may be written as either .TRUE., .FALSE., T, or F.

4. There must be a one-to-one correspondence between the list items and the data literals. Each data literal (integer constant, real constant, alphameric constant, complex constant, logical constant, double-precision constant, or octal constant) corresponds to one undimensioned variable or subscripted array reference.

Note: If it is desired to define 16 alphameric characters, say 16HDATAbTObBEbREADb starting at G(1), then G must be dimensioned to cover at least three locations and the entire literal corresponds to G(1).

5. The BLOCK DATA subprogram, which includes a DATA statement, compiles data into the common area of the program.

6. DATA defined variables that are redefined during execution will assume their new values regardless of the DATA statement.

7. Where data is to be compiled into an entire array, the name of the array (with indexing information omitted) can be placed in the list. The number of data literals must be equal to the size of the array.

For example, the statements

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

define the values of A, B(1),..., B(23) to be 4.0, and the values of B(24), B(25), and C to be 3.0, 2.0, and 1.0, respectively.

8. The DATA statement may not be used to enter data into unlabeled COMMON.

APPENDIX A: SOURCE PROGRAM STATEMENTS AND SEQUENCING

The following is a complete list of the 7090/7094 FORTRAN IV source program statements, their sequence of execution, and their order in the source program.

Statement	Normal Sequencing	Executable or Nonexecutable	Order in the Source Program[1]
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 a BLOCK DATA subprogram.
CALL	First statement of called program	Executable	May be placed anywhere.
COMMON	Next statement	Nonexecutable	Must precede any executable, any NAMELIST, and any DATA statements in the program.
COMPLEX	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers in any executable, NAMELIST, or DATA statement in 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, but it must appear in every every BLOCK DATA subprogram.
DIMENSION	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers in any executable, NAMELIST, or DATA statement in the program.
DO	Normal DO sequencing, then the next statement	Executable	May be placed anywhere.
DOUBLE PRECISION	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers in any executable, NAMELIST, or DATA statement in the program.
END	Terminates compilation of the program	Nonexecutable	Must be the physically last statement of the program.
END FILE	Next statement	Executable	May be placed anywhere.
ENTRY	Next statement	Nonexecutable	May appear only in a subprogram, but not as the first statement or in the range of a DO.
EQUIVALENCE	Next statement	Nonexecutable	Must precede any executable, any NAMELIST, and any DATA statements in the program.

Statement	Normal Sequencing	Executable or Nonexecutable	Order in the Source Program[1]
FORMAT	Next statement	Nonexecutable	May be placed anywhere.
FUNCTION	Next statement	Nonexecutable	Must be used only as the first statement of a FUNCTION subprogram.
GO TO n	Statement n	Executable	May be placed anywhere.
GO TO i, $(n_1, n_2, \ldots, n_m)$	Statement last assigned to i	Executable	May be placed anywhere.
GO TO $(n_1, n_2, \ldots, n_m)$, i	Statement n_i	Executable	May be placed anywhere.
IF $(a) n_1, n_2, n_3$	Statement n_1, n_2, n_3 if $a<0$, $a=0$, or $a>0$, respectively	Executable	May be placed anywhere.
IF (t) s	Statement s if t is true; next statement if t is false	Executable	May be placed anywhere.
INTEGER	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers in any executable, NAMELIST, or DATA statement in the program. Must also precede the first appearance of the variable(s) to which it refers in a DIMENSION statement in a subprogram when the variable(s) is used as an adjustable dimension.
LOGICAL	Next statement	Nonexecutable	Must precede the first appearance of the variable(s) to which it refers in any executable, NAMELIST, or DATA statement in the program.
NAMELIST	Next statement	Nonexecutable	Must precede any appearance of a NAMELIST name in the program.
PAUSE	Next statement	Executable	Should be placed where a temporary halt is desired.
PRINT	Next statement	Executable	May be placed anywhere.
PUNCH	Next statement	Executable	May be placed anywhere.
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 in any executable, NAMELIST, or DATA statement in the program.
RETURN	The first statement, or part of a statement, following the reference to the subprogram	Executable	Must be placed in a subprogram where a return to the calling program is desired.
RETURN i	The executable statement i in the calling program	Executable	Must appear in a SUBROUTINE subprogram where a nonstandard return is desired.
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.

Statement	Normal Sequencing	Executable or Nonexecutable	Order in the Source Program
SUBROUTINE	Next statement	Nonexecutable	Must be used only as the first statement of a SUBROUTINE subprogram.
WRITE	Next statement	Executable	May be placed anywhere.

[1]Many of the FORTRAN source statements may not end the range of a DO. See the section "DO Statement" for restrictions on statements in the range of a DO.

APPENDIX B: 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
blank	blank	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

The character $ can be used in FORTRAN IV only as alphameric text in an H field or as a prefix to the address argument in a CALL statement to a subroutine with nonstandard returns. The character ' (8-4) can be used only as alphameric text in an H field.

APPENDIX C: DIFFERENCES BETWEEN FORTRAN II AND FORTRAN IV

This section contains a summary of the differences between the FORTRAN II and FORTRAN IV languages.

1. All language items distinguished by a column 1 modal punch, except B, in FORTRAN II have been incorporated into FORTRAN IV by the Type statements as follows:

FORTRAN II Modal Punches	FORTRAN IV Type Statement (See the section, "Type Statements")
Double-precision arithmetic - D	DOUBLE PRECISION
Complex arithmetic - I	COMPLEX
F-card - F	EXTERNAL

The DATA statement may be used to enter octal constants into a FORTRAN IV program; the programmer may use certain built-in functions (see Appendix H) to handle Boolean Arithmetic. Column 1 Modal punches should not be used in FORTRAN IV.

2. The following are the differences in function naming:
   a. 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, incompatibilities may arise. In FORTRAN IV, this difficulty is handled by the Type statements REAL and INTEGER, which define a variable name or function name as floating point or fixed point, respectively (see the section "The Type Statements").
   b. The number of characters in an open, a closed, or an arithmetic statement function name in FORTRAN II is four to seven, ending in F; whereas, in FORTRAN IV, the number of characters is one to six and the final F has no meaning. In both cases, the first character of the function name must be alphabetic (see the section "Naming Subroutines").
   c. Built-in and arithmetic statement functions are not identified by a terminal F in FORTRAN IV; they are named in FORTRAN IV as described in item b above. The FORTRAN II library function is a FORTRAN IV FUNCTION subprogram.

3. The following are the differences between the COMMON and EQUIVALENCE statements.
   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 (see the section "COMMON Statement").
   b. The FORTRAN IV COMMON and Type Statements may contain dimension information.
   c. In FORTRAN IV, EQUIVALENCE and COMMON statements must precede any executable, NAMELIST, and DATA statements in the program. In FORTRAN II, EQUIVALENCE and COMMON statements may be placed anywhere in the program.

4. 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 a dummy argument of an arithmetic statement function, the explicit type applies in both contexts.

5. 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, and ( ) is any arithmetic expression within parentheses.

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

FORTRAN II Statements	FORTRAN IV Statements	
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.

7. Additional FORTRAN IV statements.
   a. DATA (see the section "DATA Statement").
   b. BLOCK DATA (see the section "BLOCK DATA Subprogram").
   c. LOGICAL, an additional Type statement that defines variables to be used in logical computation (see the section "The Type Statement").
   d. NAMELIST (see the section "NAMELIST statement").
   e. ENTRY (see the section "Multiple Entry Points into a Subprogram").

8. The following are differences in output produced by FORTRAN IV and FORTRAN II:

Programmers will find that the output produced by a source program in FORTRAN IV may not be the same as that provided by the identical program in FORTRAN II. Differences that do occur are attributed exclusively to the following differences that exist between FORTRAN IV and FORTRAN II.

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. Floating point constants that are 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 FORTRAN IV achieves a more accurate conversion and preserves more significance than does FORTRAN II.

c. The mathematical subroutines in FORTRAN IV are assembled by MAP, and those in FORTRAN II are assembled by FAP. The conversion routines in MAP 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 those 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. The arrangement of arrays in core storage should be noted by programmers who are converting FORTRAN II programs to FORTRAN IV. In FORTRAN II, arrays are arranged in decreasing absolute storage locations, whereas in FORTRAN IV, arrays are stored in increasing absolute storage locations.

9. In FORTRAN IV, EQUIVALENCE and COMMON statements must precede any executable, NAMELIST, and DATA statements in the program.

The rule that COMMON and EQUIVALENCE statements precede the first executable statement of the program is a precautionary rule. It does not inhibit compilation and execution of the object program. It can, however, cause an incorrect object program to be generated in one combination of circumstances. This is shown in the following examples.

<u>Example 1.</u>

```
    .
    .
    .
COMMON J
    .
    .
    .
DO I
CALL SUBRTN
...A(K)...
    .
    .
    .
EQUIVALENCE (K,J)
    .
    .
    .
```

<u>Example 2.</u>

```
    .
    .
    .
DO I
CALL SUBRTN
...A(K)...
    .
    .
    .
COMMON K
    .
    .
    .
```

If, as in these examples, it is the case that:

a. The COMMON and/or EQUIVALENCE statement(s) follow the first executable statement in the program, and

b. These statements bring the relevant subscript symbol (K) into COMMON, and

c. Subroutine SUBRTN changes the value of K in COMMON, then the subscript reference A (K) will not utilize the newly-defined value. The order of the CALL statement and the subscript reference A (K) in the DO loop do not affect these restrictions.

Except in this combination, no error will result from having the COMMON and EQUIVALENCE statements follow the first executable statement of the program.

10. In FORTRAN II, the sequence ENDFILE i, BACKSPACE i, positions the tape immediately before the end-of-file mark. In FORTRAN IV, this sequence positions the tape immediately before the last physical or logical record of the preceding file. For a more complete description of the FORTRAN IV BACKSPACE handling, see the section "The Manipulative Input/Output Statements."

APPENDIX D: OPTIMIZATION OF ARITHMETIC EXPRESSIONS

To optimize the object program, a sequence of operations on the multiply-divide (*, /) level may be reordered by the compiler. The reordering tends to alternate the multiply and divide operations. It occurs where all elements of the expression are of the real type. This is done on the assumption that mathematically equivalent expressions are computationally equivalent.

Where the multiply-divide expression involves mixed real and complex types, the operations on the real types occur first and are alternated.

Where the order of operations is considered significant, the programmer may use nested parentheses in the expression to specify explicitly the ordering he desires.

APPENDIX E: BINARY RECORD FORMAT

Under the form READ (i) list, an entire logical record is read. However, only as many words as are specified in the list will be transmitted to the object program. Binary records to be read in by a FORTRAN program using the standard FORTRAN input/output package (FIOCS) should be written by a FORTRAN program or should be in the proper binary record format as follows:

Consider a logical record as being any sequence of binary words to be read by any one input statement. This logical record must be divided into physical records, each of which is a maximum of 256_{10} words long. Of course, if a logical record consists of fewer than 256_{10} words, it will comprise only one physical record.

The first word of each physical record is a "signal" word that is not part of the list. The decrement portion of this word contains a count of the number of words in the physical record, exclusive of the signal word. If the count is zero, the number of words is assumed to be 127_{10}. The address portion of the signal word is zero unless it is in the last physical record of the logical record, in which case it contains a count of the number of physical records contained in the logical record.

APPENDIX F: GENERAL PROPERTIES OF A FORTRAN SOURCE PROGRAM

Writing the Source Program

The statements of a FORTRAN source program are normally written on a standard FORTRAN coding sheet as shown in Figure 12.

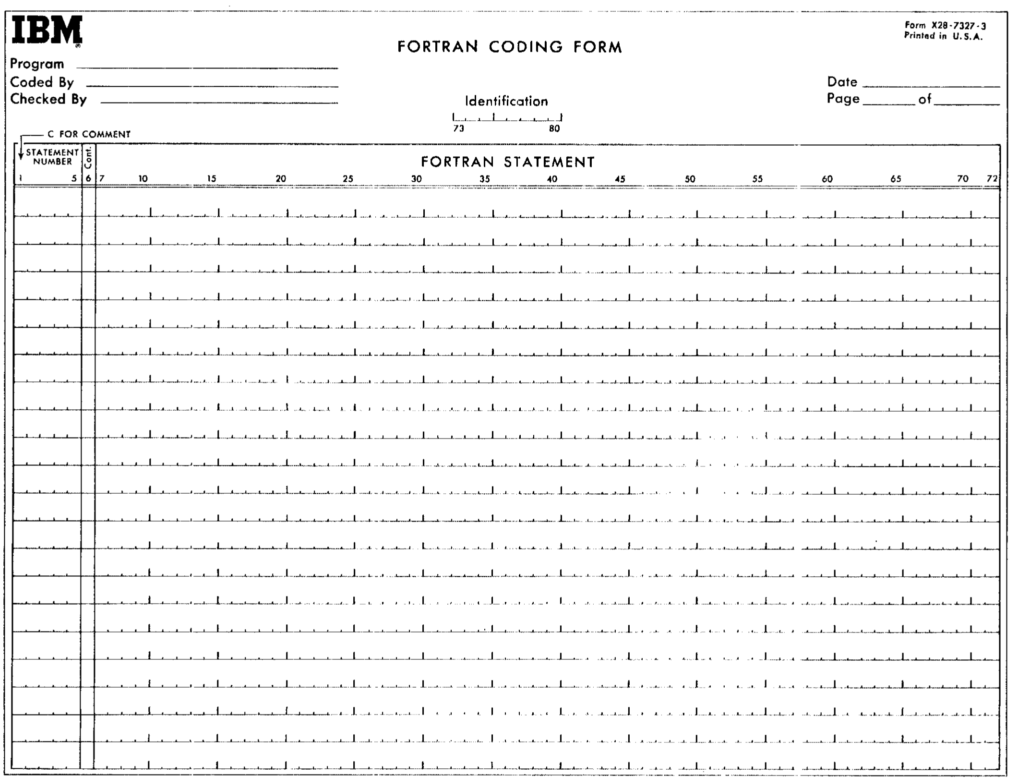

IBM
FORTRAN CODING FORM
Form X28-7327-3
Printed in U.S.A.
Program
Coded By
Checked By
Identification 73 80
Date
Page ___ of ___
C FOR COMMENT
STATEMENT NUMBER 1 5
Cont. 6
FORTRAN STATEMENT 7 10 15 20 25 30 35 40 45 50 55 60 65 70 72

Figure 12. Standard FORTRAN Coding Sheet

1. Columns 1-5 of the first line of a statement may contain a statement number that is less than 32,768 (but not zero) to identify the statement. Blanks and leading zeros are ignored in these columns.

2. Column 6 of the first line of a statement must be left blank or punched with a zero.

3. Columns 7-72 contain the actual FORTRAN statement. Blanks are ignored except in an alphameric field, which may appear in a FORMAT statement, a DATA statement, or as the argument of a CALL.

4. A statement may be continued over as many as nineteen continuation cards. Any card with a nonblank, nonzero column 6 is a continuation card.

5. Cards with a C in column 1 are not processed by FORTRAN, and columns 2-72 may be used for comments.

6. Columns 73-80 are not processed by FORTRAN and may be used for identification.

7. The order of execution of the source statements is governed by the normal sequencing of source program statements given in "Appendix A."

A sample FORTRAN IV source program, complete with control cards, is given in "Appendix G."

Punching the Source Program

Each line of the coding sheet is used to prepare a punched card. The information in column 1 of a line on the coding sheet is punched into card column 1, column 2 into card column 2, and so forth. Cards should be verified after being punched to prevent clerical errors from causing source and object program errors.

APPENDIX G: DECK SETUP FOR A FORTRAN IV JOB

Figure 13 shows the deck setup for a FORTRAN IV job. The control cards and the decks used in this figure are described in the following text.

Definitions of Terms

The following are definitions of terms used in this appendix.

Processor Monitor: The Processor Monitor is the supervisory portion of the IBJOB Processor. It provides communication between the System Monitor (IBSYS) and the components of the IBJOB Processor.

Processor Application: A processor application is the basic unit of work that can be performed by the processor. An application can consist of one or more compilations, assemblies, or the loading of relocatable programs that were assembled previously.

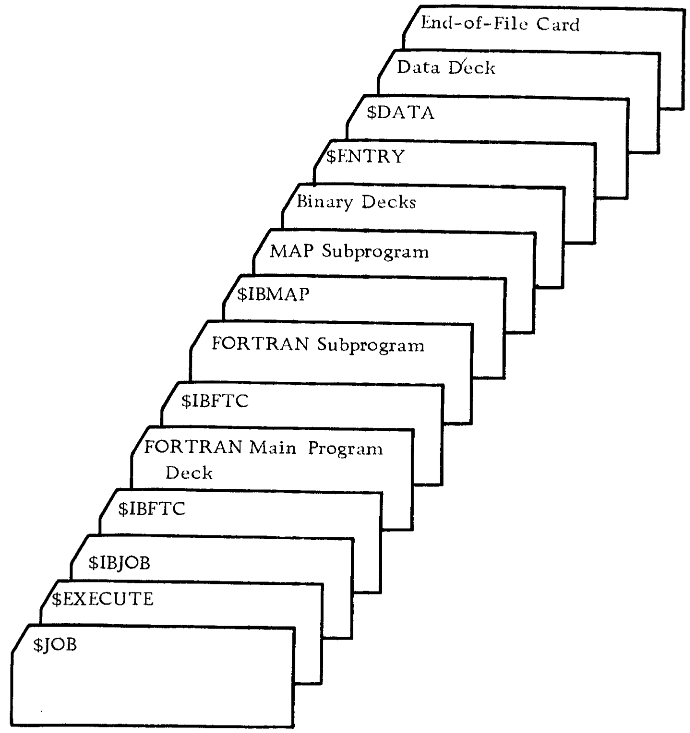


Figure 13. Deck Setup for a FORTRAN IV Job

Job: A job is one or more applications that are specified by the programmer to be executed as a logical unit. A job is delimited by a $JOB card.

IBJOB Debugging Processor: The IBJOB Debugging Processor provides a means for the user to take highly selective "snapshots" of core storage during execution of FORTRAN IV and MAP programs. Specifications for these core dumps are incorporated into the user's program at load-time.

Control Card Notation

The following notation is used in the control card formats in this appendix:

1. Brackets[ ] represent an option that may be omitted or included, at the programmer's choice.

2. Braces { } indicate that a choice of the contents is to be made by the programmer. The standard option, which is underlined, is assumed when no option is specified.

3. Upper-case specifications, if used, must be present in the form specified.

4. Lower-case specifications represent typical quantities or terms whose values must be supplied by the programmer.

5. The order in which options are specified on the various control cards is not critical unless otherwise specified.

6. Commas are used to separate options when options are present. If no options are present, a string of commas is not necessary to indicate the absence of options, unless otherwise specified.

7. Options must be punched on the control card without any blanks between words or around the punctuation.

$JOB Card

This card defines the beginning of a job. The format of the $JOB card is:

```
1               16
$JOB            any text
```

A detailed explanation of this card is in the publication IBM 7090/7094 IBSYS Operating System, System Monitor (IBSYS), Form C28-6248.

$EXECUTE Card

This card must precede a processor application within a job, if one of the following conditions is present:

1. The processor application is the first unit of work to be performed for an object program.
2. The previous processor application resulted in execution of an object program.
3. Another subsystem was in control.

The format of the $EXECUTE card is:

```
1               16
$EXECUTE        subsystem name
```

When using this card for a FORTRAN IV job, the subsystem name is IBJOB. If the name in column 16 is IBJOB, control is transferred to the IBJOB Processor unless it already has control, in which case no action is taken. If the name is anything other than IBJOB, this information is given to the System Monitor.

A more detailed explanation of the $EXECUTE card is in the publication IBM 7090/7094 IBSYS Operating System, System Monitor (IBSYS), Form C28-6248.

$IBJOB Card

The $IBJOB card must be the first control card read by the Processor Monitor for a given application. The options that can be specified in this control describe the manner in which an application is to be processed. The format of the $IBJOB card is:

```
1               16
$IBJOB          [{GO  }] [,{NOLOGIC}] [,{NOMAP}]
                [{NOGO}] [ {LOGIC  }] [ {MAP  }]
                         [ {DLOGIC }]

                [,{NOFILES}] [,{SOURCE  }]
                [ {FILES  }] [ {NOSOURCE}]

                [,{IOEX     [,{FLOW  }]}]
                [ {         [ {NOFLOW}]}]
                [ {MINIMUM             }]
                [ {BASIC               }]
                [ {LABELS              }]
                [ {FIOCS               }]
                [ {ALTIO               }]
```

The options in the variable field, which start in column 16, are described in the following text.

Execution Options

The execution options are:

1. GO -- The object program is to be executed after it is loaded.
2. NOGO -- The object program is not executed even if it is loaded.

If NOGO is specified, the object program is loaded only when LOGIC, DLOGIC, or MAP is specified in the $IBJOB card.

If neither GO nor NOGO is specified, the object program is to be executed.

Logic Options

The logic options are:

1. NOLOGIC -- A cross-reference table is not wanted.
2. LOGIC -- A cross-reference table of the program sections and the system subroutines required for execution is generated. The origin and length of each program section and subroutine and the buffer assignments are also given.
3. DLOGIC -- A cross-reference table of the program sections and the origin and length of each program section is generated. The system subroutines and buffer assignments are not given.

If neither LOGIC, DLOGIC, nor NOLOGIC is specified, a cross-reference table is not generated.

MAP Options

The MAP options are:

1. NOMAP -- A core storage map is not generated.
2. MAP -- A core storage map is generated, giving the origin and the amount of storage used by the IBSYS Operating System, the object program, and the input/output buffers. The file list and buffer pool organization are also given.

If neither MAP nor NOMAP is specified, a storage map is not generated.

File List Options

The file list options are:

1. NOFILES -- A list of the input/output unit assignments and mounting instructions to the operator are printed on-line.
2. FILES -- The list and mounting instructions are printed on-line and written off-line.

If neither FILES nor NOFILES is specified, the list is only printed on-line.

Input Deck Options

The input deck options are:

1. SOURCE -- The application contains at least one compilation or assembly.
2. NOSOURCE -- The application contains only relocatable binary program decks. These decks are loaded from the System Input Unit.

If neither SOURCE nor NOSOURCE is specified, it is assumed that a compilation or assembly is required in the application.

Input/Output Options

The IOCS options are:

1. IOEX -- The object program uses the Input/Output Executor (IOEX).
2. MINIMUM -- The minimum-level package of IOCS is to be loaded with the object program.
3. BASIC -- The basic-level package of IOCS is to be loaded with the object program.
4. LABELS -- The labels-level package of IOCS is to be loaded with the object program.
5. FIOCS -- The standard FORTRAN IV input/output package is to be loaded with the object program. (This package calls in one of the levels of IOCS.)

If none of these options are specified, IBLDR will determine the level of IOCS to be used with the object program. If the object program requires a higher level of IOCS than is specified, the specification is ignored by the Loader. The levels of IOCS are described in detail in the publication IBM 7090/7094 IBSYS Operating System, Input/Output Control System, Form C28-6345.

6. ALTIO - The alternate FORTRAN IV input/output package is to be loaded with the object program. (Since this package communicates directly with IOEX rather than calling one one of the levels of IOCS, core storage locations are saved by specifying ALTIO.)

If neither FIOCS nor ALTIO is specified for a FORTRAN IV program, the minimum-level package of IOCS is usually loaded with the object program. However, if IBLDR determines that one of the other levels of IOCS is required, it is loaded rather than the minimum package.

The two FORTRAN IV input/output packages are described in the section "The Subroutine Library" of the publication IBM 7090/7094 IBSYS Operating System, IBJOB Processor, Form C28-6389.

Overlay Options

The overlay options are:

1. Flow - Execution of the object program is not permitted if the rules concerning references between links are violated.
2. NOFLOW - Execution is permitted even though the rules governing references between links are violated.

If neither FLOW nor NOFLOW is specified, execution of the object program is not permitted when the rules governing references between links are violated.

A detailed explanation of this card is in the publication IBM 7090/7094 IBSYS Operating System, IBJOB Processor, Form C28-6389.

$IBFTC Card

The format of the $IBFTC card is:

```
1      8         16
$IBFTC deckname [{NOLIST}] [ {NODD}] [ {DECK  }]
                [{LIST  }] [,{DD  }] [,{NODECK}]
                [{FULIST}] [ {SDD }] [ {      }]

                                   [ {M90  }] [ {XR3}]
                                   [,{M94  }] [,{XRn}]
                                   [ {M94/2}] [ {   }]
```

where deck name identifies the deck that follows. A deck name of six or fewer alphameric characters must be punched in columns 8-13. Characters that cannot be used in the deck name are parentheses, commas, slashes, quotation marks, equal signs, and blanks. The deck name of a program that contains a subprogram or entry points may not be the same as the subprogram name or entry name.

The variable field starts in column 16. The options in the variable field are described in the following text.

List Options

The list options are:

1. NOLIST - A listing of the object program is not wanted.
2. LIST - A listing of the object program, three instructions per line, is generated. Only the relative locations and symbolic information are listed.
3. FULIST - A listing of the object program, one instruction per line, is generated. This listing includes generated octal information.

If neither NOLIST, LIST, nor FULIST is specified, a listing is not generated.

Debug Options

The debug options are:

1. NODD - The debugging dictionary is not generated.
2. DD - The full debugging dictionary is generated. All the symbols in the compiled program will appear in the debugging dictionary. For a FORTRAN IV program, this includes all statements numbers, all programmer-specified symbols, and all symbols generated by IBFTC.
3. SDD - The short debugging dictionary is generated. It will contain only the programmer-specified symbols and the statement numbers used in the FORTRAN IV program.

If NODD, DD, or SDD is not specified, the debugging dictionary is not generated.

Punch Options

The punch options are:

1. DECK - The object program deck is written on the system peripheral punch unit for off-line punching.
2. NODECK - A punched deck is not wanted.

If neither DECK nor NODECK is specified, the object program deck is written on the system peripheral punch unit.

Instruction Set Options

The instruction set options are:

1. M90 - The object program uses only 7090 machine instructions. Any double-precision operations are simulated by system macros, and EVEN pseudo-operations are treated as commentary.
2. M94 - The object program uses 7094 machine instructions.
3. M94/2 - The object program uses 7094 machine instructions, and EVEN pseudo-operations are treated as commentary.

If neither M90, M94, nor M94/2 is specified, it is assumed that the object program uses only 7090 machine instructions.

Index Register Options

The index register options are:

1. XR3 - The object program uses three index registers (1, 2, and 4).
2. XRn - The object program can use up to n index registers if they are required (n is a number from 4 through 7).

If neither XR3 nor XRn is specified, it is assumed that the object program uses three index registers.

A detailed explanation of the $IBFTC card is in the publication IBM 7090/7094 IBSYS Operating System, IBJOB Processor, Form C28-6389.

FORTRAN Main Program Deck

This deck consists of the sequence of FORTRAN IV source statements that constitute the main program; it does not include any subprograms or data to be read in at object time.

FORTRAN Subprograms

These are the FUNCTION or SUBROUTINE subprograms; coded by the programmer, that are referred to or called by the main program or another subprogram.

$IBMAP Card

The $IBMAP card is used in a FORTRAN job only if one or more of the subprograms referred to by the main program are coded by the programmer in the MAP language. When the Processor Monitor recognizes this card, it calls the Macro Assembly Program (IBMAP) to assemble the MAP subprogram(s). A description of the $IBMAP card is in the publication IBM 7090/7094 IBSYS Operating System, IBJOB Processor, Form C28-6389.

MAP Subprograms

The MAP subprograms are routines that are coded by the programmer in the MAP language and are referred to in either the FORTRAN main program or a FORTRAN subprogram. There are a number of ways that the programmer can get from a FORTRAN program to the MAP subprogram and then return. One method is to use an ENTRY pseudo-operation in the MAP program to establish an entry point and a SAVE pseudo-operation with an associated RETURN to return to the FORTRAN program. This method is explained in detail in the publication IBM 7090/7094 IBSYS Operating System, IBJOB Processor, Form C28-6389.

Binary Decks

Binary decks are output from previous compilations or assemblies. Since the decks are in binary, they need only be loaded. Two situations where they might occur in a FORTRAN IV job are:

1. If the programmer wants to run a previously compiled main program with different data.
2. If the programmer wants to use previously compiled subprogram(s) with his main program.

$ENTRY Card

The $ENTRY card specifies the location of the initial transfer to the object program at execution time. The variable field contains a literal, consisting of an external name to which the initial transfer is to be made. If the $ENTRY card is omitted or if the variable field is blank, the initial transfer is to either the standard entry point of the first deck retained or to an entry point whose name is '......' (the name compiled as the standard entry point to FORTRAN IV main programs).

The format of the $ENTRY card is:

```
1            16
$ENTRY       [{Exname  }]
             [{Deckname}]
```

where the variable field contains either an external name to which the initial transfer is to be made or a deck name, in which case the initial transfer is to the standard entry point of that deck.

A $ENTRY card is not needed when one of the following conditions exists:

1. The main program is a FORTRAN IV program.
2. The main program is processed first, and the desired entry point is the standard entry point of that program.

When a $ENTRY card is used, it must immediately follow the source deck. The $ENTRY card precedes either an end-of-file card or a $DATA card.

$DATA Card

The $DATA card indicates the beginning of the data deck. This card may be replaced by an end-of-file card with a 7-8 punch in column 1. One case where an end-of-file card must be used is if a (READ n, list) statement appears in the main program.

Data Deck

This deck contains the data to be used by the main program or any of the subprograms.

End-of-File Card

The end-of-file card must be the last card of a Processor application. The format for an end-of-file card is:

```
1
7
8EOF
```

This card can be replaced with any other control card that causes a file mark to be written by a peripheral program.

Sample Program and Output

Figure 14 is a sample FORTRAN IV program that computes the real roots of a set of 50 or fewer quadratic equations of the form $ax^2 + bx + c = 0$.

Figure 15 is the output that was obtained from the program in Figure 14.

```
C
O
L
1    6  8       16
$JOB
$EXECUTE        IBJOB
$IBJOB          MAP
$IBFTC DECK1    LIST,SDD,NODECK
C  PROGRAM FOR COMPUTING THE REAL ROOTS OF A SET OF 50 OR FEWER
C     QUADRATIC EQUATIONS OF THE FORM AX**2 + BX + C = 0.
      DIMENSION A(50),B(50),C(50)
      WRITE(6,103)
      READ(5,100) K,(A(I),B(I),C(I),I=1,K)
      DO 50 I=1,K
      DISC = B(I)**2 - 4.0*A(I)*C(I)
      IF(DISC) 29,39,49
   29 WRITE(6,101) A(I),B(I),C(I)
      GO TO 50
   39 ROOT1 = -B(I)/(2.0*A(I))
      ROOT2 = ROOT1
      GO TO 45
   49 RAD = SQRT(DISC)
      ROOT1 = (-B(I) + RAD)/(2.0*A(I))
      ROOT2 = (-B(I) - RAD)/(2.0*A(I))
   45 WRITE(6,102) A(I),B(I),C(I),ROOT1,ROOT2
   50 CONTINUE
  100 FORMAT(I5,/(3F12.4))
  101 FORMAT(3(E12.4,5X),19HROOTS ARE IMAGINARY)
  102 FORMAT(3(E12.4,5X),F12.4,6X,F12.4)
  103 FORMAT(1H1,2X,1HA,16X,1HB,16X,1HC,18X,5HROOT1,13X,5HROOT2,//)
      STOP
      END
$DATA
00005
1.0         -2.0        1.0
   1.0      2.0          15.0
      1             0.0 1.0
   -20.5    -11.3    4.0
7  7.1         5.3   0.0
8EOF
$IBSYS
$STOP
```

Figure 14. Sample FORTRAN IV Program

A	B	C	ROOT1	ROOT2
0.1000E 01	-0.2000E 01	0.1000E 01	1.0000	1.0000
0.1000E 01	0.2000E 01	-0.1500E 02	3.0000	-5.0000
0.1000E 01	0.	0.1000E 01	ROOTS ARE IMAGINARY	
-0.2050E 02	-0.1130E 02	0.4000E 01	-0.7963	0.2450
0.7100E 01	0.5300E 01	0.	-0.0000	-0.7465

Figure 15. Sample Program Output

APPENDIX H: MACHINE-DEPENDENT FEATURES

The built-in functions shown in Figure 16 are included only to allow the user of FORTRAN to make use of the special logical operations of the 7090/7094 Data Processing System. 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 Arguments	Type of Argument	Type of Function
Logical intersection of two 36-bit arguments	AND	2	Real or Integer	Real
Logical union of two 36-bit arguments	OR	2	Real or Integer	Real
Logical 1's complement of the 36-bit argument	COMPL	1	Real or Integer	Real
Logical 36-bit argument from signed 35-bit argument	BOOL (see note)	1	Real or Integer	Real

Note: The function BOOL is used to get results similar to those of the FORTRAN II Boolean IF statements.

Figure 16. Built-In Functions

INDEX

C28-6390-3

Printed in U.S.A C28-6390-3

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