File No. 7090-25
Form C28-6054-4

IBM **Systems Reference Library**

IBM 7090/7094 Programming Systems
FORTRAN II Programming

This publication presents the FORTRAN II language and programming rules. The IBM FORmula TRANslating System, 7090/7094 FORTRAN, is an automatic coding system for the IBM 7090/7094 Data Processing System. FORTRAN II statements may be translated into machine language statements using either the FORTRAN II Processor operating under the 7090/7094 IBSYS Basic Monitor or the independent FORTRAN Monitor System. The FORTRAN language closely resembles the ordinary language of mathematics.

In this publication, all properties attributed to FORTRAN apply to the FORTRAN II System for the IBM 7090 and the IBM 7094.

Other publications covering the 7090/7094 FORTRAN II System are as follows:

IBM General Information Manual FORTRAN, Form F28-8074-1.
IBM 7090/7094 Programming Systems: FORTRAN II Operations, Form C28-6066-4.
IBM 709/7090 Programming Systems: FORTRAN Assembly Program (FAP), Form C28-6235.
IBM 32K 709/7090 FORTRAN: Adding Built-In Functions, Form J28-6135.
IBM 709/7090 Programming System: FORTRAN Input/Output Package for the 32K Version, Form J28-6190.

© 1958, 1959, 1961, 1963 by International Business Machines Corporation

THE FORTRAN SYSTEM

The IBM FORmula TRANslating System, 7090/7094 FORTRAN II, is an automatic coding system for the IBM 7090/7094 Data Processing System. More precisely, it is a 7090/7094 program which accepts a source program written in the FORTRAN II language, a language that closely resembles the ordinary language of mathematics, and which produces a machine language object program ready to be run on a 7090/7094.

7090/7094 FORTRAN II therefore, in effect, transforms the IBM 7090/7094 into a machine with which communication can be made in a language more concise and more familiar to the programmer than the machine language itself. The result is a substantial reduction in the training required to program, as well as in the time consumed in writing programs and eliminating errors from them.

FORTRAN II Processor

The FORTRAN II Processor is composed of a Compiler, an Assembler (see the publication, FORTRAN Assembly Program (FAP), Form C28-6235), and a Monitor (see Part III of this publication) which operates under the Basic Monitor (IBSYS). Under the Basic Monitor, compilations, assemblies, and binary object programs from compilations and assemblies may be executed as parts of a single job. In addition, input to the Basic Monitor may include jobs for other Processors (e.g., COBOL) as well as FORTRAN.

The FORTRAN II Processor may also operate as an independent system operating under the control of its own Monitor.

Among the features which characterize the FORTRAN II system are the following:

Object and Source Machines

7090/7094 FORTRAN II requires the following minimum machine configuration: a 32K IBM 7090 or IBM 7094 with a card reader, an on-line printer, one disk (for the System), and six tapes or seven tapes and one additional tape for Chain jobs. The object machine requires only the amount of core storage and I/O devices required by the object program. An IBM 709 may be used as the source and object machine if it is equipped with the data channel trap feature. Programs using READ DRUM and WRITE DRUM statements can be run only on a 709 equipped with 733 Magnetic Drums.

Efficiency of the Object Program

Object programs produced by FORTRAN will generally be as efficient as those written by experienced programmers.

Scope of Applicability

The FORTRAN language provides facilities for expressing any problem of numeric computation. In particular, problems containing large sets of formulas and many variables can be dealt with easily, and any variable may have up to three independent subscripts.

The language of FORTRAN may be expanded by the use of subprograms. These subprograms may be written in the FORTRAN or FAP language, and may be called by other FORTRAN or FAP main programs or subprograms.

Inclusion of Library Routines

Pre-written routines used to evaluate functions of any number of arguments can be made available for incorporation into object programs by the use of any of several different facilities provided for this purpose.

Provision for Input and Output

Certain statements in the FORTRAN language cause the inclusion of necessary input and output routines in the object program. Those routines that deal with decimal information include conversion to or from the internal machine language, and permit considerable freedom of format in the input and output of data.

Nature of FORTRAN Arithmetic

Arithmetic in an object program will generally be performed with single-precision floating point numbers. These numbers provide about eight decimal digits of precision, and may be zero or have magnitudes between approximately 10^{-38} and 10^{38}. Fixed point arithmetic for integers is also provided.

Double-precision and complex arithmetic are provided; see Chapter 9, Part II.

MAJOR REVISION (August, 1963)

This publication, Form C28-6054-4, is a major revision of the previous edition, Form C28-6054-3. It makes the previous edition, as well as Technical Newsletter N28-0061 to that edition, obsolete. This edition, however, is equivalent to the previous edition as modified by Technical Newsletter N28-0061.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Address comments concerning the contents of this publication to:
IBM Corporation, Programming Systems Publications, Dept. D91, PO Box 390, Poughkeepsie, N.Y.

CONTENTS

CHAPTER 1. GENERAL PROPERTIES OF A FORTRAN SOURCE PROGRAM

A FORTRAN source program consists of a sequence of source statements, of which there are 38 different types. These statement types are described in detail in the chapters which follow.

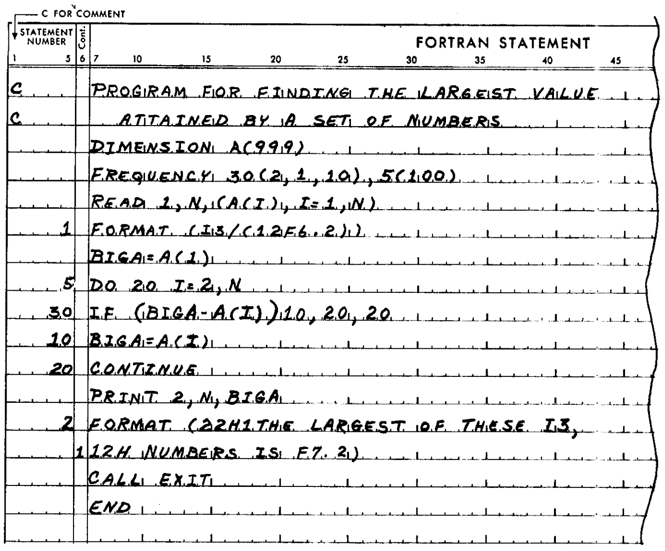
Example of a FORTRAN Program

The brief program shown in Figure 1-1 will serve to illustrate the general appearance and some of the properties of a FORTRAN program. It is shown as coded on a standard FORTRAN coding sheet.

The purpose of the program is to determine the largest value attained by a set of numbers, A_i, represented by the notation A(I), and to print the number on the attached printer. The numbers exist on punched cards, 12 to a card, each number occupying a field of six columns. The size of the set is variable, not exceeding 999 numbers. The actual size of the set is punched on the leading card and is the only number on that card.

Punching a Source Program

Each statement of a FORTRAN source program is punched into a separate card (the standard FORTRAN card form is shown in Figure 1-2); however, if a statement is too long to fit on one card, it can be continued on as many as nine "continuation cards." The order of the source statements is governed solely by the order of the normal source program statement sequencing given in Appendix A.


C FOR COMMENT
STATEMENT NUMBER
Cont.
FORTRAN STATEMENT
1 5 6 7 10 15 20 25 30 35 40 45

```
C     PROGRAM FOR FINDING THE LARGEST VALUE
C        ATTAINED BY A SET OF NUMBERS
      DIMENSION A(999)
      FREQUENCY 30(2,1,10), 5(100)
      READ 1,N,(A(I),I=1,N)
    1 FORMAT (I3/(12F6.2))
      BIGA=A(1)
    5 DO 20 I=2,N
   30 IF (BIGA-A(I))10,20,20
   10 BIGA=A(I)
   20 CONTINUE
      PRINT 2,N,BIGA
    2 FORMAT (22H1THE LARGEST OF THESE I3,
     112H NUMBERS IS F7.2)
      CALL EXIT
      END
```

Figure 1-1

Cards that contain a C in column 1 are not processed by the FORTRAN program. Therefore, such cards may be used to carry comments that will appear when the source program deck is listed.

Numbers less than 32,768 may be punched in columns 1-5 of the initial card of a statement. When such a number appears in these columns, it becomes the statement number of the statement. These statement numbers permit cross references within a source program and, when necessary, facilitate the correlation of source and object programs.

Column 6 of the initial card of a statement must be left blank or punched with a zero. Continuation cards (other than for comments), however, must

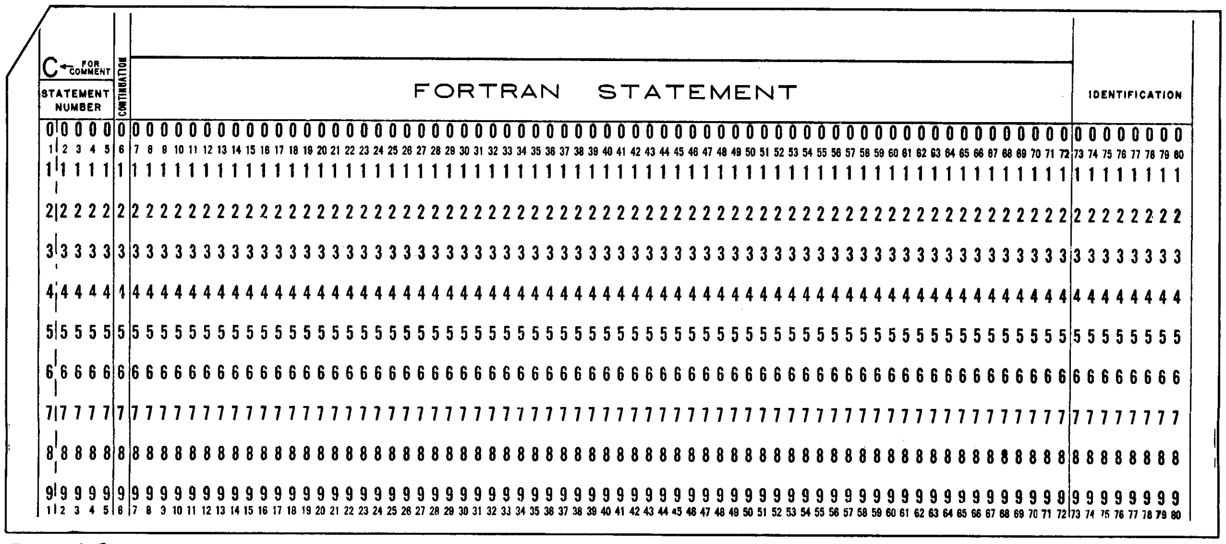


Figure 1-2

have column 6 punched with some character other than zero, and may be punched with numbers from 1 through 9. Continuation cards for comments need not be punched in column 6; only the C in column 1 is necessary.

The statements themselves are punched in columns 7-72, both on initial cards and on continuation cards. Thus, a statement may consist of not more than 660 characters (i.e., ten cards). A table of the admissible characters for FORTRAN II is given in Appendix B. Blank characters, except in column 6 and in certain fields of FORMAT statements, are simply ignored by FORTRAN, and may be freely used to improve the readability of the source program listing.

Columns 73-80 are not processed by FORTRAN; therefore, they may be punched with any desired identifying information.

The input to FORTRAN may be either the deck of source statement cards or a BCD tape prepared on off-line card-to-tape equipment. On such a tape, an end-of-file mark is required after the last card.

Types of FORTRAN Statements

The 38 types of source statements that can be used in a FORTRAN program may be classified as follows:

1. The arithmetic statement specifies a numeric computation. Part I, Chapters 2 and 3, discusses the symbols available for referring to constants, variables and functions; Part II, Chapter 4, discusses the combining of these constants, variables, and functions into arithmetic statements.
2. The 15 control statements govern the flow of control in the program. These, plus the END statement, are discussed in Part II, Chapter 5.
3. The four subprogram statements enable the programmer to define and use subprograms. The method for utilizing subprograms is discussed in Part II, Chapter 6.
4. The 13 input/output statements provide the necessary input and output routines. These statements are discussed in Part II, Chapter 7.
5. The four specification statements provide information required or desired to make the object program efficient. These are discussed in Part II, Chapter 8.

CHAPTER 2. CONSTANTS, VARIABLES, SUBSCRIPTS, AND EXPRESSIONS

As required of any programming language, FORTRAN provides a means of expressing numeric constants and variable quantities. In addition, a subscript notation is provided for expressing one-, two-, or three-dimensional arrays of variables.

Constants

Two types of constants are defined in the FORTRAN II source program language: fixed point (restricted to integers) and floating point (characterized by being written with a decimal point).

Fixed Point Constants

GENERAL FORM	EXAMPLES
A fixed point constant consists of 1 to 5 decimal digits. A preceding + or - sign is optional. The magnitude, or absolute value, of the constant must be less than 2^{17}.	3 +1 -28987

Where a fixed point constant is used for the value of a subscript, it is treated modulo the size of core storage.

Floating Point Constants

GENERAL FORM	EXAMPLES
A floating point constant consists of any number of decimal digits, with a decimal point at the beginning, at the end, or between two digits. A preceding + or - sign is optional.	17. 5.0 -.0003
A decimal exponent preceded by an E may follow a floating point constant.	5.0E3 (5.0×10^3) 5.0E+3 (5.0×10^3) 5.0E-7 (5.0×10^{-7})
The magnitude of a floating point constant must lie between the approximate limits of 10^{-38} and 10^{38}, or be zero.	

Variables

Two types of variables are defined in FORTRAN II: fixed point (restricted to integral values) and floating point. References to variables are made in the FORTRAN source language by symbolic names consisting of alphabetic and, if desired, numeric characters.

Fixed Point Variables

GENERAL FORM	EXAMPLES
A fixed point variable consists of 1 to 6 alphabetic or numeric characters (not special characters); the first character must be an I, J, K, L, M, or N.	I M2 JOBNO

A fixed point variable can assume any integral value, provided the magnitude is less than 2^{17}. Values used for subscripts, however, are treated modulo the size of core storage.

To avoid the possibility of a variable being considered a function by FORTRAN, the following two rules should be observed with respect to the naming of variables:

1. A variable cannot be given a name that coincides with the name of a function without its terminal F. For example, if a function is named TIMEF, no variable should be named TIME.

2. Unless their names are less than four characters in length, subscripted variables (see below) must not be given names ending with F, because FORTRAN will consider variables so named to be functions.

Floating Point Variables

GENERAL FORM	EXAMPLES
A floating point variable consists of 1 to 6 alphabetic or numeric characters (not special characters); the first character must be alphabetic but not I, J, K, L, M, or N.	A B7 DELTA

A floating point variable can assume any value expressible as a normalized floating point number, i.e., zero or any number whose magnitude lies between approximately 10^{38} and 10^{-38}.

The rules for naming fixed point variables also apply to floating point variables.

Subscripts

A variable can be made to represent any element of a one-, two-, or three-dimensional array of quantities by appending one, two, or three subscripts to it, respectively. The variable is then a subscripted variable. These subscripts are fixed point quantities whose values determine the member of the array to which reference is made.

GENERAL FORM	EXAMPLES
Let v represent any fixed point variable and c or c' any unsigned fixed point constant; then, a subscript is an expression in one of the forms: v c v+c or v-c c*v c*v+c' or c*v-c' (The symbol * denotes multiplication.)	I 3 MU+2 MU-2 5*J 5*J+2 5*J-2 3+K Invalid

The variable in a subscript must not itself be subscripted.

Subscripted Variables

GENERAL FORM	EXAMPLES

A subscripted variable is a fixed or floating point variable, followed by parentheses enclosing one, two, or three subscripts which are separated by commas.	A(I)
	K(3)
	BETA (5*J-2, K+2, L)

Each variable that appears in subscripted form must have the size of its array, i.e., the maximum values that its subscripts can attain specified in a DIMENSION statement preceding the first appearance of the variable in the source program.

The value of a subscript exclusive of its addend, if any, must be greater than zero and not greater than the corresponding array dimension.

Arrangement of Arrays in Storage

If an array, A, is two-dimensional, it will be stored sequentially in the order $A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, A_{2,2}, \ldots, A_{m,2}, \ldots, A_{m,n}$. Arrays are stored columnwise, with the first of their subscripts varying most rapidly and the last varying least rapidly. The same is true of three-dimensional arrays. Arrays that are one-dimensional are stored sequentially.

All arrays are stored backwards, i.e., in the order of decreasing absolute storage locations.

Expressions

A FORTRAN expression is any sequence of constants, variables (subscripted or not subscripted), and functions separated by operation symbols, commas, and parentheses. The formation of expressions must conform to the rules for constructing expressions.

The operation symbols +, -, *, /, and ** denote addition, subtraction, multiplication, division, and exponentiation, respectively, in arithmetic type operations.

Rules for Constructing Expressions

1. Since constants, variables, and functions may be fixed point or floating point, expressions may also be fixed point or floating point; however, these modes must not be mixed. This does not mean that a floating point constant, variable, or function cannot appear in a fixed point expression, etc., but rather that a quantity of one mode can

appear in an expression of another mode only in the following ways:

a. Fixed point expressions may contain floating point quantities only as arguments of a function.

b. Floating point expressions may contain fixed point quantities only as function arguments, subscripts, and exponents.

2. Constants and variables are expressions of the same mode as the constant or variable name. For example, the fixed point variable name J53 is a fixed point expression.

3. Functions are expressions of the same mode as the function name, provided that the arguments of the function are in the modes assumed in the definition of the function. For example, if SOMEF(A, B) is a function with a floating point name, then SOMEF(C, D) is a floating point expression if C and D are of the same modes as A and B, respectively.

4. Exponentiation of an expression does not affect the mode of the expression; however, a fixed point expression may not be given a floating point exponent.

Note: 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 + or - does not affect the mode of the expression produced. For example, E, +E, and -E are all expressions of the same mode.

6. Enclosing an expression in parentheses does not affect the mode of the expression. For example, A, (A), ((A)), and (((A))) are all expressions of the same mode.

7. Expressions may be connected by operators to form more complex expressions, provided:

a. no two operators appear in sequence, and

b. items so connected are all of the same mode.

Hierarchy of Operations

When the hierarchy of operations in an expression is not explicitly specified by the use of parentheses, it is understood by FORTRAN to be in the following order (from innermost operations to outermost operations):

**	Exponentiation
* and /	Multiplication and Division
+ and -	Addition and Subtraction

For example, the expression

A+B/C+D**E*F-G

will be taken to mean

$A+(B/C)+(D^E*F)-G$

Ordering Within a Hierarchy

Parentheses that have been omitted from a sequence of consecutive multiplications, consecutive divisions, consecutive additions, or consecutive subtractions will be understood to be grouped from the left. Thus, if · represents either * or / or + or - , then

A · B · C · D · E

will be taken by FORTRAN to mean

((((A · B) · C) · D) · E)

Optimization of Arithmetic Expressions

The efficiency of instructions compiled from arithmetic expressions may also be influenced by the way expressions are written. The section entitled "Optimization of Arithmetic Expressions" in Part IV, Chapter 16, mentions some of the considerations which affect object program efficiency.

Rules for Constructing Boolean Expressions

FORTRAN arithmetic expressions may be interpreted as Boolean expressions in which the arithmetic operators are treated as logical operators. To obtain this interpretation, the character B must appear in column 1 of the Boolean arithmetic statement. The following rules apply:

1. The operation symbols +, *, and - denote the operators <u>or</u>, <u>and</u> and <u>complement</u>, respectively. (The symbols / and ** are not defined for Boolean expressions.)

2. The operator * has greater binding strength than the operator +. (It is higher in the hierarchy of operations.) Because - is a unary operator it is part of the expression or symbol to which it applies. Thus, when a Boolean expression is to be complemented it must be enclosed in parentheses if it is a part of a larger expression. For example, A - B is not permitted, whereas A+(-B) is permitted.

3. In accordance with the logical usage of the expression, and to simplify the construction of masks and logical constants, constants in Boolean expressions are considered to be octal numbers. Constants must consist of no more than 12 octal digits; if there are fewer than 12, then the number will be right-adjusted. Blanks are ignored; they are not treated as zero.

4. All variables used in arithmetic statements that contain Boolean expressions must have floating point names.

5. Variable names can be subscripted in the normal FORTRAN manner.

6. All Boolean operations are performed upon the full 36-bit logical word.

CHAPTER 3. FUNCTIONS

A subroutine is considered to be any sequence of instructions which performs some desired operation. Subroutines may be function-type or subprogram-type, each type being further subdivided. This chapter contains a discussion of the four function-types which may be utilized in FORTRAN. To clarify the meaning and use of functions, they will be shown in their relation to subroutine-types as a whole. The interrelationship of the various subroutines is as follows:

FORTRAN Subroutines	Method of Calling Subroutine	Method of Naming Subroutine	Method of Defining Subroutine
Library Functions			
Built-in Functions		function	
Arithmetic Statement Functions		type	
FUNCTION Subprograms		subprogram	
SUBROUTINE Subprograms		type	

Thus, from the way they are called, or used, there are four subroutine-types (i.e., the functions) which are alike. Whereas three of these are named according to the same rules, each of the four is given its meaning (i.e., it is defined) in a different manner. The fifth subroutine-type, SUBROUTINE subprogram, is called, or used, by means of a CALL statement; however, it is named and defined in much the same manner as the FUNCTION subprogram.

Calling

As indicated in the schematic, there are two distinct ways of referencing subroutines. One type of reference is by means of an arithmetic expression. This applies to the four functions: Library, Built-In, Arithmetic Statement function, and FUNCTION subprogram. The other type of reference, which applies to SUBROUTINE subprograms, is by means of a CALL statement (discussed later).

Following are examples of arithmetic expressions that include function names:

```
Y = A - SINF(B-C)
C = MIN0F(M, L)+ABC(B*FORTF(Z), E)
```

The names of Library, Built-In, and Arithmetic Statement functions, and of FUNCTION subprograms are all used in this way. The appearance of a function name in the arithmetic expression calls the function; the value of the function is then computed, using the arguments that are supplied in the parentheses following the function name. Only one value is produced by these four functions, whereas the SUBROUTINE subprogram may produce many values. A value is here defined to be a single numeric quantity.

Naming

The following paragraphs describe the rules for naming Library, Built-In, and Arithmetic Statement functions, and FUNCTION subprograms:

Naming of Library, Built-In, and Arithmetic Statement Functions

GENERAL FORM	EXAMPLES
The name of a function consists of 4 to 7 alphabetic or numeric characters (not special characters); the last character must be an F and the first must be alphabetic. Further, the first must be an X if, and only if, the value of the function is to be fixed point. The name of the function is followed by parentheses enclosing the arguments, which are separated by commas.	ABSF (B) XMODF (M/N, K) COSF (A) FIRSTF (Z + B, Y)

Mode of a Function and Its Arguments: Consider a function of a single argument. It may be desired to state the argument either in fixed point or in floating point; similarly, the function itself may be in either of these modes. Thus, a function of a single argument has four possible mode configurations. In general, a function of n arguments will have 2^{n+1} mode configurations.

A separate name must be given and a separate routine must be available for each of the mode configurations that is used. Thus, a complete set of names for a given function might be:

SOMEF	Fixed argument, floating function
SOME0F	Floating argument, floating function
XSOMEF	Fixed argument, fixed function
XSOME0F	Floating argument, fixed function

The Xs and Fs are mandatory, but the rest of the naming is arbitrary.

Naming of FUNCTION Subprograms

Although these functions are referred to by arithmetic expressions in the same manner as the previous three types, the rules for naming them are different. Except for the fact that no name of a FUNCTION subprogram which is four to six characters long may end in F, these functions are named in exactly the same way as ordinary variables of the program. This means that the name of a fixed point FUNCTION subprogram must have I, J, K, L, M, or N for its first character.

Further details on naming FUNCTION subprograms are given later.

Definition

Each of the four types of functions is defined in a different way.

Built-In Functions

The FORTRAN II System, as distributed, contains 20 Built-In functions. It also has the capacity for ten more Built-In functions. The additional functions may be inserted into the system by the particular installation.

Following are the 20 functions that are compiled as Built-In functions into the arithmetic statement which calls them. These functions are called "open" since they appear in the object program each time they are referred to in the source program.

Type of Function	Definition	No. of Args.	Name	Mode of	
				Argument	Function
Absolute value	|Arg|	1	ABSF	Floating	Floating
			XABSF	Fixed	Fixed
Truncation	Sign of Arg times largest integer ≤|Arg|	1	INTF	Floating	Floating
			XINTF	Floating	Fixed
Remaindering (see note below)	Arg_1 (mod Arg_2)	2	MODF	Floating	Floating
			XMODF	Fixed	Fixed
Choosing largest value	Max (Arg_1, Arg_2,...)	≥ 2	MAX0F	Fixed	Floating
			MAX1F	Floating	Floating
			XMAX0F	Fixed	Fixed
			XMAX1F	Floating	Fixed
Choosing smallest value	Min (Arg_1, Arg_2,...)	≥ 2	MIN0F	Fixed	Floating
			MIN1F	Floating	Floating
			XMIN0F	Fixed	Fixed
			XMIN1F	Floating	Fixed
Float	Floating a fixed number	1	FLOATF	Fixed	Floating
Fix	Same as XINTF	1	XFIXF	Floating	Fixed
Transfer of sign	Sign of Arg_2 times $\|Arg_1\|$	2	SIGNF	Floating	Floating
			XSIGNF	Fixed	Fixed
Positive difference	Arg_1 - Min (Arg_1, Arg_2)	2	DIMF	Floating	Floating
			XDIMF	Fixed	Fixed

NOTE: The function MODF (Arg_1, Arg_2) is defined as $Arg_1 - [Arg_1 / Arg_2]\, Arg_2$, where[x] = integral part of x.

Library Functions

The Library functions are pre-written and may exist on the library tape or in prepared card decks. These functions constitute "closed" subroutines, i.e., instead of appearing in the object program for every reference that has been made to them in the source program, they appear only once, regardless of the number of references.

Hand-coded Library functions may be added to the library. Rules for coding these subroutines are given in Appendix D; those for adding them to the library are included in the FORTRAN II Operations Manual, Form C28-6066-4.

Seven Library functions are included in the FORTRAN II System. These are:

Name	Function
LOGF	Natural Logarithm
SINF	Trigonometric Sine
COSF	Trigonometric Cosine
EXPF	Exponential
SQRTF	Square Root
ATANF	Arctangent
TANHF	Hyperbolic Tangent

Arithmetic Statement Functions

Arithmetic Statement functions are defined by a single FORTRAN arithmetic statement and apply only to the particular program or subprogram in which their definition appears.

GENERAL FORM	EXAMPLES
"a=b", where a is a function name followed by parentheses enclosing its arguments; the arguments must be distinct nonsubscripted variables separated by commas. b is an expression which does not involve subscripted variables. Any functions appearing in b must be available to the program or must have been defined by preceding arithmetic statements.	FIRSTF (X) = A*X+B SECONDF (X,B) = A*X+B THIRDF (D) = FIRSTF (E)/D FOURTHF (F,G) = SECONDF (F, THIRDF (G)) FIFTHF (I,A) = 3.0*A**I SIXTHF (J) = J + K XSIXTHF (J) = J + K

Just as with the other functions, the answer will be expressed in fixed or floating point mode according to whether the name does or does not begin with X.

The right-hand side of an Arithmetic Statement function may be any expression, not involving subscripted variables, that meets the requirements specified for expressions.

In particular, it may involve functions freely, provided that any such function, if it is not built-in or available on the master tape, has been defined in a preceding function statement.

Of course, no function can be used as an argument of itself.

As many as desired of the variables appearing in the expression on the right-hand side of the Arithmetic Statement function may appear on the left-hand side as the arguments of the function. Since the arguments are really only dummy variables, their names are unimportant (except as indicating fixed or floating point mode) and may even be the same as names appearing elsewhere in the program.

Those variables on the right-hand side which are not stated as arguments are treated as parameters. Thus, if FIRSTF is defined in a function statement as FIRSTF(X) = A*X+B, then based on the current values of A, B, and Y, a later reference to FIRSTF(Y) will cause AY+B to be computed. The naming of parameters, therefore, must follow the normal rules of uniqueness.

A function defined as an Arithmetic Statement function may be used in the same manner as any other function. In particular, its arguments may be expressions and may involve subscripted variables; thus, a reference to FIRSTF(Z + Y(I)), with the above definition of FIRSTF, will cause a $(z+y_i) + b$ to be computed on the basis of the current values of a, b, y_i, and z. Functions defined by arithmetic statements are always compiled as closed subroutines.

NOTE: All the arithmetic statements defining functions to be used in a program must precede the first executable statement of the program.

FUNCTION Subprograms

This class of functions covers those subroutines which cannot be defined by only one arithmetic statement but may not be utilized frequently enough to warrant a place on the library tape; however, they may be placed on the library tape.

They are called FUNCTION subprograms because they may be conveniently defined by a conventional FORTRAN program. In this instance, compiling a FORTRAN program produces a FUNCTION subprogram in exactly the form required for object program execution.

Since FUNCTION and SUBROUTINE subprograms are defined in the same way, a discussion of the definition of FUNCTION subprograms is included in Part II, Chapter 6.

CHAPTER 4. THE ARITHMETIC STATEMENT

Arithmetic Statement

GENERAL FORM	EXAMPLES

"a=b", where a is a variable (subscripted or nonsubscripted) and b is an expression.	Q1 = K
	A(I)=B(I)+ SINF(C(I))

The arithmetic statement defines a numeric calculation. A FORTRAN arithmetic statement very closely resembles a conventional arithmetic formula. However, in a FORTRAN arithmetic statement, the = sign specifies replacement rather than equivalence. Thus, the arithmetic statement

Y = N-LIMIT (J-2)

means that the value of N-LIMIT (J-2) is to be stored in Y. The result is stored in fixed point or floating point form if the variable to the left of the = sign is a fixed point or floating point variable, respectively.

If the variable on the left is fixed point and the expression on the right is floating point, the result will first be computed in floating point and then truncated and converted to a fixed point integer. Thus, if the result is +3.872, the fixed point number stored will be +3, not +4. If the variable on the left is floating point and the expression on the right is fixed point, the latter will be computed in fixed point and then converted to floating point.

Examples of Arithmetic Statements

A = B	Store the value of B in A.
I = B	Truncate B to an integer, convert it to fixed point, and store it in I.
A = I	Convert I to floating point, and store it in A.
I = I+1	Add 1 to I and store it in I. This example illustrates the fact that an arithmetic formula is not an equation but a command to replace a value.
A = 3.0*B	Replace A by 3B.
A = 3*B	Not permitted. The expression is mixed, i.e., contains both fixed point and floating point variables.
A = I*B	Not permitted. The expression is mixed.

A Boolean arithmetic statement is an arithmetic statement in which b is a Boolean expression.

Examples of Boolean Arithmetic Statements

Col 1	Cols 7-72	Explanation
B	D=A*(-(B+C))	The inner pair of parentheses is required to indicate the scope of complementation. The outer pair of parentheses is required because the expression -(B+C) is a part of a larger expression.
B	D=-IMPF(-B, -C)	No additional parentheses are required here because the function name, as well as the argument names, are not parts of a larger expression.
B	X=X*777777000000	The constant is being used here to "mask out" the right half of word X.

CHAPTER 5. CONTROL STATEMENTS AND END STATEMENT

The FORTRAN control statements enable the programmer to state the flow of his program.

Unconditional GO TO Statement

GENERAL FORM	EXAMPLES
"GO TO n", where n is a statement number.	GO TO 3

This statement causes transfer of control to the statement with statement number n.

Computed GO TO Statement

GENERAL FORM	EXAMPLES
"GO TO (n_1, n_2,..., n_m), i", where n_1, n_2,....., n_m are statement numbers and i is a nonsubscripted fixed point variable.	GO TO (30, 42, 50, 9), I

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

This statement is used to obtain a computed many-way fork.

Assigned GO TO Statement

GENERAL FORM	EXAMPLES
"GO TO n, ($n_1, n_2, \ldots, n_m$)", where n is a nonsubscripted fixed point variable appearing in a previously executed ASSIGN statement, and $n_1, n_2, \ldots, n_m$ are statement numbers.	GO TO K, (17, 12, 19)

This statement causes transfer of control to the statement whose statement number is equal to that value of n which was last assigned by an ASSIGN statement; $n_1, n_2, \ldots, n_m$ are a list of the values which n may have assigned.

The assigned GO TO is used to obtain a pre-set many-way fork. When an assigned GO TO exists in the range of a DO, there is a restriction on the values of $n_1, n_2, \ldots, n_m$. (See the discussion of the DO statement.)

ASSIGN Statement

GENERAL FORM	EXAMPLES
"ASSIGN i TO n", where i is a statement number and n is a non-subscripted fixed point variable which appears in an assigned GO TO statement.	ASSIGN 12 TO K

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

IF Statement

GENERAL FORM	EXAMPLES
"IF (a) n_1, n_2, n_3", where a is an expression and n_1, n_2, n_3 are statement numbers.	IF(A(J, K)-B)10, 4, 30

Control is transferred to the statement numbered n_1, n_2, or n_3 if the value of a is less than, equal to, or greater than zero, respectively.

SENSE LIGHT Statement

GENERAL FORM	EXAMPLES
"SENSE LIGHT i", where i is 0, 1, 2, 3, or 4.	SENSE LIGHT 3

If i is 0, all Sense Lights will be turned Off; otherwise, only Sense Light i will be turned On.

IF (SENSE LIGHT) Statement

GENERAL FORM	EXAMPLES
"IF (SENSE LIGHT i) n_1, n_2", where n_1 and n_2 are statement numbers and i is 1, 2, 3, or 4.	IF (SENSE LIGHT 3) 30, 40

Control is transferred to the statement numbered n_1 or n_2 if Sense Light i is On or Off, respectively. If the light is On, it will be turned Off.

IF (SENSE SWITCH) Statement

GENERAL FORM	EXAMPLES
"IF (SENSE SWITCH i) n_1, n_2", where n_1 and n_2 are statement numbers and i is 1, 2, 3, 4, 5, or 6.	IF (SENSE SWITCH 3) 30, 108

Control is transferred to the statement numbered n_1 or n_2 if Sense Switch i is Down or Up, respectively.

IF ACCUMULATOR OVERFLOW Statement

GENERAL FORM	EXAMPLES
"IF ACCUMULATOR OVERFLOW n_1, n_2", where n_1 and n_2 are statement numbers.	IF ACCUMULATOR OVERFLOW 30, 49

IF QUOTIENT OVERFLOW Statement

GENERAL FORM	EXAMPLES
"IF QUOTIENT OVERFLOW n_1, n_2", where n_1 and n_2 are statement numbers.	IF QUOTIENT OVERFLOW 30, 49

Control is transferred to the statement numbered

n_1 if an overflow condition is present in either the Accumulator or the Multiplier-Quotient Register as a result of floating point arithmetic, and to n_2 if no overflow is present. That is, in 7090/7094 FORTRAN II, programming either of these statements is equivalent to programming a non-FORTRAN statement, IF OVERFLOW n_1, n_2. In 7090/7094 FORTRAN II, an internal indicator is used to denote the overflow condition; it is reset to the no-overflow condition after execution of either of these two statements.

When either the Accumulator or the Multiplier-Quotient Register overflows, the register is set to contain the highest possible quantity, i.e., 377777777777_8, with the correct sign.

If an underflow occurs in either register, that register is set to zero and the sign remains unchanged. There is no test for the underflow condition.

IF DIVIDE CHECK Statement

GENERAL FORM	EXAMPLES
"IF DIVIDE CHECK n_1, n_2", where n_1 and n_2 are statement numbers.	IF DIVIDE CHECK 84, 40

Control is transferred to the statement numbered n_1 or n_2, if the Divide Check trigger is On or Off, respectively. If it is On, it will be turned Off.

DO Statement

GENERAL FORM	EXAMPLES
"DO n i = m_1, m_2" or "DO n i = m_1, m_2, m_3", where n is a statement number, i is a nonsubscripted fixed point variable, and m_1, m_2, m_3 are each either an unsigned fixed point constant or a nonsubscripted fixed point variable. If m_3 is not stated, it is assumed to be 1.	DO 30 I = 1, 10 DO 30 I = 1, M, 3

The DO statement is a command to execute repeatedly the statements which follow, up to and including the statement with statement number n. The first time, the statements are executed with i = m_1. For each succeeding execution, i is increased by m_3. After they have been executed with i equal to the highest value in this sequence of values which does not exceed m_2, control passes to the statement following the last statement in the range of the DO.

The range of a DO is that set of statements which will be executed repeatedly; i.e., it is the sequence of consecutive statements immediately following the DO, up to and including the statement numbered n.

The index of a DO is the fixed point variable i, which is controlled by the DO in such a way that its value begins at m_1 and is increased each time by m_3 until it is about to exceed m_2. Throughout the range it is available for computation, either as an ordinary fixed point variable or as the variable of a subscript. After the last execution of the range, the DO is said to be satisfied.

Suppose, for example, that control has reached statement 10 of the program

```
 .
 .
 .
10 DO 11 I = 1, 10
11 A(I) = I * N(I)
12
 .
 .
 .
```

The range of the DO is statement 11, and the index is I. The DO sets I to 1 and control passes into the range. The value of 1·N(1) is computed, converted to floating point, and stored in location A(1). Since statement 11 is the last statement in the range of the DO and the DO is unsatisfied, I is increased to 2 and control returns to the beginning of the range, statement 11. The value of 2·N(2) is then computed and stored in location A(2). The process continues until statement 11 has been executed with I = 10. Since the DO is satisfied, control then passes to statement 12.

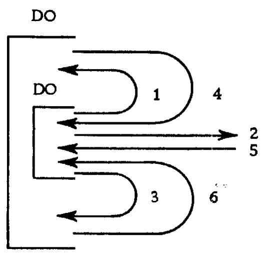
DOs within DOs: Among the statements in the range of a DO may be other DO statements. When this is so, the following rule must be observed:

Rule 1: 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.

A set of DOs satisfying this rule is called a nest of DOs.

Transfer of Control and DOs: Transfers of control from and into the range of a DO are subject to the following rule:

Rule 2: No transfer is permitted into the range of any DO from outside its range. Thus, in the configuration below, 1, 2, and 3 are permitted transfers, but 4, 5, and 6 are not.



Exception: There is one situation in which control can be transferred into the range of a DO from outside its range. Suppose control is in the range of the innermost DO of a nest of DOs which are completely nested (i.e., every pair of DOs in the nest is such that one contains the other). Suppose also that control is transferred to a section of the program, completely outside the nest to which these DOs belong, which makes no change in any of the indexes or indexing parameters (m's) in the nest. Then, after the execution of this latter section of the program, control can be transferred back to the range of the same innermost DO from which it originally came. This provision makes it possible to exit temporarily from the range of some DOs to execute a subroutine.

Restriction on Assigned GO TOs in the Range of a DO: When an assigned GO TO is in the range of a DO, the statements to which it may transfer must all be in the exclusive range of a single DO or all outside the DO nest.

Preservation of Index Values: When control leaves the range of a DO in the ordinary way, i.e., when the DO becomes satisfied and control passes on to the next statement after the range, the exit is said to be a normal exit. After a normal exit from a DO occurs, the value of the index controlled by that DO is not defined, and the index cannot be used again until it is redefined. (In this connection, see "Further Details about DO Statements.")

However, if exit occurs by a transfer out of the range by an IF or GO TO statement, the current value of the index remains available for any subsequent use. If exit occurs by a transfer which is in the ranges of several DOs, the current values of all the indexes controlled by those DOs are preserved for any subsequent use.

Restrictions on Statements in the Range of a DO: Only one type of statement is not permitted in the range of a DO, namely, any statement that redefines the value of the index or of any of the indexing parameters (m's). In other words, the indexing of a DO loop must be completely set before the range is entered.

The first statement in the range of a DO must not be one of the nonexecutable FORTRAN statements. The range of a DO cannot end with a transfer.

Exits: When a subroutine reference is executed in the range of a DO, care must be taken that the called subprogram does not alter the DO index or indexing parameters. Such an exit from a DO is not considered a transfer; thus, the current values of the indexes are not available for computation.

CONTINUE Statement

GENERAL FORM	EXAMPLES
"CONTINUE"	CONTINUE

CONTINUE is a dummy statement which 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 which are intended to begin another repetition of the DO range.

As an example of a program which requires a CONTINUE, consider the table search:

```
 .
 .
 .
10 DO 12 I = 1, 100
   IF (ARG - VALUE (I))12, 20, 12
12 CONTINUE
 .
 .
 .
```

This program will scan the 100-entry VALUE table until it finds an entry which equals the value of the variable ARG, whereupon it exits to statement 20 with the value of I available for fixed point use; if no entry in the table equals the value of ARG, a normal exit to the statement following the CONTINUE will occur.

PAUSE Statement

GENERAL FORM	EXAMPLES
"PAUSE" or "PAUSE n", where n is an unsigned octal fixed point constant.	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 FORTRAN statement.

STOP Statement

GENERAL FORM	EXAMPLES
"STOP" or "STOP n", where n is an unsigned octal fixed point constant.	STOP STOP 77777

Neither STOP nor PAUSE should be used in a source program to be executed under the FORTRAN Monitor or Basic Monitor (IBSYS).

This statement causes a halt in such a way that depressing the Start key has no effect. Therefore, in contrast to PAUSE, this statement is used where a terminal, rather than a temporary stop, is desired.

The octal number n is positioned in the address field of the Storage Register. If n is not specified, it is understood to be zero.

END Statement

GENERAL FORM	EXAMPLES

"END (I_1, I_2, I_3, I_4, I_5)", where I is 0, 1, or 2.	END (2, 2, 2, 2, 2)
	END (1, 2, 0, 1, 1)

This statement differs from the previous statements discussed in this chapter in that it does not affect the flow of control in the object program being compiled. Its application is to the FORTRAN executive program during compilation:

1. FORTRAN provides the option of running under monitor control, which allows the compilation of a number of separate FORTRAN source programs in succession. The END statement, then, marks the end of any given FORTRAN source program, separating it from the program that follows.
2. The END statement specifies the treatment of the setting of Sense Switches 1 through 5.
3. The END statement must be the physically last statement of a program. The statement may be omitted only for single program compilations. When the END statement is omitted, all I_n are assumed equal to 2. For each I of the statement's list, see below.

I = 0	Ignore actual sense switch setting. Assume it to be Up.
I = 1	Ignore actual sense switch setting. Assume it to be Down.
I = 2	Note actual sense switch setting and act accordingly.

The END statement does not, of course, physically change the setting of a sense switch.

The sense switch options are given in Appendix C.

CHAPTER 6. SUBPROGRAM STATEMENTS

It is possible, in the FORTRAN language, to program subroutines that are referred to by other programs. These subroutines may, in turn, refer to still other lower level subroutines that may also be coded in FORTRAN language. It is possible, therefore, by means of FORTRAN, to code problems using several levels of subroutines. This configuration may be thought of as a total problem consisting of one main program and any number of subprograms.

Because of the interrelationship among several different programs, it is possible to include a block of hand-coded instructions in a sequence, including instructions compiled from FORTRAN source programs. It is only necessary that hand coded instructions conform to rules for subprogram formation, since they will constitute a distinct subprogram.

This chapter presents a discussion of the two types of FORTRAN coded subprograms: the FUNCTION subprogram and the SUBROUTINE subprogram. Four statements, described subsequently, are necessary for their definition and use. Two of these, SUBROUTINE and FUNCTION, are discussed in Section A; the other two, CALL and RETURN, are discussed in Section B.

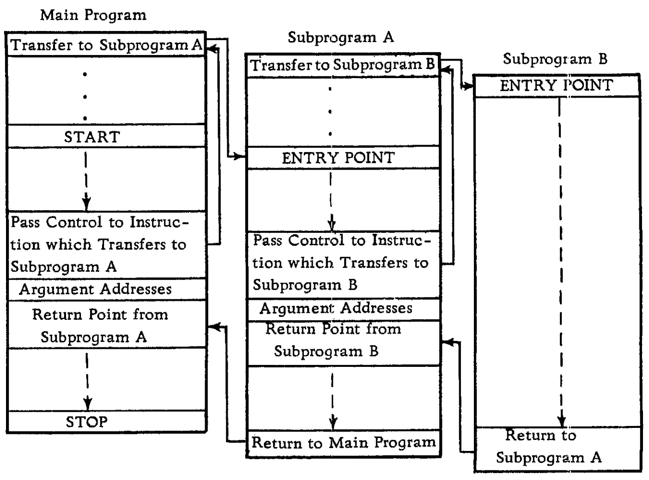
Illustrations of, and the rules for, hand-coding subprograms are given in Appendix D.

Although FUNCTION subprograms and SUBROUTINE subprograms are treated together and may be viewed as similar, it must be remembered that they differ in two fundamental respects.

1. The FUNCTION subprogram is always single-valued, whereas the SUBROUTINE subprogram may be multi-valued.
2. The FUNCTION subprogram is called or referred to by the arithmetic expression containing its name; the SUBROUTINE subprogram can only be referred to by a CALL statement.

Each of these two types of subprograms, when coded in FORTRAN language, must be regarded as independent FORTRAN programs. They conform in all respects to rules for FORTRAN programming. However, they may be compiled with the main program, of which they are parts, by means of multiple program compilation. In this way, the results of a multiple program compilation will be a complete main program-subprogram sequence, ready to be executed.

Schematically, the relationship among nested main programs and subprograms can be shown as follows. This diagram also indicates the main division of the internal structure of each program.



Section A: FUNCTION and SUBROUTINE Statements

FUNCTION Statement

GENERAL FORM	EXAMPLES
"FUNCTION Name (a_1, a_2, ..., a_n)", where Name is the symbolic name of a single-valued function, and the arguments a_1, a_2, ..., a_n, of which there must be at least one, are non-subscripted variable names or the dummy name of a SUBROUTINE or FUNCTION subprogram. The function name consists of 1 to 6 alphameric characters; the first character must be alphabetic. The first character must be I, J, K, L, M, or N if, and only if, the value of the function is to be fixed point; the final character must not be F if there are more than three characters in the name.	FUNCTION ARCSIN (RADIAN) FUNCTION ROOT (B, A, C) FUNCTION INTRST (RATE, YEARS)

The FUNCTION statement must be the first statement of a FUNCTION subprogram and defines it to be such.

In a FUNCTION subprogram, the name of the function must appear at least once as the variable on the left-hand side of an arithmetic statement, or alternately in an input statement list, e.g.:

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

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

This type of program may be either compiled independently or it may be multiple-compiled with others. A FUNCTION subprogram must never be inserted between two statements of any other single program.

The arguments following the name in the FUNCTION statement may be considered "dummy" variable names. That is, during object program execution, other actual arguments are substituted for them. Therefore, the arguments which follow the function reference in the calling program must agree with those in the FUNCTION statement in the subprogram in number, order, and mode. Furthermore, when a dummy argument is an array name, the corresponding actual argument must also be an array name. Each of these array names must appear with the same dimensions in DIMENSION statements of their respective programs.

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

SUBROUTINE Statement

GENERAL FORM	EXAMPLES
"SUBROUTINE Name (a_1, a_2, ..., a_n)", where Name is the symbolic name of a subprogram, and each argument, if any, is a nonsubscripted variable name or the dummy name of a SUBROUTINE or FUNCTION subprogram. The name of the subprogram must consist of 1 to 6 alphamerrc characters. The first character must be alphabetic. The last character must not be F if there are more than three characters in the name.	SUBROUTINE MATMPY (A, N, M, B, L, C) SUBROUTINE QDRTIC (B, A, C, ROOT1, ROOT2)

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

A SUBROUTINE subprogram must be referred to by a CALL statement in the calling program. The CALL statement specifies the name of the subprogram and its arguments.

Unlike the FUNCTION subprogram which returns only a single numeric value, the SUBROUTINE subprogram uses one or more of its arguments to return output. The arguments so used must, therefore, appear on the left side of an arithmetic statement or in an input statement list within the program.

The arguments of the SUBROUTINE statement are dummy names which are replaced, at the time of execution, by the actual arguments supplied in the CALL statement. There must, therefore, be correspondence in number, order, and mode, between the two sets of arguments. Furthermore, when a dummy argument is an array name, the corresponding actual argument must also be an array name. Each of these array names must appear in DIMENSION statements of their respective programs with the same dimensions.

For example, the subprogram headed by

```
SUBROUTINE MATMPY (A,N,M,B,L,C)
```

could be called by the main program through the statement

```
     CALL MATMPY (X,5,10,Y,7,Z)
```

where the dummy variables A,B,C are the names of matrices. A,B, and C must appear in a DIMENSION statement in the subprogram, and X, Y, and Z must appear in a DIMENSION statement in the calling program. The dimensions assigned must be the same in both statements.

None of the dummy variables may appear in an EQUIVALENCE statement in the SUBROUTINE subprogram. These subprograms may be compiled independently or they may be multiple-compiled with others.

Subroutine Names as Arguments of Subprograms

FORTRAN will accept Library function, FUNCTION subprogram, and SUBROUTINE subprogram names as arguments in other SUBROUTINE and FUNCTION subprograms. This permits the subroutine name specified as an argument to be different, depending upon the arguments specified in the subprogram reference.

The terminal F of a Library function name must be dropped only when this name appears in the argument list of a CALL or SUBROUTINE statement or FUNCTION subprogram reference. This terminal F, however, must appear whenever the Library function name appears within an arithmetic expression.

When a subroutine name appears in the argument list of a SUBROUTINE or FUNCTION subprogram, the corresponding subroutine name in the subprogram reference must appear in an F card. The F card defines a subprogram argument to be a subroutine name, and the F card may appear anywhere in the program containing the subprogram reference.

The letter F must appear in column 1, and the subroutine name(s) must appear, separated by commas, in columns 7-72. For example,

```
F     SIN, COS, FUN, SUBP
```

where SIN and COS are Library function names, FUN is a FUNCTION subprogram name, and SUBP is a SUBROUTINE subprogram name.

This sample F card indicates that SIN, COS, FUN, and SUBP are subroutine names appearing in argument lists of subprogram references. Note that the terminal F required for the Library functions SIN and COS is omitted when these names appear in a F card.

Consider the subprogram

```
      SUBROUTINE BOB (DUMMY, Y, A)
      A=DUMMYF(Y)
      RETURN
      END
```

and the calling program

```
F     SIN, COS
      CALL BOB (SIN, S, X)
      CALL BOB (COS, S, Z)
```

SUBROUTINE BOB permits the function DUMMYF to vary, depending on the CALL statements of the calling program.

The statement

```
     CALL BOB (SIN, S, X)
```

causes the SINF(S) to be computed and placed in storage location X. Similarly,

```
     CALL BOB (COS, S, Z)
```

causes COSF(S) to be stored in location Z.

Section B: CALL and RETURN Statements

The CALL statement has reference only to the SUBROUTINE subprogram, whereas the RETURN statement is used by both the FUNCTION and SUBROUTINE subprograms.

CALL Statement

GENERAL FORM	EXAMPLES
"CALL Name (a_1, a_2,..., a_n)", where Name is the name of a Subroutine subprogram, and a_1, a_2,..., a_n are arguments which take one of the forms described below.	CALL MATMPY (X, 5, 10, Y, 7, Z) CALL QDRTIC (P*9.732, Q/4.536, R-S**2.0, X1, X2)

This statement is used to call SUBROUTINE subprograms; the CALL transfers control to the subprogram and presents it with the parenthesized arguments. Each argument may be one of the following types:

1. Fixed point constant.
2. Floating point constant.
3. Fixed point variable, with or without subscripts.
4. Floating point variable, with or without subscripts.
5. Arithmetic expression.
6. 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 characters when used in alphameric fields.
7. The name of a FUNCTION of SUBROUTINE subprogram.

The arguments presented by the CALL statement must agree in number, order, mode, and array size with the corresponding arguments in the SUBROUTINE statement of the called subprogram, and none of the arguments may have the same name as the SUBROUTINE subprogram being called.

RETURN Statement

GENERAL FORM	EXAMPLES
"RETURN"	RETURN

This statement terminates any subprogram, whether of the type headed by a SUBROUTINE statement or a FUNCTION statement, and returns control to the calling program.

A RETURN statement must, therefore, be the last executed statement of the subprogram. It need not be the physically last statement of the subprogram; it can be any point reached by a path of control, and any number of RETURN statements may be used.

CHAPTER 7. INPUT/OUTPUT STATEMENTS

Thirteen FORTRAN statements are available for specifying the transmission of information between storage and magnetic tapes, drums, card reader, card punch, and printer. These input/output statements can be grouped as follows:

1. Five statements, READ, READ INPUT TAPE, PUNCH, PRINT, and WRITE OUTPUT TAPE, cause transmission of a specified list of quantities between storage and an external input/output medium: cards, printed sheet, or magnetic tape, for which information is expressed in alphameric punching, alphameric print, or binary-coded-decimal (BCD) tape code, respectively.
2. One statement, FORMAT, is a nonexecutable statement that specifies the arrangement of the information in the external input/output medium with respect to the five source statements of group 1 above.
3. Four statements, READ TAPE, READ DRUM, WRITE TAPE, and WRITE DRUM, cause information to be transmitted in binary machine-language.
4. Three statements, END FILE, BACKSPACE, and REWIND, manipulate magnetic tapes.

Specifying Lists of Quantities

Of the thirteen input/output statements, nine call for the transmission of information and must, therefore, include a list of the quantities to be transmitted. This list is ordered and its order must be the same as the order in which the words of information exist (for input) or will exist (for output) in the input/output medium.

The formation and meaning of a list is best described by the following example:

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

Suppose that this list is used with an output statement. Then the information will be written on the input/output medium in this order:

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

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

Thus, the list reads from left to right, with repetition for variables enclosed within parentheses. Only variables, and not constants, may be listed. The execution is exactly that of a DO-loop, as though each opening parenthesis (except subscripting parentheses) were a DO, with indexing given immediately before the matching closing parenthesis, and with the DO range extending up to that indexing information. The order of the above list can thus be considered the equivalent of the "program":

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

Note that indexing information, as in DOs, consists of three constants or fixed point variables, and that the last of these may be omitted, in which case it is taken to be 1.

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

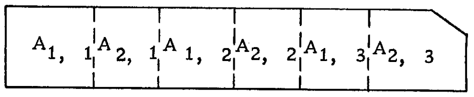
Input/Output in Matrix Form

As outlined previously, FORTRAN treats variables according to conventional matrix practice. Thus, the input/output statement

READ 1, ((A(I,J), I = 1, 2), J = 1,3)

causes the reading of I x J (in this case 2 x 3) items of information. The data items will be read into storage in the same order as they are found on the input medium.

For example, if the data is punched on a card in the form



the data will be stored in locations N, N-1, N-2,.., N-5, respectively, where N is the highest absolute location used for the array of information to be read in.

Input/Output of Entire Matrices

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

Thus, if A has previously been listed in a DIMENSION statement, the statement,

READ 1, A

is sufficient to read in all of the elements of the array A. In 7090/7094 FORTRAN II, the elements read in by this notation are stored in their natural order, i.e., in order of decreasing storage locations. If A has not previously appeared in a DIMENSION statement, only the first element will be read in.

NOTE: Certain restrictions to these rules exist with respect to lists for the statements READ DRUM and WRITE DRUM, for which the abbreviated notation mentioned immediately above is the only one permitted.

FORMAT Statement

GENERAL FORM	EXAMPLES
"FORMAT ($s_1, \ldots, s_n$)", where each s is a format specification as described below.	FORMAT (I2/ (E12.4, F10.4))

In addition to the list of quantities to be transmitted, the five input/output statements of group 1 contain the statement number of a FORMAT statement describing the information format to be used. It also specifies the type of conversion to be performed between the internal machine language and external notation. FORMAT statements are not executed; their function is merely to supply information to the object program. Therefore, they may be placed anywhere in the source program, except as the first statement in the range of a DO.

For the sake of clarity, examples are given below for printing. However, the description is valid for any case simply by generalizing the concept of "printed line" to that of unit record in the input/output medium. A unit record may be:

1. A printed line with a maximum of 120 characters.
2. A punched card with a maximum of 72 characters.
3. A BCD tape record with a maximum of 120 characters.

Numeric Fields

Four forms of conversion for numeric data are available:

INTERNAL	$\overset{\leftarrow}{\underset{\rightarrow}{\text{TYPE}}}$	EXTERNAL
Floating point variable	E	Floating point decimal
Floating point variable	F	Fixed point decimal
Fixed point variable	I	Decimal integer
Binary representation of the octal integer	O	Octal integer

These types of conversion are specified in the forms:

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

where w and d are unsigned fixed point constants.

Format specifications are used to describe the format of input and output. The format is specified by giving, from left to right, beginning with the first character of the record:

1. The control character (E, F, I, or O) for the field.
2. The width (w) of the field. The width specified may be greater than required, to provide spacing between numbers.
3. For E- and F-type conversions, the number of positions (d) of the field which appear to the right of the decimal point. (Note: d is treated modulo 10.)

Specifications for successive fields are separated by commas. No format specification that provides for more characters than the input/output unit record should be given. Thus, a format statement for printed output should not provide for more than 120 characters per line, including blanks.

Information to be converted by O-type format specifications may be given fixed point or floating point variable names.

Example: The statement FORMAT (I2, E12.4, O8, F10.4) might cause printing of the line:

```
I2    E12.4        O8     F10.4
27b-0.9321Eb0257734276bbb-0.0076
```

(b is included here to indicate blank spaces.)

Alphameric Fields

FORTRAN provides two ways by which alphameric information may be read or written; the specifications for this purpose are Aw and wH. Both result in storing the alphameric information internally in BCD form. The basic difference is that information handled with the A specification is given a variable array name and hence can be referred to by means of this name for processing and/or modification. Information handled with the H specification is not given a name and may not be referred to or manipulated in storage in any way.

The specification Aw causes w characters to be read into, or written from, a variable or array name. The name must be constructed in the same manner as a fixed point or floating point variable name.

The effect of nAw depends on whether it is used for input or output.

1. Input -- The next n successive fields of w characters each are to be stored as BCD information. If $w > 6$, only the six rightmost characters will be significant; if $w < 6$, the characters will be left-adjusted and the word filled out with blanks.
2. Output -- The next n successive fields of w characters each of output are to be the result of transmission from storage without conversion. If $w > 6$, only six characters will be transmitted, preceded by w-6 blanks; if $w < 6$, the w leftmost characters of the word will be transmitted.

The specification wH is followed in the FORMAT statement by w alphameric characters; for example

```
24H THIS IS ALPHAMERIC DATA
```

Note that blanks are considered alphameric characters and must be included as part of the count w.

The effect of wH depends on whether it is used for input or output.

1. Input -- w characters are extracted from the input record and replace the w characters included with the specification.
2. Output -- The w characters following the specification, or the characters which replaced them, are written as part of the output record.

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

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

(b is used to indicate blank characters.)

This example assumes that there are steps in the source program which 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.

Blank Fields

Blank characters may be provided in an output record, and characters of an input record may be skipped by means of the specification wX, where $0 < w \leq 120$ (w is the number of blanks provided or characters skipped). When the specification is used with an input record, w characters are considered to be blank, regardless of what they actually are, and these w characters are skipped over. (The control character X need not be separated by a comma from the specification of the next field.)

Repetition of Field Format

It may be desired to print n successive fields within one record, in the same fashion. This may be specified by giving n, where n is an unsigned fixed point constant, before E, F, I, O, or A. Thus, the statement FORMAT (I2, 3E12.4) would give the printed line

```
27b-0.9321Eb02b-0.7580E-02bb0.5536Eb00
```

Repetition of Groups

A limited parenthetical expression is permitted in order to enable repetition of data fields according to certain format specifications within a longer FORMAT statement specification. Thus, FORMAT (2(F10.6, E10.2), I4) is equivalent to FORMAT (F10.6, E10.2, F10.6, E10.2, I4).

Scale Factors

To permit more general use of F-type conversion, a scale factor followed by the letter P may precede the specification. The scale factor is defined so that:

$$\text{Printer number} = \text{Internal number} \times 10^{\text{scale factor}}$$

Thus, the statement FORMAT (I2, 1P3F11.3) used with the data of the preceding example, would give

```
27bbb-932.096bbbbb-0.076bbbbbb5.536
```

whereas FORMAT (I2, -1P3F11.3) would give

```
27bbbbb-9.321bbbbb-0.001bbbbbb0.055
```

A positive scale factor may also be used with E-type conversion to increase the number and decrease the exponent. Thus, with the same data, FORMAT (I2, 1P3E12.4) would produce

```
27b-9.3210Eb01b-7.5804E-03bb5.5361E-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 E- and F-type conversions following the scale factor within the same FORMAT statement. This applies to both single-record and multiple-record formats (see below). Once a scale factor has been given, a subsequent scale factor of zero in the same FORMAT statement must be specified by 0P. Scale factors have no effect on I-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 line. Thus, FORMAT (3F9.2, 2F10.4/8E14.5) would specify a multi-line 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 such that the first two lines will 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 open parenthesis.

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

Blank lines may be introduced into a multi-line FORMAT statement by listing consecutive slashes. N + 1 consecutive slashes produce N blank lines.

Format and Input/Output Statement Lists

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).

Ending a FORMAT Statement

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 numeric 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 will be brought to an end when a specification for a numeric field or the end of the FORMAT statement is encountered, and there are no items remaining in the list.

FORMAT Statements Read in at Object Time

FORTRAN accepts a variable FORMAT address. This provides the facility of specifying a list at object time. Example:

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

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

The name of the variable FORMAT specification must appear in a DIMENSION statement even if the array size is only 1. The FORMAT name must consist of one to six alphameric characters and the first character must be alphabetic.

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.

Carriage Control

The WRITE OUTPUT TAPE statement prepares a decimal tape which can later be used to obtain off-line printed output. The off-line printer is manually set to operate in one of the three modes: single space, double space, and Program Control. Under Program Control, which gives the greatest flexibility, the first character of each BCD record controls spacing of the off-line printer; the first character of a BCD record is not printed.

The control characters and their effects are:

Blank	Single space before printing
0	Double space before printing
+	No space before printing
1 - 9	Skip to printer control channels 1-9*
J - R	Short skip to printer control channels 1-9*

Thus, a FORMAT specification for WRITE OUTPUT TAPE for off-line printing with Program Control will usually begin with 1H followed by the appropriate control character. This is required for the PRINT statement since on-line printing simulates off-line printing under Program Control.

* See the section entitled "Carriage Control" in the IBM Reference Manual, IBM 709 Data Processing System, Form A22-6536.

Data Input to the Object Program

Decimal input data to be read by means of a READ or READ INPUT TAPE when the object program is executed must be in essentially the same format as given in the previous examples. Thus, a card to be read according to FORMAT (I2, E12.4, F10.4) might be punched

```
    27  -0.9321E 02  -0.0076
```

Within each field, all information must appear at the extreme right. Plus signs may be omitted or indicated by a blank or +. Minus signs may be punched with an 11-punch or an 8-4 punch. Blanks in numeric fields are regarded as zeros. Numbers for E- and F-type conversion may contain any number of digits, but only the high-order 8 digits of accuracy will be retained. For numbers whose magnitude is greater than 2^{27}, it is preferable to use E, rather than F conversion. Numbers for I-type conversion will be treated modulo 2^{17}.

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

1. Numbers of E-type conversion need not have 4 columns devoted to the exponent field. The start of the exponent field must be marked by an E, or if that is omitted, by a + or - (not a blank). Thus E2, E02, +2, +02, E 02, and E+02 are all permissible exponent fields.
2. Numbers for E- or F-type conversion need not have their decimal point punched. If it is not 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 indicated position in the FORMAT specification.

READ Statement

GENERAL FORM	EXAMPLES
"READ n, List", where n is the statement number of a FORMAT statement, and List is as described in Chapter 7.	READ 1, ((ARRAY (I,J), I = 1, 3), J = 1, 5)

The READ statement causes the reading of cards from the card reader. For 7090 FORTRAN, the Data Synchronizer Channel to which the card reader is attached must be specified by the installation (see "Symbolic Input/Output Unit Designation"). Successive cards are read until the complete list has been "satisfied," i.e., all data items have been read, converted, and stored in the locations specified by the list of the READ statement. The FORMAT statement to which the READ refers describes the arrangement of information on the cards and the type of conversion to be made.

READ INPUT TAPE Statement

GENERAL FORM	EXAMPLES
"READ INPUT TAPE i, n, List", where i is an unsigned fixed point constant or a fixed point variable; n is the statement number of a FORMAT statement, and List is as described in Chapter 7.	READ INPUT TAPE 24, 30, K, A(J) READ INPUT TAPE N, 30, K, A(J)

The READ INPUT TAPE statement causes the object program to read BCD information from symbolic tape unit ($0<i<81$). Record after record is brought in, in accordance with the FORMAT statement, until the complete list has been satisfied.

The object program tests for the proper functioning of the tape reading process. In the event that the tape cannot be read properly, the object program halts.

Symbolic Input/Output Unit Designation

Tape Units: In order to enable 7090/7094 FORTRAN II to accept source programs written in connection with other programming systems, a distinction is made between the logical tape unit numbers specified in the source program, and the actual tape units which will be affected by the resulting object program. Logical/actual equivalences for the 7090/7094 FORTRAN II System are specified in the system as distributed, but these may be changed by the installation in accordance with its own needs. The equivalences are established by the insertion of an IOU subroutine into the edit deck of the 7090/7094 FORTRAN II System. (See " The FORTRAN II Editing Program," in the 7090/7094 FORTRAN II Operations manual.)

Card Reader, On-Line Printer, and Card Punch: One each of these input/output units can be attached to Data Synchronizer Channels A, C, or E of the 7090. The card reader, on-line printer, or card punch which will actually be involved in the execution of READ, PRINT, or PUNCH, respectively, is specified by the system as distributed and may be changed by the installation. At the time that the

7090/7094 FORTRAN II object program is executed, the equivalence between the logical and actual input/output units must be known.

READ TAPE Statement

GENERAL FORM	EXAMPLES
"READ TAPE i, List", where i is an unsigned fixed point constant or a fixed point variable, and List is as described in Chapter 7.	READ TAPE 24, (A(J), J = 1, 10) READ TAPE K, (A(J), J = 1, 10)

The READ TAPE statement causes the object program to read binary information from symbolic tape unit i ($0<i<81$), into locations specified in the list. A record is read completely only if the list specifies as many words as the tape record contains; no more than one record will be read. The tape, however, always moves to the beginning of the next record.

Binary tapes read by a 7090/7094 FORTRAN II Compiled program should have been written by a 7090/7094 FORTRAN II object program. However, it is possible to use a non-FORTRAN written binary tape, provided the tape records are in the proper format. The following is a description of this record format.

Consider a logical record as being any sequence of binary words to be read by any one input statement. This logical record must be broken 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 constitute only one physical record. The first word of each physical record is a "signal" word that is not part of the list. This word contains zero for all but the last physical record of a logical record. The first word of the last physical record contains a number designating the number of physical records in this logical record.

FORTRAN handles binary tape operations according to the following rules:

1. Binary records are read under logical record control and written under count control.
2. The list for binary tape operations may be less than or equal to, but not greater than, the length of the logical record.

The object program checks tape reading. In the event that a record cannot be read properly, the object program halts.

READ DRUM Statement

GENERAL FORM	EXAMPLES
"READ DRUM i, j, List", where i and j are each either an unsigned fixed point constant or a fixed point variable, with the value of i between 1 and 8 inclusive, and List is as described below.	READ DRUM 2, 1000, A, B, C, D (3) READ DRUM K, J, A, B, C, D (3)

The READ DRUM statement causes the object program to read words of binary information from consecutive locations on drum i, beginning with the word in drum location j, where $0 \leq j < 2048$. (If $j > 2047$, it is interpreted modulo 2048.) Reading continues until all words specified by the list have been read in. If the list specifies an array, the array is stored in inverse order.

The list for the READ DRUM and WRITE DRUM statements can consist only of variables without subscripts or with only constant subscripts, such as A, B(5), C, D. Variables consisting of only one element of data will be read into storage in the ordinary way; those which are arrays will be read with indexing obtained from their DIMENSION statements. Thus, the statement READ DRUM, i, j, A, where A is an array, causes the complete array to be read. The array A is stored in inverse order.

PUNCH Statement

GENERAL FORM	EXAMPLES
"PUNCH n, List", where n is the statement number of a FORMAT statement, and List is as described in Chapter 7.	PUNCH 30, (A(J), J = 1, 10)

The PUNCH statement causes the object program to punch alphameric cards. Cards are punched in accordance with the FORMAT statement until the complete list has been satisfied.

PRINT Statement

GENERAL FORM	EXAMPLES
"PRINT n, List", where n is the statement number of a FORMAT statement, and List is as described in Chapter 7.	PRINT 2, (A(J), J = 1, 10)

The PRINT statement causes the object program to print output data on an on-line printer. Successive lines are printed in accordance with the FORMAT statement until the complete list has been satisfied.

WRITE OUTPUT TAPE Statement

GENERAL FORM	EXAMPLES
"WRITE OUTPUT TAPE i, n, List", where i is an unsigned fixed point constant or a fixed point variable, n is the statement number of a FORMAT statement, and List is as described in Chapter 7.	WRITE OUTPUT TAPE 42, 30, (A(J), J = 1, 10) WRITE OUTPUT TAPE L, 30, (A(J), J = 1, 10)

The WRITE OUTPUT TAPE statement causes the object program to write BCD information on symbolic tape unit i $(0 < i < 81)$. Successive records are written in accordance with the FORMAT statement until the complete list has been satisfied. An end of file is not written after the last record.

WRITE TAPE Statement

GENERAL FORM	EXAMPLES
"WRITE TAPE i, List", where i is an unsigned fixed point constant or a fixed point variable, and List is as described in Chapter 7.	WRITE TAPE 24, (A(J), J = 1, 10) WRITE TAPE K, (A(J), J = 1, 10)

The WRITE TAPE statement causes the object program to write binary information on the tape unit with symbolic tape number i $(0 < i < 81)$. One logical record is written consisting of all the words specified in the list.

The object program checks tape writing. In the event that a record cannot be written properly, the object program halts. When the object program is operating under the Monitor, the EXEM subroutine handles the error (see the 7090/7094 FORTRAN II Operations manual).

WRITE DRUM Statement

GENERAL FORM	EXAMPLES
"WRITE DRUM i, j, List", where i and j are each either an unsigned fixed point constant or a fixed point variable, with the value of i between 1 and 8 inclusive, and List is as described for READ DRUM.	WRITE DRUM 2, 1000, A, B, C, D(6) WRITE DRUM K, J, A, B, C, D(6)

The WRITE DRUM statement causes the object program to write words of binary information onto consecutive locations on drum i, beginning with drum location j. (If $j > 2047$, it is interpreted modulo 2048.) Writing continues until all the words specified by the list have been written.

The list of the WRITE DRUM statement is subject to the same restrictions that apply to READ DRUM.

END FILE Statement

GENERAL FORM	EXAMPLES

"END FILE i", where i is an unsigned fixed point constant or a fixed point variable.	END FILE 29
	END FILE K

The END FILE statement causes the object program to write an end-of-file mark on symbolic tape unit i $(0 < i < 81)$.

REWIND Statement

GENERAL FORM	EXAMPLES
"REWIND i", where i is an unsigned fixed point constant or a fixed point variable.	REWIND 3 REWIND K

The REWIND statement causes the object program to rewind symbolic tape unit i $(0 < i < 81)$.

BACKSPACE Statement

GENERAL FORM	EXAMPLES
"BACKSPACE i", where i is an unsigned fixed point constant or a fixed point variable.	BACKSPACE 18 BACKSPACE K

The BACKSPACE statement causes the object program to backspace symbolic tape unit i $(0 < i < 81)$.

CHAPTER 8. SPECIFICATION STATEMENTS

The final type of FORTRAN statement consists of the four specification statements: DIMENSION, FREQUENCY, EQUIVALENCE, and COMMON. These are nonexecutable statements that supply necessary information or information to increase object program efficiency.

DIMENSION Statement

GENERAL FORM	EXAMPLES
"DIMENSION $v_1, v_2, v_3, \ldots$ ", where each v is the name of a variable, subscripted with 1, 2, or 3 unsigned fixed point constants. Any number of v's may be given.	DIMENSION A(10), B(5, 15), CVAL(3, 4, 5)

The DIMENSION statement provides the information necessary to allocate storage for arrays in the object program.

Each variable which appears in subscripted form in a program or subprogram must appear in a DIMENSION statement of that program or subprogram; the DIMENSION statement must precede the first appearance of that variable. The DIMENSION statement lists the maximum dimensions of arrays; in the object program, references to these arrays must never exceed the specified dimensions.

The above example indicates that B is a two-dimensional array for which the subscripts never exceed 5 and 15. The DIMENSION statement, therefore, causes 75 (i.e., 5 x 15) storage locations to be set aside for the array B.

A single DIMENSION statement may specify the dimensions of any number of arrays. A program must not contain a DIMENSION statement that includes the name of the program itself, or any program that it calls.

FREQUENCY Statement

GENERAL FORM	EXAMPLES
"FREQUENCY n (i, j, ...), m(k, l, ...), ... ", where n, m, ... are statement numbers, and i, j, k, l, ... are unsigned fixed point constants.	FREQUENCY 30(1, 2, 1), 40 (11), 50(1, 7, 1, 1), 10 (1, 7, 1, 1)

The FREQUENCY statement has no direct effect upon the execution of the object program. Its purpose is to inform FORTRAN of the number of times that the programmer believes that each branch of one or more specified control branchings will be executed.

The purpose of the statement is to make the object program as efficient as possible in terms of execution time and storage locations required. In no case will the logical flow of an object program be altered by a FREQUENCY statement.

A FREQUENCY statement can be placed anywhere in the FORTRAN source program, except as the first statement in the range of a DO, and it may be used to give frequency estimates for any number of branch-points. For each branch-point, the information consists of the statement number of the statement causing the branch, followed by parentheses enclosing the estimated frequencies which are separated by commas.

In a program including the above example, statement 30 might be an IF, and statement 50, a computed GO TO. In these cases, the probability of going to each of the three or four branch-points in statements 30 and 50, respectively, is given by the corresponding entry of the FREQUENCY statement. Statement 40 must be a DO, in which at least one of the parameters is variable and the value of the variable parameter is not known in advance. An estimate is made that the DO range will be executed 11 times before the DO is satisfied.

All frequency estimates, except those about DOs, are relative. Thus, the example given above could have been FREQUENCY 30(2, 4, 2), 40(11), 50(3, 21, 3, 3), with equivalent results. A frequency can be estimated as 0; this will be taken to mean that the expected frequency is very small.

Applicable Statements

The following table lists the seven FORTRAN statements about which frequency information may be given.

STATEMENT	No. of Branches	REMARKS
(Computed) GO TO	≥ 2	Frequencies must appear in the same order as the branches. If no frequencies are given, they are assumed to be equal for all branches.
IF	3	
IF (SENSE SWITCH)	2	
IF ACCUMULATOR OVERFLOW	2	
IF QUOTIENT OVERFLOW	2	
IF DIVIDE CHECK	2	
DO	1	The frequency need be given only when m_1, m_2, or m_3 is variable.

A frequency estimate concerning a DO is ignored unless at least one of the indexing parameters of that DO is variable. Moreover, such frequency estimates should be based only on the expected values of those variable parameters; in other words, even if the range of a DO were to contain transfer exits, the frequency estimate should specify the number of times the range must be executed to cause a normal exit. A DO with variable indexing parameters, and for which no FREQUENCY statement is given, will be treated by FORTRAN as though a frequency of 5 has been estimated.

EQUIVALENCE Statement

GENERAL FORM	EXAMPLES
"EQUIVALENCE (a, b, c, ...), (d, e, f, ...), ...", where a, b, c, d, e, f, ... are variables optionally followed by a single unsigned fixed point constant in parentheses.	EQUIVALENCE (A, B(1), C(5)), (D(17), E(3))

The EQUIVALENCE statement provides the option of controlling the allocation of data storage in the object program. In particular, when the logic of the program permits it, the number of storage locations used can be reduced by causing locations to be shared by two or more variables. The EQUIVALENCE statement should not be used to equate mathematically two or more elements.

An EQUIVALENCE statement may be placed anywhere in the source program, except as the first statement of the range of a DO. Each pair of parentheses of the statement list encloses the names of two or more quantities which are to be stored in the same locations during execution of the object program; any number of equivalences, i.e., sets of parentheses, may be given.

In an EQUIVALENCE statement, the meaning of C(5) would be "the fourth storage location following the one that contains C or, if C is an array, the fourth storage location following the location that contains C_1, $C_{1,1}$, or $C_{1,1,1}$." In general, C(p) is defined for $p > 0$ to mean the (p-1)th location after C or after the beginning of the C-array, i.e., the pth location in the array. If p is not specified, it is taken to be 1.

Thus, the above sample statement indicates that the A, B, and C arrays are to be assigned storage locations such that the elements A, B(1), and C(5) are to occupy the same location. In addition, it specifies that D(17) and E(3) are to share the same location.

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

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

The sharing of storage locations cannot be planned safely without a knowledge of which FORTRAN statements, when executed in the object program, will cause a new value to be stored in a location. There are seven such statements:

1. Execution of an arithmetic statement stores a new value in the variable on the left-hand side of the statement.
2. Execution of an ASSIGN i TO n stores a new value in n.
3. Execution of a DO will, usually, store a new indexing value. (It will not always do so, however; see the section entitled "Further Details about DO Statements.")
4. Execution of a READ, READ INPUT TAPE, READ TAPE, or READ DRUM will store new values for the variables mentioned in the statement list.

COMMON Statement

GENERAL FORM	EXAMPLES
"COMMON A, B, ...", where A, B, ... are the names of variables and nonsubscripted array names.	COMMON X, ANGLE, MATA, MATB

Variables, including array names, appearing in COMMON statements are assigned to upper storage. They are stored in locations completely separate from the block of program instructions, constants, and data. This area is assigned separately for each program compiled. For 7090/7094 FORTRAN II, the area is assigned beginning at location 77461_8 and continuing downwards. This separate COMMON area may be shared by a program and its subprograms. In this way, COMMON enables data storage area to be shared between programs in a way analogous to that by which EQUIVALENCE permits data storage sharing within a single program. Where the logic of the programs permits, this can result in a large saving of storage space.

NOTE: When a FORTRAN II program is executed under control of IBSYS, COMMON is relocated to start at 77361_8. Array names appearing in a COMMON statement must also appear in a DIMENSION statement in the same program.

The programmer has complete control over the locations assigned to the variables appearing in COMMON. The locations are assigned in the sequence in which the variables appear in the COMMON statements, beginning with the first COMMON statement of the program.

Arguments in Common Storage

Because of the above, COMMON statements may be used to serve another important function. They may be used to transmit arguments from the calling program to the called FUNCTION or SUBROUTINE subprogram. In this way, they are transmitted implicitly rather than explicitly by being listed in the parentheses following the subroutine name.

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

Notes:

1. In order to force correspondence in storage locations between two variables which otherwise will occupy different relative positions in COMMON storage, it is valid to place dummy variable names in a COMMON statement. These dummy names, which may be dimensioned, will cause reservation of the space necessary to cause correspondence.
2. While implicit arguments can take the place of all arguments in CALL-type subroutines, there must be at least one explicit argument in a FUNCTION subprogram. Here, too, a dummy variable may be used for convenience.

The entire COMMON area may be relocated downward for any one problem by means of a control card (see the 7090/7094 FORTRAN II Operations manual).

When a variable is made equivalent to a variable that appears in a COMMON statement, the equivalenced variable will also be located in COMMON storage. When COMMON variables also appear in EQUIVALENCE statements, the ordinary sequence of COMMON variables is changed. Priority is given to those variables in EQUIVALENCE statements in the order in which they appear in EQUIVALENCE statements. For example,

```
     COMMON A, B, C, D
     EQUIVALENCE (C, G), (E, B)
```

will cause storage to be assigned in the following way:

77461_8	C and G
77460_8	B and E
77457_8	A
77456_8	D

CHAPTER 9. DOUBLE-PRECISION AND COMPLEX ARITHMETIC

Double-precision arithmetic is a technique for carrying out floating point calculations with twice the normal number of significant decimal places. Only single-precision floating point numbers may be input/output; however, output data may be more accurate as a result of using double-precision operations internally. Provision is made for program constants of up to 16 digits; this permits increased accuracy when dealing with a critical value such as π.

Complex arithmetic is a technique for carrying out floating point calculations with the real and imaginary parts of complex numbers. No provision is made for the input/output of complex numbers; however, since each part is represented internally as a separate single-precision floating point number, each part may be input/output separately.

Designating a Double-Precision or Complex FORTRAN Statement

A statement will be interpreted as double-precision arithmetic if the indicator D is placed in column 1 of the statement card; a statement will be interpreted as complex arithmetic if the indicator I is placed in column 1 of the statement card.

Generally, in statements in which the indicator (D or I) has no meaning, such as input/output statements, the indicator will be ignored. Such usage, however, may result in a less efficient object program or in error messages.

Constants, Variables, Subscripts, and Expressions

Constants

Double-precision floating point constants are defined in the same way as are single-precision constants and may contain up to 16 decimal digits. The magnitude of such a number must be between 10^{38} and 10^{-38}, or be zero. In the range 10^{38} to 10^{-29}, 16 decimal places are significant. Due to the way these numbers are represented internally by the computer, only 8 decimal places are significant in the range 10^{-29} to 10^{-38}.

Complex numbers are written as two single-precision floating point numbers, separated by a comma and enclosed in parentheses. The first number is the real part, the second is the imaginary part; each part may include up to eight significant digits and may have a magnitude between 10^{38} and 10^{-38}, or be zero.

For example the complex number 2.0+7.6i would be written in a complex arithmetic statement as (2.0, 7.6).

The sine of this number could be written in a complex arithmetic statement as SINF ((2.0,7.6)).

Note that the complex constant must have its own set of parentheses in addition to the usual set that encloses the argument.

Variables

Double-precision and complex variables must have floating names. Any floating point variable name that appears in a D or I statement will be considered to be a double-precision or complex variable name, respectively.

Subscripted Variables

Subscripts are written according to the normal rules, with the following differences:

1. All nonsubscripted floating point variables that appear in double-precision or complex statements are treated by 709/7090 FORTRAN II as one-dimensional arrays consisting of two elements: the first element contains the most significant part of a double-precision number or the real part of a complex number, and the second element contains the least significant part of the double-precision number or the imaginary part of the complex number. Reference to the variable name in a D or I statement will give both parts of the number.

Reference to the variable name in a statement which is not a D or I statement will give only the most significant (or real) part. The least significant (or imaginary) part may be referenced in a non-D or I statement by subscripting, but only if the variable appears in a double-precision or complex DIMENSION statement (input/output lists are an exception).

For example, assume that B has been defined as follows:

```
  I      B=(3.4,2.1)
```
Then, the statement
```
  I      A = B
```
would give A the value (3.4,2.1), whereas the statements
```
         A = B or A = B(1)
```
would give A the value 3.4, and the statement
```
         A = B(2)
```
would give A the value 2.1.

2. Each array of double-precision or complex numbers is stored in two blocks. The most significant (or real) parts are stored in the first block; the least significant (or imaginary) parts are stored in the second block.

As with nonsubscripted variables, the appearance of a subscripted variable in a D or I statement automatically refers to both parts of the number.

If desired, it is possible to refer to each part of a double-precision or complex number in non-D or I statements. The first part may be obtained by conventional subscript notation; the second part may be obtained by including an addend in the last subscript of the variable. The addend is the dimension of the last subscript of the variable.

For example, assume that A is a three-dimensional complex array with dimensions of 5 x 5 x 5. The arithmetic product of these dimensions is 125; however, 250 locations will be set aside to contain both parts of the number.

The statement
```
  I      B = A(2,3,4)
```
would refer to both parts of the number. The statement
```
         B = A(2,3,4)
```
would refer to the real part of the same number. The statement
```
         B = A(2,3,9)
```
would refer to the imaginary part of the same number.

Expressions

The rules for forming double-precision and complex arithmetic expressions are the same as those for forming single-precision floating point expressions, except that a complex quantity may only be given a fixed point exponent.

Functions and Subprograms

The normal rules for functions and subprograms are valid, with the following exceptions:

1. Double-precision or complex Library, Built-In, and Arithmetic Statement functions, may not have fixed point names.
2. The names of double-precision or complex Library or Built-In functions must contain four to six alphameric characters, ending with F.
3. The names of double-precision and complex Library and Built-In functions must appear in the FORTRAN system and on the symbolic object program listing prefaced with a D or I, respectively; this prefix must not be used in source program references.

For example, the name of the Library function to compute the square root of X, using double-precision arithmetic, appears on the library program card as

DSQRT

This function might be used in the source program in the following manner:

```
D      AREAF (R) = SQRTF (PI*R**2)
```

Note that the D prefacing SQRT is not written, but the following F is written.

4. When a subroutine name appears as an argument of a subprogram reference, and thus on an F card, the terminal F of the subroutine name must be omitted; in addition, if the subroutine is a Library or Built-In function, the name must be prefaced with a D or I exactly as it appears in the FORTRAN system. For example:

Calling Program	Called Program
D Z = BOBF(DSIN, S)	FUNCTION BOB (FUNC, X)
F DSIN	D BOB = FUNCF(X)

5. All floating point arguments in a subroutine reference are considered to be double-precision or complex if there is a D or I in column 1 of the statement containing the arguments.

6. If a double-precision or complex subscripted variable appears in an argument list of a subprogram reference, the corresponding dummy variable appearing in the subprogram definition must have identical dimensions.

However, where the dummy argument of a subprogram is not dimensioned and it is desired to use a subscripted variable as the corresponding argument, a statement may appear in the calling program setting the subscripted variable equal to a nonsubscripted variable. This non-subscripted variable may then be used in the argument list. For example:

```
Calling Program                  Called Program
D DIMENSION A(10), B(5,5)        SUBROUTINE NAME
                                    (X, Y)
    .                            D DIMENSION Y(5,5)
    .                                .
    .                                .
D C = A(5)                           .
D CALL NAME (C, B(1,3))          D RETURN
    .
    .
    .
```

Note that A may not be subscripted in the CALL statement because the corresponding dummy variable, X, does not have identical dimensions. B, however, may be subscripted in the CALL statement since the corresponding dummy variable, Y, does have identical dimensions.

7. To ensure a true double-precision or complex result in a FUNCTION subprogram, punch a D or an I in column 1 of the RETURN statement. Modal punches in RETURN statements of SUBROUTINE subprograms have no effect.

Arithmetic Statements

Arithmetic statement usage is unchanged. The following rules apply when a fixed point quantity appears on either side of the equal sign in a D or I statement:

1. If the variable on the left is fixed point and the expression on the right is double-precision or complex, the expression will be evaluated in the manner indicated and the most significant (or real) part will be truncated to an integer, converted to fixed point, and stored as the value of the variable on the left.

2. If the variable on the left is double-precision or complex and the expression on the right is fixed point, the expression will be evaluated in fixed point, converted to a single-precision floating point number, and stored as the value of the most significant (or real) part of the variable on the left. The least significant (or imaginary) part of the variable on the left will be set equal to zero.

Control Statements

The arithmetic expression in an IF statement may be given double-precision or complex significance by placing a D or I in column 1. The expression will be evaluated in the manner indicated; however, the test will be made only on the most significant (or real) part of the quantity produced as a result of the computations. If it is desired to test the least significant (or imaginary) part or to test the relationship between the two parts, this can be accomplished with single-precision IF statements, using the subscripting techniques discussed in Chapter 9.

Input/Output Statements

Input/output statements are not given any double-precision or complex arithmetic significance; a D or I in column 1 is ignored. Reference to a double-precision or complex arithmetic variable in an input/output list is interpreted to mean the most significant (or real) part only. The least significant (or imaginary) part may be obtained by using the subscripting technique.

Since each part of a complex number is a separate single-precision floating point number, each part may be input/output using E- or F-conversion. In the case of double-precision numbers, the least significant part only has meaning with respect to the most significant part. Since the most significant part undergoes rounding during input/output, the least significant part may no longer be related to it. However, the two parts of a double-precision number may be input/output in octal or binary. This is of particular importance in dealing with a large block of data that may have to be moved to an intermediate tape, and then later brought back; no precision is lost by the operation.

Example: Assume that A is a double-precision variable; then A may be written out as follows:

```
    10    FORMAT (2O12)
          WRITE OUTPUT TAPE 6, 10, A(1),A(2)
or
          WRITE TAPE 7, A(1),A(2)
```

The abbreviated notation for input/output of complete arrays may be used with double-precision and complex numbers; however, only the most significant (or real) parts will be read/written.

Specification Statements

DIMENSION Statement: Names for double-precision or complex arrays must appear in DIMENSION statements with either a D or I in column 1. The stated dimensions must refer to the most significant (or real) parts of the numbers only. FORTRAN will double the storage allotment.

EQUIVALENCE and COMMON Statements: The variables in EQUIVALENCE and COMMON statements may be single-precision, double-precision, or complex; an indicator in column 1 will be ignored. Generally, single-precision variables should not be made equivalent to double-precision or complex variables, nor should they be given the same location in COMMON.

Limitations on Source Program Size

All FORTRAN tables remain the same, with the following additions:

1. DLIST 1 Table. The maximum number of unique, double-precision or complex array names is 150.
2. DLIST 2 Table. The maximum number of unique, double-precision or complex nonsubscripted variables is 400.

Note also that available core storage may be reduced by the additional compiled instructions and added subroutines required for double-precision and complex arithmetic.

Available Functions

The following table summarizes the Library and Built-In functions available for use in double-precision and complex arithmetic expressions and their single-precision counterparts.

Single-Precision	Double-Precision	Complex
*ABSF	*ABSF	ABSF
*INTF	INTF	---
*MODF	MODF	---
*FLOATF	*FLOATF	*FLOATF
*SIGNF	*SIGNF	*SIGNF
*XFIXF	*FIXF	*FIXF
ATANF	ATANF	---
---	**ATAN2F	---
EXPF	EXPF	EXPF
LOGF	LOGF	LOGF
***LOG10F	***LOG10F	---
SINF	SINF	SINF
COSF	COSF	COSF
SQRTF	SQRTF	SQRTF

* Built-In functions.
** ATAN2F evaluates the arctangent of (arg 1, arg 2).
*** LOG10 F evaluates the log (x) to the base 10.

PART III. PROGRAMMING FOR THE FORTRAN II MONITOR

CHAPTER 10. INTRODUCTION TO THE FORTRAN II MONITOR SYSTEM

The 7090/7094 FORTRAN II Processor consists of three basic programs: a Monitor, a Compiler, and an Assembler. The Compiler accepts a source program written in the FORTRAN II language and produces a machine language object program. The Assembler accepts symbolic machine language and produces a machine language object program. The function of the Monitor is to coordinate compiler and assembler processing and simultaneously to provide means for initiating execution of object programs. Thus, continuous machine operation is possible regardless of what combinations of source and object programs the machine encounters.

A series of 7090/7094 FORTRAN or FAP (FORTRAN Assembly Program for the IBM 709/7090) source programs can be continuously compiled and assembled without halts between processing individual programs. Also, a series of object programs may be continuously executed, again without halting between programs. A third possibility, allowing continuous machine operation, is a mixture of source programs for compiling/assembling and of object programs for execution. Still a fourth possibility exists: a single source program can be compiled/assembled and executed with no machine halts between compiling/assembling and execution. From the programmer's point of view, this is equivalent to entering a source program into the machine as an object program. A fifth possibility allows continuous execution of a program too large to fit into core storage as a series of subsections, called links.

Thus, the Monitor is a supervisory program for 7090/7094 FORTRAN II, FAP, and object programs. It calls in the various Processor programs as needed. It is necessary only to inform the Monitor what type of processing is to be expected.

The FORTRAN II Processor may operate under the control of the Basic Monitor (IBSYS) or independently under the control of the FORTRAN Monitor. See Part II of the 7090/7094 FORTRAN II Operations manual. Under the Basic Monitor, FORTRAN II jobs may be stacked as input along with jobs for other processors.

The FORTRAN II Processor may reside on tape or disk. Substantial savings in setup time are achieved when the processor operates under the Basic Monitor and resides on disk.

FORTRAN II Monitor Operations

The FORTRAN II Monitor permits the following operations:

1. FORTRAN compilation.
2. FAP (FORTRAN Assembly Program) assembly.
3. Execution of object programs.
4. Execution of programs in links, a procedure necessary where the total program is too large to fit into storage and a link is a section of it that does fit into storage.

FORTRAN II Monitor Input

Input to the FORTRAN II Processor consists not only of the source program, but may include the following as well:

1. FAP symbolic cards.
2. Object program cards.
3. Data cards.
4. FORTRAN II Monitor control cards.

With one exception, the relative order of a series of different types of input does not matter, provided that each separate deck, whether source program, object program, etc., is preceded by appropriate control cards. This exception is described below under "Ordering of Job Input Deck."

The 7090/7094 FORTRAN II Compiler proper may be considered a subsection of the Monitor. Under FORTRAN control a single source program may be compiled. Nothing further, including execution, can be done. If multiple compilation of a series of FORTRAN source programs is desired, Monitor control is required.

Definition of Job

A job may be considered as the basic unit being processed by the Monitor at any one time; it consists of one or more programs. A job is either an Execute job or a Non-Execute job. As an Execute job, it is to be executed immediately after whatever processing is required. This means that the programs of the job are related to each other. A Non-Execute job contains programs which need not be dependent. Each program is processed as the control cards for the job specify. The "processing" that is given a program is one of the following:

Execute	Non-Execute	
1. FORTRAN Compilation	1. FORTRAN Compilation	(object program input is ignored)
2. FAP Assembly	2. FAP Assembly	
3. Relocation of object program input		
4. For jobs divided into links, treatment of chain links		

A job may be considered to be one of the following five types:

Non-Execute Jobs

1. One or more FORTRAN source programs to be compiled. This is simply multiple compilation. The programs may be main programs or subprograms.
2. One or more FAP symbolic programs to be assembled. These may be main programs or subprograms.
3. An intermixture of job types 1 and 2. This results in multiple compilation and assembly of FORTRAN and FAP source programs, with object program output for each source program input. There may be any combination of main programs and subprograms.

Execute Jobs

4. A sequence of input programs for immediate execution. The input programs may be of job types 1 and 2, together with relocatable column binary object program cards. Data cards, to be used during execution, follow the input programs. Input programs each consist of a single main program-subprogram sequence not larger than the available core storage. This sequence constitutes a "machine load."
5. A sequence of input programs meant for execution where each input program is a job of type 4. The data cards are placed at the end of all the input programs. This is called a chain job and each of the jobs of type 4 is a chain link. This permits a single object program execution to consist of more than one "machine load."

CHAPTER 11. FORTRAN II MONITOR FEATURES

1. The first logical record of the FORTRAN II Monitor is the "Sign-On" record. This may be programmed by the installation to process identifying information pertaining to a job. It reads and interprets the I.D. card, which is the first card for any given job. In addition, it recognizes the END TAPE card which signals that no more jobs follow. The IBM version of the Sign-On record prints the I.D. card on-line, writes it on tape for off-line printing, and signals the beginning of a job. It also prints and writes on tape the total number of lines of output of a job. This number includes output from both compilation and execution of the job. If an installation elects to program this record, it will be useful to have certain locations left undisturbed at all times in which to save desired information. For this purpose, the Monitor leaves available locations 3-7 and $11\text{-}137_8$ when operating independently under the FORTRAN II Monitor and the top 64_{10} locations when operating under IBSYS.

2. There is a complete set of control cards for the FORTRAN II Monitor. These are distinguished by an asterisk (*) in column one. In general, they are of two types: one type governs the job as a whole, telling what it consists of, and the other type governs output options. In addition to this set of control cards, there are the DUMP card, the START card, and the RESTART card, all of which are self-loading binary cards. Each of these three card types permits processing to be restarted when an unexpected stop occurs. They are discussed in detail in the <u>7090/7094 FORTRAN II Operations</u> manual.

3. The FORTRAN II Processor uses eight tapes on two channels. These tapes are A1, A2, A3, A4, B1, B2, B3, B4. A2 is the input tape and A3 is the output tape. It should be noted that the correspondence between logical tape designations used in FORTRAN source program input/output statements and the actual tape assignments at object time is set in the Unit table (IOU) in the FORTRAN library. In the Unit table distributed with the IBM System, the correspondence is as given below. For your information, the actual unit when operating under IBSYS is also included:

Logical Designation	Actual Unit when Operating as an Independent System	Actual Unit when Operating Under IBSYS *
1	A1	SYSLBx
2	B2	SYSUT3
3	B3	SYSUT4
4	A4	SYSUT1
5	A2	SYSIN1
6	A3	SYSOU1
7	B4	SYSPP1
8	B1	SYSUT2

* This unit designation is standard for IBSYS.

Each installation may alter the logical correspondences. For compatibility purposes an installation may allow more than one logical tape designation to apply to each of the input and output tapes. This is done through the Unit table (IOU) in the FORTRAN Library. (See "Description of DSU Channel-Unit Table for FORTRAN" in the <u>7090/7094 FORTRAN II Operations</u> manual.)

If a job is not a chain job, fewer tapes are required by the FORTRAN Monitor.

4. FORTRAN programs written for use under Monitor control should adhere to the following conventions:

a. The instructions for reading input tape and writing output tape should refer to the Monitor input and output tapes, respectively.

b. The STOP and PAUSE statements should not be used. Programs must be terminated by a CALL EXIT or CALL DUMP statement, or by a READ INPUT TAPE statement when there is no more input data.

c. The END card program option controls may be superseded by Monitor control cards. The END card itself is still necessary.

5. Monitor control card information and diagnostic information are written on tape and printed on-line.

6. Object programs in column binary form (and row binary if the *ROW control card is used) are stacked on tape B4 for peripheral punching. The binary output for each job is contained in one file which is preceded by a file containing the contents of the I.D. card for that job. In a chain job, the compiled binary output for each link of the job is contained in a separate file. After the binary output file, a file is written containing an END TAPE card.

Ordering of Job Input Deck

All program decks containing symbolic cards or control cards (except DATA) must precede all binary decks that are part of the job. Once a binary card has appeared in the job input deck, a symbolic card, with the exception of the DATA card, may not subsequently appear. In a chain job, this ordering refers to each link of the chain separately.

CHAPTER 12. THE CHAIN JOB

In the chain job, a program that is too large to fit into core storage is executed as a sequence of smaller programs. Each smaller program, called a link, consists of a main program together with all its subprograms and constitutes a "machine load."

For execution, the links are stacked on any of three possible tapes. The first link in the input deck is called in first for execution by the Monitor executive routine. The other links are executed as they are called by a preceding link.

There are two requirements for distinguishing individual links: (a) The start of each link must be distinguished when preparing the input deck; (b) Each link must make provision for calling the following link during execution of the chain job.

1. The control card CHAIN (R, T) must precede the physically first program (or subprogram) of each chain link, regardless of whether the link is composed of source or object programs. In the card CHAIN (R, T), T specifies the tape on which the chain link is to be kept at execution time. It should be 2, 3, or 4. If it is B2, B3, or A4, the A or B will be deleted; if it is a reference to any other tape, it will be changed to 4. Previously written FORTRAN source programs which refer to B1 will be accepted and the tape reference changed to A4. R is a fixed point number greater than 0 but less than 32,768 which denotes an identifying label for that link by which it is called. (Note: The sequence in which links are stored is in no way determined by the number R. The sequence follows from the relative position in the input deck.) Under IBSYS, although the programmer refers to tape B2, B3, and A4, as CHAIN tapes, the actual unit assignment is altered.

2. The last executable statement of a link which is to call a succeeding link for execution must be of the form:

    CALL CHAIN (R, T)

This will then cause the link, which at compilation time had been specified by the control card:

    CHAIN (R, T)

to be read into core storage and executed.

Chain Job Deck Ordering

The rule given in the previous chapter for ordering within a job applies separately to each link of a chain job.

Selection of Tapes for Link Stacking

Chain links may be stacked on tapes B2, B3, and/or A4 for object time execution. If PDUMP is called in a link, succeeding links should not be stacked on B2. The selection of tapes may be a function of object time needs to minimize tape reading time. For example, if it is desired to execute the links only once and in succession, they may be placed in that order on one tape. If, however, one of these links is to be executed repeatedly while the others are executed only once, then it should be on a separate tape to minimize tape backspace and search time.

Programming for Chain Problems

Data and Common: Data may be passed from one link to another by means of COMMON. Therefore, when it is intended that data be used by the programs of two or more links, the appropriate COMMON and EQUIVALENCE statements must be written. If a link, A, in storage is overwritten by the next link in sequence, the next time link A is read in for execution, it will be in the same form as before its first execution. This means that any program modification or storage of non-COMMON data resulting from the first execution will not exist for the second execution. In this connection, it should be mentioned that FORTRAN compiled programs do not cause program modifications.

Relative Constants: As in the case of main programs and subprograms within a link, relative constant values may be passed on from one link to another merely by placing them in COMMON statements. This means that if I is used as a subscript in one link and its value is defined in another, the appropriate COMMON entries will ensure the proper subscript values at the time the subscript is used.

CHAPTER 13. LIMITATIONS ON SOURCE PROGRAM SIZE

In translating a source program into an object program, FORTRAN internally forms and utilizes various tables containing certain items of information about the source program. These tables are of finite size and thus place restrictions on the volume of certain kinds of information which the source program may contain. If a table size is exceeded, a diagnostic message is issued and compilation is terminated.

A description of the relevant tables is given below. The term "literal appearance" means that if the same item appears more than once, it must be counted each time it appears. Table size limitations are given following the table descriptions. (See the DLIST table sizes in Chapter 9. The table sizes given are for the independent version of the FORTRAN II Processor. For FORTRAN operating under IBSYS, most tables are reduced one-eighth in size.)

Alphameric Arguments

HOLARG Table: Entries are made in this table when a CALL statement lists alphameric arguments. For every nH in a CALL statement, divide n by 6. Add 1 to the quotient if there is a remainder. Add 1 to this. The maximum number of entries is 3600.

Arithmetic Statements

ALPHA Table: This table is computed for each arithmetic statement as follows:

Set the initial value of a counter to 3.

Scanning the right-hand side of the statement in question, add 4 to the value of this counter for each left parenthesis encountered and subtract 4 for each right parenthesis encountered.

Compilation will stop if overflow occurs.
The maximum table size is 556.

BETA Table: This table limits the size of arithmetic expressions which appear both on the right-hand side of arithmetic statements and as the arguments of IF and CALL statements. Using the values computed for the LAMBDA Table (below):

$B = \lambda + 1 - n - f$

The maximum table size is 1208.

LAMBDA Table: This table limits the size of arithmetic expressions which appear on both the right-hand side of arithmetic statements and as the arguments of IF and CALL statements. For each expression:

$\lambda = n+4b+4a-3f+3p+2t+e+3$

where:

n = the number of literal appearances of variables and constants, except those in subscripts.

b = the number of open parentheses, except those introducing subscripts.

p = the number of appearances of + or -, except in subscripts or as unary operators (the + in A*(+B) is a unary operator).

t = the number of appearances of * or /, except in subscripts.

e = the number of appearances of **.

f = the number of literal appearances of function names.

a = the number of arguments of functions (for SINF(SINF (X)), a = 2).

The maximum table size is 4800.

Arithmetic Statements: Fixed Point Variables

FORVAL Table: An entry is made for each literal appearance of nonsubscripted fixed point variables on the left-hand side of arithmetic statements, in input lists, in COMMON statements, and in the argument list for FUNCTION and SUBROUTINE subprograms. The maximum number of entries is 2000.

FORVAR Table: An entry is made for each literal appearance of nonsubscripted fixed point variables on the right-hand side of arithmetic statements and in the arguments of IF and CALL statements. The maximum number of entries is 3000.

Arithmetic Statement Function

FORSUB Table: An entry is made for each distinct Arithmetic Statement function. The maximum number of entries is 200.

CALL Statement

CALLFN Table: An entry is made for each CALL statement appearing in the source program. The maximum number of entries is 2400.

COMMON Statement

COMMON Table: An entry is made for each literal appearance of variables in COMMON statements. The maximum number of entries is 6000.

DIMENSION Statement

DIM Tables: An entry is made for each one-, two-, and three-dimensional variable mentioned in DIMENSION statements. The maximum number of entries for each is as follows:

one-dimensional	400
two-dimensional	400
three-dimensional	360

DO Statement

DOTAG Table: An entry is made for each DO in a nest of DOs. The maximum number of entries is 200.

TDO Table: An entry is made for each DO. (A DO-implying parenthesis counts as a DO.) The maximum number of entries is 600.

EQUIVALENCE Statement

EQUIT Table: An entry is made for each literal appearance of variables in EQUIVALENCE statements. The maximum number of entries is 3000.

Fixed Point Constants

FIXCON Table: An entry is made for each different fixed point constant. For this purpose, constants differing only in sign are not considered different. The maximum number of entries is 400.

Floating Point Constants

FLOCON Table: An entry is made for each different floating point constant in any one arithmetic statement and in any one source program. For this purpose, constants differing only in sign or format (e.g., 4., 4. 0, 40.E-1) are not considered different. The maximum number of entries is 1800.

FORMAT Statement

FMTEFN Table: An entry is made for each literal appearance of a FORMAT statement number in an input/output statement. The maximum number of entries is 2000.

FORMAT Table: For each FORMAT statement included in the source program, compute as follows:

Count all characters, including blanks, following the word FORMAT, up to and including the final right parenthesis. Divide this count by 6. Add 1 to the quotient if there is a remainder.

All values thus computed are entered in the table. The maximum number of entries is 6000.

FREQUENCY Statement

FRET Table: An entry is made for each number mentioned in FREQUENCY statements. The maximum number of entries is 3000.

Non-Executable Statements

NONEXC Table: An entry is made for each non-executable statement in the source program. The maximum number of entries is 1200.

Statement Numbers

TEIFNO Table: An entry is made for each source statement that has a statement number. (An input/output statement that has a statement number and whose list contains controlling parentheses counts as 2.) The maximum number of entries is 3000.

STOP Statement

TSTOPS Table: An entry is made for each STOP and RETURN statement in the source progrrm. The maximum number of entries is 1200.

Subprogram Arguments

SUBDEF Table: The SUBDEF Table arises from the SUBROUTINE and FUNCTION statements. An entry is made for the name of the subprogram being defined and for each "dummy" argument contained in the argument lists. The maximum number of entries is 180.

Subprograms Functions and Input/Output Statements

CLOSUB Table: One entry is made in this table for each closed subroutine, FUNCTION, and SUBROUTINE subprogram called in the source program. In addition, as many as three entries may be made for each input/output statement. The maximum sizes are as follows:

Total entries	6000
Total different entries	3000

Subscripted Variables

FORTAG Table: An entry is made in this table for each literal appearance of subscripted variables. The maximum number of entries is 6000.

Subscripts

SIGMA Table: An entry is made for each literal appearance of variables whose subscripts contain one or more unique addends in any one arithmetic expression. The maximum number of entries is 120.

TAU Tables: An entry is made for each different one-, two-, and three-dimensional subscript combination. Subscript combinations are considered different if corresponding subscripts, exclusive of addends, or corresponding "leading dimensions" of the subscripted arrays differ. "Leading dimensions" are the first dimension of a two-dimensional array, and the first and second dimensions of a three-dimensional array. The maximum number of entries for each is as follows:

one-dimensional	400
two-dimensional	360
three-dimensional	300

Transfer Statements

NLIST Table: An entry is made in this table for each different fixed point variable in an assigned GO TO statement. The maximum number of entries is 200.

TIFGO Table: An entry is made in this table for each ASSIGN, IF, and GO TO-type statement in the source program. The maximum number of entries is 1200.

TRAD Table: An entry is made for each literal appearance of statement numbers mentioned in assigned GO TO and computed GO TO statements. The maximum number of entries is 1000.

CHAPTER 14. FORTRAN II MONITOR CONTROL CARDS

All Monitor control cards must have an * in column 1. With the exception of the I. D. card, the specific control instruction of the card is punched in columns 7-72. Punching may be done according to normal FORTRAN rules, which means that blanks are ignored. Nothing may follow the control word on the control card unless separated from it by a left parenthesis; e.g., *PAUSE (MOUNT TAPE X ON A5).

Governing the Entire Job: Type 1 Control Cards

1. I. D. Card. This card must be present for every job, and, if there is no DATE card, it must be the first card for the job. If there is a DATE card, it is first and the I. D. card immediately succeeds it. Columns 2-72 may contain anything that the installation's Sign-On record is prepared to process.
2. XEQ. This card must follow the I. D. card of a job which is to be executed.
3. DATA. This card must immediately precede the data, if any, for jobs that are to be executed. It is not needed for jobs that do not require data.
4. CHAIN (R, T). This card is used to separate links within a single chain job and specifies the tape on which the link object program is to be stored for execution. It must precede the physically first program (or subprogram) of each chain link, regardless of whether the program is a source program or an object program. R is a fixed point number greater than 0 but less than 32,768 which denotes an identifying label for the tape record which contains the link, and T is the actual unit designation of the tape on which the link is to be stored at execution time.
5. DATE. This card permits the programmer to obtain the date as an additional part of the heading for each printed page of output. Following are examples of the date field, which is specified after the DATE word of the control card: 4/2/61; 11/4/61; 3/19/61. There must be two slashes (/) in the date field plus two characters for the year. (As usual, blanks are ignored.) The DATE card may appear in two places:
   a. Preceding the I. D. card for a job. The DATE card is the only card which may precede the I. D. card.
   b. Following the Monitor START card read on-line. The date specified in this manner will be used throughout the Monitor run. However, a DATE card appearing with a job, as in a., takes precedence over the DATE card read on-line for that job only.

   The date may also be specified by an IBSYS DATE card.

6. DEBUG. This card follows the last source program, if any, of a job and precedes the Debug cards for each job or each link of a chain job. See the IBM 7090/7094 FORTRAN II Operations manual for a description of FORTRAN's debugging facility.
7. IOP. This card prevents the zeroing out of the FORTRAN Common Input/Output Package (IOP) by the FORTRAN Monitor just prior to execution, thus making it available to object programs. COMMON storage is relocated downwards to prevent overlap with

IOP. For a description of the use of IOP, see the reference manual, FORTRAN: Input/Output Package for the 32K Version, Form J28-6190.

Governing Compilation of Individual Programs: Type 2 Control Cards

Under Monitor control, there are two ways by which the programmer may specify his output options for FORTRAN compilations. These are the END card and the type 2 Monitor control cards. If specifications are given by both means, the Monitor control cards take precedence. In fact, the END card specifications will then be overwritten, and the END statement which appears in the source program listing will be that fabricated by the Monitor from the control cards. Another result of the precedence of type 2 control cards over the END card is that the END statement for programs to be compiled by the Monitor need not have options specified following the word END.

If no specifications are given in the END statement or in Monitor control cards for a FORTRAN compilation, a standard output is produced.

This consists of the following:

1. The output tape, A3 or SYSOU1, when operating under IBSYS, contains a listing of the source program and the map of object program storage. Page headings are printed on each page of FORTRAN output. This heading is derived from the information punched in columns 2-72 of the first card of the source deck that does not have an * in Column 1. In addition, each page of output is numbered.
2. The object program in relocatable binary is stacked on tape B4(IBSYS SYSPP1) for peripheral punching without the required library subroutines. The binary output of each job is contained in one file which is preceded by a file containing the contents of the I. D. card for that job. In a chain job the binary output for each link is contained in a separate file.

The type 2 Monitor control cards and their effects are:

1. CARDS ROW. This card causes the Processor to punch on-line standard FORTRAN relocatable row binary cards, preceded by a BSS loader for a main program.

2. CARDS COLUMN. This card causes the Processor to punch on-line column binary relocatable cards (no loader).

Note that CARDS COLUMN supersedes CARDS ROW when used with the same source program.

3. LIST or LIST8. Each of these cards causes the Processor to write the object program in FAP-type language following the storage map. Both appear on the output tape. The LIST card produces listings in three columns without octal instruction representation; the LIST8 card produces listings in two columns with octal representation of each instruction and its relocation bits. If both cards are used, the LIST8 card takes precedence. The LIST card option corresponds to END card setting 2; the LIST8, to END card setting 8.

4. LIBE. This card causes the Processor to search the FORTRAN library for subroutines and includes them with the object program.

5. LABEL. This card causes labeling and serialization of the off-line output cards. The contents of columns 2 through 7 of a card in the input deck are taken as the label if:

a. It is the first card of the program that does not have an asterisk (*) in column 1;

b. It has a C punch in column 1; and

c. At least one of the columns 2 through 7 is not blank. This label, with blanks treated as zero, is then placed in columns 73 through 80 of the off-line output cards, with columns 79 and 80 used for serialization. Serialization begins with 00 and recycles when 99 is reached. If, however, the label does not require all of columns 73-78, serialization begins with zero and increases to 99...9, filling all remaining columns, through column 80, before it recycles. The Symbol Table and all subroutines obtained with the program are serialized and labeled with their own names.

If conditions a., b., and c. are not met, the labeling is applied as follows: for a subprogram, the name of the subprogram is used; for a main program, 000000 is used.

The LABEL card option corresponds to END card setting 7. Labeling may be obtained on the off-line output cards of a FAP assembly. The information in columns 2 through 7 of the page title card will appear as the label. Serialization will occur as in the FORTRAN compilation. See also a description of the LBL pseudo-operation in the reference manual, FORTRAN Assembly Program (FAP), Form C28-6235.

6. PACK. This card causes FORTRAN to pack records on the off-line listing tape. There will be up to five 120-character lines per record.

7. PRINT. This card causes the Processor to print on-line the information on the BCD output tape (see the 7090/7094 FORTRAN II Operations manual).

8. ROW. This card causes the Processor to stack row binary cards on the Monitor punch tape for

peripheral punching. This option may not be used in an EXECUTE job; the Monitor will delete execution if a ROW card appears.

9. SYMBOL TABLE. This card causes the Processor to punch the Symbol Table. The Symbol Table is used only for object time debugging. See the 7090/7094 FORTRAN II Operations manual.

Other Control Cards

There are three other Monitor control cards: FAP, END TAPE, and PAUSE.

1. FAP. This card is placed immediately before the FAP program cards that are input to the Monitor. It specifies that those cards are to be assembled by FAP. The FAP card follows any type 2 Monitor control cards that may be used.
2. END TAPE. This card designates the end of the last Monitor job. It must be a separate file on the input tape.
3. PAUSE. This card is placed in the job input deck at any point(s) at which the programmer wishes the machine to halt during the reading of the input tape. In this way, a pause for such purposes as tape reel mounting may be obtained. Processing may be restarted by depressing the START key.

Other cards, not strictly control cards, may be used as input to the Monitor.

1. Cards with an asterisk in column 1 may be included with the control cards, but their information field will be treated in the manner of comments. When read, they will be printed on-line and written on tape for off-line printing.
2. End of File -- This is not a Monitor control card. When input is on-line, this card is necessary to signal the FORTRAN card-to-tape simulator to write an end-of-file mark to separate jobs on the input tape. An end-of-file card is specified by a 7- and 8-punch in column 1. All other columns are ignored.

CHAPTER 15. PROGRAMMING FORTRAN PROBLEMS FOR THE MONITOR

This chapter deals with programming in the FORTRAN II language. However, the same requirements, as reflected in machine language, apply to FAP assembly programs and to input object programs resulting from a previous symbolic assembly program.

Further details on arrangement of input decks for Monitor operations are given in the 7090/7094 FORTRAN II Operations manual. In general, all ordinary FORTRAN problems may be used with the Monitor. There are, however, three ways in which FORTRAN Monitor programs must differ: tape usage, terminating execution, and the END statement.

Differences Concerning Tape Usage

1. BCD Tape: All input BCD data must be called by the statement READ INPUT TAPE A, n, List. Output is effected by a WRITE OUTPUT TAPE B, n, List statement, where A and B are the proper logical tape designations for the Monitor input and output tapes, respectively.

If BCD information is to be written for intermediate storage during program execution, a tape not used by the Monitor must be used.

2. Binary Information: READ TAPE and WRITE TAPE statements must address tapes not used by the Monitor system. However, when the programmer knows the complete disposition of the various tapes used during Monitor operation, those tapes not being used may also be addressed. For example, if a binary tape is to be used for intermediate storage during execution of the program, a Monitor tape may be available for that particular object program run.

Differences Concerning End of Program

The STOP and PAUSE statements should not be used. Instead, the last executable source program statement must be one of the following:

1. CALL EXIT. This statement causes immediate termination of the job. IOP is restored and control goes to the Sign-On record to process the next job.

2. CALL DUMP (A_1, B_1, $F1$, . . . , A_n, B_n, F_n), where A and B are variable data names indicating limits of core storage to be dumped. Either A_i or B_i may represent upper or lower limits. F_i is a fixed point number indicating the format desired, as follows:

F	= 0	dump in octal
	= 1	dump in floating point
	= 2	interpret decrement as decimal integer
	= 3	octal with mnemonics

The storage dump is effected as specified, and then a CALL EXIT is executed. If no arguments are given, all of core storage is dumped in octal. The last format indication, F_n, may be omitted, in which case it will be assumed to be octal.

Example: Consider the FORTRAN source program

```
      DIMENSION          A(100), C(100), B(100),N(100)
      COMMON B
      DO 22 I = 1, 100
      A(I) = FLOATF (I)
      B(I) = A(I)
      N(I) = I
22    C(I) = N(I)
      CALL DUMP              ?
      END
```

a. To dump the array A in floating point, the CALL DUMP statement would be
CALL DUMP (A, A(100), 1)

b. To dump in octal that portion of core storage which contains the arrays A, C, B, and N, the CALL DUMP statement would be
CALL DUMP (N(100), A, 0) or CALL DUMP (A, N(100))

c. To dump both 1 and 2, the CALL DUMP statement would be
CALL DUMP (A, A(100), 1, N(100), A, 0)

d. To dump in octal with mnemonics from absolute location 100_{10} up to, but not including, the array N, another statement is required:
L = XLOCF(N) - 100
CALL DUMP (N(L), N(101), 3)
The library function XLOCF(N) returns the location of N to the accumulator as a fixed point constant.

3. CALL CHAIN (R, T). This statement can be used only as the last executable statement of a chain link. It calls the next chain link into core storage to be executed. Thus, each link or job runs to its conclusion without stopping and progresses to the next link or job without operator intervention.

4. READ INPUT TAPE. This statement terminates execution if all data on the input tape has been previously read. Thus, a programmer may utilize the technique of reiterating the reading and processing of data until all the data is exhausted.

Use of END Statement

The END statement may be used without any of the indicated program options following it. Thus, END, which must be the physically last statement of every FORTRAN source program, may appear in either of the two following forms:

1. END -- If this form is used, indicators for the actual program options will be inserted by the Monitor according to the type 2 Monitor control cards used or according to the standard FORTRAN output.
2. END (I_1, I_2, ..., I_{15}) where I_i may have the values 0, 1, or 2. There are two possibilities with respect to each option indicator.
   a. No Monitor control card is present to control the Sense Switch I_i. The setting prescribed by "standard" FORTRAN output is inserted.
   Where $I_i = 2$, FORTRAN is instructed to interrogate the actual sense switch setting. Physical sense switch settings, however, are not available under Monitor control. The setting of 2, therefore, will instruct the Monitor to make its setting represent that given on the control card or that given by the standard setting, as above.
   b. A Monitor control card for the indicator is present, in which case the setting prescribed by this card is inserted.

The END card switch settings correspond to the type 2 Monitor control cards as follows:

Control Card	END Card Setting
CARDS ROW	Switches 1 and 4 UP
CARDS COLUMN	Switch 1 UP and Switch 4 DOWN
LIST	Switch 2 DOWN
LIBE	Switch 5 DOWN
LABEL	Switch 7 DOWN
LIST8	Switches 2 and 8 DOWN
PRINT	Switch 3 DOWN
PACK	Switch 10 DOWN
ROW	Switch 9 DOWN
SYMBOL TABLE	Switch 6 DOWN

Dumping During Execution

The following statement may be used anywhere in the source program. CALL PDUMP (A_1, B_1, F_1, ..., A_n, B_n, F_n). The argument formats for A, B, and F are the same as those given for the CALL DUMP statement.

The difference between PDUMP and DUMP is that after PDUMP is executed, the machine is restored to its condition upon entry, and control is returned to the next executable statement. The storage dumps appear on tape A3 with other output from the job.

PDUMP is a primary name appearing on the program card of the library subprogram, DUMP.

Restriction on use of PDUMP. The CALL PDUMP statement should not be used when there is a chain link on tape B2 to be executed subsequently. Tape B2 is used by the PDUMP program for intermediate storage of the contents of core storage where PDUMP is loaded.

General Rules

Monitor Operations

Under Monitor control, a FORTRAN compilation may produce row binary cards; however, the only cards acceptable for Monitor execution are column binary cards. All non-Monitor hand-coded subprograms to be used must have correct associated program cards in proper column binary form.

If an error occurs during any of the nonexecution phases of the Monitor, the Monitor will continue to process as much as possible of the remainder of the current job.

1. If the error is in the source program (whether FORTRAN or FAP), an on-line printout occurs. This particular program of the job will be skipped and the next program of the job will be brought in via the Source Program Error Record.

WARNING: Where a nonexecution phase error occurs in any program of a job, there is the danger that succeeding programs of the job will be compiled needlessly. If the job is an XEQ job and object programs of succeeding compiled/assembled programs are not called for by the control cards, there is no purpose in continuing to these programs. Therefore, the operator, in this case, at the time of the source program error diagnostic, should be prepared to continue to the next job by means of the appropriate RESTART card.

2. If the stop is a machine error stop, the ordinary diagnostic option will be presented by the Machine Error Record. The option of continuing will enable the next program of the job to be brought in. If the job is an XEQ job, the warning given above applies here also.

3. When operations are independent of IBSYS and an unlisted stop occurs, RESTART cards and a DUMP card may be used. These cards are described in the _7090/7094 FORTRAN II Operations_ manual.

4. For unexpected stops occurring during object program execution, the DUMP or RESTART cards may be used when operating independently of IBSYS; IBSYS provides its own cards for these functions.

Program Limitations

1. Care must be exercised on jobs involving both compilation/assembly and execution to avoid the overlapping of program and COMMON data and to avoid the overlapping of program and BSS control. If either occurs, execution will be omitted. COMMON data may overlap BSS control and the Generalized I/O package.

2. A list of missing subroutines is accumulated during a job or during each chain link of a job. If more than 50 are missing, a diagnostic printout occurs and the job is deleted.

3. Corrections and patches to binary programs can be made in the usual way when under Monitor control. That is, the necessary control and relocatable correction cards can be added to the binary deck if patches are desired.

CHAPTER 16. MISCELLANEOUS DETAILS ABOUT 7090/7094 FORTRAN II

Arrangement of the Object Program

A main object program and its associated subprograms may each be considered as a separate, but complete block, containing everything, except COMMON data, necessary for execution of the program. These blocks are placed continuously in lower core storage, with a variable-length area separating them from COMMON in upper core storage.

Each program block consists of a transfer list, program instructions, constants, formats, erasable storage, and data, which are stored in that order in ascending storage locations. The data is separated into nondimensioned variables, dimensioned variables, and variables appearing in EQUIVALENCE statements.

COMMON data starts at 77461_8, and continues downward in storage. The area above 77461_8 is available for erasable storage for library and hand-coded subroutines. When a FORTRAN program is to be executed under the control of IBSYS, COMMON is relocated to 77361_8 during loading.

When a source program is compiled, FORTRAN produces a printed "storage map" of the arrangement of storage locations in the object program.

Fixed Point Arithmetic

The use of fixed point arithmetic is governed by the following considerations:

1. Fixed point constants specified in the source program must have magnitudes $< 2^{17}$.
2. Fixed point data read in by the object program itself is treated modulo 2^{17}.
3. The output from fixed point arithmetic in the object program is modulo 2^{17}. However, if during computation of a fixed point arithmetic expression, an intermediate value occurs which is $\geq 2^{18}$, it is possible that the final result will be inaccurate.
4. Indexing in the object program is modulo the size of core storage and never greater than 2^{15}.

Optimization of Arithmetic Expressions

Considerable attention is given by FORTRAN to the efficiency of the object program instructions arising from an arithmetic expression, regardless of how the expression is written. Thus, although the expression

$A \cdot B \cdot C \cdot D \cdot E$

is taken to mean

$((((A \cdot B) \cdot C) \cdot D) \cdot E)$

(where $\cdot$ represents / or * or + or -)

FORTRAN assumes that mathematically equivalent expressions are computationally equivalent. Hence, a sequence of consecutive multiplications, consecutive divisions, consecutive additions, or consecutive subtractions, not grouped by parentheses will be reordered, if necessary, to minimize the number of storage accesses in the object program.

Although the assumption concerning mathematical and computational equivalence is virtually true for floating point expressions, special care must be taken to indicate the order of fixed point multiplication and division, since fixed point arithmetic in FORTRAN is "greatest integer" arithmetic (i.e., truncated or remainderless.) Thus, the expression

5*4/2

which by convention is taken to mean $[(5 \times 4)/2]$, is computed in a FORTRAN object program as

((5/2)*4)

i.e., it is computed from left to right after permutation of the operands to minimize storage accesses.

The result of a FORTRAN computation in this case would be 8. On the other hand, the result of the expression (5 x 4)/2 is 10. Therefore, to insure accuracy of fixed point multiplication and division, it is suggested that parentheses be inserted into the expression involved.

One important type of optimization, involving common subexpressions, takes place only if the expression is suitably written. For example, the arithmetic statement

```
Y = A*B*C + SINF (A*B)
```

will cause the object program to compute the product A*B twice. An efficient object program would compute the product A*B only once. The statement is correctly written

```
Y = (A*B) * C + SINF (A*B)
```

By parenthesizing the common subexpression, A*B will be computed only once in the object program.

In general, when common subexpressions occur within an expression, they should be parenthesized.

There is one case in which it is not necessary to write the parentheses, because FORTRAN will assume them to be present. These are the type discussed in "Hierarchy of Operations," and need not be given. Thus

```
Y = A*B+C+SINF (A*B)
```

is, for optimization purposes, as suitable as

```
      Y = (A*B)+C+SINF(A*B)
```

However, the parentheses discussed in "Ordering with a Hierarchy," must be supplied if optimization of common subexpressions is to occur.

Subroutines on the System Tape

Various library subroutines in relocatable binary form are available on the FORTRAN master tape. As mentioned previously, further subroutines can be placed on the tape by each installation in accordance with its own requirements. To do so, the following steps are necessary:

1. Produce the subroutine in the form of relocatable binary cards.
2. Produce a program card in accordance with specifications outlined in the 7090/7094 FORTRAN II Operations manual.
3. Edit in accordance with instructions in the 7090/7094 FORTRAN II Operations manual.

Tape subroutines may include FUNCTION and SUBROUTINE subprograms. The program card compiled by FORTRAN with these programs will be in the format required for tape subroutines.

If the name of a function defined by a library tape subroutine is encountered while FORTRAN is processing a source program, that subroutine will be included in the object program. Only one such inclusion will be made for a particular function, regardless of how many times that function occurs in the source program.

Input and Output of Arguments

When control is transferred to a library subroutine other than a FORTRAN FUNCTION or SUBROUTINE subprogram, the argument(s) will be located as follows: Arg_1 will be located in the AC, Arg_2 (if any) in the MQ, Arg_3 (if any) in relocatable location 77775_8, Arg_4 in relocatable location 77774_8, etc. Locations down through 77462_8 are available for common erasable storage for library subroutines.

The output of any function, which is a single value, must be in the accumulator when control is returned to the calling program. All index registers that were stored at the beginning of the subroutine must be restored prior to returning control.

The arguments for FUNCTION and SUBROUTINE subprograms are listed in the object program after the transfer to the subroutine (see Appendix D).

Relative Constants

A relative constant is defined as a subscript variable which is not under control of a DO or a DO-implying parentheses in a list. For example, in the sequence:

```
      A = B(K)
      DO 10 I = 1, 10
      X = B(I) + C(I, 3*J+2)
```

K and J are relative constants, but I is not.

The appearance of a relative constant in any of the following ways will be called a relative constant definition.

1. On the left side of an arithmetic statement.
2. In the list of an input statement.
3. As an argument for a FUNCTION or SUBROUTINE subprogram.
4. In a COMMON statement.

The following paragraphs describe methods for assuring that the computation for relative constants occurs at the proper point between the definition and the use of the relative constant. A relative constant must be explicitly defined for each logical path to a program.

The variable in a Computed GO TO is treated as a relative constant.

Relative Constants in an Input List

In the object program, some computation will take place at each relative constant definition in an input list. In the case of READ, READ TAPE, and READ INPUT TAPE lists, the computation may not precede the use of a relative constant in the list unless the relative constant appearance is handled properly.

Where the relative constant definition appears in the same READ, READ TAPE, or READ INPUT TAPE list with its relative constant and precedes it, extra parentheses may be required in the list. In such a list, it is necessary that there be a left parenthesis, other than the left parenthesis of a subscript combination, between the relative constant definition and its relative constant. If the list does not contain the parenthesis, it should be obtained by placing parentheses around the symbol subscripted by the relative constant.

Examples:

```
   A, B, K, M, (C(J), J = 1, 10), G(K)
   A, B, K, M, G(K)
```

The first of these two input lists is correct. The second is incorrect, but may be made correct with extra parentheses; i.e.,

```
   A, B, K, M, (G(K))
```

A relative constant definition must not appear to the left of the name of an array in the list of a READ DRUM statement.

Relative Constants in an Argument List

A variable defined in one program may have its value transmitted to another program, where it is a relative constant and where the value is used by placing it in an argument list. Thus, the appearance of a relative constant in an argument list is sufficient to provide the necessary computation for the relative constant definition.

Relative Constants in Common Statements

A relative constant value may be transmitted from one program to another by placing it in a COMMON statement only if it is being transmitted from the calling to the called subprogram. In the example below, note that K in the calling program and I in the called program share the same location.

Example:

```
Calling Program      Called Program
      .
      .
      .              SUBROUTINE ABC
COMMON K             COMMON I
                     DIMENSION B(10)
K = 5                A = B(I)
CALL ABC                   .
      .                    .
C = A(K)                   .
      .                    .
      .                    .
```

Constants in Argument Lists

A constant may not appear as an argument in the call to a SUBROUTINE or FUNCTION subprogram if the corresponding dummy variable in the definition of the subprogram appeared either on the left side of an arithmetic statement or in an input list.

Further Details About DO Statements

Triangular Indexing

Indexing such as

```
    DO        I = 1,10
    DO        J = I,10
or
    DO        I = 1,10
    DO        J = 1,I
```

is permitted in a source program and simplifies work with triangular arrays. These are special cases of an index under control of a DO and available for general use as a fixed point variable.

The diagonal elements of an array may be picked out by the following type of indexing:

```
    DO        I = 1,10
    A(I,I,I) = (some expression)
```

A DO nest of the form:

```
    DO n1     K = 1,D3
    DO n2     J = 1,D2
    DO n3     I = 1,D1
```

for a three-dimensional array $A(D_1, D_2, D_3)$, where $A(I, J, K)$ is referred to within the inner DO, must be tested against the following criterion:

The expression $(D_1 * D_2) + (D_1 * D_2 * D_3)$ must be less than or equal to 32,767; otherwise, improper indexing will result.

The DO Index

A DO loop with index I does not affect the contents of the object program storage location for I, except under the following circumstances:

1. An IF-type or GO TO-type transfer exit occurs from the range of the DO.
2. I is used as a variable in the range of the DO.
3. I is used as a subscript in combination with a relative constant whose value changes within the range of the DO.

Therefore, if a normal exit occurs from a DO to which cases 2 and 3 do not apply, the I cell contains what it did before the DO was encountered. After normal exit, where 2 or 3 do apply, the I cell contains the current value of I.

What has just been said applies only when I is referred to as a variable. When it is referred to as a subscript, I is undefined after any normal exit and is the current value after any transfer exit.

Parentheses in I/O Lists

An I/O list should not contain unnecessary or extraneous parentheses. An incorrect compilation may result if this rule is not followed.

APPENDIX A. SOURCE PROGRAM STATEMENTS AND SEQUENCING

The precise rules which govern the order in which the source program statements of a FORTRAN program will be executed can be stated as follows:

1. Control originates at the first executable statement.
2. If control has been with statement S, then control will pass to the statement indicated by the normal sequencing properties of S. (The normal sequencing properties of each FORTRAN statement are given below. If, however, S is the last statement in the range of one or more DOs which are not yet satisfied, then the normal sequencing of S is ignored and DO-sequencing occurs.)

Nonexecutable Statements

The statements FORMAT, DIMENSION, EQUIVALENCE, FREQUENCY, and COMMON are nonexecutable statements. In questions of sequencing they can simply be ignored.

If the last executable statement in the source program is not a STOP, RETURN, IF-type, or GO TO-type statement, then the object program is compiled to give the effect of depressing the Load Cards key following the last executable statement.

Every executable statement in a FORTRAN source program, except the first, must have some path of control leading to it.

Table of Source Program Statement Sequencing

Statement	Normal Sequencing
a = b	Next executable statement.
GO TO n	Statement n.
GO TO n, $(n_1, n_2, \ldots, n_m)$	Statement last assigned to n.
ASSIGN i TO n	Next executable statement.
GO TO $(n_1, n_2, \ldots, n_m)$, i	Statement n_i.
IF (a) n_1, n_2, n_3	Statement n_1, n_2, or n_3 if (a) $<$ 0, (a) = 0, or (a) $>$ 0, respectively.
SENSE LIGHT i	Next executable statement.
IF (SENSE LIGHT i) n_1, n_2	Statement n_1, n_2 if Sense Light i is On or Off, respectively.
IF (SENSE SWITCH i) n_1, n_2	Statement n_1, n_2 if Sense Switch i is Down or Up, respectively.
IF ACCUMULATOR OVERFLOW n_1, n_2	Statement n_1, n_2 if the 7090/7094 FORTRAN II internal overflow indicator is On or Off, respectively.
IF QUOTIENT OVERFLOW n_1, n_2	Statement n_1, n_2 if the 7090/7094 FORTRAN II internal overflow indicator is On or Off, respectively.
IF DIVIDE CHECK n_1, n_2	Statement n_1, n_2 if the Divide Check indicator is On or Off, respectively.
PAUSE or PAUSE n	Next executable statement.
STOP or STOP n	Terminates program.
DO n i = m_1, m_2 or DO n i = m_1, m_2, m_3	Do-sequencing, then next executable statement.
CONTINUE	Next executable statement.
END $(I_1, I_2, I_3, \ldots, I_{15})$	No sequencing; this statement terminates a problem.
CALL Name $(a_1, a_2, \ldots, a_n)$	First statement of subroutine Name.
SUBROUTINE Name $(a_1, a_2, \ldots, a_n)$	Next executable statement.
FUNCTION Name $(a_1, a_2, \ldots, a_n)$	Next executable statement.
RETURN	The statement or part of statement following the call to the subprogram.
READ n, List	Next executable statement.
READ INPUT TAPE i, n, List	Next executable statement.
PUNCH n, List	Next executable statement.
PRINT n, List	Next executable statement.
WRITE OUTPUT TAPE i, n, List	Next executable statement.
FORMAT (Specification)	Not executed.
READ TAPE i, List	Next executable statement.
READ DRUM i, j, List	Next executable statement.
WRITE TAPE i, List	Next executable statement.
WRITE DRUM i, j, List	Next executable statement.
END FILE i	Next executable statement.
REWIND i	Next executable statement.
BACKSPACE i	Next executable statement.
DIMENSION $v_1, v_2, v_3, \ldots$	Not executed.
EQUIVALENCE (a, b, c, ...), (d, e, f, ...), ...	Not executed.
FREQUENCY n (i, j, ...), m (k, l, ...), ...	Not executed.
COMMON A, B, ...	Not executed.

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

NOTE: There are two - signs. Only the 11-punch minus sign can be used in FORTRAN source program cards. Either minus sign may be used in input data to the object program; object program output uses the 11-punch minus sign.

The character $ can be used in FORTRAN only as alphameric text in a FORMAT statement.

APPENDIX C. SENSE SWITCH SETTINGS FOR 7090/7094 FORTRAN II

Sense Switch 1	UP	When Sense Switch 1 is Up, cards containing the object program(s) are punched on-line. If under Monitor control, tape B4 contains the object program; if not under Monitor control, tape B3 contains the object program.
	DOWN	When Sense Switch 1 is Down, tape unit B3 contains the object program for the last or only source program compiled. If under Monitor control, tape unit B4 contains the object programs for all the source programs compiled, in the order compiled. No cards are punched.
Sense Switch 2	UP	When Sense Switch 2 is Up and the source program is compiled in the single compile mode, two files, containing the source program and a map of program storage, are produced on tape unit B2. If under Monitor control, tape unit A3 will contain one file for each job compiled.
	DOWN	When Sense Switch 2 is Down, the object program for each program compiled in FAP language is written on tape unit B2 if the compilation is not under Monitor control; if the compilation is under Monitor control, the object program is written on tape unit A3.
Sense Switch 3	UP	When Sense Switch 3 is Up, no on-line listings are produced.
	DOWN	When Sense Switch 3 is Down, the first two files of tape unit B2 are listed on-line.
Sense Switch 4	UP	When Sense Switch 4 is Up, punched output, if any, is relocatable row binary cards.
	DOWN	When Sense Switch 4 is Down, punched output is relocatable column binary cards.
Sense Switch 5	UP	When Sense Switch 5 is Up, library subroutines will not be punched on cards or written on actual tape unit B3.
	DOWN	When Sense Switch 5 is Down, it causes library subroutines to be punched on cards or written on tape unit B3, depending upon whether Sense Switch 1 is Up or Down.

APPENDIX D. USING HAND-CODED SUBROUTINES WITH 7090/7094 FORTRAN COMPILED OBJECT PROGRAMS

FUNCTION and SUBROUTINE subprograms coded by hand or by a system other than FORTRAN can also be linked to FORTRAN programs. If coded in FAP and assembled through the FORTRAN Monitor, the linkage instructions will occur automatically. For hand-coding other than by FAP, rules for providing this linkage are given below.

It is necessary for hand-coded subprograms to conform to FORTRAN programs with regard to the following five conditions:

1. Transfer lists to called subroutines, if any.
2. Method of obtaining the variables (arguments) given in the calling sequence.
3. Saving and restoring index registers.
4. Storing results.
5. Method of returning to the calling program.

Calling Sequence

A calling sequence for a subprogram produced by FORTRAN consists of the following:

```
     TSX        NAME, 4
     TSX        LOCX1
     TSX        LOCX2
      .          .
      .          .
      .          .
     TSX        LOCXn
```

The calling sequence consists of n+1 words. The first is an instruction which causes transfer of control to the subprogram. The remaining n words include one for each argument. The TSX in these words is never executed. In case an argument consists of an array, one instruction determines the entire array; the address of that instruction specifies the location of the first element of the array, i.e., element $A_{1,1,1}$. If the argument is alphameric data, the location given is that of the first word of the block containing the data.

Transfer List Prologue, and Index Register Saving

The first group of instructions in a subprogram is the transfer list and the prologue, in that order. The transfer list contains the symbolic names of the lower level subprograms and functions, if any, that the subprogram calls. The prologue obtains and stores the locations given in the calling sequence. It will consist of the CLA and STA instructions necessary for each argument. If it is desired, index registers may be saved.

The instructions below show a typical transfer list and prologue.

```
SUBP1  BCD   1SUBP1
SUBP2  BCD   1SUBP2              Transfer List
  .     .      .
  .     .      .
  .     .      .
SUBPN  BCD   1SUBPN

       HTR               Storage for contents of IR4
       HTR               Storage for contents of IR2
       HTR               Storage for contents of IR1
NAME   SXD   NAME-3,4    Save IR4 contents in (NAME-3)
       SXD   NAME-2,2    Save IR2 contents in (NAME-2)
       SXD   NAME-1,1    Save IR1 contents in (NAME-1)
       CLA   1,4
       STA   X1          Location of 1st argument→X1 21-35
       CLA   2,4
       STA   X2          Location of 2nd argument→X2 21-35
       CLA   N,4
       STA   Xn          Location of nth argument→Xn 21-35
```

Results

A FUNCTION subprogram must place its (single) result in the accumulator prior to returning control to the calling program.

A SUBROUTINE subprogram must place each of its results in a storage location. (Such a subprogram need not return results.) A result represented by the nth argument of a CALL statement is stored in the location specified by the address field of location (n,4).

Return

Transfer of control to the calling program is effected by:

1. restoring the Index Registers to their condition prior to transfer of control to the subprogram, and
2. transferring to the calling program.

The required steps are as follows:

```
LXD    NAME-3,4     RESTORE CONTENTS OF XR4
LXD    NAME-2,2     RESTORE CONTENTS OF XR2
LXD    NAME-1,1     RESTORE CONTENTS OF XR1
TRA    N+1,4        RETURN. N=NUMBER OF ARGUMENTS
```

Entry

Unlike a FORTRAN compiled subprogram, a hand-coded subprogram may have more than one entry point. A hand-coded subprogram used with a FORTRAN calling program may be entered at any desired point, provided that a subprogram name acceptable to FORTRAN is assigned to each selected entry point. All the above mentioned conditions must, of course, be satisfied at each entry point.

System Tape Subroutines

As discussed previously, hand-coded subprograms as well as Library functions may be placed on the System tape of the FORTRAN System. When a FORTRAN source program mentions the name of such a subprogram, it is handled in exactly the same way as a library function.

Alphameric Information

Hand-coded subprograms may handle alphameric information. This information is supplied as an argument of a CALL statement. The form of an alphameric argument is:

$nHx_1x_2 \ldots x_n$

The following example illustrates the method of storing alphameric information.

CALL TRMLPH (8, C, 13HFINAL RESULTS)

The characters 13H are dropped and the remaining information stored as follows:

Location	Contents
X	FINALb
X+1	RESULT
X+2	Sbbbbb (b represents a blank, 60_8)
X+3	777777777777_8

The address X is given in the calling sequence for the CALL statement.

C28-6054-4

Printed in U.S.A. C28-6054-4

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York