



IBM

Reference Manual

709/7090 FORTRAN

Programming System



IBM

**Reference Manual
709/7090 FORTRAN
Programming System**

MAJOR REVISION
(January, 1961)

This manual is a major revision of, and obsoletes: 709 FORTRAN Automatic Coding System for the IBM 709 Data Processing System, form C28-6054-1.

It includes material from:

IBM 709/7090 FORTRAN Monitor, form C28-6065, pp. 1-12

704 and 709 FORTRAN: Additional Format features, form J28-6081

704 and 709 FORTRAN: Using Function and Subroutine Names as Arguments,
form J28-6082

704 and 709 FORTRAN: Boolean Expressions, form J28-6087

The initial printing of this revision included an errata sheet. Errors listed on that sheet have been corrected in this printing; no other changes have been made.

In this manual, all properties attributed to "FORTRAN" apply to the FORTRAN System for the IBM 704, 709 and 7090 Data Processing Systems; properties attributed to "709/7090 FORTRAN" apply to that system only. In the latter case, certain rules are applicable to the 32K System only. These are explicitly stated as such.

This manual presents the FORTRAN language and programming rules. Other materials covering the 709/7090 FORTRAN System are:

Programmer's Primer for FORTRAN
(Form F28-6019)

709/7090 FORTRAN Operations Manual
(Form C28-6066-2)

FORTRAN Assembly Program (FAP) for the IBM 709/7090
(Form J28-6098)

THE FORTRAN SYSTEM

The IBM FORMula TRANslating System, 709/7090 FORTRAN, is an automatic coding system for the IBM 709/7090 Data Processing System. More precisely, it is a 709/7090 program which accepts a source program written in the FORTRAN language, closely resembling the ordinary language of mathematics, and which produces a machine language object program ready to be run on a 709/7090.

709/7090 FORTRAN therefore, in effect, transforms the IBM 709/7090 into a machine with which communication can be made in a language more concise and more familiar 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.

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

Object and Source Machines

FORTRAN is available in versions for all sizes of storage. Each version produces programs which can be used on any size 709/7090, provided sufficient storage is available for the object program. Object programs which are too large for the 709/7090 on which they are to be used must be subdivided by the user.

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 numerical 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 or subprograms. The language may be expanded by the use of subprograms to any desired depth.

Inclusion of Library Routines

Pre-written routines 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 in the object program of the necessary input and output routines. Those which 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 8 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.

TABLE OF CONTENTS

| | <u>Page</u> |
|---|-------------|
| PART I. GENERAL CONCEPTS | 1 |
| Chapter 1. The General Properties of a FORTRAN Source Program | 3 |
| Chapter 2. Constants, Variables, Subscripts and Expressions | 6 |
| Chapter 3. Functions | 12 |
| PART II. THE FORTRAN LANGUAGE | 17 |
| Chapter 1. The Arithmetic Statements | 19 |
| Chapter 2. Control Statements and END Statement | 21 |
| Unconditional GO TO | 21 |
| Computed GO TO | 21 |
| Assigned GO TO | 21 |
| ASSIGN | 22 |
| IF | 22 |
| SENSE LIGHT | 22 |
| IF (SENSE LIGHT) | 22 |
| IF (SENSE SWITCH) | 23 |
| IF ACCUMULATOR OVERFLOW | 23 |
| IF QUOTIENT OVERFLOW | 23 |
| IF DIVIDE CHECK | 24 |
| DO | 24 |
| CONTINUE | 27 |
| PAUSE | 27 |
| STOP | 27 |
| END | 28 |
| Chapter 3. Subprogram Statements | 29 |
| SECTION A: FUNCTION and SUBROUTINE | 31 |
| FUNCTION | 31 |
| SUBROUTINE | 32 |
| SECTION B: CALL and RETURN | 35 |
| CALL | 35 |
| RETURN | 36 |
| Chapter 4. Input/Output Statements | 37 |
| FORMAT | 39 |
| READ | 46 |

| | |
|---|-----|
| READ INPUT TAPE | 46 |
| READ TAPE | 47 |
| READ DRUM | 48 |
| PUNCH | 49 |
| PRINT | 49 |
| WRITE OUTPUT TAPE | 49 |
| WRITE TAPE | 50 |
| WRITE DRUM | 50 |
| END FILE | 50 |
| REWIND | 51 |
| BACKSPACE | 51 |
| Chapter 5. Specification Statements | 52 |
| DIMENSION | 52 |
| FREQUENCY | 52 |
| EQUIVALENCE | 54 |
| COMMON | 56 |
| PART III. PROGRAMMING FOR THE 709/7090 FORTRAN MONITOR | 59 |
| Chapter 1. Introduction to the Monitor System | 61 |
| Chapter 2. Monitor Features | 64 |
| Chapter 3. The Chain Job | 66 |
| Chapter 4. Monitor Control Cards | 68 |
| Chapter 5. Programming FORTRAN Problems for the Monitor | 72 |
| PART IV. GENERAL RULES FOR FORTRAN PROGRAMMING | 77 |
| Chapter 1. Miscellaneous Details about 709/7090 FORTRAN | 79 |
| Chapter 2. Limitations on Source Program Size | 86 |
| APPENDICES | |
| A. Source Program Statements and Sequencing | 93 |
| B. Table of Source Program Characters | 97 |
| C. Sense Switch Settings for 709/7090 FORTRAN | 99 |
| D. Using Hand-Coded Subroutines with 709/7090 FORTRAN Compiled Object Programs | 101 |
| INDEX | 105 |

PART I

GENERAL CONCEPTS



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 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 6 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.

| C FOR COMMENT | | FORTRAN STATEMENT |
|------------------|---------------|--|
| STATEMENT NUMBER | 1 2 3 4 5 6 7 | |
| 1 | 5 6 7 | |
| C | | PROGRAM FOR FINDING THE LARGEST VALUE |
| C | | ATTAINED BY A SET OF NUMBERS. |
| | | DIMENSION A (999) |
| | | FREQUENCY, 30(2,1,10),5(10,0) |
| | | READ 1,N,(A(I),I=1,N) |
| | 1 | FORMAT(13,(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(22H1 THE LARGEST OF THESE 13, 12H NUMBERS IS, F7.2) |
| | | STOP 77,777 |

Figure 1

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 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 statement cards.

Types of
FORTRAN
Statements

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

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

CHAPTER 2 — CONSTANTS, VARIABLES, SUBSCRIPTS, AND EXPRESSIONS

As required of any programming language, FORTRAN provides a means of expressing numerical 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 permissible in the FORTRAN 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 |
|--|-------------------|
| 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 (size of core storage).

Floating Point Constants

| GENERAL FORM | EXAMPLES |
|--|--|
| 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 number thus expressed must lie between the approximate limits of 10^{-38} and 10^{38} , or be zero. | |

VARIABLES

Two types of variables are permissible: 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, numerical characters.

Fixed Point Variables

| GENERAL FORM | EXAMPLES |
|--|------------------|
| 1 to 6 alphabetic or numerical characters (not special characters), of which the first is 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 (size of core storage).

To avoid the possibility that a variable may be considered by FORTRAN to be a function (see page 13), the following two warnings should be observed with respect to the naming of variables:

Warning: A variable cannot be given a name which coincides with the name of a function without its terminal F. Thus, if a function is named TIMEF, no variable should be named TIME.

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 |
|---|------------------|
| 1 to 6 alphabetic or numerical characters (not special characters), of which the first is 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 with magnitude between approximately 10^{38} and 10^{-38} .

Note: The restrictions on 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 1, 2, or 3 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 |
|--|--|
| <p>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:</p> <p style="margin-left: 40px;">v c $v+c$ or $v-c$ $c*v$ $c*v+c'$ or $c*v-c'$</p> <p>(The symbol $*$ denotes multiplication.)</p> | <p style="text-align: center;">I 3 MU+2 MU-2 5*J 5*J+2 5*J-2</p> |

The variable in a subscript must not itself be subscripted.

Subscripted Variables

| GENERAL FORM | EXAMPLES |
|---|---|
| <p>A fixed or floating point variable, followed by parentheses enclosing 1, 2, or 3 subscripts which are separated by commas.</p> | <p style="text-align: center;">A (I) K (3) BETA (5*J-2, K+2, L)</p> |

Each variable which appears in subscripted form must have the size of its array (i. e., the maximum values which 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 2-dimensional, it will be stored sequentially in the order $A_{1,1}, A_{2,1}, \dots, A_{m,1}, A_{1,2}, A_{2,2}, \dots, A_{m,2}, \dots, A_{m,n}$. Arrays are thus stored "columnwise," with the first of their subscripts varying most rapidly, and the last varying least rapidly. The same is true of 3-dimensional arrays. Arrays which are 1-dimensional are of course simply 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, which complies with 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, subscripted variables, and functions may be fixed point or floating point, expressions may also be fixed point or floating point; however, they must not be mixed mode. 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 arguments, subscripts, and exponents.
2. Constants, variables, and subscripted variables are also 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.
 - b. Items so connected are all of the same mode.

Rules for
Constructing
Boolean
Expressions
(Applicable to
32K 709/7090
FORTRAN Only)

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

1. The operation symbols +, *, and - denote the operators or, and, and complement, 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, although 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 taken 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 which 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.

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):

| | |
|---------|-----------------------------|
| ** | 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 which have been omitted from a sequence of consecutive multiplications and divisions (or consecutive additions and subtractions) will be understood to be grouped from the left. Thus, if • represents either * or / (or either + or -), then

$$A \bullet B \bullet C \bullet D \bullet E$$

will be taken by FORTRAN to mean

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

Optimization of Arithmetic Expressions

The efficiency of instructions compiled from arithmetic expressions may also be influenced by the way in which the expressions are written. The section on Optimization of Arithmetic Expressions in Part IV, Chapter 2 mentions some of the considerations which affect object program efficiency.

$$Y = A - \text{SINF}(B-C)$$

$$C = \text{MIN0F}(M, L) + \text{ABC}(B * \text{FORTF}(Z), E)$$

The names of Open, Closed, Arithmetic Statement, and Fortran functions are all used in this way. The appearance in the arithmetic expression serves to call the function; the value of the function is then computed, using the arguments which 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 numerical quantity.)

NAMING

The following paragraphs describe the rules for naming Open, Closed, and Arithmetic Statement functions.

Naming of Open, Closed, and Arithmetic Statement Functions

| GENERAL FORM | EXAMPLES |
|--|---|
| <p>The name of the function consists of 4 to 7 alphabetic or numerical characters (not special characters), of which the last must be F and the first must be alphabetic. Further, the first must be 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 separated by commas.</p> | <p>ABSF (B) XMODF (M/N, K) COSF (A) FIRSTF (Z + B, Y)</p> |

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 or floating point; similarly the function itself may be in either of these modes. Thus a function of a single argument has 4 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 which 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 X's and F's are mandatory, but the rest of the naming is arbitrary.

Naming of Fortran Functions

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 Fortran function which is 4 to 6 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 Fortran function must have I, J, K, L, M, or N for its first character.

Further details on naming Fortran functions are given on page 31.

DEFINITION

Each of the four types of functions is defined (or generated) in a different way.

Open (or Built-in) Functions

The FORTRAN System, as distributed, contains 20 built-in subroutines. It, further, has the capacity for 7 more built-in subroutines. The additional subroutines may be inserted into the system by the particular installation.

Following are the 20 functions that are compiled as open subroutines 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 \leq 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 | MAXOF | Fixed | Floating |
| | | | MAX1F | Floating | Floating |
| | | | XMAXOF | Fixed | Fixed |
| | | | XMAX1F | Floating | Fixed |
| Choosing smallest value | Min (Arg ₁ , Arg ₂ , ...) | ≥ 2 | MINOF | Fixed | Floating |
| | | | MIN1F | Floating | Floating |
| | | | XMINOF | 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 (Arg₁, Arg₂) is defined as Arg₁ - \lfloor Arg₁ / Arg₂ \rfloor Arg₂, where $\lfloor x \rfloor$ = integral part of x.

Closed (or Library) Functions

These are functions which 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 closed 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 Operations Manual.

Seven library functions are included in the FORTRAN System, as distributed. These are:

| <u>Name</u> | <u>Type of Function</u> | <u>Name</u> | <u>Type of Function</u> |
|-------------|-------------------------|-------------|-------------------------|
| LOGF | Natural Logarithm | SQRTF | Square Root |
| SINF | Trigonometric Sine | ATANF | Arctangent |
| COSF | Trigonometric Cosine | TANHF | Hyperbolic Tangent |
| EXPF | Exponential | | |

Arithmetic Statement Functions

These are functions which are defined by a single FORTRAN arithmetic statement and apply only to the particular program or sub-program in which their definition appears.

| GENERAL FORM | EXAMPLES |
|--|--|
| "a = b" where a is a function name followed by parentheses enclosing its arguments (which must be distinct non-subscripted variables) separated by commas, and b is an expression which does not involve subscripted variables. Any functions appearing in b must be available to the program or already 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 according as the name does or does not begin with X.

The right-hand side of a function statement 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 may be stated on the left-hand side to be 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 $\text{FIRSTF}(X) = A * X + B$ then a later reference to FIRSTF (Y) will cause $ay+b$, based on the current values of a, b, and y, to be computed. The naming of parameters, therefore, must follow the normal rules of uniqueness.

A function defined by a function statement may be used just as any other function. In particular, its arguments may be expressions and may involve subscripted variables; thus a reference to $\text{FIRSTF}(Z + Y(I))$, 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.

Fortran Functions

This class of functions covers those subroutines which on the one hand cannot be defined by only one arithmetic statement, and on the other, are not utilized frequently enough to warrant a place on the library tape.

They are called Fortran functions because they may conveniently be defined by a conventional FORTRAN program. In this instance compiling a FORTRAN program produces a Function subroutine in exactly the form required for object program execution.

Since Fortran functions and Subroutine subprograms are defined in the same way, a discussion of the definition of Fortran functions is included in Part II, Chapter 3.

PART II

THE FORTRAN LANGUAGE



CHAPTER 1 — THE ARITHMETIC STATEMENT

Arithmetic Statement

| GENERAL FORM | EXAMPLES |
|--|--------------------------------|
| “a = b” where a is a variable (subscripted or not subscripted) and b is an expression. | Q1 = K A(I)=B(I)+SINF(C(I)) |

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

$$Y = N\text{-LIMIT } (J-2)$$

means that the value of N-LIMIT (J-2) is to be stored in Y. The result is stored in fixed point or in floating point form if the variable to the left of the = sign is a fixed point or a 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 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 to fixed point, and store in I. |
| A = I | Convert I to floating point, and store in A. |
| I = I+1 | Add 1 to I and store in I. This example illustrates the fact that an arithmetic formula is not an equation, but is a command to replace a value. |
| A = 3.0*B | Replace A by 3B. |
| A = 3*B | <u>Not permitted.</u> The expression is mixed, i. e., contains both fixed point and floating point variables. |
| A = I*B | <u>Not permitted.</u> The expression is mixed. |

A Boolean arithmetic statement is an arithmetic statement in which b is a Boolean expression (see page 10).

Examples of Boolean Arithmetic Statements

| <u>Col. 1</u> | <u>Cols. 7-72</u> | |
|---------------|--------------------------|---|
| 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 | $D=-\text{IMPF}(-B, -C)$ | 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 here being used to "mask out" the right half of word X. |

CHAPTER 2 — CONTROL STATEMENTS AND END STATEMENT

The second class of FORTRAN statements is the set of sixteen control statements which enable the programmer to state the flow of his program.

Unconditional
GO TO

| 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

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

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

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

Assigned
GO TO

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

This statement causes transfer of control to the statement with statement number equal to that value of n which was last assigned by an ASSIGN statement; n_1, n_2, \dots, 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, \dots, n_m . (See page 26.)

ASSIGN

| 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, \dots, n_m) to transfer control to statement number i where i is included in the series n_1, \dots, n_m .

IF

| 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 with the statement number n_1, n_2 , or n_3 if the value of a is less than, equal to, or greater than zero, respectively.

SENSE LIGHT

| 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 Sense Light i only will be turned On.

IF (SENSE LIGHT)

| 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 with statement number 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)

| 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 with statement number n_1 or n_2 if Sense Switch i is Down or Up, respectively.

IF ACCUMULATOR OVERFLOW

| 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

| 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 with statement number n_1 if an overflow condition is present in either the Accumulator or the Multiplier-Quotient Register, and to n_2 if no overflow is present at all. That is, in 709/7090 FORTRAN, programming either of these statements is equivalent to programming a non-FORTRAN statement, IF OVERFLOW n_1 , n_2 . In 709/7090 FORTRAN, 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 Multiplier-Quotient Register overflows, the register is set to contain the highest possible quantity, i. e., 377777777777₈. The sign is unchanged.

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

IF DIVIDE CHECK

| 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 with statement number 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

| 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 non-subscripted fixed point variable, and m_1, m_2, m_3 are each either an unsigned fixed point constant or non-subscripted fixed point variable. If m_3 is not stated, it is taken 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 of 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 $1 \cdot N(1)$ 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 $2 \cdot N(2)$ 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.

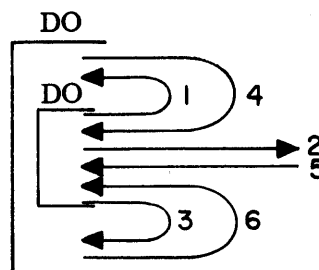
DOs within DOs. Among the statements in the range of a DO may be other DO statements. When this is so, the following rule must be observed:

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 (i. e. , among those statements in the range of a DO which are not in the range of any DO in its range), 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," page 85.)

However, if exit occurs by a transfer out of the range, 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 which 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 non-executable FORTRAN statements. The range of a DO cannot end with a transfer.

Exits. When a CALL statement 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. This applies as well when a Fortran function is called for in the range of a DO.

CONTINUE

| 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

| GENERAL FORM | EXAMPLES |
|--|----------------------|
| "PAUSE" or "PAUSE n" where n is an unsigned <u>octal</u> 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 0. Depressing the Start key causes the program to resume execution of the object program with the next FORTRAN statement.

STOP

| GENERAL FORM | EXAMPLES |
|--|--------------------|
| "STOP" or "STOP n" where n is an unsigned <u>octal</u> fixed point constant. | STOP STOP 77777 |

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 0.

END

| 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 (see below).

For each I of the statement's list,

| | |
|-------|---|
| 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 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 3 — SUBPROGRAM STATEMENTS

It is possible to program, in the FORTRAN language, subroutines which are referred to by other programs. These subroutines may, in turn, refer to still other lower level subroutines which may also be coded in FORTRAN language. It is therefore possible, 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 comprise a distinct subprogram.

This chapter presents a discussion of the two types of FORTRAN coded subprograms possible. These are 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 dealt with in Section A; the other two, CALL and RETURN, are discussed in Section B.

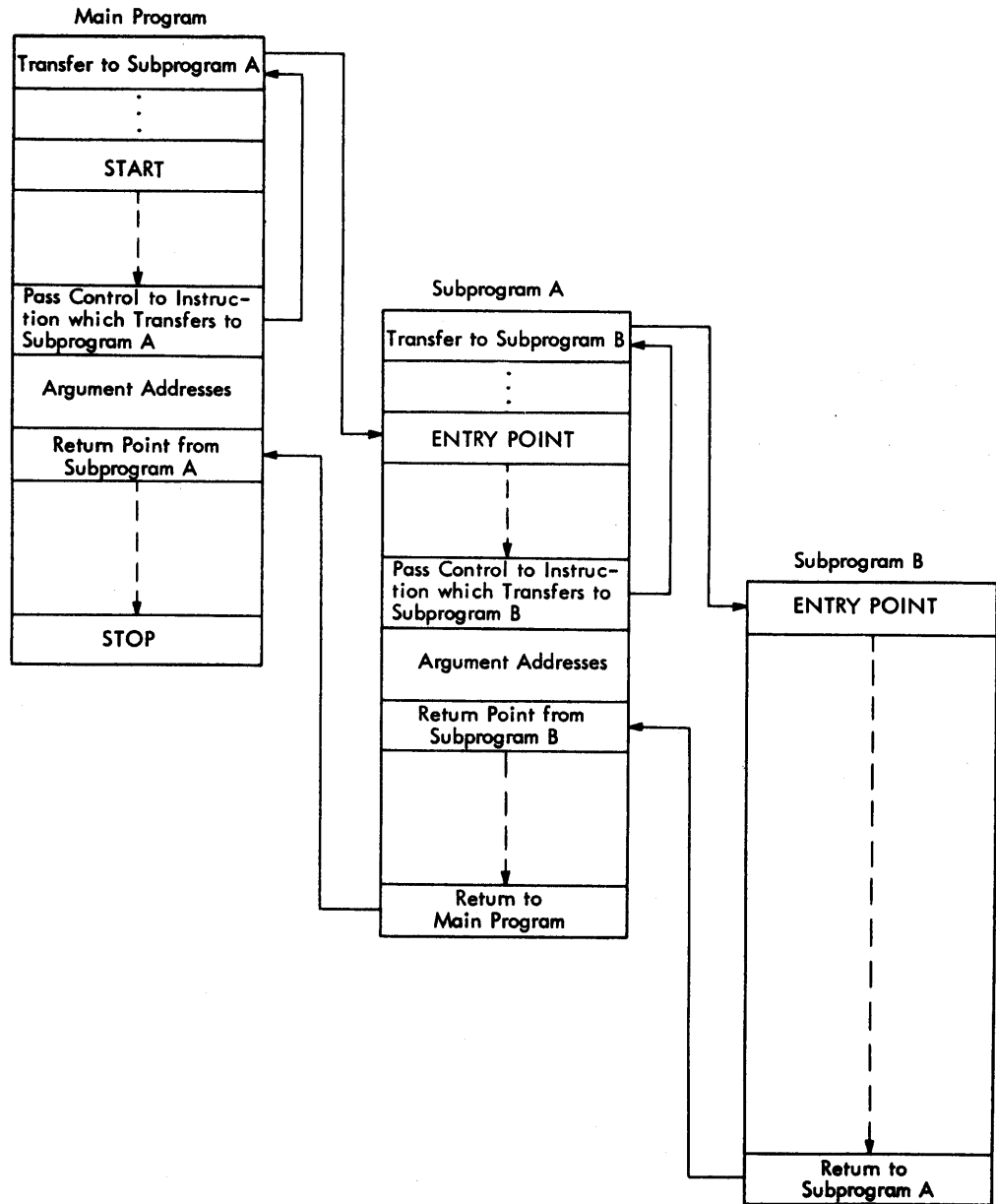
Illustrations of, and the rules for hand-coding subprograms are given on page 101.

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, which results in a Fortran function as defined on page 16, 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 subprogram, when coded in FORTRAN language must be regarded as independent FORTRAN programs. In all respects, they conform 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 and subprograms can be shown as follows. This diagram, further, indicates the main division of the internal structure of each program.



SECTION A: FUNCTION and SUBROUTINE

FUNCTION

| GENERAL FORM | EXAMPLES |
|--|--|
| <p>“FUNCTION Name (a_1, a_2, \dots, a_n)” where Name is the symbolic name of a single-valued function, and the arguments a_1, a_2, \dots, a_n, of which there must be at least one, are non-subscripted variable names, the name of a Subroutine subprogram, or the name of a Fortran function.</p> <p>The function name consists of 1 to 6 alphanumerical characters, the first of which 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, and the final character must not be F if there are more than three characters in the name.</p> <p>Every variable name used as an argument must occur in at least one executable statement of the subprogram.</p> | <p>FUNCTION ARCSIN (RADIAN) FUNCTION ROOT (B, A, C) FUNCTION INTRST (RATE, YEARS)</p> |

The FUNCTION statement must be the first statement of a Fortran 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 either be compiled independently, or 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 as "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 in DIMENSION statements of their respective programs with the same dimensions.

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

SUBROUTINE

| GENERAL FORM | EXAMPLES |
|--|--|
| <p>“SUBROUTINE Name (a_1, a_2, \dots, a_n)” where Name is the symbolic name of a subprogram, and each argument, if any, is a non-subscripted variable name, the name of another Subroutine subprogram, or the name of a Fortran function.</p> <p>The name of the subprogram may consist of 1 to 6 alphanumerical characters, the first of which is alphabetic; its final character must not be F if there are more than three characters in the name.</p> <p>Every variable name used as an argument must occur in at least one executable statement in the subprogram.</p> | <p>SUBROUTINE MATMPY (A, N, M, B, L, C)</p> <p>SUBROUTINE QDRTIC (B, A, C, ROOT1, ROOT2)</p> |

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 (see page 35) in the calling program. The CALL statement specifies the name of the subprogram and its arguments.

Unlike the FUNCTION-type subprogram which returns only a single numerical 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 someplace in the program (or alternately, 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 subprogram MATMPY 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 EQUIVALENCE statements in the Subroutine subprogram. These subprograms may be compiled independently or multiple-compiled with others.

FORTRAN will accept Function and Subroutine names as arguments in other Subroutine and Function subprograms. This permits the Function or Subroutine specified as an argument to be different depending upon the arguments specified in the CALL statements. Thus, the statements

```
SUBROUTINE BOB (DUMMY, Y)
```

```
·  
·  
·
```

```
A=DUMMYF (Y)
```

will permit the function DUMMYF to vary depending on the CALL statements of the main program.

The statement

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

in the main program would cause the SIN (S) to be computed and placed in storage location A.

Notice that the terminal 'F' of the subroutine name must be dropped when the subroutine name appears as the argument of a CALL or SUBROUTINE statement. This terminal 'F' however, must appear whenever the subroutine name appears within an arithmetic statement.

F CARD

In addition, when a subroutine name is to appear in an argument list of a SUBROUTINE or CALL statement, the subroutine name must appear in an F card. The F card may appear anywhere in the deck containing the CALL statement.

F must appear in column 1. Thus, the F card

```
F SIN, COS
```

would permit either SIN (Y) or COS (Y) to be computed by Subroutine BOB, depending upon which was specified in the CALL statement.

SECTION B: CALL AND RETURN

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

CALL

| GENERAL FORM | EXAMPLES |
|--|--|
| “CALL Name (a_1, a_2, \dots, a_n)” where Name is the name of a Subroutine subprogram, and a_1, a_2, \dots, 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. Alphanumerical 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 alphanumerical fields.

Alphanumerical arguments can, of course, only be used as input to hand-coded programs. (See Appendix D.)

7. The name of a FORTRAN function or another 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

| GENERAL FORM | EXAMPLES |
|--------------|----------|
| "RETURN" | RETURN |

This statement terminates any subprogram, whether of the type headed by a SUBROUTINE 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 physically the 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 4 — INPUT/OUTPUT STATEMENTS

There are thirteen FORTRAN statements available for specifying the transmission of information during execution of the object program, between storage on the one hand, and magnetic tapes, drums, card reader, card punch, and printer on the other hand. These input/output statements can be grouped as follows:

1. Five statements (READ, READ INPUT TAPE, PUNCH, PRINT, and WRITE OUTPUT TAPE) which 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 Hollerith punching, alphanumerical print, or binary-coded-decimal (BCD) tape code, respectively.
2. One statement (FORMAT), which is a non-executable 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) which cause information to be transmitted in binary machine-language.
4. Three statements (END FILE, BACKSPACE, and REWIND) that 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 an 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 were 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 on page 8, 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 $I \times J$ (in this case 2×3) 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 punched on a data card in the form:

| | | | | | |
|------------|------------|------------|------------|------------|------------|
| $A_{1, 1}$ | $A_{2, 1}$ | $A_{1, 2}$ | $A_{2, 2}$ | $A_{1, 3}$ | $A_{2, 3}$ |
|------------|------------|------------|------------|------------|------------|

the items will be stored in locations $N, N-1, N-2, \dots, N-5$, 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 709/7090 FORTRAN, 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

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

The five input/output statements of group 1 (listed on page 37) contain, in addition to the list of quantities to be transmitted, 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.

**Numerical
Fields**

Four forms of conversion for numerical data are available:

| INTERNAL | ← 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-0.9321Eb02b7734276bbb-0.0076 | | | |

(b is included here to indicate blank spaces.)

Alphanumerical
Field

FORTRAN provides two ways by which alphanumerical information may be read or written; the specifications for this purpose are Aw and wH. Both result in storing the alphanumerical information internally in BCD form. The basic difference is that information handled with the A specification is given a variable or 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 or floating point variable name.

The specification wH is followed in the FORMAT statement by w alphanumerical characters. For example

28H THIS IS ALPHANUMERICAL DATA

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

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

1. 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-93.210bbbbbbbbb
XY=9999.999bbOVFLOW
XY=bb28.768bbbbbbbbb
```

(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 stores 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 $0 \leq w \leq 120$ (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 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) might give:

```
27 -0.9321E 02 -0.7580E-02 0.5536E 00
```

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 such 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

```
27 -932.096 -0.076 5.536
```

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

27 -9.321 -0.001 0.055

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

27 -9.3210E 01 -7.5804E-03 5.5361E-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 a 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 alphanumerical 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 numerical field is found and list items remain to be transmitted, input/output takes place according to the specification and scanning of the FORMAT statement resumes. If no items remain, transmission ceases and execution of that particular input/output statement is terminated. Thus, a decimal input/output operation will be brought to an end when a specification for a numerical 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

FORTTRAN will accept a variable FORMAT address. This provides the facility of specifying a list at object time.

Example:

| C FOR COMMENT | | | FORTRAN STATEMENT |
|------------------|---|----|---|
| STATEMENT NUMBER | 5 | 67 | |
| 1 | | | D I M E N S I O N , F M T , (1 , 2) |
| | 1 | | F O R M A T , (1 , 2 A , 6) |
| | | | R E A D , 1 , (F M T (1) , I = 1 , 1 , 2) |
| | | | R E A D , F M T , A , B , (C (1) , I = 1 , 5) |

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

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

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 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 and that character 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.

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 numerical 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. 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.

* See the section on Carriage Control in the Reference Manual for the IBM 709 Data Processing System (Form A22-6536).

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 over-rides the indicated position in the FORMAT specification.

READ

| GENERAL FORM | EXAMPLES |
|--|--|
| "READ n, List" where n is the statement number of a FORMAT statement, and List is as described on page 37. | READ 1, ((ARRAY (I, J), I = 1, 3), J = 1, 5) |

The READ statement causes the reading of cards from the card reader. For 709 FORTRAN, the Data Synchronizer Channel to which the card reader is attached, must be specified by the installation (see "Symbolic Input/Output Unit Designation" page 47). 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

| 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 on page 37. | 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 i (709, 0 < i < 49; 7090, 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 709/7090 FORTRAN 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 709/7090 FORTRAN 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 a control card into the edit deck of the 709/7090 FORTRAN system. (See "The 709/7090 FORTRAN Editing Program," in the 709/7090 FORTRAN 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 709. 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 the 709/7090 FORTRAN object program is executed, the equivalence between the logical and actual input/output units must be known.

READ TAPE

| GENERAL FORM | EXAMPLES |
|--|--|
| <p>"READ TAPE i, List" where i is an unsigned fixed point constant or a fixed point variable, and List is as described on page 37.</p> | <p>READ TAPE 24, (A(J), J = 1, 10) READ TAPE K, (A(J), J = 1, 10)</p> |

The READ TAPE statement causes the object program to read binary information from symbolic tape unit i (709, $0 < i < 49$; 7090, $0 < i < 81$), 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 709/7090 FORTRAN compiled program should have been written by a 709/7090 FORTRAN object program. It is, however, 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 128_{10} words long. Of course, if a logical record consists of fewer than 128_{10} words, it will comprise only 1 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.

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

READ DRUM

| 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

| GENERAL FORM | EXAMPLES |
|---|--------------------------------|
| "PUNCH n, List" where n is the statement number of a FORMAT statement, and List is as described on page 37. | PUNCH 30, (A(J), J = 1, 10) |

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

PRINT

| GENERAL FORM | EXAMPLES |
|--|-------------------------------|
| "PRINT n, List" where n is the statement number of a FORMAT statement and List is as described on page 37. | 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.

PK
**WRITE
OUTPUT
TAPE**

| 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 on page 37. | 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 (709, 0 < i < 49; 7090, 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

| GENERAL FORM | EXAMPLES |
|---|--|
| <p>"WRITE TAPE i, List" where i is an unsigned fixed point constant or a fixed point variable, and List is as described on page 37.</p> | <p>WRITE TAPE 24, (A(J), J = 1, 10)</p> <p>WRITE TAPE K, (A(J), J = 1, 10)</p> |

The WRITE TAPE statement causes the object program to write binary information on the tape unit with symbolic tape number i (709, 0<i<49; 7090, 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.

WRITE DRUM

| GENERAL FORM | EXAMPLES |
|--|--|
| <p>"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.</p> | <p>WRITE DRUM 2, 1000, A, B, C, D(6)</p> <p>WRITE DRUM K, J, A, B, C, D(6)</p> |

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

| GENERAL FORM | EXAMPLES |
|---|--------------------------------------|
| <p>"END FILE i" where i is an unsigned fixed point constant, or a fixed point variable.</p> | <p>END FILE 29</p> <p>END FILE K</p> |

The END FILE statement causes the object program to write an end-of-file mark on symbolic tape unit i (709, $0 < i < 49$; 7090, $0 < i < 81$).

REWIND

| 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 (709, $0 < i < 49$; 7090, $0 < i < 81$).

BACKSPACE

| 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 (709, $0 < i < 49$; 7090, $0 < i < 81$).

end of file

CHAPTER 5 — SPECIFICATION STATEMENTS

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

DIMENSION

| GENERAL FORM | EXAMPLES |
|--|---|
| <p>"DIMENSION v_1, v_2, v_3, \dots" 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.</p> | <p>DIMENSION A(10), B(5, 15), CVAL(3, 4, 5,)</p> |

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

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 2-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 which includes the name of the program itself, or any program which it calls.

FREQUENCY

| GENERAL FORM | EXAMPLES |
|---|---|
| <p>"FREQUENCY $n(i, j, \dots), m(k, l, \dots), \dots$" where n, m, \dots are statement numbers and i, j, k, l, \dots are unsigned fixed point constants.</p> | <p>FREQUENCY 30(1, 2, 1), 40(11), 50(1, 7, 1, 1) 10(1, 7, 1, 1)</p> |

The FREQUENCY statement has no direct effect upon the execution of the object program. Its sole purpose is to inform FORTRAN about the number of times which 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 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 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 3 or 4 branch points, 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 which 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.

Statements
to Which
Applicable

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 | 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 (see page 25), 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

| 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 4th storage location following the one which contains C, or (if C is an array) contains C₁, 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, 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:

- A. Execution of an arithmetic formula stores a new value of the variable for the left-hand side of the formula.
- B. Execution of an ASSIGN i TO n stores a new value in n.
- C. Execution of a DO will in general store a new indexing value. (It will not always do so, however; see the section, "Further Details about DO statements," page 85.)
- D. Execution of a READ, READ INPUT TAPE, READ TAPE, or READ DRUM each stores new values for the variables mentioned in the statement list.

COMMON

| GENERAL FORM | EXAMPLES |
|--|-----------------------------|
| "COMMON A, B, ..." where A, B, ... are the names of variables and non-subscripted 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 (see page 79). This area is assigned separately for each program compiled. For 709/7090 FORTRAN, the area is assigned beginning at location 77461₈ 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.

DONT BELIEVE IT!

Array names appearing in COMMON 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 as a medium by which to transmit arguments from the calling program to the called Fortran 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 Fortran function. 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 FORTRAN Operations Manual.)

When a variable is made equivalent to a variable which appears in a COMMON statement, the first variable will also be located in COMMON storage. When COMMON variables also appear in EQUIVALENCE statements, the ordinary sequence of COMMON variables is changed and 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.

```
774618      C and G
```

```
774608      B and E
```

```
774578      A
```

```
774568      D
```



PART III

PROGRAMMING FOR THE 709/7090 FORTRAN MONITOR



CHAPTER 1 — INTRODUCTION TO THE MONITOR SYSTEM

The 709/7090 FORTRAN Monitor System consists of three basic programs: a Monitor, a Compiler, and an Assembler. The Compiler accepts a source program written in FORTRAN 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 709/7090 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 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 709/7090 FORTRAN, FAP, and object programs. It calls in the various System programs as needed. It is necessary only to inform the Monitor what type of processing is to be expected.

MONITOR OPERATIONS

The Monitor permits the following operations:

1. FAP (FORTRAN Assembly Program) assembling.
2. Execution of object programs.
3. 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 which does fit into storage.

MONITOR INPUT.

Input to the FORTRAN Monitor System 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 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 applies to the 32K System and is described below under "Ordering of Job Input Deck."

The 709/7090 FORTRAN 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 or 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

1. FORTRAN Compilation
2. FAP Assembly
3. Relocation of object program input
4. For jobs divided into links, treatments of chain links

Non-Execute

1. FORTRAN Compilation
2. FAP Assembly
(object program input is ignored)

A job may be considered as falling into 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.

Execute Jobs

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.
4. A sequence of input programs for immediate execution. The input programs may be of job types 1 and 2, together with relocatable columnar 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 2 — MONITOR FEATURES

1. The first record of the Monitor is the "Sign-On" record. This may be programmed by the installation to handle accounting or other 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 both from 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 3-18₁₀ and 49-99₁₀ in the 32K system, and 34-36₁₀, 38-39₁₀ in the 8K system.
2. There is a complete set of control cards for the 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 governs output options. In addition to this set of control cards, there are the DUMP card, the START card, and the RESTART cards 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 709/7090 FORTRAN Operations Manual.
3. The FORTRAN Monitor System uses eight tapes on two channels. These 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 follows:

| <u>Logical Designation</u> | <u>Actual Unit</u> |
|----------------------------|--------------------|
| 1 | A1 |
| 2 | B2 |
| 3 | B3 |
| 4 | A4 |
| 5 | A2 |
| 6 | A3 |
| 7 | B4 |
| 8 | B1 |

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, A2 or A3. This is done through the Unit table (IOU) in the FORTRAN Library. (See Description of DSU Channel-Unit Table for FORTRAN in the "709/7090 FORTRAN Operations Manual," form C28-6066-2.)

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 will be the same as conventional FORTRAN programs, with three exceptions.
 - a. The instructions for reading input tape and writing output tape must refer to tapes A2 and A3, 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 columnar binary form are stacked on tape B4 for peripheral punching if on-line card output is not called for. In the 8K System this stacking may be prevented by placing Sense Switch 6 in the Down position. In the 32K System 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. If a chain job, the compiled binary output for each link of the job is contained in a separate file. At the end of the binary output, an END TAPE file is placed.

Ordering of
Job Input Deck
(Applicable to
32K System
Only)

All program decks containing symbolic cards (including control cards) must precede all binary decks which are part of the job. Once a binary card has appeared in the job input deck (or link, for chain job) a symbolic card, with the exception of the DATA card, may not subsequently appear.

CHAPTER 3 — THE CHAIN JOB

In the Chain job, one program which 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 must be either B2, B3 or A4. 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.)
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 cores and executed.

Chain Job Deck
Ordering
(Applicable to
32K System Only)

The rule given on page 65 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, A4 for object time execution. If PDUMP is used, links may 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 one of these links, however, 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

1. 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. HA!
2. Relative Constants. As in the case of main programs and sub-programs within a link, relative constant values may be passed on from one link to another merely by placing them in COMMON. 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 assure the proper subscript values at the time the subscript is used.

CHAPTER 4 — 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.

Governing
the Job as
a Whole;
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.
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 or 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. (Applicable to 32K System only.) This card precedes the I. D. card for a job and is the only card which may precede it. It 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.)

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 over-written, and the END statement which appears in the source program listing will be that fabricated by the Monitor from the control cards. For FAP assemblies, only the first two of these

control cards apply. 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. It may be only

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:

- a. The output tape, A3, contains the information of which the first two files of a compilation in the single compile mode are composed; that is, the source program and the map of object program storage.
- b. The object program in relocatable binary is stacked on tape B4 for peripheral punching without the required library sub-routines.

The type 2 Monitor control cards, and their effects are:

1. CARDS ROW. This card causes the Monitor to punch on-line standard FORTRAN relocatable row binary cards, preceded by a BSS loader, if a main program.
2. CARDS COLUMN. This card causes the Monitor to punch on-line columnar binary relocatable cards (no loader). It prevents stacking of binary output on tape B4 for peripheral punching.

Note that CARDS ROW and CARDS COLUMN cannot be used with the same source program.

3. LIST or LIST8. (The LIST8 card is applicable only to the 32K System.) Each of these cards causes the Monitor 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 4; the LIST8, to END card setting 8.

4. LIBE. This card causes the Monitor to search the FORTRAN library for subroutines during compilation and include these with the object program.
5. LABEL. (Applicable to 32K System only.) This card permits labeling to be obtained on the output cards of a FORTRAN compilation. The label is designated by the programmer in this way: The contents of columns 2-7 inclusive of a card are taken as the label if (a) it is the first card of the program which does not have an * in column 1, (b) the card has a c punch in column 1, and (c) at least one of the columns 2-7 does not contain blank. This label, with blanks treated as zeros, is then placed in columns 73-78 of the output cards with columns 79 and 80 used for serialization. Serialization begins with 00 and re-cycles when 99 is reached. All subroutines obtained with the program are serialized in the same way with their own names in columns 73-78.

If conditions (a), (b), and (c) do not all hold, then labeling is applied in the following way with the LABEL control card present: for a subprogram, the name of the subprogram is used; for a main program, 000000 is used, in columns 73-78. The LABEL card option corresponds to END card setting 7.

Other
Control
Cards

1. There are three other Monitor control cards: FAP, END TAPE and PAUSE.
 - a. 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 FAP assembled. The FAP card follows any Type 2 Monitor control cards that may be used.
 - b. END TAPE. This card designates the end of the last Monitor job. It must be a separate file on the input tape.
 - c. PAUSE. (Applicable to 32K System only.) This card constitutes, in a sense, an executable statement. When this card is processed the machine halts and may be restarted by depressing the Start key. In this way a pause for such purposes as tape reel mounting may be obtained.
2. Other cards, not strictly control cards, may be input to the Monitor.

a. 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.

b. End of File.

This is not a Monitor control card. It is used only when input to the Monitor is on-line. When input is on-line this card is necessary to signal the FORTRAN card-to-tape simulator to write an end-of-file mark which must 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 5 — PROGRAMMING FORTRAN PROBLEMS FOR THE MONITOR

This chapter deals with programming in the FORTRAN language. However, the same requirements, as reflected in machine language, apply to FAP assembly programs as well, 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 709/7090 FORTRAN 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.

Differences Concerning Tape Usage

1. BCD Tape. All input BCD data must be called by the statement `READ INPUT TAPE α , n, List`. Output is effected by a `WRITE OUTPUT TAPE β , n, List` statement, where α and β are the proper logical tape designations for tapes A2 (input) and A3 (output), respectively.

If BCD information is to be written for intermediate storage during program execution, a tape not used by the Monitor must be used (or one the programmer knows the Monitor is not using).

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. 1 - CS is restored and control goes to the Sign-On record to process the next job.
2. CALL DUMP ($A_1, B_1, F_1, \dots, A_n, B_n, F_n$) 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

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

The core dump is effected as specified, 1 - CS is restored, and control is transferred to Sign-On to initiate the next job. 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

| C FOR COMMENT | | | FORTRAN STATEMENT |
|------------------|-----|-----|---|
| STATEMENT NUMBER | 5 | 6 7 | |
| 1 | | | D I M E N S I O N , A (1 0 0) , , C (1 0 0) , , B (1 0 0) , , N (1 0 0) , |
| | | | C O M M O N , B , |
| | | | D O , 2 2 , I = 1 , , 1 0 0 , |
| | | | A (I) , = , F L O A T F (I) , |
| | | | B (I) , = , A (I) , |
| | | | N (I) , = , I , |
| | 2 2 | | C (I) , = , N (I) , |
| | | | C A L L , D U M P , (?) , |
| | | | E N D , |

- 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 includes the array A and the array N as well,

CALL DUMP (N(100), A, 0) or CALL DUMP (A, N(100))

- c. To dump both 1 and 2,

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) simply 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
2. END (I_1, I_2, \dots, I_{15}) where I_i may have the values 0, 1, or 2.

If the first form is used, indicators for the actual program options will be inserted by the Monitor. There are two possibilities with respect to each option indicator.

- a. No Monitor control card is present. The setting prescribed by "standard" FORTRAN output (see page 69) 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 by the standard setting, as above.
- b. A Monitor control card for the indicator precedes the program. In this case, the setting prescribed by this card is inserted.

○ Dumping
During
Execution

The following statement may be used anywhere in the source program. CALL PDUMP (A₁, B₁, F₁ . . . , 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

- a. Although when under Monitor control, a FORTRAN compilation, if desired, will produce row binary cards, the only cards acceptable for Monitor execution are columnar binary. All non-Monitor hand-coded subprograms to be used must have correct associated program cards in proper columnar binary form.
- b. If an error occurs during any of the non-execution 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 print-out 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 non-execution 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 if 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. For the case of unlisted stops, RESTART cards and a DUMP card, which are loaded at the point of stop, are provided. These cards are described in the 709/7090 FORTRAN Operations Manual.
4. For unexpected stops occurring during object program execution, the DUMP or RESTART cards may be used.

Program
Limitations

1. Care must be exercised on jobs involving both compilation/assembly and execution to avoid overlapping of program and common data and to avoid overlapping of program and BSS control. If either occurs, execution will be omitted. Common data may overlap BSS control. (Overlapping of program and BSS control is permitted in the 8K System.)
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 print-out 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, when prepared as Monitor input, if patches are desired.

PART IV

GENERAL RULES FOR FORTRAN PROGRAMMING



CHAPTER 1 — MISCELLANEOUS DETAILS ABOUT FORTRAN

SOURCE AND OBJECT MACHINES

The source machine is that which is used to translate a FORTRAN source program into the object program. The object machine is that on which the object program is executed.

For 709/7090 FORTRAN, the source machine must be an IBM 709/7090 Data Processing System which includes at least 8,192 storage locations, 5 tape units, 1 on-line card punch, 1 on-line card reader, and 1 on-line printer. When multiple-program compiling, 3 additional tape units are required.

The object machine may be of any size. The information produced at compiling time by FORTRAN includes a count of the storage locations required by the object program. From this information it can be determined whether an object program, together with its subprograms, is too large for a given object machine.

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 contiguously by the FORTRAN BSS loader in lower storage with a variable length area separating them from COMMON in upper storage.

Each program block consists of program instructions, constants, erasable storage, and data, which are stored in that order in ascending storage locations. The data is separated into non-dimensioned 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 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 $\geq 2^{19}$, it is possible that the final result will be inaccurate. (The inaccuracy will occur only when the arithmetic expression contains a divide.)
4. Indexing in the object program is modulo (size of core storage) — 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 and/or divisions (or additions and/or 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 is by convention 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 $(5 \times 4)/2$ 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 sub-expressions, takes place only if the expression is suitably written. For example, the arithmetic statement

$$Y = A*B*C + \text{SINF}(A*B)$$

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

$$Y = (A*B) * C + \text{SINF} (A*B)$$

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

In general, when common sub-expressions 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," page 11), and need not be given. Thus

$$Y = A*B+C+\text{SINF}(A*B)$$

is, for optimization purposes, as suitable as

$$Y = (A*B)+C+\text{SINF}(A*B)$$

However, the parentheses discussed in "Ordering within a Hierarchy," on page 11, must be supplied if optimization of common sub-expressions is to occur.

SUBROUTINES ON THE SYSTEM TAPE

Various library subroutines in relocatable binary form are available on the FORTRAN master tape. As mentioned on page 15, 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 709/7090 FORTRAN Operations Manual.
3. Transcribe the resulting card deck onto the master tape by means of the 9LIB program included in the 709/7090 FORTRAN Editor Deck.

Tape subroutines may include Fortran functions 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: Arg₁ will be located in the AC, Arg₂ (if any) in the MQ, Arg₃ (if any) in relocatable location 77775₈, Arg₄ in relocatable location 77774₈, etc. Locations down through 77462₈ are available for common erasable storage for library subroutines.

The output of any function called by an arithmetic statement which is a single value, must be in the Accumulator when control is returned to the calling program. All Index Registers which were stored at the beginning of the subroutine, must be restored prior to returning control.

The arguments for Fortran functions 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 variable in a subscript, which is not under control of a DO, or a DO-implying parentheses in a list. For example, in the sequence:

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

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 Fortran function or Subroutine subprogram.
4. In a COMMON statement.

The following paragraphs describe methods for assuring that the computation for relative constants occur at the proper point between the definition and the use of the relative constant.

Relative Constants in an Input List

In the object program, some computation will take place at each such definition. 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, consequently, the value is used. This may be done by placing it in an argument list. The appearance of a relative constant in an argument list is sufficient to provide the necessary computation for the relative constant.

Relative
Constants in
Common
Statements

A relative constant value may be transmitted from one program to another by placing it in COMMON, but only if it is being transmitted from the calling to the called subprogram.

Example

Main Program

```
.  
. .  
COMMON K  
  
K = 5  
CALL ABC  
  
. .  
. .  
. .
```

```
SUBROUTINE ABC  
COMMON I  
DIMENSION B(10)  
A = B(I)  
  
. .  
. .  
. .
```

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 and simplifies work with triangular arrays. These are simply special cases of the fact that an index under control of a DO is 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)
```


Status of the Cell Containing I

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.

CHAPTER 2 — 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 halt will occur during compilation.

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.

Alphanumeric Arguments

HOLARG Table. Entries are made in this table when a CALL statement lists alphanumerical 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. Maximum table size is 900 (8K), 3600 (32K).

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.

Maximum table size is 139 (8K), 556 (32K).

BETA Table. This table limits the size of arithmetic expressions 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):

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

Maximum table size is 297 (8K), 1197 (32K).

LAMBDA Table. This table limits the size of arithmetic expressions both on 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 = number of literal appearances of variables and constants, except those in subscripts.

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

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

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

e = number of appearances of **.

f = number of literal appearances of function names.

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

Maximum table size is 1185 (8K), 4785 (32K).

Arithmetic
Statements:
Fixed Point
Variables

FORVAL Table. An entry is made for each literal appearance of non-subscripted fixed point variables on the left-hand side of arithmetic statements, in input lists, in COMMON statements, and in argument list for Fortran functions and Subroutine subprograms.

Maximum table size is 1000 (8K), 4000 (32K).

FORVAR Table. An entry is made for each literal appearance of non-subscripted fixed point variables on the right-hand side of arithmetic statements, and in the arguments of IF and CALL statements.

Maximum table size is 1500 (8K), 6000 (32K).

Arithmetic
Statement
Function

FORSUB Table. An entry is made for each distinct Arithmetic Statement function.

Maximum table size is 35 (8K), 140 (32K).

CALL

CALLFN Table. An entry is made for each CALL statement appearing in the source program.

Maximum table size is 600 (8K), 2400 (32K).

COMMON

COMMON Table. An entry is made for each literal appearance of variables in COMMON statements.

Maximum table size is 600 (8K), 2400 (32K).

DIMENSION

DIM Tables. An entry is made for each 1-, 2-, and 3-dimensional variable mentioned in DIMENSION statements.

Maximum table sizes are:

| | (8K) | (32K) |
|---------------|------|-------|
| 1-dimensional | 100 | 400 |
| 2-dimensional | 100 | 400 |
| 3-dimensional | 90 | 360 |

SIZ Table. An entry is made for each array mentioned in a source program.

Maximum table size is 580 (8K), 2320 (32K).

DO

DOTAG Table. An entry is made for each DO in a nest of DOs.

Maximum table size is 50 (8K), 200 (32K).

TDO Table. An entry is made for each DO. (A DO-implying parenthesis counts as a DO.)

Maximum table size is 150 (8K), 600 (32K).

EQUIVALENCE

EQUIT Table. An entry is made for each literal appearance of variables in EQUIVALENCE statement.

Maximum table size is 1500 (8K), 6000 (32K).

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.

Maximum table size is 100 (8K), 400 (32K).

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.

Maximum table size is 450 (8K), 1800 (32K).

○ **FORMAT**

FMTEFN Table. An entry is made for each literal appearance of a **FORMAT** statement number in an input/output statement.

Maximum table size is 500 (8K), 2000 (32K).

FORMAT Table. For each **FORMAT** statement included in the source program, compute *f* 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 *f* values thus computed are entered in the table.

Maximum table size is 1500 (8K), 6000 (32K).

FREQUENCY

FRET Table. An entry is made for each number mentioned in **FREQUENCY** statements.

Maximum table size is 750 (8K), 3000 (32K).

○ **Non-executable Statements**

NONEXC Table. An entry is made for each non-executable statement in the source program.

Maximum table size is 300 (8K), 1200 (32K).

Statement Numbers

TEIFNO Table. An entry is made for each source statement which has a statement number. (An input/output statement which has a statement number and whose list contains controlling parentheses counts as 2.)

Maximum table size is 750 (8K), 3000 (32K).

STOP

TSTOPS Table. An entry is made for each **STOP** and **RETURN** statement in the source program.

Maximum table size is 300 (8K), 1200 (32K).

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.

Maximum table size is 180 (8K and 32K).

Subprograms,
Functions and
Input/Output
Statements

CLOSUB Table. One entry is made in this table for each closed subroutine, Fortran function, and Subroutine subprogram called in the source program. In addition, as many as three entries may be made for each input/output statement.

Maximum table sizes are:

| | |
|-------------------------|-----------------------|
| Total entries | 1500 (8K), 6000 (32K) |
| Total different entries | 750 (8K), 3000 (32K) |

Subscripted
Variables

FORTAG Table. An entry is made in this table for each literal appearance of subscripted variables.

Maximum table size is 1500 (8K), 6000 (32K).

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.

Maximum table size is 30 (8K), 120 (32K).

TAU Tables. An entry is made for each different 1-, 2-, and 3-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 2-dimensional array, and the first and second dimensions of a 3-dimensional array.

Maximum table sizes are:

| | (8K) | (32K) |
|---------------|------|-------|
| 1-dimensional | 100 | 400 |
| 2-dimensional | 90 | 360 |
| 3-dimensional | 75 | 300 |

Transfer
Statements

NLIST Table. An entry is made in this table for each different fixed point variable in an assigned GO TO statement.

Maximum table size is 50 (8K), 200 (32K).

TIFGO Table. An entry is made in this table for each ASSIGN, IF-, and GO TO-type statement in the source program.

Maximum table size is 600 (8K), 2400 (32K).

TRAD Table. An entry is made for each literal appearance of statement numbers mentioned in assigned GO TO and computed GO TO statements.

Maximum table size is 250 (8K), 1000 (32K).



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 DO's which are not yet satisfied, then the normal sequencing of S is ignored and DO-sequencing occurs.)

Non-Executable Statements

The statements **FORMAT**, **DIMENSION**, **EQUIVALENCE**, **FREQUENCY**, and **COMMON** are non-executable 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 | |
|---|---|
| <u>Statement</u> | <u>Normal Sequencing</u> |
| a = b | Next executable statement |
| GO TO n | Statement n |
| GO TO n, (n ₁ , n ₂ , ..., n _m) | Statement last assigned to n |
| ASSIGN i TO n | Next executable statement |
| GO TO (n ₁ , n ₂ , ..., 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 if (a) > 0, respectively. |

| <u>Statement</u> | <u>Normal Sequencing</u> |
|--|---|
| 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 709/7090 FORTRAN internal overflow indicator is On or Off, respectively |
| IF QUOTIENT OVERFLOW n ₁ , n ₂ | Statement n ₁ , n ₂ if the 709/7090 FORTRAN 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 , I ₄ , I ₅) | No sequencing; this statement terminates a problem. |
| CALL Name (a ₁ , a ₂ , ..., a _n) | First statement of subroutine Name. |
| SUBROUTINE Name (a ₁ , a ₂ , ..., a _n) | Next executable statement. |

| <u>Statement</u> | <u>Normal Sequencing</u> |
|--|--|
| FUNCTION Name (a_1, a_2, \dots, a_n) | Next executable statement. |
| RETURN | The statement or part of statement following call. |
| 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, v, v, ... | 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 Hollerith text in a FORMAT statement.



APPENDIX C — SENSE SWITCH SETTINGS FOR 709/7090 FORTRAN

| | | |
|----------------|------|--|
| Sense Switch 1 | UP | Cards containing the object program(s) are punched on-line. Actual tape unit B3 contains the object program of the source program compiled, or, if under Monitor control, of the last source program compiled. |
| | DOWN | Actual 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 | Produces, on actual tape unit B2, two files for the source program compiled, containing the source program and a map of object program storage. If under Monitor control, actual tape unit A3 will contain two files for each program compiled and actual tape unit B2 will contain two files for the last program compiled. |
| | DOWN | Adds a third file for each program compiled (see above) containing the object program in terms of the symbolic code FAP (FORTRAN Assembly Program) on actual tape unit B2 (and A3, if under Monitor control). |
| Sense Switch 3 | UP | No on-line listings are produced. |
| | DOWN | Lists on-line the first two or three files of tape unit B2, depending upon the setting of Sense Switch 2. |
| Sense Switch 4 | UP | Punched output, if any, is relocatable row binary cards. |
| | DOWN | Punched output is relocatable columnar binary cards. |

Sense Switch 5

UP

Library subroutines will not be punched on cards or written on actual tape unit B3.

DOWN

Causes library subroutines to be punched on cards or written on actual tape unit B3, depending upon whether Sense Switch 1 is Up or Down.

APPENDIX D — USING HAND-CODED SUBROUTINES WITH 709/7090 FORTRAN COMPILED OBJECT PROGRAMS

Fortran function subprograms 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 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 Hollerith 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 instructions of a subprogram will consist of a transfer list and a 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 such a transfer list and prologue.

| * FOR REMARKS | | | |
|---------------|-----------|-------------------------------|--|
| LOCATION | OPERATION | ADDRESS, TAG, DECREMENT/COUNT | COMMENTS |
| 1 | 2 | 3 | 4 |
| 12 | 6 7 8 | 14, 15, 16 | |
| S.U.B.P.1 | B.C.D. | 1.S.U.B.P.1 | |
| S.U.B.P.2 | B.C.D. | 1.S.U.B.P.2 | |
| | | | Transfer List |
| S.U.B.P.N | B.C.D. | 1.S.U.B.P.N | |
| | H.T.R. | | Storage for contents of index register 4 |
| | H.T.R. | | Storage for contents of index register 2 |
| | H.T.R. | | Storage for contents of index register 1 |
| NAME | S.X.D. | NAME-3, 4 | Save IR4 contents in location (NAME-3) |
| | S.X.D. | NAME-2, 2 | Save IR2 contents in location (NAME-2) |
| | S.X.D. | NAME-1, 1 | Save IR1 contents in location (NAME-1) |
| | CLA | 1, 4 | |
| | STA | X.1 | Location of 1st argument → X1 ₂₁₋₃₅ |
| | CLA | 2, 4 | |
| | STA | X.2 | Location of 2nd argument → X2 ₂₁₋₃₅ |
| | | | |
| | CLA | n, 4 | |
| | STA | X.n | Location of nth argument → Xn ₂₁₋₃₅ |

RESULTS

A Fortran function 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, of course, 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.
2. Transferring to the calling program. The required steps are:

| * FOR REMARKS | | | |
|---------------|-----------|-------------------------------|-------------------------------|
| LOCATION | OPERATION | ADDRESS, TAG, DECREMENT/COUNT | COMMENTS |
| | 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. The entry point name by which a FORTRAN calling program refers to a FAP subprogram need not have been used in the original symbolic FAP coding.

System Tape Subroutines

As discussed on page 81, 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.

Alphanumerical Information

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

$$nHx_1x_2\dots x_n$$

The following example illustrates the method of storing alphanumerical information.

Example:

```
CALL TRMLPH (8, C, 13HFINAL RESULTS)
```

the characters 13H are dropped, and the remaining information stored:

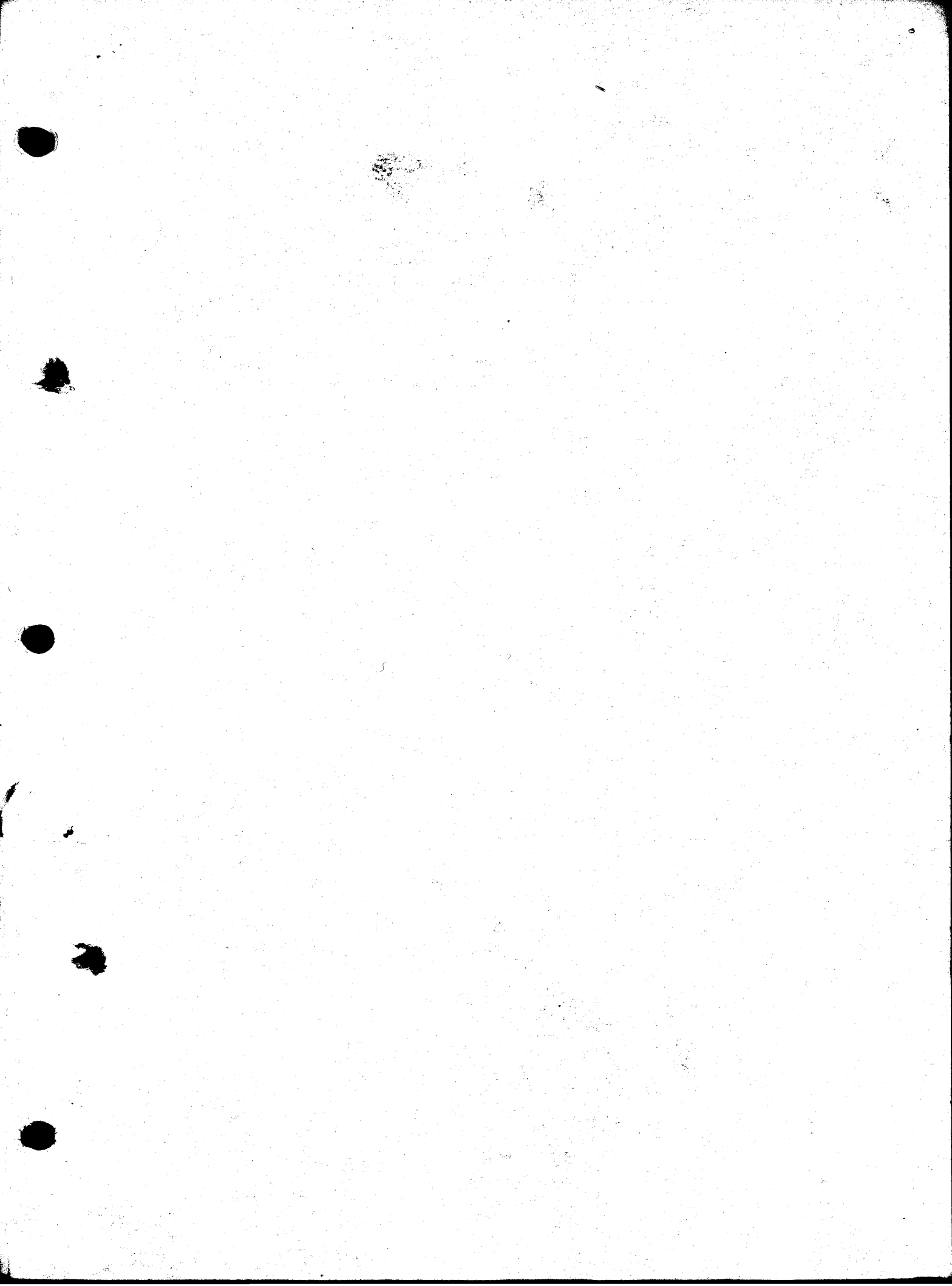
| Location | Contents |
|----------|---|
| X | F I N A L b |
| X+1 | R E S U L T |
| X+2 | S b b b b b (b represents a blank - 60 ₈) |
| X+3 | 777777777777 ₈ |

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

INDEX

| | | | |
|---------------------------------------|--------------------|---|---------------------------|
| Admissible Characters | 97 | Defining Functions | 14 |
| Alphanumeric Fields, conversion | 41 | Diagnostics | 75 |
| Arguments | | DIMENSION | 8, 32, 39, 44, 48, 52, 56 |
| alphanumeric | 35 | DO | 24, 39, 54 |
| in common storage | 56 | index | 24, 26, 54, 85 |
| of a function | 13, 16, 32, 35 | nests | 25 |
| of a library subroutine | 82 | range | 24 |
| (see also "Subprogram, arguments") | | satisfied | 24 |
| Arithmetic Expressions, optimizing | 88 | sequencing | 93 |
| Arithmetic Statements | 5, 19 | transfer exit from | 26 |
| Boolean expressions in | 20 | transfers within range of | 25, 53 |
| functions (see "Functions") | | Dummy Variables | 16, 32, 56 |
| meaning of = | 19 | DUMP Card | 64 |
| mode of result | 19 | Efficiency of Object Program | 5 |
| optimization of | 11, 80 | END | 5, 21, 28, 68, 74 |
| truncation of floating point quantity | 19 | END FILE | 37, 50 |
| Arrays | 6 | End of File Card | 71 |
| arrangement in storage | 8, 56 | END TAPE Card | 64, 70 |
| in FUNCTION subprograms | 32 | EQUIVALENCE | 32, 52, 54, 67, 69 |
| in SUBROUTINE subprograms | 33 | not to be used to equate quantities | 55 |
| Assembler | 61 | Execute Jobs | 63 |
| ASSIGN | 21, 55 | Exponentiation | 9 |
| Assigned GO TO, in range of DO | 26 | Expressions | 6, 9, 15 |
| (see also "GO TO, Assigned") | | F Card | 34 |
| BACKSPACE | 37, 51 | FAP Card | 70 |
| Blank Fields, output records | 42 | Fixed Point | |
| Blanks, ignored | 4 | arithmetic | 79 |
| Boolean | | constants | 6 |
| expressions | 10 | variables | 7 |
| operators | 10 | Floating Point | |
| BSS Loader | 79 | constants | 6 |
| Built-In Functions | | variables | 7, 10 |
| (see "Open Functions") | | FORMAT | 4, 37, 39, 46, 49 |
| CALL | 12, 29, 33, 57 | Format | 40 |
| CALL CHAIN (R, T) | 74 | for alphanumerical conversion | 41 |
| CALL DUMP | 72, 75 | for numerical conversion | 40 |
| CALL EXIT | 72 | lists | 43 |
| CALL PDUMP | 67, 75 | multiple record | 43 |
| CARDS ROW Card | 69 | repetition | 42 |
| CARDS COLUMN Card | 69 | variable | 44 |
| Carriage Control | 44 | FORTRAN | |
| CHAIN | 66, 68 | assembly program (FAP) | 61 |
| Chain Job | 63, 66 | card | 3 |
| Closed Functions | 15 | coding form | 3 |
| Coding Forms | | functions (see "Subprograms, FUNCTION-type") | |
| (see "FORTRAN, coding form") | | types of statements | 5 |
| Comment Cards | 4, 71 | FREQUENCY | 52 |
| COMMON | 52, 56, 67, 79, 84 | FUNCTION | 29, 31 |
| Compiler | 61 | (see also "Subprograms, FUNCTION-type") | |
| Computed GO TO | | Functions | 12 |
| (see "GO TO, Computed") | | arithmetic statement | 12, 15 |
| Control Statements | 5, 21 | closed (library) | 12 |
| Constants | 6 | open (built-in) | 12, 14 |
| Continuation Cards | 3, 4 | modes of | 13, 15 |
| CONTINUE | 27 | naming | 7, 12 |
| DATA Card | 65, 68 | | |
| DATE | 68 | | |
| Data Input to Object Programs | 45 | | |

| | | | |
|--|-------------|---|------------------------|
| GO TO | | | |
| Assigned | 21 | Parentheses | 10, 83 |
| Computed | 21, 54 | PAUSE | 27, 70, 72 |
| Unconditional | 21 | Physical Record | 48 |
| Hand Coded Subroutines | 29, 81, 101 | PRINT | 37, 45, 47, 49 |
| Hierarchy of Operations | 10, 80 | PUNCH | 37, 47, 49 |
| I. D. Card | 64, 68 | READ | 37, 46, 55, 83 |
| IF | 22, 54 | READ DRUM | 37, 48, 55 |
| IF ACCUMULATOR OVERFLOW | 23, 54 | lists | 39, 48, 83 |
| IF DIVIDE CHECK | 24, 54 | READ INPUT TAPE | 37, 46, 55, 72, 74, 83 |
| IF (SENSE LIGHT) | 22 | READ TAPE | 37, 47, 55, 72, 83 |
| IF (SENSE SWITCH) | 23, 54 | Relative Constants | 67, 82 |
| IF QUOTIENT OVERFLOW | 23, 54 | in argument lists | 83 |
| Individual Compilations | 68 | in input list | 83 |
| Input/Output | | RESTART Card | 64, 75 |
| lists | 37 | RETURN | 29, 35 |
| of matrices | 38, 39 | REWIND | 37, 51 |
| statements | 5, 37 | Scale Factors | 42 |
| unit designation | 47 | Selection of Tapes for Link Stacking | 67 |
| Job | 62 | SENSE LIGHT | 22 |
| LABEL | 70 | Sense Switch Settings | 28, 74, 99 |
| LIBE Card | 70 | Sequencing of Statements | 93 |
| Library Functions | | Sign-On Record | 64 |
| (see "Closed functions") | | Source Machine | 79 |
| LIST Card | 69 | Source Program | 3, 79 |
| LIST8 Card | 69 | Source Program Characters | 97 |
| Lists of Quantities | | Source Statements | 3, 4 |
| abbreviated form | 39 | Specification Statements | 5, 52 |
| for transmission | 37 | START Card | 64 |
| for drum | 39, 48 | Statement | |
| Logical Record | 48 | cards | |
| Machine Load | 66 | (see "FORTRAN, card") | |
| Mode of a Function | 13 | numbers | 4 |
| Monitor | 28, 60 | STOP | 27, 72 |
| control cards | 64, 68 | Storage Allocation | 79 |
| diagnostics | 65 | Subprogram | 12, 30 |
| input | 62, 75 | arguments | 32, 34, 56, 84 |
| options | 61 | FUNCTION-type | 12, 14, 16, 29, 31 |
| output | 69 | statements | 5, 29 |
| patches | 76 | SUBROUTINE-type | 12, 29, 32 |
| program limitations | 76 | SUBROUTINE | 29, 32, 34 |
| programming | 65, 72 | (see also "Subprogram, SUBROUTINE-type") | |
| tape usage | 72 | Subroutines | 12 |
| Multiple Program Compilation | 62 | on system tape | 81 |
| Naming | | Subscripts | 6, 10 |
| functions | 13 | Subscripted Variables | 7, 8, 52 |
| FORTRAN functions | 31 | Symbolic Input/Output | |
| subroutines | 32 | Unit Designation | 47, 64 |
| Non-executable Statements | 93 | Symbolic Machine Instructions | 61 |
| Non-execute Job | 62 | Table Sizes | 86 |
| Numerical Fields, conversion | 40 | Triangular Indexing | 84 |
| Object Machine | 79 | Unconditional GO TO | |
| Object Program | 79 | (see "GO TO, Unconditional") | |
| Octal Numbers | 10, 27 | Unit Table (IOU) | 64 |
| Open Functions | 14 | Variables | 6 |
| Operation Symbols | 9, 11 | WRITE DRUM | 37, 49 |
| Ordering of Job Input Deck | 62, 65 | lists | 39 |
| Ordering Within a Hierarchy | 11 | WRITE OUTPUT TAPE | 37, 44, 49, 72 |
| Optimization of Arithmetic Expressions | 11 | WRITE TAPE | 37, 50, 72 |
| | | XEQ | 68, 75 |





IBM[®]

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

