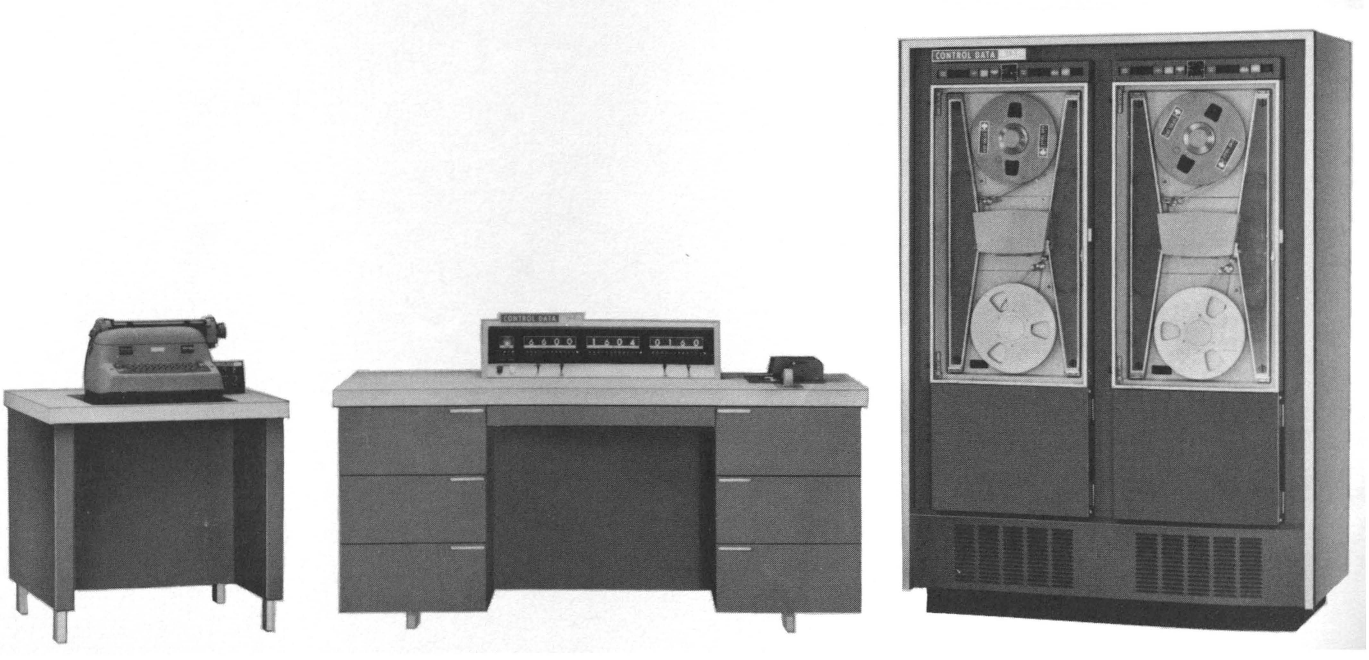
CONTROL DATA

160 COMPUTER

160

160 FORTRAN/REFERENCE MANUAL

CONTROL DATA 160 COMPUTER



160 FORTRAN REFERENCE MANUAL

CONTROL DATA CORPORATION
8100 34th Avenue South
Minneapolis 20, Minnesota

April, 1962

ACKNOWLEDGEMENT

FORTRAN is an abbreviation for Formula Translation. The original FORTRAN was developed by International Business Machines Corporation.

160 FORTRAN REFERENCE MANUAL

Contents

Contents Page

Contents Page

Contents Page

List of Figures

CHAPTER 1

INTRODUCTION

The Control Data 160 FORTRAN system has been developed to alleviate the difficulties normally associated with machine-language coding. The form of its source language resembles the semantics and syntax of the more familiar mathematical notations.

The 160 FORTRAN compiler translates this language to a form intelligible to the 160 computer. The result is a program which precisely reflects the intent of the statements comprising the original source program. At no point in the process, from statement of the problem to its solution, is the user required to have more than a cursory knowledge of the computer.

This manual describes in detail all phases of the system and how they must be applied to effect problem solutions.

A short glossary is provided at the back of this manual; however, it may be well to discuss the usage of a few words before getting into the detailed descriptions. Within 160 FORTRAN, numbers may be represented in two distinct ways. Integers or fixed point numbers (the terms are used interchangeably) are whole numbers from 0 to 2047 and the negatives of these numbers. Floating point numbers are numbers consisting of two parts; one part contains the significant digits of the number; the other part contains a scale factor. Floating point numbers may take on the value zero, values in the range from (and including) 10^{-32} to (but excluding) 10^{31}, and the negatives of these values.

The term "constant" is used only in connection with positive numbers or zero. A negative number may be considered an arithmetic expression, but never a constant. The term "variable" is used to mean a single quantity whose value may change from time to time. Variables may be lumped together in "arrays", but in this manual such an array is never called a variable. One element of an array is a variable, a subscripted variable. A variable which is not an element of an array is known as a simple variable. The term "operand" is used to mean something that is being operated on, usually by an arithmetic operator (+, -, etc.). Variables and constants can be used as operands in arithmetic expressions.

There are many cross references to other sections and chapters within this manual. The user who reads through from the beginning will probably find it unnecessary to look up the references. Certainly it would be best not to go to the forward references until the intervening material has been read. The manual will also be found to be sufficiently repetitious to allow the more experienced user to consult only those sections in which he is interested at the moment.

CHAPTER 2

FORTRAN LANGUAGE MEDIA

2.1 Coding Form

The FORTRAN coding form is illustrated in Figure 1.

The forms are available as 8-1/2 by 11 inch padded sheets. There are 80 columns in which the FORTRAN characters (including blank) are to be written, one character per column. The columns are numbered at the top and bottom of the page and the form contains lines and captions to indicate the correct use of each column.

2.2 Statements

FORTRAN statements, the instructions and descriptions in the FORTRAN language, must be written in columns 7 through 72. Each statement must begin on a new line. Although there must only be one statement on a line, any statement may extend over as many lines as necessary or desirable. Blanks are ignored and may, therefore, be used freely in FORTRAN statements to enhance readability. Do not end any line with slash (/) (section 2.8).

2.3 Statement Type

Certain statements must have a "type" designation. The character which designates the type must be written in column 1 of the first or only line of the statement.

2.4 Statement Number

Any statement may be assigned a statement number for identification, but only those statements to which reference is made from elsewhere in the program are required to have numbers. The statement numbers need not be in any sequence, but within the main program or within any subroutine, no two statements are permitted to have the same number. If there is no type designation other than blank, digits of the statement number are written in any columns from 1 through 5 of the first or only line of the statement. If there is a type designation other than blank, the statement number is limited to columns 2 through 5 of the first line of the statement. A statement number must not be greater than 2047.

2.5 Continuation

Column 6 is used to indicate continuation of a statement from line to line. If a statement is contained on one line, column 6 must be blank or zero. If the statement occupies more than one line, the first line of the statement must have a blank or a zero in column 6. All subsequent lines of a statement must have some FORTRAN character other than blank or zero in column 6.

2.6 Identification Field

Columns 73 through 80 are ignored in the translation process. Many programmers use these columns for identification purposes when the program is to be transcribed on punched cards.

2.7 Punched Cards

Although 160 FORTRAN does not use punched cards, the coding form was designed with 80-column cards in mind. Each line on the coding form corresponds to one card. The terms "line" and "card" are often used interchangeably.

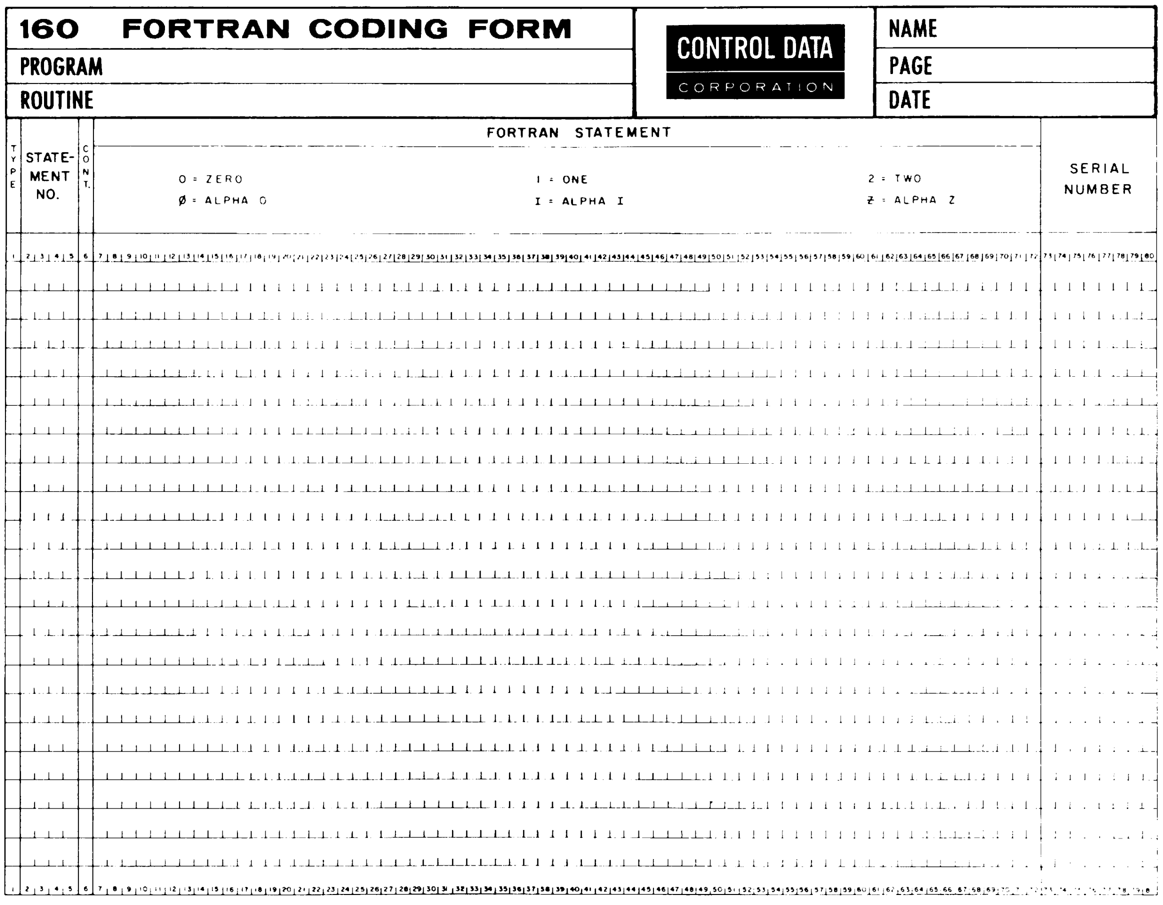
2.8 Punched Paper Tape

Punched paper tape is used for both the input and output media in the 160 FORTRAN system. Input tape is prepared on a Flexowriter. The Flexowriter operator punches certain characters on the tape which are not included in the FORTRAN character set, but which affect the translation process. "Upper Case" and "lower case" codes are needed to distinguish between some special characters, but the case codes do not take up a column of the coding sheet. Until the first case code in the program is encountered, it is assumed that all character codes are in lower case.

The "tab" code does not take up a column, but it has the effect of spacing over to column 7. The "carriage return" code must be used to end a line of coding. It does not occupy a column and it may be used whenever nothing remains in the line other than blanks. If the "tab" code is used at column 6 or beyond, all punching is ignored until the next "carriage return" code.

For the convenience of the Flexowriter operators, the 160 FORTRAN compiler will ignore any line ending in slash followed by carriage return. Therefore, the programmer must not write a slash (/) as the last character on any line of the coding form.

All the Flexowriter codes and the space they occupy within the computer must be taken into consideration when planning input and output to be under control of the FORTRAN Program.



160 FORTRAN CODING FORM

CONTROL DATA CORPORATION

PROGRAM	NAME
ROUTINE	PAGE
	DATE

TYPE	STATEMENT NO.	CONT.	FORTRAN STATEMENT	SERIAL NUMBER
			0 = ZERO Ø = ALPHA O 1 = ONE I = ALPHA I 2 = TWO Ƶ = ALPHA Z	
1	2 3 4 5	6	7 – 72	73 – 80

Figure 1.

CHAPTER 3

FORTRAN CHARACTERS

3.1 Standard FORTRAN II Characters

The FORTRAN language is written using the letters A through Z, the digits 0 through 9, and the special characters, comma, period, blank, *, /, +, -, =, and left and right parentheses. Notice in Figure 1 that standards are prescribed for writing the digits zero, one, and two, and the letters I, O, and Z, in order to avoid confusion.

3.2 Differences in 160 FORTRAN Characters

The Flexowriter has no asterisk (*); therefore, the Flexowriter apostrophe (') is read and written by the compiler in place of the asterisk. Flexowriter operators must be instructed to punch apostrophe in positions which correspond to asterisk on the coding form.

FORTRAN II ignores blanks within statements (columns 7 through 72) except within Hollerith Fields. In 160 FORTRAN, blanks are ignored within statements except that they must not occur in H fields (sections 9.11 and 9.18).

For some input/output conversion (sections 9.8 and 9.16) all Flexowriter codes are acceptable, whether or not they are FORTRAN characters.

CHAPTER 4

THE ELEMENTS OF 160 FORTRAN

4.0 Introduction

The 160 FORTRAN language will be described in terms of the standard FORTRAN characters without regard to differences introduced by the use of Flexowriter tape. Wherever the discussion is in terms of Flexowriter codes, this will be clearly indicated by context.

4.1 Primitive FORTRAN Words

The following words have special meanings in FORTRAN:

ASSIGN	INPUT
CALL	NONLOCAL
CONTINUE	OUTPUT
DIMENSION	PAUSE
DO	PUNCH
END	READ
FORMAT	RETURN
GOTO	STOP
IF	SUBROUTINE

GOTO may be written GO TO and NONLOCAL may be written NON LOCAL since spaces are ignored in FORTRAN statements.

4.2 Names

Names are used to identify simple variables, arrays, subroutines, and functions. Names consist of from 1 to 6 letters and digits. The first character must be a letter. The primitive FORTRAN words must not be used as names and they must not occur as the leading characters of any name. Each name may be used to identify only one simple variable, array, subroutine, or function.

EXAMPLES:

```
B58
MATRIX
SORT
SINF
```

There are further restrictions on names depending on whether they are used to identify simple variables, arrays, subroutines, or functions. These additional restrictions will be explained in the appropriate sections.

4.3 Integer Constants

Integer constants are written using from 1 to 4 digits. They must not contain a period (decimal point) between digits nor at either end. They may have values from 0 through 2047 inclusive.

EXAMPLES:

```
   1        2046
2047          17
  23        1999
```

4.4 Boolean Constants

Boolean constants are octal numbers which represent patterns of twelve bits (binary digits). They are written using from 1 to 4 digits, where each digit is a number from 0 through 7. The following examples show a twelve-bit binary number on the left and the corresponding Boolean constant on the right.

EXAMPLES:

000000000000	0
111111111111	7777
101010101010	5252
011011011011	3333
000000000100	4

Boolean constants can occur only in Boolean statements. Boolean statements are indicated by placing the letter B in column 1 (the type column) of the first line of the statement.

Notice that many Boolean constants look exactly like integer constants. They are distinguished entirely by context. Boolean statements can encompass both integer constants and Boolean constants in distinct roles. This will be explained in the section on Boolean expressions (section 5.5).

4.5 Floating Point Constants

A floating point constant may be written as a string of digits with one leading, trailing, or embedded period which indicates the decimal point. The decimal point must be present explicitly.

EXAMPLES:	2.
	2.5
	.2
	0003.14159

A floating point constant may also be written as described above, followed by one of the following:

E and a one or two-digit integer
E + and a one or two-digit integer
E - and a one or two-digit integer

EXAMPLES	314159.E-5
	.0000314159E05

The number following E indicates the power of ten to be multiplied by the number preceding E, to arrive at the value for the floating constant. Although any number of digits (at least one) may precede the E, the permissible values for floating constants are zero and values in the range from, and including, 1.E-32 to, but excluding 1.E31. Except for positioning the decimal point, only the first eight significant digits are considered.

4.6 Simple Integer Variables

Quantities whose values may change during the operation of a FORTRAN program are known as variables. If such a variable will always be an integer in the range from -2047 to 2047, inclusive, it may be represented by a simple integer variable. Simple integer variables are represented by names of from 1 to 6 letters and digits which begin with one of the letters I, J, K, L, M, or N. General rules for the formation of names are given in section 4.2. The names of variables are further restricted in that, if the name consists of more than three characters, the last one must not be F. The following are examples of simple integer variables.

EXAMPLES:	N	K2S04
	MARY	J567A
	NACL	LOOK

4.7 Simple Floating Point Variables

Simple floating point variables may take on values greater than -1.E31 and less than 1.E31. They are identified by names of 1 to 6 letters and digits which begin with a letter other than I, J, K, L, M, or N. The names of such variables follow the general rules given in section 4.2; and if the name consists of more than three characters, the last one must not be F. The following are examples of simple floating point variables.

EXAMPLES:	A	CHOSEN
	X	PDQ
	U235	RB47

4.8 Arithmetic and Boolean Operators

There are two types of operations in 160 FORTRAN, arithmetic and Boolean. The FORTRAN operators and their arithmetic and Boolean meanings are summarized in Figure 2. In arithmetic A**B means A raised to the power B. Multiplication must always be indicated by the * operator.

The operators are considered Boolean only if they occur in statements with type designation B (in column one of the coding form) and not always then. For instance, subscripts are never Boolean, even if they are present in Boolean statements. The details of the "not always then" are explained in section 5.5. Boolean operators apply to constants and integer variables considered as patterns of 12 binary digits (bits) since these quantities are stored in computer words which are 12 bits long. In Boolean expressions, I**J means shifting the bit pattern represented by I, J places to the left. The shift is end around, which means that bits leaving the pattern at the left reappear at the right. If J is negative, that is, if the bit pattern of J has a one at the left, then I is shifted right by -J, the complement of J. -I means the complement of I, that is, the pattern obtained by changing all the zeros of I to ones and all the ones to zeros. The other three Boolean operators are logical connectives known as AND (*), exclusive OR (/), and inclusive OR (+). Each of these operators applies, bit by bit, to the two twelve-bit patterns represented by the expressions on either side. Figure 3 consists of three operations tables. For bit values shown at the left and top, the intersection of row and column shows the result of the indicated Boolean operation. Figure 4 shows some examples of the results of Boolean operations.

4.9 Subscripts

Subscripts are used, normally, to designate a particular element of an array (which is explained in the next section). A subscript is written in any one of the following seven forms:

```
n
i
m * i
i + n
i - n
m * i + n
m * i - n
```

where m and n represent integer constants and i represents an integer variable. The operators are to be understood in the arithmetic sense even if the subscript occurs in a Boolean statement. The value of a subscript must be kept within the range 1 to 2047, inclusive, whenever the statement containing the subscript is being executed. In general, subscripts are more severely restricted. Each subscript must stay within the corresponding range assigned at the beginning of the program. The DIMENSION statement, explained in section 7.1, is used to assign these ranges.

EXAMPLES:	23	J9F - 165
	N	7 * MQ + 16
	16 * KON	1000 * LOLA - 2
	L99 + 2047	

4.10 Arrays, Subscripted Variables

An array is an associated group of variables called by a single name. The naming rules for arrays are the same as those for simple variables. An array name is 1 to 6 letters or digits, the first of which is a letter. It must not end with F if it is longer than three characters. An integer array, or array of integer variables, must start with I, J, K, L, M, or N. A floating point array must start with some <u>other</u> letter.

A subscripted variable is one element of an array. It is designated by the array name followed by one, two, or three subscripts, enclosed in parentheses and separated by commas. There are three forms, depending on the number of dimensions of the array:

a (s)
a (s,t)
a (s,t,u)

where a represents an array name and s, t, and u represent subscripts.

EXAMPLES:	VECTOR (25 * MOM - 17)
	MATRIX (KILL - 99, I + 3)
	TENSOR (8 * JOY, K9, 88)

4.11 Integer Variables

The term integer variable will mean either a simple integer (fixed point) variable (section 4.6), or a subscripted integer (fixed point) variable (section 4.10).

EXAMPLES:	INDEX
	KARD (80)
	MATRIX (I,J)

4.12 Floating Point Variables

A floating point variable will mean either a simple floating point variable (section 4.7) or a subscripted floating point variable (section 4.10).

EXAMPLES:	A
	EMTRX (MROW,NCOL)
	X (3,5,27)

4.13 Integer Operands

An integer operand is an integer variable (section 4.11) or an integer constant (section 4.3).

EXAMPLES:	99
	I
	J(K,L,M)

4.14 Boolean Operands

A Boolean operand is an integer variable (section 4.11) or a Boolean constant (section 4.4). Boolean operands are distinguished from integer operands in that they occur only in Boolean statements, which are distinguished by having type designation B.

EXAMPLES:	7777
	I
	J(7,8,9)

Note that the subscripts which may help to identify particular Boolean operands are not themselves Boolean.

4.15 Floating Point Operands

A floating point operand means a floating point constant (section 4.5), a floating point variable (section 4.12), or a function (section 5.4).

EXAMPLES:	3.14159	constant
	A(9,M)	array element
	SINF(B + C)	function

FORTRAN Operator	Arithmetic Operation	Boolean Operation
**	Exponentiation	Left Shift
*	Multiplication	AND
/	Division	Exclusive OR
+	Addition	Inclusive OR
-	Subtraction	Complementation

Figure 2.

*	0	1
0	0	0
1	0	1

/	0	1
0	0	1
1	1	0

+	0	1
0	0	1
1	1	1

Figure 3. Boolean Operation Tables

Left Operand	Boolean Operator	Right Operand	Effective Result
000001101010	**	000000000011	001101010000
111000111000	*	000000111111	000000111000
111000111000	/	000000111111	111000000111
111000111000	+	000000111111	111000111111
	-	000000111111	111111000000

Figure 4.

CHAPTER 5

ARITHMETIC EXPRESSIONS AND OTHER PHRASES

5.0 Introduction

This chapter discusses the constructions in 160 FORTRAN which cause manipulation of operands. In general the operations available are those of arithmetic and of Boolean algebra. There are also special procedures, or sequences of operations, known as functions and subroutines. The use of functions will be described in this chapter, but discussion of subroutines will be deferred until chapter 8.

5.1 Arithmetic Expressions

The arithmetic operators have been discussed previously in section 4.8. Exponentiation means raising to a power. For example, ALPHA**3 means ALPHA cubed, or ALPHA raised to the third power. Division of one integer (fixed point) operand by another integer operand yields a truncated integer result, but only if every operand in the expression is fixed point. For example, 8/3 yields 2 as a result.

Arithmetic expressions are made up of strings of operands connected with operators. Operands and operators may be grouped by the use of parentheses. Within a group, the ordinary rules of precedence apply: exponentiation is done first, followed by multiplication and division, followed by addition and subtraction. Within the rules of precedence, operations are performed starting at the left. In the following examples, assume that 3 is the current value of INDEX.

EXAMPLES:	INDEX * 3/2	result is 4
	INDEX * (3/2)	result is 3

Note that multiplication must be indicated by the * operator. There is no operation in FORTRAN expressed by placing operands or expressions next to each other. In any arithmetic expression, an arithmetic operator must stand between any two operands. This is not mitigated by the presence of parentheses. An operand must stand between two operators unless a left parenthesis stands between them and the operator on the right is a minus sign. Parentheses may be included in an expression only if the number of left parentheses equals the number of right parentheses.

The following examples illustrate the various rules governing the order of evaluation. Processing is described step by step in some of the examples.

EXAMPLES:

Operator Priorities

```
A + (B/C**D) - E * F
```

a. Raise C to the D power and store in temporary location TEMP 1.
b. Divide B by the results of (a) and store in TEMP 1.
c. Multiply E times F and store in TEMP 2.
d. To A, add the results of (b), and subtract the results of (c).

Left to Right Rule

```
A * B/C * D
```

a. A is multiplied by B.
b. The results of (a) are divided by C.
c. The results of (b) are multiplied by D.

Parentheses Evaluation Rule

```
((A + B)/C - D) * E
```

a. Add B to A.
b. Divide by C.
c. Subtract D from the results of (b).
d. Multiply by E.

Operand Placement Rule

RIGHT	WRONG
A * B + C	AB + C
A * (B + D)	A(B + C)
(A + B) * (C + D)	(A + B)(C + D)

Operator Placement Rule

RIGHT	WRONG
A * (-B)	A * -B
B/(-A)	B/-A

Arithmetic expressions are either floating point or fixed point as defined in the next two sections.

5.2 Integer Expressions

Integer expressions form a subset of arithmetic expressions. They are strings of integer operands (integer variables and integer constants) separated by arithmetic operators and formed into groups by means of parentheses. In order to be more precise, consider the three following forms:

```
i
i a j
(-i)
```

where i and j represent integer operands (section 4.13) and a represents an arithmetic operator (section 4.8). Then the above three forms are integer expressions.

EXAMPLES:

```
7
K(3,2)
J2 * 2
3 - MAN(2 * I,J,K)
(-28)
(-KID)
```

Let k and m represent integer expressions of the above three forms. Then the following three forms are also integer expressions:

```
k a m
(k)
(-k)
```

EXAMPLES:

```
I + J + K + 3
(I * J)
(-I/J)
```

Extend the meaning of k and m to represent integer expressions of the above forms. This makes the definition of integer expression recursive, allowing strings of operands and operators, nestings, and groupings of any necessary complexity.

```
EXAMPLES:             (I + J) / (K + 3)
                      (-I -J) * (-7 / K)
                      (((K4 * I + K3) * I + K2) * I + K1) * I + K0
```

One more form completes the definition of integer expression: -k

```
EXAMPLES:             -2047
                      -K / (3 * I - J)
                      -(I * I + J * J)
```

Every operand occurring in an integer expression must be an integer operand. Integer expressions may take on values only in the range -2047 to 2047 inclusive. This means that the programmer must avoid calculations which exceed the permissible range at any step. (See section 5.1 for order of calculations.) Within an integer expression, division yields a truncated integer result. For example -15/4 yields -3 as the result.

In evaluating fixed point expressions, exponentiation returns a value of 1 when the right operand (the exponent) is non-positive; for positive exponents the algebraic value of the left operand (the base) is used. This differs from exponentiation in a floating point expression where the absolute value of the base is used (section 5.3).

5.3 Floating Point Expressions

Any arithmetic expression which includes at least one floating point operand is a floating point expression. Let g represent any floating point operand, that is, a floating constant, a floating variable, or a function (section 5.4). Let x represent any floating point or integer operand. Let a represent any arithmetic operator. Then the following four forms are floating point expressions:

```
g
g a x
x a g
(-g)
```

```
EXAMPLES:             G
                      2.7E4
                      1.1 + I
                      3 * ALPHA (I,J,K)
                      (-B29)
```

Let e and f represent floating point expressions of the above four forms. Let k represent any integer expression which does not begin with a minus sign (section 5.2). Then the following five forms are also floating point expressions:

```
e a f
e a k
k a e
(e)
(-e)
```

EXAMPLES:

```
A * B + C * D
A * B - I / J
I / J / ( - X)
(5 * ALPHA)
(-3.2 / I)
```

Now extend the meaning of e and f to include all nine of the above forms. Then the definition of floating point expression becomes recursive, allowing expressions of any necessary degree of complexity. There is one more form which completes the definition of floating point expression:

```
-e
```

EXAMPLES:

```
-.1
-(A(I,J + 3) ** 4)
```

Note that at least one operand occurring in each floating point expression must be a floating point operand. In evaluating floating point expressions, exponentiation uses the absolute value of the left operand even if the right operand is an integer. Floating point expressions are permitted to take on values only greater than -1.E31 and less than 1.E31. Therefore, the programmer must avoid calculations which exceed the permissible range at any step. The order of calculations is given in section 5.1. In the calculation of integer expressions which form part of a floating point expression, the range -2047 to 2047 does not apply and division does not truncate the quotient.

5.4 Functions

A function is a special set of instructions which operates on one or more arguments, yielding a floating point result. There are eight functions presently available in 160 FORTRAN. Each installation can change its 160 FORTRAN library tape to change, add, or delete functions. Details are given in another manual.

A function is invoked by writing a function name followed by the arguments, enclosed in parentheses and separated by commas. Each argument must be an integer or floating point expression. A function name is a name (section 4.2) of four, five, or six characters, the last of which is the letter F.

```
EXAMPLES:          SINF (BETA)
                   COSF (-23 ** SKIDOO)
                   SIGNF (I * A, - J * B)
```

Every function yields a single floating point value depending on the value of its arguments. For purposes of explanation, a function is considered to be a floating point operand. Therefore, a function can be an element of a floating point expression as defined in section 5.3.

```
EXAMPLES:          K * SINF(BETA(2))
                   -SQRTF(X ** X + Y ** Y) / 2
```

Furthermore, the argument of a function can be a floating point expression which includes a function.

```
EXAMPLE:           SQRTF (SINF(ATANF(X))/COSF(ATANF(Y)))
```

Figure 5 is a table listing the eight current function names with the appropriate number of acceptable arguments. The value of each function is also listed. The programmer is still responsible for ensuring meaningful values within the acceptable range. For instance, one should not attempt to obtain the exponent function of a value greater than 71.

5.5 Boolean Expressions

160 FORTRAN provides for manipulation of integer variables by means of Boolean operators as well as arithmetic operators. The operators used are the same FORTRAN operators displayed in Figure 2, but with their Boolean meanings.

A Boolean operand is an integer variable (section 4.11) or a Boolean constant (section 4.4). Let p and q represent any Boolean operands. Let b represent one of the four Boolean operators **, *, /, or + (shift circular left, AND, exclusive OR, inclusive OR). Note that b does not include complementation. Then the following three forms represent Boolean expressions:

```
p
p b q
(-p)
```

EXAMPLES:

```
7777
I(J,K) * M(J,K)
(-MASK)
```

Let r and s represent Boolean expressions of the above three forms. Then the following three forms also represent Boolean expressions:

```
r b s
(r)
(-r)
```

EXAMPLES:

```
I(K - 9) + J(K - 9) + J8(K - 9)
(-MASK8(9 * K) ** 7)
```

Notice in the above examples, that the subscripts of the variables are not Boolean expressions; they are integer expressions. Now extend the meaning of r and s to include Boolean expressions of all the above six forms. Then any necessary complexity of Boolean expressions is possible. There is one more form needed to complete the description of possible Boolean expressions:

```
-r
```

An expression, of the proper form, is Boolean only if it occurs in a statement which has a B in column 1 of the first line (or the only line) of the statement. This B is not part of the statement. It designates the type of the statement and is placed in the type column on the coding form. Any statement which does not have a B in the type column does not contain any Boolean expressions. Boolean expressions can occur in replacement statements (section 6.1) and in IF statements (section 6.7). In such Boolean statements, the subscripts which identify variables are never Boolean and the statement numbers in the IF statement are never Boolean. Also statement numbers in the statement number columns are never Boolean.

The Boolean operators are shown in Figure 2 in Chapter 4. Boolean operators operate on fixed point variables and octal constants considered as patterns of bits (binary digits). The value of a Boolean expression is the integer value represented by the resulting pattern of bits. The use or existence of "truth values" is not implied by the use of Boolean expressions. To understand the Boolean operations, it is necessary to be aware of the way that fixed point numbers are represented by object programs produced by the 160 FORTRAN compiler. Positive numbers are represented in the binary system by 12 bits, the first bit of which is zero. Zero is represented by 12 zero bits. Negative numbers are represented by the complements of the corresponding positive numbers. That is, to obtain a negative number, change all ones to zeros and change all zeros to ones in the corresponding positive number. Hence, the first bit of every negative number is one.

The complement of zero, which consists of 12 zero bits, is 12 one bits. Thereby arises an ambiguity. In testing a value for negative, zero, or positive, the pattern of 12 ones is considered negative, but in performing arithmetic with such a value, it will be treated as zero.

Except for the complement of zero, there is no question, in Boolean expressions, of exceeding the permissible range for integers. In the following examples, the number on the left is the value of a FORTRAN operand, in decimal, and the pattern on the right is the representation of that value within the computer.

EXAMPLES:	3	000000000011
	-3	111111111100
	2047	011111111111
	-2047	100000000000

5.6 Boolean Operations

The ** operator in Boolean expressions means circular left shift. More specifically, I ** J means shift the pattern represented by I circularly to the left J places. If J is negative, the shift is -J places circularly to the right. Circular shift means that bits shifted off one end reappear at the other end. In section 5.5, we have seen the patterns representing 3 and -3. The following examples show patterns which result from use of the shift operator.

EXAMPLES:	-3 ** 3	111111100111
	3 ** 3	000000011000
	-3 ** (-3)	100111111111
	3 ** (-3)	011000000000

The complementation operator (-) in Boolean expressions is a unary operator; it can never stand between two operands unless it immediately follows a left parenthesis.

In Boolean expressions, just as in arithmetic expressions, parentheses are used to enclose subexpressions which are to be evaluated and then treated as single operands in the larger expression. Within parenthetical groups, the precedence rules require that shift operations (**) be performed first, followed by AND and exclusive OR (* and /), followed by inclusive OR and complementation (+ and -). Within the order established by the precedence rules, operations are performed from left to right.

EXAMPLES:	-I * J	means	-(I * J)
	-I + J	means	(-I) + J

SINF (A)	sine of A radians
COSF (B)	cosine of B radians
ATANF (C)	arctangent in radians ($-\pi/2$ to $\pi/2$) of C
EXPF (U)	e raised to the power U †
LOGF (V)	logarithm of $\|V\|$ to the base e †
SQRTF (W)	square root of $\|W\|$
ABSF (X)	$\|X\|$ (absolute value of X)
SIGNF (Y,Z)	$\|Y\|$ if Z is not negative, $-\|Y\|$ if Z is negative

† e is the base of the natural logarithms.

Figure 5.

CHAPTER 6

160 FORTRAN STATEMENTS

6.0 Introduction

The basic statements of FORTRAN are those which calculate the values of variables based on the current values of other variables, examine the results, and change or interrupt the sequence of computations based on the results of these examinations.

6.1 Replacement Statement, The Basic FORTRAN Statement

The fundamental statement in FORTRAN causes the current value of a variable to be replaced by a new value, the currently calculated value of an expression. Let v stand for a variable; simple or subscripted, integer or floating point (sections 4.6, 4.7, 4.10, 4.11, 4.12). Let e stand for an expression; integer, floating point, or Boolean (sections 5.2, 5.3, 5.4, 5.5). Then the following form represents the replacement statement:

v = e

EXAMPLES:

```
A(I,J) = 2.3
I = J * K
N = N + 1
```

Note that this is not the same as an equation in the mathematical sense. The last example above causes N to take on a new value, 1 greater than the old value.

To distinguish between Boolean and integer expressions at the right of the = sign, a B is placed in the type designation column of statements containing Boolean expressions (section 2.3).

The replacement statement may consist of a floating point variable on the left and a Boolean or integer expression on the right. In such a case, the integer value of the expression is converted to floating point in order to replace the value of the variable. There may also be an integer variable on the left with a floating point expression on the right. In this situation, the expression is evaluated in floating point. Any integer operands in the expression are converted to floating point for purposes of calculation. Then the value of the expression is converted and truncated to an integer for replacement as the new value of the integer variable on the left. Usually, however, no conversion is needed because the left hand variable is floating point and the expression is floating point, or because there is an integer variable on the left and a Boolean or integer expression on the right.

The replacement statement, as any statement, may be given a statement number for reference (section 2.4).

6.2 Limitation of Operands

The number of different operands in a replacement statement is limited to 63. This includes the variable to the left of the = sign, each constant, simple variable, array element, and function, whether in the main expression or in the arguments of functions. It does not include operands which occur only within subscripts. If the expression is floating point, the occurrence of exponentiation counts as one operand. Different elements of the same array count as separate operands. Even the same element, if subscripted in different ways, will be considered as separate operands.

The same limitation applies to the number of operands in the expression enclosed in parentheses in an IF statement (section 6.7).

6.3 Unconditional Transfer of Control, GOTO Statement

Normally statements in a FORTRAN program are executed sequentially in the order in which they occur on the coding sheets (hence, on the Flexowriter tape). However, certain statements interrupt this progression at a specific point and transfer control to a specific numbered statement in the program. Statements of the form

```
GO TO n
```

where n is a statement number, cause control to be transferred to the statement whose number, in columns 1 to 5, is n. If the GO TO statement is in the main program, the number must refer to a statement in the main program. If the GO TO statement is in a subroutine, the number must refer to a statement in the same subroutine. The space between GO and TO is entirely optional.

EXAMPLE: GO TO 2047

Statement numbers must be in the range from 1 to 2047, inclusive. Throughout this chapter the discussion will involve statement numbers. In each case, the number must refer to a statement in the same subroutine as that in which the reference occurs. Or if the reference is in the main program, the referenced statement must also be in the main program.

Subroutines are discussed in chapter 8.

6.4 Computed GOTO

Consider statements of the following forms:

```
GO TO (n1, n2),i
GO TO (n1, n2, n3),i
GO TO (n1, n2, n3, n4),i
etc.
```

where i is a simple (non-subscripted) integer variable and n1, n2, n3, n4, etc. are statement numbers. This causes control to be transferred to a point which depends on the current value of the variable i. If i equals 1, control goes to statement n1; if i equals 2, the sequence of computations is transferred to statement n2; etc.

EXAMPLE: GO TO (1962, 832, 1, 17, 23), JUMBO

For values of JUMBO equal to 1, 2, 3, 4, or 5, control is transferred to statement number 1962, 832, 1, 17, or 23, respectively.

At the time of execution of a computed GO TO statement, the value of the associated variable must not be zero and it must not be greater than the number of statement numbers within the parentheses.

6.5 ASSIGN Statement

Statements of the form

```
ASSIGN n TO i
```

where n is a statement number and i is an integer variable, are used to associate the location of a numbered statement with the named variable. The variable may then be used in an assigned GO TO statement (section 6.6).

```
EXAMPLE:              ASSIGN 25 TO JUMP
```

Note that in the above example, JUMP does not have the arithmetic value 25. The actual number that JUMP equals depends on the location of the statement numbered 25. After the value of an integer variable has been set by means of an ASSIGN statement, that same variable may be referenced only in assigned GO TO statements. The integer variable may be set, at any time, to an integer value by means of the replacement statement (section 6.1) or a PAUSE or DO statement, to be explained later. After being given an arithmetic integer value, an integer variable must not be used in an assigned GO TO statement.

```
EXAMPLE:              ASSIGN 25 TO JUMP
                            ...
         25           A = B ** C
                            ...
                      GOTO JUMP, (25,29)
                            ....

EXAMPLE:              JUMP = I + 1
                      INDEX = JUMP * 5
                            ....
```

The question of before or after is directly dependent on the order of execution of the object program. It is only indirectly dependent on the order in which statements appear on the coding sheets.

6.6 Assigned GO TO Statement

Consider statements of the following forms:

```
GO TO    i
GO TO    i(n1,n2)
GO TO    i(n1,n2,n3)
etc.
GO TO    i,(n1,n2)
GO TO    i,(n1,n2,n3)
etc.
```

where i is a simple integer variable and n1, n2, n3, etc. are statement numbers. Whenever such a statement is executed, it is required that the current value of i be the result of execution of an ASSIGN statement (section 6.5).

```
EXAMPLE:          GOTO  KIKI(27, 105, 3, 9, 7)
```

The parenthesized list of statement numbers is entirely unnecessary. It may be included, however, to list the possible statement numbers to which the variable might have been assigned at this point in the program.

The effect of the assigned GOTO statement is to transfer control to the statement designated by the associated integer variable, the current value of the variable having been affected by execution of an ASSIGN statement.

6.7 Conditional Transfer, IF Statement

Statements of the form

```
                         IF (e) n1, n2, n3
```

where n1, n2, and n3 are statement numbers and e is an expression (sections 5.2, 5.3, 5.5), provide conditional transfer of control. If the current value of the expression is negative, zero, or positive, control is transferred to statement numbered n1, n2, or n3, respectively.

```
EXAMPLES:          IF   (ALPHA)     1,2,3
                   IF   (I - 5)     4,5,4
           B       IF   (J * 77)    8,8,9
```

In the last example the B goes in column 1 to indicate that the expression is Boolean. If the rightmost six bits of J are zeros, control goes to statement 8. Otherwise control goes to statement 9. The Boolean expression J * 77 cannot be negative since the constant has a zero in the sign position. In the first example control goes to statement 1 if ALPHA is negative, statement 2 if ALPHA is zero, and statement 3 otherwise. In the second example, if I - 5 is zero control goes to statement 5; otherwise control goes to statement 4. The expression is limited to 63 operands (section 6.2).

6.8 IF OVERFLOW

At any step in evaluation of a floating point expression (section 5.3), if the computed value gets outside the range greater than -1.E31 and less than 1.E31, a special indicator called the overflow switch is turned on and the value of the expression is replaced by a special number called the overflow number. The appropriate sign accompanies the overflow number, which might be considered tantamount to infinity.

EXAMPLE:

```
ALPHA = BETA ** BETA
```

In the above example, if BETA is 30.E1, the value of the expression is about 2.06E44. Therefore, the overflow switch would be turned on and the overflow number would replace the value of ALPHA.

The condition of the overflow switch is tested by statements of the following form:

```
IF  ACCUMULATOR  OVERFLOW  n1, n2
IF  QUOTIENT     OVERFLOW  n1, n2
```

where n1 and n2 are statement numbers. The effects of the two forms above are identical. If the overflow switch is off, control is transferred to statement n2. If the overflow switch is on, it is turned off and control is transferred to statement n1.

EXAMPLE:

```
IF  ACCUMULATOR  OVERFLOW  17,3
```

Evaluation of floating point expressions can only turn the overflow switch on, never off. If one of the values being used in a computation is already the overflow number, the overflow switch will not be turned on. For instance, in the following

EXAMPLES:

```
A = B * C * D
E = -D * C * B
```

if B is 1.E16, C is 1.E17, and D is the overflow number, evaluation of A would cause the overflow switch to be turned on. Calculation of the value of E would <u>not</u> cause the overflow switch to be turned on, although the new value of E would be minus the overflow number.

6.9 IF DIVIDE CHECK

In attempting to divide by zero at any step in evaluation of a floating point or integer expression, a special indicator called the divide check switch is turned on. The resulting value, in the case of floating point, is the overflow number (section 6.8). In the case of integer expressions, the result is zero.

The condition of the divide check switch is tested by statements of the form:

```
IF DIVIDE CHECK n1, n2
```

where n1 and n2 are statement numbers. If the divide check switch is off, control is transferred to statement numbered n2. If the divide check switch is on, it is turned off and control is transferred to statement n1.

EXAMPLE: `IF DIVIDE CHECK 1024, 2047`

6.10 PAUSE Statement

Any of the following forms:

```
PAUSE  n,i
PAUSE  n
PAUSE  ,i
PAUSE
```

where n is a four digit octal number and i is a simple integer variable, will, during execution of the object code, cause the computer to halt. n will be displayed in the accumulator register of the 160 console. If n has not been specified, zero will be displayed.

At this time, if the operator activates the run switch, computation will continue with the next statement. Also, if an integer variable has been specified, its value will be replaced by the contents of the accumulator register.

EXAMPLE: `PAUSE 0707, ILIAD`

6.11 STOP Statement

Either of the following forms:

```
STOP n
STOP
```

where n is a four-digit octal number, will, during execution of the object code, cause the computer to halt. n will be displayed in the accumulator register of the 160 console. If n has not been specified, zero will be displayed. Computation cannot be continued past this point.

EXAMPLE:

```
        STOP 0001
```

6.12 DO Loops

Consider statements of the forms:

```
DO  n i = ml, m2
DO  n i = ml, m2, m3
```

where n is a statement number, i is a simple integer variable, and ml, m2, and m3 are either integer constants or simple integer variables. If m3 does not occur in the statement, it is considered to be the constant 1. n must be the number of a statement beyond the DO statement.

The DO statement causes all the following statements, up to and including the statement numbered n, to be repeated a number of times depending on the values of ml, m2, and m3. During each repetition i takes on one of a series of values also dependent on ml, m2, and m3. The first value of i is ml. For each repetition of the loop, i is increased in value by m3, so long as it does not become greater than m2. The last value of i is less than or equal to m2, unless ml is greater than m2. If ml is greater than m2, the sequence is executed just once and the last value of i is ml.

EXAMPLE:

```
        DO  1  JACK = 2, 9, 4
               A(JACK) = B(JACK, INDEX)
     1         SUM = SUM + A(JACK)
```

EXAMPLE:

```
        DO  2  JILL = 1, 10
     2         V = V + ALPHA(K,JILL) * BETA(JILL,N)
```

In the first example, the short loop consisting of two statements is executed twice and JACK is left with the value 6. In the second example, the one-statement loop is executed ten times, leaving JILL with the value 10. The effect of a DO loop can be achieved with other coding. For instance, the repetitions can be strung out as in the first example below or the loop can be turned by means of a conditional transfer statement as in the second example below. The two following examples accomplish the same effects as the two examples above.

```
EXAMPLE:               A(6) = B(6,INDEX)
                       SUM = SUM + B(2,INDEX) + A(6)
                       JACK = 6

EXAMPLE:               JILL = 1
             20        V = V + ALPHA(K,JILL) * BETA(JILL,N)
                       JILL = JILL + 1
                       IF (10 - JILL) 30, 20, 20
             30        JILL = JILL - 1
```

Within a DO loop, there can be statements which change the value of i. Such statements, of course, can affect the number of repetitions of the loop. For instance, in the following example, the coding indicated by the ellipsis will be executed four times, with K taking on the values 1, 5, 7, and 8.

```
EXAMPLE:               DO  1  K = 1, 8, 5
                                ...
              1        K = K / 2
```

In the following example, the loop would be repeated endlessly with K taking on the values 1, 5, 7, 8, 9, 9, 9.....

```
EXAMPLE:               DO  1  K = 1, 9, 5
                                ...
              1        K = K / 2
```

If m1, m2, and m3 are variables, they too can be changed within the loop. Changing m1 will not affect the number of repetitions after i has been initialized, but changing m2 or m3 will be significant. In the following example, the loop will be executed five times with K taking the values 1, 3, 6, 10, 15.

EXAMPLE:

```
              INC = 1
              DO 3K = 1, 20, INC
                    ...
     3        INC = INC + 1
```

In the following example, the loop will be executed seven times with K taking the values 1, 1, 2, 3, 5, 8, 13.

EXAMPLE:

```
              J = 0
              DO  4  K = 1, 20, INC
                    ...
              INC = J
     4        J = K
```

Within a DO loop, i is not in any way a special variable. It may be set outside the loop; control may be transferred into the midst of the loop; and exit may be made from the loop by means of any conditional or unconditional transfer. All these points are illustrated in the following

EXAMPLE:

```
              K = 5
              GOTO  7
              DO  10  K = 1, 100, 3
     7        X = A(K) / B(K)
                    ...
              IF(X)  7, 15, 9
     9        Y = X ** 3
                    ...
    10        Z = 0
    15        LIMIT = K
```

After exit from the loop, whether by transfer from within or by exceeding m2, i will have the last value it had within the loop. This is illustrated by the two following examples, which are entirely equivalent except for the necessary statement numbers:

EXAMPLE:

```
              BETA = ALPHA
              DO  10  I = M1, M2, M3
              X = SINF(Y(I))
                    ...
    10        Y(I) = COSF(X)
              ALPHA = BETA
```

```
EXAMPLE:             BETA = ALPHA
                     I = M1
          20         X = SINF(Y(I))
                          ...
                     Y(I) = COSF(X)
                     I = I + M3
                     IF(M2 - I)  30, 20, 20
          30         I = I - M3
                     ALPHA = BETA
```

Ordinarily the last statement of a DO loop, Y(I) = COSF(X) in the above example, should not be a conditional or unconditional transfer. The incrementing and testing of the loop variable does not correspond to any statement in the FORTRAN program at the end of the loop. Therefore, it is impossible to put a statement label on the increment and test. Therefore, if the last statement of the loop is a transfer, there is no way to get to the increment and test of the loop variable. In section 6.14 a dummy statement is introduced which can be used to provide a label at the end of a loop.

6.13 Nested DO Loops

It is correct for a DO loop to contain another DO loop. It is often useful to initialize the inner loop variable by means of the current value of the outer loop variable as in the following

```
EXAMPLE:             DO  9  I = 1,20
                     DO  9  J = I,20
                     TEMP = A(I,J)
                     A(I,J) = A(J,I)
           9         A(J,I) = TEMP
```

As in the above example, the nested DO loops may end with the same statement. In this situation, the loop variable initialized last is incremented and tested first. It is also permissible for the inner loop to end before the outer loop as in the following

```
EXAMPLE:             DO  8  I = 1,20
                     DO  9  J = I,20
                          ...
           9         ALPHA(J) = 0
           8         BETA(I) = 0
```

Normally, there is nothing useful to be gained by overlapping the ranges of different DO statements. That is, in general, the DO loop which starts later must end earlier. If this rule is violated as in the following

```
EXAMPLE:             DO  8   I = 1,20
                     DO  9   J = I,20
                                ...
            8        BETA(I) = 0
            9        ALPHA(J) = 0
```

the later loop variable really does not control a loop. It is simply initialized every time around the first loop.

There is no limitation on the depth of nesting permitted in 160 FORTRAN, except for practical considerations of the space available for the program. The following example of three nested DO loops illustrates multiplication of a 4 by 3 matrix by a 3 by 6 matrix:

```
EXAMPLE:             DO  5   I = 1,4
                     DO  5   J = 1,6
                     C(I,J) = 0
                     DO  5   K = 1,3
            5        C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

The following example is another program to transpose a matrix which takes less computation time than the first example:

```
EXAMPLE:             DO  9   I = 1,19
                     J = I + 1
                     GOTO  8
                     DO  9   J = 1,20
            8        TEMP = A(I,J)
                     A(I,J) = A(J,I)
            9        A(J,I) = TEMP
```

6.14 CONTINUE Statement

The complete statement is

```
CONTINUE
```

This statement calls for no computation; it only establishes a statement number. It is normally used following a transfer statement which would otherwise be at the end of a DO loop.

EXAMPLE:

```
              DO  2047  N = 2, 68, 4
                    ...
              IF(A(N))  2047, 13, 2047
      2047    CONTINUE
        13    RAJAH = X ** Y
```

6.15 END Statement

The complete statement is

```
END
```

This statement calls for no computation. Its purpose is to signal the compiler at the end of every program. For this purpose, two END statements in succession are required.

EXAMPLE:

```
                    ...
              STOP  7007
              END
              END
```

The END statement is also used at the end of every subroutine as explained in section 8.4.

CHAPTER 7

DATA STORAGE

7.0 Introduction

Every distinct constant which occurs in a 160 FORTRAN source program is included within the object code. For every distinct simple variable, space is provided. In the case of arrays, it is impossible for the compiler to infer the amount of storage required from examination of the arithmetic statements in the source program. Therefore, a special statement is provided explicitly to reserve space for arrays. This is discussed in the next section. Another statement which provides space for headings and titles to be output is discussed in chapter 9.

7.1 DIMENSION Statement

Consider the following three forms:

```
a(n1)
a(n1, n2)
a(n1, n2, n3)
```

where a is an array name (section 4.10) and n1, n2, and n3 are integer constants greater than one. These forms are descriptions of one, two, and three dimensional arrays, respectively. The constants give the extent of each dimension of the named array and the number of constants gives the number of dimensions. Let d1, d2, d3, d4, etc., be array descriptions in the above forms. Then the following are the forms of the DIMENSION statement:

```
DIMENSION  d1
DIMENSION  d1, d2
DIMENSION  d1, d2, d3
etc.
```

EXAMPLES:

```
DIMENSION ALICE (5,2,18), JACK(3,2)
DIMENSION B29(100), I(5,10,15), ALF(9,6)
```

The DIMENSION statements name all arrays which occur in the main program and give the number and extent of the dimensions of each. The DIMENSION statements must all be placed at the beginning of the FORTRAN source program, following the FORMAT statements (section 9.21).

There may be as many DIMENSION statements as convenient or the arrays may all be described in one DIMENSION statement, but <u>all</u> arrays in the program must be so described except those which are formal parameters of subroutines. Throughout the program, each reference to an array element must have the number of subscripts occurring in the original description and the value of each subscript must stay within the limits in the original description. There are exceptions in the case of input, output, and subroutines, where an entire array may be referenced just by naming it.

CHAPTER 8

SUBROUTINES

8.0 Introduction

In many programs, there are sections of coding, or groups of statements, which are essentially repetitions of other sections or groups, except that different data may be involved. Certain such groups which are relatively common have been put in the 160 FORTRAN library. These are the functions, the use of which is described in section 5.4. This chapter describes another means whereby the 160 FORTRAN user may set aside sections of coding and then call on these sections from various points in his program, thus avoiding the repetitions.

These sections of coding which are set aside are called subroutines. The SUBROUTINE statement introduces a group of statements to be regarded as a subroutine. The END statement terminates the group. The CALL statement calls the subroutine into use from any point within the program. These, and other statements pertinent to subroutines, are discussed in this chapter. 160 FORTRAN permits the use of no more than seven subroutines in any program.

8.1 SUBROUTINE Statement

This statement is one of the forms:

```
SUBROUTINE  s
SUBROUTINE  s(v1)
SUBROUTINE  s(v1, v2)
SUBROUTINE  s(v1, v2, v3)
etc.
```

where s is a subroutine name and v1, v2, v3, etc., known as formal parameters, are the names of arrays or of simple variables. The SUBROUTINE statement introduces a block of statements to be regarded as a subroutine.

```
EXAMPLES:          SUBROUTINE  MULT (A,B,C)
                   SUBROUTINE  SORT
```

The first example might be the beginning of a subroutine which multiplies two matrices. A, B, and C are two-dimensional arrays, the left multiplier, the right multiplier and the product matrix, respectively.

A, B, and C would be known to be two-dimensional arrays because within the body of the subroutine these names would occur only as doubly-subscripted variables and because the actual parameters which correspond to them would be two-dimensional arrays. The second example above might be the heading of a subroutine which sorts the entries of a table, always the same table since there are no parameters.

8.2 NONLOCAL Statement

Statements of the forms:

```
NONLOCAL  m1
NONLOCAL  m1, m2
NONLOCAL  m1, m2, m3
etc.
```

where m1, m2, m3, etc., are the names of arrays or of simple variables, are used within subroutines to designate the names of those variables and arrays which are used within a subroutine, but which belong to the main program.

```
EXAMPLES:        NONLOCAL  TEMP
                 NONLOCAL  TABL, TEMP
```

Many subroutines have need for temporary variables, the values of which are significant at some time during the execution of the subroutine, but not at the time of entering or leaving. It saves space in the object code for every subroutine to use the same temporaries, declaring them to be non-local. Information may be passed between subroutines and the main program by non-local variables instead of parameters. In the second example above, TABL might be the name of the array always rearranged by a SORT subroutine.

Simple variables used within a subroutine must be local, non-local, or they must be formal parameters. Subscripted variables (array elements) used within a subroutine must be elements of arrays which are either non-local or formal parameters. If there are any non-local arrays or simple variables referenced within a subroutine, they must all be listed in a single NONLOCAL statement immediately following the SUBROUTINE statement.

Simple variables used in subroutines, which are not formal parameters and are not listed in the NONLOCAL statement, are automatically local to the subroutine.

8.3 RETURN Statement

The complete statement is

```
RETURN
```

It is used to return control to the main program, or whatever program called the subroutine, when the work of the subroutine is completed.

8.4 END Statement

The complete statement is

```
END
```

It is used to signal the end of the block of coding which constitutes a subroutine. One and only one END statement must be used at the end of every subroutine. It calls for no computation. The END statement is also used to terminate a program (section 6.15).

8.5 Structure of a Subroutine

Every subroutine must begin with a SUBROUTINE statement (section 8.1) and end with an END statement (section 8.4). Somewhere within the subroutine there must be one or more RETURN statements (section 8.3) to get out of the subroutine when its job is done. If any of the variables used by the subroutine are non-local, they must be declared in a NONLOCAL statement immediately following the SUBROUTINE statement. When the subroutine is called, computation begins with the statement immediately following the SUBROUTINE statement or the NONLOCAL statement if there is one.

Besides the four statement types named in the last paragraph, a subroutine may contain all types of 160 FORTRAN statements except READ, PUNCH, and FORMAT (chapter 9) and DIMENSION (section 7.1). Formal parameters must not be used as the loop variable in a DO loop nor as a list variable in an I/O list. All statement numbers which occur in a subroutine are independent of statement numbers in other subroutines or in the main program. Therefore, the statement numbers within a subroutine may, with impunity, duplicate statement numbers outside.

Elements of non-local arrays referenced within the subroutine must be within the range specified by the DIMENSION statements of the program. Elements of formal parameter arrays must be within the range specified for the corresponding actual parameter arrays (section 8.6) by the DIMENSION statements of the program. No other subscripted variables may be used within the subroutine. If simple variables are used which appear neither in the list of formal parameters nor the list of non-local variables, they will be given locations within the object code for the subroutine. The names of such local variables may be the same as the names of variables or arrays in the main program or in other subroutines. The names of formal parameters may also be the same as the names of simple variables or arrays outside the subroutine.

It is impossible to transfer control into a subroutine except by means of the CALL statement (section 8.6). It is impossible to get out of a subroutine except by means of the RETURN statement (section 8.3) or, temporarily, by a CALL statement to another subroutine.

8.6 CALL Statement, Use of Subroutines

The CALL statement is used to invoke a subroutine. It names the subroutine and it specifies the actual parameters to be used in the computation. The forms of the CALL statement are as follows:

```
CALL  s
CALL  s(p1)
CALL  s(p1, p2)
CALL  s(p1, p2, p3)
etc.
```

where s is the name of a subroutine and p1, p2, p3, etc., known as actual parameters, are constants, variables, or array names. There must be the same number of actual parameters in the CALL statement as there are formal parameters in the SUBROUTINE statement with the same subroutine name. The actual parameters replace the corresponding formal parameters wherever they occur in the subroutine. The correspondence is established by relative positions in the lists following the subroutine name in the CALL and SUBROUTINE statements.

```
EXAMPLES:           CALL  MULT (X,Y,Z)
                    CALL  SORT
```

When the subroutine is finished, control is returned to the statement following the CALL statement. If the CALL statement is at the end of a DO loop, control is returned to the increment and test of the loop variable.

Figure 6 lists compatible characteristics between formal parameters and the corresponding actual parameters. 160 FORTRAN permits no more than seven subroutines in any program.

Within a subroutine, there may be a CALL statement invoking another subroutine.

```
EXAMPLE:            SUBROUTINE A(B,C,D)
                    B = C + D
                    RETURN
                    END
                    SUBROUTINE E(F,G,H)
                    X = G / H
                    CALL A(F,X,H)
                    RETURN
                    END
                      ...
                    CALL E(X,Y,Z)
                      ...
```

In the above example, the effect of the call on subroutine E is the same as

```
X = Y / Z + Z
```

Notice that the X in the main program is distinct from the X in subroutine E because X is <u>not</u> declared NONLOCAL.

The depth to which subroutines may call each other is limited to four levels. That is, there may be no more than four CALL statements executed before execution of a RETURN statement. This limitation applies to actual depth during execution of the object code and not to potential depth which may be apparent in the FORTRAN source program. Subroutines may even call themselves either directly or indirectly through other subroutines. The limitation to four levels applies here also, but the parameters at the initial levels are not preserved. This means that any subroutine which is called recursively must not have any output parameters because the reference to the original actual output (result) parameter is destroyed when the subroutine is entered at a lower level. References to the actual input parameters are also destroyed. However, if no reference is made to the formal input parameter after the recursive CALL nor as an actual parameter in the recursive CALL statement, no difficulty will ensue.

EXAMPLE:

```
              SUBROUTINE ARITH (M,N,JOB)
              NONLOCAL Z
              IF (JOB - 2) 1, 2, 4
      1       Z = M + N
              RETURN
      2       J2 = 0
              N2 = N
              DO  3  I = 1, N2
      3       J2 = Z
              RETURN
      4       IF (JOB - 4) 5,5,7
      5       J3 = 1
              N3 = N
              DO  6  J = 1, N3
              CALL ARITH (M1,J3,2)
      6       J3 = Z
              RETURN
      7       J4 = 1
              N4 = N
              DO  8  K = 1,N4
              CALL ARITH (M1,J4,3)
      8       J4 = Z
              RETURN
              END
```

In the above example, it was desired to have an output parameter, Z, to represent the final result. Since this is impossible, Z is used as a nonlocal variable. The value of the input parameter, M, does not change anywhere in the subroutine, but it is incorrect to use M in the recursive CALL statements. Therefore, it is necessary to set another variable, M1, to the value of M and use M1 in the CALL statement. At each stage a new value is used as the N parameter and the current value of N must be saved in three different places. JOB controls the depth of the subroutine and need not be saved after initial entry.

The above subroutine does one of the four arithmetic operations of addition, multiplication, exponentiation, and tetration, depending on the value of JOB which must be 1, 2, 3, or 4, respectively. Tetration is repeated exponentiation. At each level the operation is carried out by repetitions of the next lower level, down to addition. M and N must be positive integers. The answer is given in floating point. The example is for illustrative purposes only.

The following

```
EXAMPLE:            CALL ARITH(Z,4,3,4)
```

calls for tetration of 4 by 3. The required answer is 4 ** 64, which is just beyond the capacity of the 160 FORTRAN system. Furthermore, if the computation could be carried out, it would require in excess of 250,000, 000,000,000,000,000,000,000,000 years. ARITH(Z,3,3,4), which is 3 ** 27, requires only 600 years. ARITH(Z,2,4,4) which is 2 ** 16, requires an hour or two.

Characteristics of formal parameters	Requirements for actual parameters
Integer	Integer
Floating Point	Floating Point
Simple Variable, the value of which is changed by the subroutine	Variable (simple or subscripted)
Simple Variable, the value of which is <u>not</u> changed by the subroutine	Constant or Variable
Array Name (the subroutine uses the name as a subscripted variable)	Array Name (the DIMENSION statement must assign the number of subscripts used within the subroutine)

Figure 6.

CHAPTER 9

INPUT/OUTPUT IN 160 FORTRAN

9.0 Introduction

The acquisition of problem data and the output of answers are discussed in this chapter. All inputs and outputs are handled by paper tape. The READ statement provides for input of data from a tape prepared on a Flexowriter. The PUNCH statement provides punched tape which can be listed on a Flexowriter. The INPUT and OUTPUT statements read and punch, respectively, tape in binary machine language format. The items for input or output are listed in the above named statements. In addition, the READ and PUNCH statements refer to FORMAT statements which specify the details of conversion between the printed data and their binary representations.

9.1 Input-Output (I/O) Lists

Let e and f stand for constants, simple variables, array names, or subscripted variables. Then call e and f I/O lists. Such I/O lists are singular, since they only have one element each; however, the following forms are also I/O lists.

```
e, f
(e, i = m1, m2)
(e, i = m1, m2, m3)
```

Where i is a simple integer variable (called a list variable) and m1, m2, and m3 are integer constants or simple integer variables. Now let e and f include I/O lists of the above three forms. Then apply these forms recursively. This results in quite general structures for I/O lists. The above description of I/O lists includes constants as elements, although it would be inappropriate to input to a constant.

EXAMPLES:

```
A
A,JOE
B(LIST), C(5,3),29.E5,33
(D(5,J),J = 1, 10),(I(J),J = 2, 10,2)
((MAN,15,RILL(MAN,J),MAN = 1, J),J = K,99,3)
(A,B,I,I = I,10)
```

As indicated in the above examples, i may or may not occur as an element of the associated I/O list or as a subscript of such an element. By associated I/O list is meant the entire list preceding i back to the left parenthesis matching the right parenthesis which follows m2, or m3. Parentheses must be used only to enclose subscripts and to enclose a list variable with its associated I/O list.

I/O lists appear in INPUT, OUTPUT, READ, and PUNCH statements and they indicate precisely the constants and/or variables involved in input and/or output. The intended meaning is obvious in the case of I/O lists not containing the "i=" constructions. The "i=" construction with its associated I/O list is treated entirely analogously to a DO loop (section 6.13). If m3 is not specified, it is taken as 1. The following two examples illustrate the effects of the last two examples above:

```
EXAMPLE:          1,15,RILL(1,K),2,15,RILL(2,K),...,K,15,RILL(K,K),1,15,
                  RILL(1,K + 3),2,15,RILL(2,K + 3),...,K + 3,15,RILL(K + 3,
                  K + 3),1,15,RILL(1,K + N * 3),...,K + N * 3,15,R(K + N * 3,
                  K + N * 3)

EXAMPLE:          A,B,I,A,B,I + 1,A,B,I + 2,...,A,B,10
```

Except for subscripts (section 4.9), the arithmetic expressions such as K + N * 3 and I + 2 are not permitted in I/O lists. They are included in the two examples above just to illustrate the effects of the previous examples of I/O lists with loops.

When an array name occurs without subscripts in an I/O list, the effect is the same as listing all the elements of the array, with the first subscript varying most rapidly. To illustrate, if the dimensions of A are 3,2,2, the following three I/O lists are equivalent:

```
EXAMPLES:         A
                  (((A(I,J,K),I = 1,3),J = 1,2),K = 1,2)
                  A(1,1,1),A(2,1,1),A(3,1,1),A(1,2,1),A(2,2,1),A(3,2,1),
                  A(1,1,2),A(2,1,2),A(3,1,2),A(1,2,2),A(2,2,2),A(3,2,2)
```

9.2 INPUT Statement

The forms are

```
INPUT e
INPUT, e
```

where e is an I/O list as described in section 9.1. None of the elements of the I/O list should be constants, or the value of the constant will be changed.

EXAMPLES:

```
INPUT  N,A(N)
INPUT  N,(A(I),I = 1,N)
INPUT  (N,A(I),I = N,10)
INPUT  (A(N),N,B(N),N = 1,10)
```

The first example causes input of a value to N, then input to an element of A, the particular element designated by the value of N just read in. The second example inputs N and then N elements of A. In the third example, I takes on the value of N before any input takes place. Then a value is read into N. Then a value goes to the element of A designated by I, which is the element designated by N _before_ any input. Then I is incremented by 1. What happens to I is independent of any new values of N. The loop is then repeated; another new value goes to N and a new value is input to the next element of A. The loop is repeated for each value of I, up to and including 10. I has the value 10 after input is complete. If the original value of N is greater than 10, the loop is executed once and I ends up with the old value of N. Such an I/O list structure is not likely to be used since it calls for several values to be input to N, only the last of which can have any effect on later computations.

In the fourth example N is set to 1, then input goes to A(1), to N, and to B(N), depending on the new value of N. Suppose the new value of N is 6, N is then incremented, making it 7, and input goes to A(7) and to N. Suppose this makes N equal to 10, then input goes to B(10). Since N is now 10, the input is done. The loop will continue to be repeated until a value of 10 or greater is input to N.

The information which the INPUT statement reads from tape always begins with a special symbol which is not put into any of the listed items. The tape is moved forward until the special symbol, a punch in the seventh level, is found. Following the special symbol there must be floating point and integer numbers to match the floating point and integer quantities specified by the I/O list. The binary patterns on the input tape are put into the specified places in exactly the same configuration, without any conversion. One machine word of twelve bits is provided for each integer variable. Three words are provided for each floating point number. Thus two frames are required on the tape for integer input and six frames for floating point input. There can be input for several INPUT statements on one tape or on more than one tape, but all the input to be processed by one execution of an INPUT statement must be on a single tape with no extra blank tape within the significant information. Blank tape is read as zeros. The computer operator must somehow be informed when and if a new input tape is required.

9.3 OUTPUT Statement

The forms are

```
OUTPUT  e
OUTPUT, e
```

where e is an I/O list as described in section 9.1.

EXAMPLES:

```
OUTPUT  18,N,A(N)
OUTPUT  (N,A(I),I = N,10)
OUTPUT  (A(N),N,B(N),N = 1,10)
```

The first example causes output of three quantities: 18,N, and the Nth element of array A. The second example causes output of N, A(N), N, A(N + 1), N, A(N + 2), N, ..., N, A(10). It also causes I to end up with the value 10, unless N is greater than 10. Note that N is output every time a different element of A is output. If N is greater than 10, the output consists just of N and A(N) and I is left with the value N. In the third example, the output consists of A(1), 1, B(1), A(2), 2, B(2), ..., A(10), 10, and B(10), while N ends up with the value 10.

The OUTPUT statement first writes a special character on the paper tape which consists of a single punch in channel 7. The output specified by the I/O list then follows in binary, just as it is found within the computer, six binary digits (bits) per frame. A frame is a single line across the paper tape. Fixed point quantities occupy one word of twelve bits within the computer. Floating point quantities occupy three words. Therefore, on the paper tape, integer numbers take up two frames and floating point numbers take up six frames. A punched hole represents a 1-bit, absence of a hole represents a 0-bit. A frame of blank tape represents half a word of zero bits.

If it is expected that the information produced by an OUTPUT statement will be read by an INPUT statement, it must all be punched on one tape.

9.4 Conversion Specifications

For input from or output to a Flexowriter tape, it is necessary to relate the configuration of data outside the computer to the variables (or constants in the case of output) within the computer. There must be a conversion specification for each quantity to be read or punched. This includes the requirement for an explicit conversion specification for each element of an array even though only the array name occurs in the I/O list, without subscripts. There are exceptions since some of the conversion specifications will be repeated if necessary. This is explained in section 9.20.

There are conversion specifications which do not correspond to items in the I/O list of the READ (section 9.23) or PUNCH (section 9.22) statement. These are specifically the X and H conversions. X and H conversions are not to be counted in matching conversion specifications to I/O list items.

Conversion specifications are listed in FORMAT statements (section 9.21). The FORMAT statements are referenced in READ and PUNCH statements. The same conversion specifications can apply to both input and output, but the effects are not always compatible. In the following sections, the output effects are given for all conversion specifications. Then the input effects are given.

9.5 Punch Conversions

The I and O conversions apply to integer (fixed point) variables and constants. The A conversion applies to integer variables which contain the representations of Flexowriter characters and control codes. The E and F conversions apply to floating point variables and constants. The H conversion is a specification of a string of Flexowriter characters; no variable or constant is involved. The X conversion is just a specification for spaces to be inserted in the output; no variable or constant is involved. The next seven sections discuss these conversions in more detail.

9.6 Punching by I Conversion

The form of the specification is

I n

where n is an integer, greater than zero, which specifies the width of the field, that is, the number of character spaces provided on the typed output page. A specification of this form may be used only for punching of an integer (fixed point) variable or constant.

EXAMPLES:

```
I 10
I 6
I 3
I 1
```

If the value to be punched is positive, its basic width is 4. If it is negative, its basic width is 5, which includes the minus sign. If the width of the field is greater than the basic width, the number is printed at the right of the field, preceded by blanks. If the width of the field is less than the basic width, leading zeros are squeezed out. If squeezing out all lead zeros still leaves a number too big to fit in the field, the digits at the right are truncated. Leading zeros are replaced by blanks except for a zero value. Figure 7 (page 67) shows the effects of various width fields on the printing of several values.

The following form:

m I n

where m and n are each integers greater than zero, represents m repetitions of In.

EXAMPLE: 3I6

The above example is equivalent to the following

EXAMPLE: I6, I6, I6

9.7 Punching by O Conversion

The form of the specification is

On

where n is an integer, greater than zero, which specifies the width of the field and O is the letter. Specifications of this form may only be used for punching of integer variables or constants. The O specification causes printing of the octal equivalent of the twelve-bit representation of the integer value. Leading zeros are not suppressed. If n is less than 4, the n leading digits are printed. If n is greater than 4, the field begins with n-4 blank spaces.

EXAMPLES: 010
0 4
0 2

The following form:

mOn

where m and n are both integers greater than zero, represents m repetitions of On.

EXAMPLE: 307

The above example is equivalent to the following

EXAMPLE: 07, 07, 07

9.8 Punching by A Conversion

The forms of the specification are

A2
Al

These specifications apply to punching of integer variables or constants only. The tape is punched with the bit pattern of the integer value. A2 causes punching of two frames corresponding to the left and right halves, respectively, of the 12-bit integer. Al conversion causes punching of just one frame of six bits from the right half of the 12-bit integer.

The A conversions are intended to be used for the punching of Flexowriter codes. The extreme freedom allowed permits the punching of all Flexowriter codes, including blank tape, "delete", "stop", "color shift", and "back space" codes.

The following forms:

n Al
n A2

where n is an integer greater than 1, represent n repetitions of A1 or A2, respectively.

EXAMPLE: 2A2

The above example is equivalent to the following

EXAMPLE: A2, A2

9.9 Punching by E Conversion

The form of the specification is

E m.n

where n is an integer between 1 and 8 inclusive and m is an integer greater than or equal to n + 6. The field width is given by m. The number of significant digits is n. The E conversion specifies punching of floating point variables and constants only.

EXAMPLE: E15.7

The printing format specified by E conversion is minus sign or blank, decimal point, n digits (the first one not zero), E, minus sign or blank, two digits. If the field width is greater than n + 6, extra blanks are used at the left. The number following E is the power of ten to be multiplied by the decimal fraction preceding E to arrive at the true value. Ordinarily the first digit following the decimal point is not zero, but if the floating point number is zero, all digits in the decimal fraction will be zero and the power of 10 will be -32. Output of the overflow number (section 6.8) results in a zero in the first position after the decimal point and 31 as the power of 10. Figure 8 (page 68) shows the effects of printing several values under control of E15.7 conversion. In figure 8, b represents a blank space.

Floating point numbers are stored within the computer with an accuracy of eight significant decimal digits. If the output conversion calls for fewer than eight digits, the excess significance is truncated without rounding.

The following form:

k E m.n

where k is an integer greater than 1 and m and n are integers as described above, stands for k repetitions of Em.n.

EXAMPLE: 4E8.2

The above example is equivalent to the following

EXAMPLE: E8.2, E8.2, E8.2, E8.2

9.10 Punching by F Conversion

The form of the specification is

F m.n

where n is zero or a positive integer less than 9 and m is an integer greater than or equal to 3 and at least as large as n + 2. The field width is given by m. The number of digits after the decimal point is given by n. There is no multiplying factor printed. The number of digits before the decimal point is determined by the value to be printed. Two spaces are required for a sign and a decimal point. Therefore, the number of integer digits is limited to m-n-2. The F conversion applies to punching of floating point variables and constants only.

EXAMPLES: F12.6
F12.10
F10.4

The printing format is more easily described starting at the right of the m character field: n fractional digits, a decimal point, required number of integer digits, blank or minus sign, blanks to fill out the field. There will never be more than eight digits printed. Excess digits after the decimal point are replaced by blanks. If conversion of a value requires more than eight or more than m-n-2 integer digits and if m and n are compatible with an E conversion (section 9.9) then the value will be punched as if an E conversion had been specified. If the number cannot be printed under the F specification and if m and n are <u>not</u> compatible with an E specification, x's will print instead of the number.

The following form:

k F m.n

where k is an integer greater than 1 and m and n are integers as described above, stands for k repetitions of Fm.n.

EXAMPLE: 2F12.2

The example above is equivalent to the following

EXAMPLE: F12.2, F12.2

9.11 Punching by H Specification

The forms of the specification are as follows:

```
1Ha
2Hab
3Habc
etc.
```

where a, b, c, etc., are semicolons or any 160 FORTRAN characters except blanks.

EXAMPLES:

```
 6HTITLES
 7HNAMES;=
24HAND;**WILD;ANIMAL**;ACTS
11H*-/)URK(/-*
```

This specification prescribes a string of printed characters to appear in the Flexowriter listing. The number before H tells how many characters are to be printed and this many characters must be written following H. Parentheses are among the characters which may be specified. Asterisk (*) will print as apostrophe ('). Blanks must not appear in the specification. At places where a blank is to appear in the printed output, a semicolon (;) is used in the specification.

The H specification provides all the information for output. It does not need and does not correspond to any variable or constant in the program, nor to any element of an I/O list. Repetition of an H specification can be expressed by enclosing the specification in parentheses, preceded by the number of intended occurrences.

EXAMPLES: 2(3HTUT)

The above example is equivalent to the following

EXAMPLE: 3HTUT, 3HTUT

The normal mode of printing is lower case, but if the last character of a printed H specification is upper case, the following printing will be upper case. Semicolon, standing for space, is accompanied by lower case code.

9.12 Punching by X Specification

The form of the specification is

nX

where n is an integer greater than zero.

EXAMPLES: 1X
27X

This specification prescribes that n space codes be punched on tape, and, therefore, that n blank spaces appear at the corresponding positions on the typed output. The X specification does not need and does not correspond to any element of an I/O list.

9.13 Read Conversions

The I, O, and A conversions apply to input to integer (fixed point) variables. The E and F conversions apply to input to floating point variables. The H conversion provides, directly, space within the computer for storage of Flexowriter codes; no variable or constant is involved. The X conversion just specifies that characters on the input tape be skipped. All of the field widths in read specifications refer to characters. Blank frames and delete codes are skipped without counting. Except in the A conversion, case codes do not count, but they are examined to help in identifying the following characters.

9.14 Reading by I Conversion

The form of the specification is

I n

where n is an integer, greater than zero, which specifies the number of characters to be read from the input tape. Case codes and delete codes are ignored and they do not count as characters read. If a slash (/) is encountered it means that no more characters are in the field even if fewer than n characters have been read. It is expected to find only spaces, digits, and (if required) a minus sign. All other characters, including the plus sign, are illegal. Decimal points will count as characters read, but they will not enter into the conversion. Other illegal characters, including carriage return, will be treated as zeros. For instance .1.P6 on an input data tape to be converted according to I5 will read in as the value 106. The I conversion applies to reading of values for integer variables only. Integer variables may only take on values in the range from -2047 to 2047, inclusive.

The following form:

mIn

where m is an integer greater than 1, represents m repetitions of In.

EXAMPLE: 4I2

The above example is equivalent to the following

EXAMPLE: I2, I2, I2, I2

9.15 Reading by O Conversion

The form of the specification is

On

where n is an integer, greater than zero, which specifies the number of characters to be read from the input tape, and O is the letter. Case codes and delete codes are ignored and they do not count as characters read. If a slash (/) is encountered, it means that no more characters are in the field even if fewer than n characters have been read. It is expected to find only spaces, digits from 0 to 7, and (possibly) a minus sign. All other characters, including the plus sign, are illegal. Decimal points will count as characters read, but they will not enter into the conversion. Although the conversion is octal, the digits 8 and 9 will be included at face value. Other illegal characters, including carriage return, will be treated as zeros.

EXAMPLES:	3849	is equivalent to	4051
	A03.2	is equivalent to	00[illegible]

The O conversion applies to reading of values for integer variables only. The conversion treats the numbers read as octal, [illegible] the value of 1234 will be 1*512+2*64+3*8+4.

The following form:

mOn

where m is an integer greater than 1, represents m repetitions of On.

EXAMPLE: 9O4

The above example is equivalent to the following

EXAMPLE: O4,O4,O4,O4,O4,O4,O4,O4,O4

9.16 Reading by A Conversion

The forms of the specification are

A2
A1

This specification applies to reading of data for integer variables only. A2 causes reading of the bit pattern from two frames of the input tape into the corresponding integer variable. A1 causes reading of the bit pattern from one frame of the input tape into the right half of the corresponding integer variable. The left half of the variable is set to six binary zeros. Any six-bit pattern except delete code or blank tape is read from each frame of the input tape, including the codes for carriage return, slash, and case shifts.

The following forms:

nA1
nA2

where n is an integer greater than 1, represent n repetitions of A1 or A2, respectively.

9.17 Reading by E and F Conversion

The forms of the specification are

```
Em.n
Fm.n
```

where m is an integer greater than zero, n is an integer greater than or equal to zero, and m is at least as great as n. These specifications apply to reading of data for floating point variables only. They cause the reading and conversion of m characters from the data input tape. If a slash (/) is encountered, it means that no more characters are in the field even if fewer than m characters have been read. It is expected to find only spaces, digits, minus signs, and the letter E. All other characters, including the plus sign, are illegal. Illegal characters, including carriage return, and imbedded blanks will be treated as zeros. Delete code is skipped and does not count as a character.

Let s, t, and u represent strings of digits. Then the following forms represent possible fields on the input tape to be converted by E or F conversion:

```
s
.s
s.
s.t
-s
-.s
-s.
-s.t
```

Now let f represent any of the above eight forms. Then the following forms also represent possible fields on the input tape to be converted by E or F conversion:

```
fEu
f-u
fE-u
```

There must not be more than two digits in u. If there is no decimal point in the number on the input tape, then n tells how many of the digits of s are to be considered fractional. If a decimal point is present n is ignored. m tells how many characters of the input tape are to be considered, ignoring case codes. No more than eight digits before the E or the minus sign are converted. Values represented by any additional digits are lost. Figure 9 gives some examples of specifications, input fields, and the resulting values. If the absolute value of the input is less than 1.E-32, it is converted to floating point zero. If the input is as great as 1.E31 in absolute value, it is converted to the overflow number (section 6.8). Repetition of an E or F specification can be indicated by preceding the specification with the desired number of occurrences.

EXAMPLE: 6F4.2

The above example is equivalent to the following

EXAMPLE: F4.2,F4.2,F4.2,F4.2,F4.2,F4.2

9.18 Reading by H Specification

The forms of the specification are as follows:

```
1Ha
2Hab
3Habc
etc.
```

where a, b, c, etc., are semicolons or any 160 FORTRAN characters except blanks. For input, there is no significance to the particular characters appearing after the H. They just serve to reserve space into which characters from the input tape may go. A computer word containing two Flexowriter codes, case code and the particular character code, is set aside for each character after H in the specification. On input, if no case code is found before the character, lower case is assumed. Delete code is skipped. On input, space codes are read as spaces and semicolons are read as semicolons. Repetition of an H specification may be indicated by enclosing the specification in parentheses and preceding with the number of desired occurrences.

EXAMPLE: 3(9HLOOK;AWAY)

The above example is equivalent to the following

EXAMPLE: 9HLOOK;AWAY, 9HLOOK;AWAY, 9HLOOK;AWAY

9.19 Reading by X Specification

The form of the specification is

nX

where n is an integer greater than zero.

EXAMPLES:
```
15X
137X
```

This specification prescribes that n characters of the input tape be skipped. Case codes, delete codes, and blank tape are skipped without being counted. The X specification does not need and does not correspond to any element of an I/O list.

9.20 Specification Lists

The conversion specifications discussed in the previous sections of this chapter can be arranged in lists of individual specifications separated by commas or slashes.

EXAMPLES:
```
E12.6,F8.4,3X,4HMIFF
E20.8/F9.0,3HEND
2E10.4,3F5.0/2I4
```

Parts of the list can effectively be repeated by enclosing the repeated part in parentheses and preceding the parenthetical group with the number, greater than one, of desired occurrences. However, such parenthetical groups must not be nested.

EXAMPLES:
```
2(I6,(3A2/1X))
2(E8.2,F4.2),3(1X/5X,I4)
```

Specification lists can also include, at the end, one parenthetical group without a preceding number. This unnumbered parenthetical group may stand alone at the end of the list, or it may be within a numbered parenthetical group, or it may include a numbered parenthetical group. Specification lists are theoretically infinite; that is, the list is used repetitively as many times as necessary during execution of the READ or PUNCH statement. If the list includes a parenthetical list without a number, it is this unnumbered parenthetical specification list that is considered to be repeated after the complete specification list has been exhausted. In these implied repetitions, the implied separator is the slash rather than the comma. For instance, the fourth example in this section is equivalent to the following

EXAMPLE: I6,A2,A2,A2/1X,I6,A2,A2,A2/1X/A2,A2,A2/1X/A2,A2,A2/1X/...

Specification lists occur in FORMAT statements (section 9.21) which are referenced by PUNCH and READ statements (sections 9.22 and 9.23). As explained above, every specification list is equivalent to a list (without parentheses and without implied repetitions) of specifications separated by commas or slashes. The commas serve only to separate specifications. The slashes separate specifications and affect the processing in a way

I5	I4	I3	I2	I1
0000	0000	000	00	0
5	5	5	5	5
86	86	86	86	8
743	743	743	74	7
1395	1395	139	13	1
- 6	- 6	- 6	-6	-
- 23	- 23	-23	-2	-
- 456	-456	-45	-4	-
-1789	-178	-17	-1	-

Figure 7.

which depends on whether the specification list is being used to control conversion during punching or reading. During punching, a slash means to punch a carriage return character before proceeding to the next conversion. During reading, a slash or the implication of one through repetition of a specification list, means to move the paper tape forward to the next carriage return character and then resume conversion with the frame after the carriage return.

9.21 FORMAT Statement

All format statements must appear at the beginning of the program. The form is as follows:

n FORMAT (s)

where n is a statement number (in the statement number field of the coding form) and s is a specification list as explained in section 9.20. Every FORMAT statement must have a unique statement number. The FORMAT statement is not executed as such. Its purpose is to provide a specification list which is used in converting the variables and constants listed in PUNCH and READ statements (sections 9.22 and 9.23).

EXAMPLES:
```
 6   FORMAT (11HJ;DOE;;;RUN,I3)
23   FORMAT (I4,(E14.8))
```

Value	Printed Output
.0123456	bbb.1234560E-01
.1234567	bbb.1234567Eb00
-5960352.	bb-.5960352Eb07
-.000005	bb-.5000000E-05
27.345	bbb.2734500Eb02

Figure 8. Printing Under E15.7 Specification

In the above examples FORMAT statement 6 might be used in connection with a PUNCH statement in order to identify the output. The run number would be calculated or read in with the input data. FORMAT statement 23 might be used in connection with a READ statement to bring in a number representing the length of an array and then the specified number of floating point values for the array elements, the elements being separated by carriage return. The E14.8 specification would effectively be repeated the required number of times.

Any FORMAT statement may be referenced by either PUNCH or READ statements; however, the effects of the listed specifications in the FORMAT statement may not give compatible results in punching and reading. The meanings of the specifications for punching are described in sections 9.6, 9.7, 9.8, 9.9, 9.10, 9.11, and 9.12. The meanings of the specifications for reading are described in sections 9.14, 9.15, 9.16, 9.17, 9.18, and 9.19.

9.22 PUNCH Statement

The forms are as follows:

```
PUNCH n
PUNCH n,e
PUNCH ne
```

where n is the statement number of a FORMAT statement (section 9.21) and e is an I/O list (section 9.1).

EXAMPLE:

```
PUNCH 6,N
PUNCH 2
```

The effect of the PUNCH statement is to cause punching of paper tape for later listing on a Flexowriter. First a carriage return character is punched. Then all the variables and constants of the I/O list in the PUNCH statement are matched with the conversion specifications of the specification list in the referenced FORMAT statement and conversion and punching proceeds until the I/O list is exhausted. Each item of the I/O list must match its corresponding conversion specification as to being integer or floating point.

Whenever a slash is encountered as a separator between specifications, a carriage return is punched. Whenever an H or X specification is encountered in the specification list, printing or spacing is done according to the specification, but consideration of the next item in the I/O list is delayed until the next E, F, I, O, or A specification is reached. If the end of the specification list in the FORMAT statement is reached before the end of the I/O list in the PUNCH statement, part or all of the FORMAT list is considered repeatedly (section 9.20). If the end of the I/O list is reached before the end of the FORMAT list, punching in accordance with slashes and H and X specifications will proceed until the next E, F, I, O, or A specification or the end of the FORMAT list is reached. If there is no I/O list in the PUNCH statement, the specifications in the FORMAT statement must all be H or X specifications, or they must at least begin with H or X specifications if any punching is to occur.

EXAMPLES:

```
2     FORMAT (20X,3HABC,I5,I5)
4     FORMAT (20X,I5,I5,3HXYZ)
      PUNCH 2
      PUNCH 2,JA,JB,JC
      PUNCH 4,KAK,KBK,KCK
      PUNCH 4,LA,LB,LC,LD
```

If we assume that JA, JB, JC, KAK, KBK, KCK, LA, LB, LC, and LD contain the values 1, 2, 3, 11, 12, 13, 21, 22, 23, and 24, respectively, then the above four examples of PUNCH statements, with their referenced FORMAT statements, will cause punching, for ultimate printing of seven lines, as follows:

```
ABC
ABC    1    2
ABC    3
   11   12XYZ
   13
   21   22XYZ
   23   24XYZ
```

If the I/O list in the PUNCH statement contains list variables (section 9.1), the values of these variables will be affected by execution of the PUNCH statement. For instance, the value of NIMBLE will be 9 after execution of the following

EXAMPLE:

```
PUNCH 2,(JACKBE(NIMBLE),NIMBLE = 1,10,2)
```

9.23 READ Statement

The forms are as follows:

```
READ n
READ n,e
READ ne
```

where n is the statement number of a FORMAT statement (section 9.21) and e is an I/O list (section 9.1).

EXAMPLES:

```
READ2
READ 23, N, (ALPHA(J),J = 1,N)
```

The effect of the READ statement is to read paper tape for data required in further execution of the program. The information on the tape is read and converted according to the specifications in the FORMAT statement (referenced in the READ statement) and the resulting value is stored in the variables named in the I/O list of the READ statement. All the items of the I/O list must be variables, not constants. The items of the I/O list are matched with the specifications of the specification list in the FORMAT statement except for the H and X specifications, which stand by themselves. The items of the I/O list and the corresponding specifications must match as to being integer or floating point.

The data must begin with a carriage return character. Upon execution of the READ statement, the input tape is searched forward for a carriage return and conversion begins with the character following the carriage return. Whenever a slash is encountered as a separator between specifications, the tape is moved forward to the next carriage return and conversion resumes with the next character after the carriage return. A slash found on the input tape while converting a field according to E, F, I, or O conversion means that the field is finished even if the specified number of characters have not been found. Whenever an H or X specification is encountered in the specification list, characters are skipped or read into the space reserved, but consideration of the next variable in the I/O list is delayed until the next E, F, I, O, or A specification is reached. If the end of the specification list in the FORMAT statement is reached before the end of the I/O list in the READ statement, part or all of the specification list is considered repeatedly (section 9.20). If the end of the I/O list is reached before the end of the specification list, reading will proceed in accordance with slashes, H specifications, and X specifications until the next E, F, I, O, or A specification or the end of the specification list is reached. If there is no I/O list in the READ statement, the specification list in the FORMAT statement must begin with H or X specifications if any reading is to occur.

EXAMPLES:

```
6    FORMAT (6HA;B;C;,F9.6,E12.6,I3)
8    FORMAT (I4,(E8.6,E8.6,A2))
     READ  6
     READ  6,ALPHA,BETA,JACK
     READ  8,IDENT,(BMBRS(J),FGTRS(J),KSYMB(J),J=1,M)
```

Execution of the first READ statement among the above examples would cause the six computer words occupied by three space codes and the three letters A, B, and C, and associated case codes, to be replaced by six characters from the input tape. Execution of the second READ statement would also cause input of six characters to those reserved computer words. But then input and conversion of three numbers to become the new values of ALPHA, BETA, and JACK, would proceed according to the last three specifications in FORMAT statement 6. Notice that the conversion specifications are for two floating point numbers and one integer, and that those

Specification	Input Data Field	Value
F6.4	-13906	-1.3906
F6.4	279.37	279.37
F6.4	3G59E1	3.059
F6.4	5.R7-1	.507
E10.3	2468357988	2468357.9
E10.3	1234.1234E	1234.1234
E10.3	-27.27E 2	-2727
E10.3	39XYZE- 01	3.9

Figure 9.

are the respective types of the variables named in the READ statement. FORMAT statement 8, if referenced by a READ statement, calls for conversion and input of values of an integer variable and as many lines as necessary of two floating point variables and one fixed point variable. The last READ statement of the examples above satisfies this requirement. It calls for input to the integer variable IDENT and to M elements of the floating point arrays BMBRS and FGTRS and the integer array KSYMB.

Notice that execution of a READ statement can change the value of a variable which is not an element of the I/O list. In the last example above, J will be left with the value which M has, even though there is no input to J.

CHAPTER 10

A PROGRAM IN 160 FORTRAN

10.0 Introduction

Basically, a useful computation requires that data be read in, computations be performed on the data, and results be punched out for listing on a Flexowriter. Complexities begin to creep in immediately in that alternative computations must be provided and intermediate results must be examined in order to choose among the alternatives. The IF statements and the conditional GO TO statements have been provided for the task of choosing (sections 6.4, 6.5, 6.6, 6.7, 6.8, 6.9). Other seeming complexities are actually simplifications. DO loops and SUBROUTINES have been provided so that some tasks to be done repeatedly may be programmed only once (sections 6.12, 6.13, 6.14, and Chapter 8).

10.1 Structure of a Program

The following rules apply to all 160 FORTRAN programs:

1. All FORMAT statements must appear first in the program.

2. All FORMAT statements must have unique statement numbers.

3. All DIMENSION statements must appear next in the program.

4. All subroutines must appear next in the program.

    a. Each subroutine must begin with a SUBROUTINE statement.

    b. If there is a NONLOCAL statement in the subroutine, it must appear next.

    c. There must not be any READ or PUNCH statements inside a subroutine.

    d. There must not be any FORMAT or DIMENSION statements inside a subroutine.

    e. There must not be any SUBROUTINE or END statements inside a subroutine.

    f. There must be an END statement at the end of each subroutine.

5. The next statement is the first one to be executed in the object program.

6. There must be a PAUSE or STOP statement as the last one to be executed.

7. There must be an END statement as the next-to-last statement in the program.

8. There must be an END statement as the last statement of the program.

10.2 Comments in 160 FORTRAN Programs

It may be desirable to insert comments within a FORTRAN program to explain the processing at any point. A means has been provided to do this so that the comments will be listed along with the rest of the source code, but will be ignored in the compilation process. If a C is written in column 1, the type column, of the coding sheet (section 2.3) the entire line will be ignored by the compiler. The C designation does not carry over from line to line; if there are several consecutive comment lines, each must have a C in column one.

10.3 A Sample 160 FORTRAN Program

This section contains a sample program in 160 FORTRAN source language as it would appear in a Flexowriter listing of the input tape. Notice that there is a prime (') wherever an asterisk (*) was written on the coding sheet. There are two statements which are continued onto a second line. They are the statements following statements 7 and 8, respectively. It is important that the statement after statement 8 breaks between N and the plus sign rather than between the slash and the N. As a convenience to the Flexowriter operator, in case an error occurs in transcribing, a slash followed by a carriage return is a signal to the compiler to ignore the whole line. Therefore, if a line ends in slash, this, combined with the following carriage return code, will cause the line to be left out. The letters in the following example are all upper case; however, it is immaterial to the compiler if the letters are upper or lower case. The sample program follows:

```
100  FORMAT (I4/(F5.1,F11.8))
101  FORMAT (F12.3/2F12.3/2F12.3/F12.3)
102  FORMAT (3F12.3)
     DIMENSION X(10),Z(10)
     READ100, N, (X(I),Z(I), I=1,N)
     READ101, TN2, CHI1,CHI2,A,B,ZO
     XBAR=0
     ZBAR=0
```

```
      DO 1 I=1,N
      XBAR=XBAR+X(I)/N
1     ZBAR=ZBAR+Z(I)/N
C     ESTIMATE REGRESSION COEFFICIENTS AND VARIANCE
      BETA=0
      ALPHA=0
      TEMP=0
      DO 2 I=1,N
      BETA=(X(I)-XBAR)'(Z(I)-ZBAR)+BETA
2     TEMP=TEMP+(Z(I)-ZBAR)''2
      BETA=BETA/TEMP
      ALPHA=XBAR-BETA'ZBAR
      VAR=0
      DO 3 I=1,N
3     VAR=VAR+(X(I)-ALPHA-BETA'Z(I))''2/N
      PUNCH102, ALPHA, BETA, VAR
C     CONFIDENCE INTERVAL FOR ALPHA
      TEMP=0
      TEMP1=0
      DO 4 I=1,N
      TEMP=TEMP+Z(I)''2
4     TEMP1=TEMP1+(X(I)-ALPHA-BETA'Z(I))''2
      TEMP1=TEMP1'TEMP
      TEMP=0
      DO 5 I=1,N
5     TEMP=TEMP+N'(N-2)'(Z(I)-ZBAR)''2
      TEMP=SQRTF(TEMP/TEMP1)
      TEMP2=(TN2+ALPHA'TEMP)/TEMP
      TEMP1=(-TN2+ALPHA'TEMP)/TEMP
      PUNCH102, TEMP1, TEMP2
C     CONFIDENCE INTERVAL FOR BETA
      TEMP1=0
      TEMP2=0
      DO 6 I=1,N
      TEMP1=TEMP1+(X(I)-ALPHA-BETA'Z(I))''2
6     TEMP2=TEMP2+(N-2)'(Z(I)-ZBAR)''2
      TEMP=SQRTF(TEMP1/TEMP2)
      TEMP1=(-TN2+BETA'TEMP)/TEMP
      TEMP2=(TN2+BETA'TEMP)/TEMP
      PUNCH102, TEMP1,TEMP2
C     CONFIDENCE INTERVAL FOR VAR
      TEMP1=(N'VAR)/CHI2
      TEMP2=(N'VAR)/CHI1
      PUNCH102,TEMP1,TEMP2
      TEMP=0
C     COMPUTE F FOR ALPHA=A AND BETA=B
      DO 7 I=1,N
```

```
7     TEMP=TEMP+(Z(I)''2)'(BETA-B)
      TEMP1=(N'(ALPHA-A)+2'N'ZBAR'(ALPHA-A)'
     1(BETA-B)+TEMP)/(N'VAR)
      PUNCH102, TEMP1
C     PREDICTION INTERVAL FOR X CORRESPONDING TO Z
      TEMP=0
      DO 8 I=1,N
8     TEMP=TEMP+(Z(I)-ZBAR)''2
      TEMP=TN2'VAR'SQRTF((N/(N-2))'((N+1)/N
     1+(ZO-ZBAR)''2/TEMP))
      TEMP1=ALPHA+BETA'ZO-TEMP
      TEMP2=ALPHA+BETA'ZO+TEMP
      PUNCH102, TEMP1, TEMP2
      STOP 7707
      END
      END
```

CHAPTER 11

ENVIRONMENT OF 160 FORTRAN

11.0 Introduction

The 160 FORTRAN system consists of the FORTRAN programming language as described in the previous chapters of this manual, the Control Data 160 (or 160-A) Computer, the program tapes to compile the FORTRAN source program, and the program tape to interpret the FORTRAN object program. The following sections describe aspects of the system that may help the user to realize the maximum benefits from the system.

11.1 Source Program

The original program is written in the 160 FORTRAN language on standard forms (Figure 1). From this manuscript a punched paper tape and a listing are prepared on a Flexowriter. The listing is for reference. The punched paper tape is for input to the compiler. The method of accomplishing compilation is described in the operating instructions at the end of this manual.

11.2 Results of Compilation

The main output of the compiler is a binary punched tape of the object code which is used in executing the program. It is also possible to obtain two auxiliary tapes for listing on a Flexowriter. The first is the IDLIST Tape, which contains lists of the variables, constants, and other entities used in the program. The second auxiliary tape is an interpreted listing of the object code.

11.3 The IDLIST Tape

The first list is the list of simple variables. This variable list gives the first of two or four locations associated with each variable in the program and the name of the variable. The locations are given in octal as are all locations on the list tape and the object code tape. The first location for a variable contains its type: 2 for integer, 5 for floating point, 3 for integer formal parameter of a subroutine, 7 for floating point formal parameter. For formal parameters or local variables there is a number, from 1 to 7, in the list, after the variable name, that tells to which subroutine it belongs. If the variable type is 2 there is one word, immediately following the type, which will contain the value of the integer variable. Variable type 5 means the three following words contain the floating point value. Variable type 3 or 7 means the following word contains the location of the actual parameter.

There will always be a variable named IF listed. This is not a legal variable name for the programmer to use. It is included by the compiler for use as an index in input and output of arrays.

If there are any arrays, they are listed next under the heading, "Array Storage". A location is given along with the name and dimensions of the array. The given location is the fourth of several allocated to the array. The first three words contain the dimensions of the array. The fourth word (the one given in the listing) contains the type: 1 for integer, 4 for floating point. Following this there is one word for each element of an integer array and three words for each element of a floating point array.

Next there are lists of the integer and floating point constants, type 2 or 5, respectively, with two or four locations allocated to each. The lists are in two columns giving location and the value of the constant.

Next comes a list of statement labels. The list is headed, "Labels". It consists of four columns giving a reference location, the object code location, the programmer's statement number, and the rank. The reference, or "Varlist" location, is needed only because the compiler is one-pass. Rank of zero means the statement number is in the main program. Ranks from 1 to 7 tell which subroutine. The compiler always supplies statement number zero to the first executable statement in the program.

Following the statement labels there may be a list called "EAPACK". This is a list of all the different ways of referring to array elements. The listed location is the first of several assigned to the corresponding subscripted variable. This first location contains codes indicating the number of subscripts and the structure of each. Following this there are the locations of the array and the locations of the variable subscript elements and the values of the constant subscript elements.

Next are lists of the library functions and the subroutines. PUNCH, READ, and ** (with floating point operands) make use of the library functions PU, RE, and EXPF, respectively. PU and RE are library functions which cannot be called explicitly by the programmer. The listings give the reference location, the object code location and the name of the function or subroutine.

11.4 The Object Code

The object code produced by the 160 FORTRAN compiler is <u>not</u> in 160 machine language. It is a special language designed to be read and interpreted by a program (part of the 160 FORTRAN system called the interpreter) which is in the computer at the same time as the object code. The Control Data 160 Computer contains 4096 words, numbered (in octal) from 0000 through 7777. All locations are given in octal. The interpreter occupies locations 0000 through 3200 if there are no FORMAT statements. Otherwise it continues to 3500. The object code, starting with the subroutines and FORMAT statements, follows the interpreter. The library functions follow the object code. The variable list is built down from 7777. In order for the program to operate, it must be small enough that the object code or library functions do not overlap the variable list. The approximate lengths of the library functions are, in octal, 355 for READ, 520 for PUNCH, and from 100 to 200 for each of the others.

An interpretive listing of the object code may be obtained as explained in the operating instructions. This consists of a symbolic listing of the macro-instructions of the object code, together with their octal locations. The macro-instructions are the sentences of the language read by the interpreter. The listing uses mnemonic symbols which, it is hoped, will convey some meaning.

11.5 Samples of the IDLIST and Object Code List

This section contains excerpts, printed on a Flexowriter, of the listings described in sections 11.3 and 11.4:

```
VARIABLE LIST

7470  L
7474  K
7502  K, 1
7504  JD, 1
7506  N4, 1
```

```
7560  LPLP
7572  LP
7574  JOB, 1
7576  N, 1
7600  M, 1
7773  IF


ARRAY STORAGE

7642  1(10)


INTEGER CONSTANTS

7547  1789
7551     9
7556   456
7560     8
7565    23
7567     7
7574     6


LABELS
VARLIST LCN   OBJCODE LCN    NAME    RANK

7450          4620             25    0
7454          4541             24    0
7456          4446             23    0
7460          4353             22    0
7464          4260             21    0
7500          4203              8    1
7512          4130              6    1

7742          7717              3    0
7761          7744              2    0
7767          7763              1    0
7771          4220              0    0

EAPACK

7535 I(J)
7544 I(+9)
7553 I(+8)
```

```
LIBRARY FUNCTION NAMES
VARLIST LCN   OBJCODE LCN    NAME

7476          4710           PU


SUBROUTINE NAMES
VARLIST LCN   OBJCODE LCN    NAME

7602          3700           ARITH
```

```
OBJECT CODE


3700  SUBROUTINE FOLLOWS, NUMBER OF PARAMETERS:  3
      M, 1
      N, 1
      JOB, 1
3704  ARITH
      MODE: INTEGER
3710  ADD LP
      ADD 1
      STO LP
      END
3714  ARITH
      MODE: INTEGER
3720  ADD LP
      SUB 1000
      END
3723  IF
      32      1
      31      1
      31      1
```

```
4172  INIT
      1
      K, 1
4175  CALL ARITH
      3
      M1, 1
      J4, 1
      JD, 1
4203  ARITH
      MODE: FLTNG
4207  ADD Z
      STO J4, 1
      END
4212  INCR
      K, 1
      N4, 1
      1
      4175
4217  RETURN



4663  CALL ARITH
      3
      2
      4
      4
4671  IO
      PUNCH
      6         0
      PU
4675  IOC
      5
      2
      4
      Z
      LPLP
      LP
4704  STOP
      7776
4706  STOP
      0000
4710  PU 4710-5531
      VARIABLE LIST 7450-7776
```

SUMMARY OF STATEMENTS IN 160 FORTRAN

1. Control

   a. Unconditional GO TO
   b. ASSIGN
   c. Assigned GO TO
   d. Computed GO TO
   e. IF
   f. IF ACCUMULATOR OVERFLOW
   g. IF QUOTIENT OVERFLOW
   h. IF DIVIDE CHECK
   i. PAUSE
   j. STOP
   k. DO
   l. CONTINUE

2. Subroutine

   a. SUBROUTINE
   b. CALL
   c. NONLOCAL
   d. RETURN
   e. END

3. Declaration

   a. DIMENSION

4. Input/Output

   a. READ
   b. PUNCH
   c. INPUT
   d. OUTPUT
   e. FORMAT

FORTRAN SPECIAL CHARACTERS

1.	**	Power, Shift
2.	*	Multiply, Logical Product
3.	/	Divide, Exclusive OR
4.	+	Add, Inclusive OR
5.	-	Subtract, Complement
6.	(	Left Parenthesis
7.	)	Right Parenthesis
8.	,	Separator
9.	.	Floating Point Indicator
10.	=	Replacement symbol
11.	;	Space Designator in an H Specification

FLEXOWRITER CHARACTERS

Character	Code	Case Shift*
A	30	UC, LC
B	23	UC, LC
C	16	UC, LC
D	22	UC, LC
E	20	UC, LC
F	26	UC, LC
G	13	UC, LC
H	05	UC, LC
I	14	UC, LC
J	32	UC, LC
K	36	UC, LC
L	11	UC, LC
M	07	UC, LC
N	06	UC, LC
O	03	UC, LC
P	15	UC, LC
Q	35	UC, LC
R	12	UC, LC
S	24	UC, LC
T	01	UC, LC
U	34	UC, LC
V	17	UC, LC
W	31	UC, LC
X	27	UC, LC
Y	30	UC, LC
Z	31	UC, LC
0	56	UC, LC
1	74	UC, LC
2	70	UC, LC
3	64	UC, LC
4	62	UC, LC
5	66	UC, LC
6	72	UC, LC
7	60	UC, LC
8	33	UC, LC
9	37	UC, LC

* UC = Upper Case, LC = Lower Case

Character	Code	Case Shift
blank (space bar)	04	UC, LC
' (apostrophe)	44	UC
/	44	LC
-	52	UC, LC
(	54	UC
)	54	LC
, (comma)	46	LC
+	46	UC
.	42	LC
=	42	UC
;	50	LC
:	50	UC
Carriage return	45	UC, LC
Tab bar	51	UC, LC
Color Shift	02	UC, LC
Back space	61	UC, LC

Upper case (UC)	47
Lower case (LC)	57
Skip for tape feed	Blank tape

GLOSSARY

array:	A collection of variables. Usually the members of the collection are related in some way. One member of the collection is known as an array element. In FORTRAN there may be one, two, and three dimensional arrays. The array elements are identified by one, two, or three subscripts, respectively, and are also called subscripted variables.
binary:	Characterized by two, as 1) a binary operator, such as +, which operates on two quantities, or 2) the binary number system which has just two digit symbols, 0 and 1. The binary system uses positional notation in which each position represents a power of two instead of a power of ten.
bioctal:	Characterized by two octal digits. Sometimes used to designate six level paper tape punched with internal computer code. In this manual such tape is called binary.
bit:	A binary digit, which may be 0 or 1.
Boolean:	Originally the symbolic logic developed by the mathematician George Boole. In 160 FORTRAN, the operations of shifting, logical multiplication (AND), logical addition (inclusive OR), exclusive OR, and complementation, carried out in parallel on the twelve bits of integer operands.
constant:	A positive numerical quantity or zero, not subject to change. In FORTRAN, constants are referred to by their values.
conversion specification:	A stated relationship between internal and external representations of numbers. Used in carrying out READ and PUNCH operations.
fixed point:	The ordinary notation for numbers, in which the decimal point or binary point is explicitly stated and there is no scale factor. In FORTRAN fixed point numbers are all integers.
Flexowriter:	A brand of electric typewriter with paper tape punching and reading mechanisms attached.

floating point:	A representation for numbers in which the binary point or decimal point is always in a standard position and a scale factor is included which indicates a power of 2 or 10 to be used as a multiplier.
frame:	A line of punched holes across a paper tape.
Hollerith:	Originally a coding scheme for representing letters, digits, and special characters on punched cards, named for the designer. Now used to mean almost any scheme, associated with computers, for representing letters, digits, and special characters.
integer:	A whole number, without any fractional part. In FORTRAN, used to refer to fixed point numbers, which are required to be integers.
I/O list:	A list of variables, arrays, and constants for input or output by an INPUT, OUTPUT, READ, or PUNCH statement.
list:	I/O list or specification list. In this manual, context will always clearly indicate which. In other works on FORTRAN, list often means only I/O list.
one-pass:	Characteristic of compilers or assembly programs which only look at the source code once.
operand:	In arithmetic or other operations, the quantities on which the operations are performed.
operator:	That which indicates the arithmetic or other operation to be performed.
recursive:	Done again. A recursive subroutine is one which calls itself or which calls a chain of subroutines which ultimately call the original subroutine. To be useful the subsequent call must somehow differ from the previous call so that the subroutine can eventually complete its work. A recursive definition is a definition by formula in which the defined term can be used as a term in the formula, thereby extending the definition.
specification list:	A list of conversion specifications occurring in a FORMAT statement.

unary:	Characterized by one, as a unary operator such as negation or complementation which acts on only one operand.
variable:	A numeric quantity which is allowed to change value during execution of a program. A simple variable or a subscripted variable (array element). In this manual, an array is not called a variable.
word:	A memory cell of a computer. In the Control Data 160 Computer, memory is organized into 4096 words of 12 binary digits (bits) each.

OPERATING PROCEDURES FOR 160 FORTRAN

I. Operating equipment.

A. Control Data 160 with paper tape punch and reader.

B. Flexowriter.

II. Listings and paper tapes invloved in a typical run.

A. The FORTRAN program tapes -- furnished by Control Data.

1. Compiler I.
2. Compiler II.
3. Compiler III (the interpreter).

B. Tapes and listings furnished/generated by user for a given problem.

1. Source program -- Flexowriter tape with FORTRAN statements in standard format.
2. Listing of IDLIST (optional).
3. Compiler output -- a binary punched paper tape (odd parity) consisting of:
   a. Format specifications, if any, corresponding to format statements in the source code.
   b. The object code.
   c. Library subroutines, if any. (Object code format.)
   d. Varlist.
4. Interpretive listing of object code (optional).
5. Input-output tapes (during execution). If the compiled program involves input-output, this will be via paper tape reader and punch. These input-output tapes are of two types:
   a. Binary tapes, internal format.
   b. Flexowriter tapes, external format.

III. Detailed operating procedures.

We shall assume the source code has been generated on a Flexowriter and proceed to do a compilation and execution.

A. Compilation.

1. Machine load Compiler Tape I at P=0000. Check sum: 0000
2. Position the source code tape in the reader, turn on punch, clear and run (from P=0000).
3. If no source code errors have been detected during compilation, a stop will occur at P=5431 and 0000 will be displayed in the A-register. DO NOT CLEAR. At this point, a binary tape has been output which consists of Format specifications (if any) and the bulk of the object code; however, some "end" coding still remains in memory. Do not remove the punched tape from the punch at this point.

   Error Stops:

   When an error stop occurs (at P=5252 or P=5427), the A-register will contain the type of error. Run from here and another stop will occur (P=5255.) Now the A-register contains the value of the last encountered statement number in the source coding. (This second stop does not occur for error types 20, 27 and 30.) Memory cell #40 will contain the number of statements between the statement given in the A-register and the statement where the error stop occurred.

   Pressing run after an error stop will start compilation of the next FORTRAN statement. Either another error stop will occur or the final program stop at P=5431. At this point the A-register contains the number of errors detected during compilation. Thus for each error detected during compilation it is possible to determine the type of error and the source code statement in which the error occurs.

   If error stops occur, the program cannot be executed, but it may be helpful to continue with steps 4 and 5 for diagnostic purposes. If there is a format statement at some position other than first in the program, an error stop will occur at P=6262; no restart is possible from this start.

4. Position Compiler Tape II in the reader and run. (P=5431). The remainder of the object code is punched out, including the required library functions and the variable list.

   Stop at P = 0240 DO NOT CLEAR!

   Remove the binary tape from punch. (This will be used during execution and step 5 following.) The operator can proceed directly to execution at this point or implement step 5.

Error Stops:
P=0162, parity error stop.

Note: If a parity error stop occurs at this time, it indicates reader/machine errors. (All the library functions and final service routines have been verified.) To reload the record in which a parity error is indicated, position the tape in the reader on the blank frames preceding the section in question, clear, set P=0134 and run.

In addition, Compiler error stops ERR20, ERR27, and ERR30 can occur during this step.

5. Optional service routines.
   a. For a listing of IDLIST, press Run (P=0240). A Flexowriter tape is generated and a stop occurs at P=2343. DO NOT CLEAR.
   b. For an interpretive listing of the object code position the binary object code tape in the reader and run (from P=2343). A Flexowriter tape is then punched out and a final stop occurs at P=4374.

   Error Stops:

P=2521	Name of macro not in table
P=4001	Parity error. (recompile. Usually indicates punch trouble.)
P=4055	Search failure. (Varlist location not in IDLIST.)

Note: These Flexowriter tapes are designed to be listed on a Flexowriter with the standard OSAP tab settings (7, 10, 14, 17, 22, 26, 30, 40). However, it is possible to modify the tab settings to one tab per field of information. Also the select codes can be altered to permit listing on an on-line typewriter.

This is done preceding step 5.a. by inserting an appropriate number into the A-register as follows (P=0240 at this point):

A=1 Type IDLIST
A=2 Type object code
A=3 Change tab settings

After entering the appropriate parameter in the A-register, press run (P=0240). A stop occurs at P=0240, A=0000. Another change can be made in the dump routine by putting another parameter in the A-register, or step 5.a. can be implemented (with A = 0000).

B. Execution

1. Machine load Compiler Tape III (the interpreter) at P = 0000. Check sum = 0000.
2. Clear, position the binary object tape in the reader and run (from P = 0000). Stop at P = 0120. DO NOT CLEAR!

   Error Stop:

P = 0052	Parity error stop. Usually indicates punch trouble.

   Note: If a punch failure has caused three or more blank frames to be inserted in a binary object tape the program will hang up when the blank frames are encountered.

3. At this point the FORTRAN object program is in memory and ready to be executed. Turn on the punch, position input tapes in the reader as required, and run (from P = 0120).

   Error Stop:

P = 0445	Invalid operation code. (Compiler Error.)

PROGRAM ERRORS DETECTED DURING COMPILATION

Contents of A-register	Type of Error
1	No variable following operation symbol
2	No (after function
3	Two adjacent variables
4	Two adjacent operation symbols
5	Unequal (and)
6	Two ** **
7	Initial **, *, or /
10	Compiler error
11	Unknown character
12	Array name used as variable
13	Wrong IF format
14	Format error in GO TO
15	Format error in NON LOCAL
16	Not implemented in 160 FORTRAN
17	Problem too large
20	Undefined label
21	Format error in I/O list
22	Format error in I/O list
23	Numeric conversion
24	Too many characters in identifier
25	Identifier of wrong form
26	Format error in DIMENSION
27	Object code too large
30	Library routine not on tape
31	Error in Format Statement

Pub. No. 60050300

Litho in U.S.A.