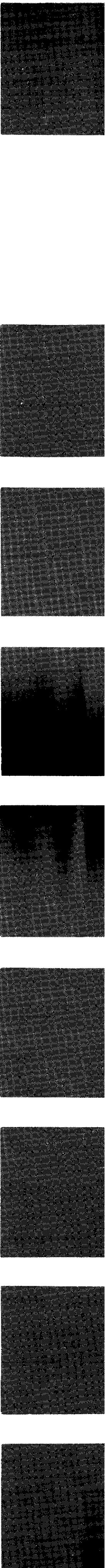
File No. 1130-25
Form C26-5933-2



Systems Reference Library

IBM 1130 FORTRAN Language

This publication presents the IBM 1130 FORTRAN language and programming rules. The FORTRAN language closely resembles the language of mathematics and is designed to be used for mathematically-oriented computer applications.

PREFACE

FORTRAN (FORmula TRANslation) is a coding system with a language that closely resembles the language of mathematics. It is a system designed primarily for scientific and engineering computations. Since this system is essentially problem oriented rather than machine oriented, it provides scientists and engineers with a method of communication that is more familiar, easier to learn, and easier to use than actual computer language.

This publication presents the IBM 1130 Fortran language and programming rules; it should not be used as a Fortran primer. For general information about Fortran, refer to the IBM FORTRAN General Information Manual (Form F28-8074).

Machine Configuration and Feature Requirements

The minimum machine configuration and feature requirements needed to compile programs with the IBM 1130 Card/Paper-Tape Fortran Programming System are:

- IBM 1130-1A Central Processing Unit.
- IBM 1442 Card Read Punch, or IBM 1054 Paper-Tape Reader and IBM 1055 Paper-Tape Punch.

This publication supersedes and makes obsolete prior publication C26-5933-1 and TNL N26-0109.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
A form has been provided at the back of this publication for readers' comments.
If the form has been detached, comments may be directed to :
IBM, Programming Publications Dept. 234, San Jose, Calif. 95114

© 1964, 1965 by International Business Machines Corporation

CONTENTS

IBM 1130 FORTRAN PROGRAMMING SYSTEM 1
Constants, Variables, and Subscripts 1
Arithmetic Statements 6
Control Statements 6
Input/Output Statements 9
Specification Statements 17
Subprogram Statements 20

MONITOR FORTRAN 28

APPENDIX A. TABLE OF SOURCE PROGRAM CARD CHARACTER CODES 30

APPENDIX B. NONSTANDARD ITEMS 31

APPENDIX C. SOURCE PROGRAM STATEMENTS AND SEQUENCING 32

APPENDIX D. IBM SYSTEM/360 RESERVED WORDS 33

INDEX 34

The IBM 1130 Fortran Programming System consists of two parts: the language and the compiler. The language is a set of statements, composed of expressions and operators, which are used in writing the source program. The 1130 Fortran Compiler, provided by IBM, is a program which translates the source program statements into a form suitable for execution on the IBM 1130 System. The translated statements are known as the object program. The compiler detects certain errors in the source program and writes appropriate messages on the typewriter. At the user's option, the compiler also produces a listing of the symbol table.

Coding Form

The statements of a Fortran source program are normally written on a standard Fortran coding sheet (Form No. X28-7327). Fortran statements are written one to a line in columns 7-72. If a statement is too long for one line, it may be continued on a maximum of five successive lines by placing any character other than a blank or a zero in column 6 of each continuation line. For the first line of a statement, column 6 must be blank or zero.

Columns 1-5 of the first line of a statement may contain a statement number. This statement number consists of 1-5 digits of any value; leading zeros are ignored. Statement numbers may appear anywhere in the statement number field but must not contain any special characters. The statement numbers may be assigned in any order; the sequence of operations is always dependent upon the order of the statements in the program, not on the value of the statement numbers.

NOTE: Superfluous statement numbers may decrease efficiency during compilation and should, therefore, be avoided.

Columns 73-80 are not used by the Fortran compiler and may, therefore, be used for program identification, sequencing, or any other purpose.

Comments to explain the program may be written in columns 2-72 of a line if the character C is placed in column 1. Comments may appear anywhere except before a continuation line or after an END statement. The comments are not processed by the Fortran compiler. Blank cards in a source deck are ignored by the Fortran compiler.

Statements

The Fortran statements are the instructions used in the Fortran language. There are five categories of Fortran statements:

- Arithmetic Statements, which are used to define calculations to be performed.
- Control Statements, which are used to govern the sequence of execution of the program statements.
- Input/Output Statements, which are used to transmit information between the computer and input or output units.
- Specification Statements, which are used to provide information about the data that the object program is to process.
- Subprogram Statements, which are used to define and use subprograms.

Blanks may be used freely to improve the readability of a Fortran program listing. For example, the following statements have a valid format:

```
GObTO(1,2,3,4),I
GObTObb(1,2,3,4),bbI
```

where b represents a blank.

CONSTANTS, VARIABLES, AND SUBSCRIPTS

Fortran provides a means of expressing numeric constants, variable quantities, and subscripted variables. The rules for expressing these quantities are quite similar to the rules of ordinary mathematical notation.

Arithmetic calculations specified in an object program are performed with binary numbers; since decimal fractions cannot be represented exactly, exact decimal results of arithmetic calculations should not be expected.

Constants

A constant is any number which is used in a computation without change from one execution of the program

to the next. A constant appears in numeric form in the source statement. For example, in the statement

```
J = 3 + K
```

the 3 is a constant, since it appears in actual numeric form. Two types of constants may be written in Fortran: integer and real.

Integer Constants

An integer constant is a number written without a decimal point. The magnitude of an integer constant must not be greater than 32767 ($2^{15}-1$).

Commas are not permitted within any Fortran constants. A preceding plus sign is optional for positive numbers. Any unsigned constant is assumed to be positive.

The following examples are valid integer constants:

```
  0
 91
-173
+327
```

The following are not valid integer constants:

3.2	(contains a decimal point)
27.	(contains a decimal point)
31459036	(exceeds the magnitude permitted by the compiler)
5,496	(contains a comma)

Real Constants

A real constant is a number written with a decimal point and consisting of 1-7 or 1-10 significant decimal digits (the precision to be selected at compile time). The magnitude of a real constant must not be greater than 2^{127} or less than 2^{-129} (approximately 10^{38} and 10^{-39}) or zero.

A real constant may be followed by a decimal exponent written as the letter E followed by a one- or two-digit integer constant (signed or unsigned) indicating the power of 10.

The following examples are valid real constants:

105.	
3.14159	
5.E3	(5.0×10^3)
5.0E3	(5.0×10^3)
-5.0E03	(-5.0×10^3)
5.0E-3	(5.0×10^{-3})
5.0E1	(5.0×10)

The following are not valid real constants:

325	(no decimal point; however, this is a valid integer constant)
5.0E	(no exponent)
5.0E003	(exponent contains three digits)

Variables

A Fortran variable is a symbolic representation of a quantity that may assume different values. The value of a variable may change either for different executions of a program or at different stages within the program. For example, in the statement:

```
A = 5.0 + B
```

both A and B are variables. The value of B is determined by some previous statement and may change from time to time. The value of A varies whenever this computation is performed with a new value for B.

Variable Names

A variable name consists of 1-5 alphameric characters, the first of which must be alphabetic.

Examples:

```
M
DEV86
I2
```

Variable Types

The type of variable corresponds to the type of data the variable represents (i.e., integer or real). Variables can be specified in two ways: implicitly or explicitly.

Implicit Specification. Implicit specification of a variable is made as follows:

1. If the first character of the variable name is I, J, K, L, M, or N, the variable is an integer variable.
2. If the first character of the variable name is not I, J, K, L, M, or N, the variable is a real variable.

Explicit Specification. Explicit specification of a variable type is made by using the Type statement (see Type Statements). The explicit specification

overrides the implicit specification. For example, if a variable name is ITEM and a Type specification statement indicates that this variable is real, the variable is handled as a real variable, even though its initial letter is I.

Naming Variables

The rules for naming variables allow for extensive selectivity. In general, it is easier to follow the flow of a program if meaningful symbols are used wherever possible. For example, to compute distance it would be possible to use the statement:

```
X = Y*Z   (Asterisk denotes multiplication)
```

but it would be more meaningful to write:

```
D = R*T
```

or:

```
DIST = RATE * TIME
```

Similarly, if the computation were to be performed using integers, it would be possible to write:

```
I = J*K
```

or:

```
ID = IR * IT
```

or:

```
IDIST = IRATE * ITIME
```

In other words, variables can often be written in a meaningful manner by using an initial character to indicate whether the variable is integer or real and by using succeeding characters as an aid to the user's memory.

Arrays and Subscripts

An array is a group of quantities arranged in a particular order. It is often advantageous to be able to refer to this entire group by one name, and to refer to each individual quantity in this group in terms of its position in the group. For example, assume that the following is an array named NEXT:

15
12
18
42
19

Suppose it is desired to refer to the second quantity in the group; in ordinary mathematical notation, this would be $NEXT_2$. In Fortran this would be NEXT(2). The quantity 2 is called a subscript. Thus, NEXT(2) has the value 12 and NEXT(4) has the value 42.

Similarly, an ordinary mathematical notation might use $NEXT_n$ to represent any element of the array NEXT. In Fortran, this is written as NEXT (I) where I equals 1, 2, 3, 4, or 5.

The array could be two-dimensional; for example, the array LIST:

	COLUMN1	COLUMN2	COLUMN3
ROW1	82	4	7
ROW2	12	13	14
ROW3	91	1	31
ROW4	24	16	10
ROW5	2	8	2

Suppose it is desired to refer to the number in row 2, column 3; this would be:

LIST(2, 3)

where 2 and 3 are the subscripts. Thus, LIST(2, 3) has the value 14 and LIST(4, 1) has the value 24.

Ordinary mathematical notations might use $LIST_{i,j}$ to represent any element of the array LIST. In Fortran, this is written as LIST(I, J) where I equals 1, 2, 3, 4, or 5 and J equals 1, 2, or 3.

Fortran allows up to three subscripts (i.e., three-dimensional arrays). For example, a three-dimensional array might be used to store statistical data on the urban and rural population of each state for a period of 10 decades.

The use of an array in the source program must be preceded by either a DIMENSION statement, a COMMON statement, or a Type statement in order to specify the size of the array. The first of these statements that refers to the array must specify its size (see Specification Statements).

Arrangement of Arrays in Storage

Arrays are stored in column order in descending storage addresses, with the value of the first of their subscripts increasing most rapidly and the value of the last increasing least rapidly. In other words, arrays are stored with element (1, 1, 1) in a higher core location than element (2, 3, 4). In scanning the array from element (1, 1, 1), the left indices are advanced

more rapidly than those on the right. A one-dimensional array, J(5), in address 0504 appears in storage as follows:

Address	Element
0500	J (5)
0501	J (4)
0502	J (3)
0503	J (2)
0504	J (1)

A two-dimensional array, K (5, 3), appears in storage in single-array form in ascending storage addresses in the following order reading from left to right:

K (5, 3) K (4, 3) K (3, 3) K (2, 3) K (1, 3) K (5, 2)
K (4, 2) K (3, 2) K (2, 2) K (1, 2) K (5, 1) K (4, 1)
K (3, 1) K (2, 1) K (1, 1)

If K (5, 3) is in core address 0200, K (1, 1) will be in core address 0214.

The following list is the order of a three-dimensional array, A(3, 3, 3):

A(3, 3, 3) A(2, 3, 3) A(1, 3, 3) A(3, 2, 3) A(2, 2, 3)
A(1, 2, 3) A(3, 1, 3) A(2, 1, 3) A(1, 1, 3) A(3, 3, 2)
A(2, 3, 2) A(1, 3, 2) A(3, 2, 2) A(2, 2, 2) A(1, 2, 2)
A(3, 1, 2) A(2, 1, 2) A(1, 1, 2) A(3, 3, 1) A(2, 3, 1)
A(1, 3, 1) A(3, 2, 1) A(2, 2, 1) A(1, 2, 1) A(3, 1, 1)
A(2, 1, 1) A(1, 1, 1)

Subscript Forms

Subscripts may take the following forms:

v
c
v+c
v-c
c*v
c*v+c'
c*v-c'

where: v represents an unsigned, nonsubscripted, integer variable; c and c' represent unsigned integer constants.

Examples:

The following are valid subscripts:

IMAX
19
JOB+2
NEXT-3
8*IQUAN
5*L+7
4*M-3

The following are not valid subscripts:

-I	(the variable may not be signed)
A+2	(A is not an integer variable unless defined as such by a Type statement)
I+2.	(2. is not an integer constant)
-2*J	(the constant must be unsigned)
I(3)	(a subscript may not be subscripted)
K*2	(for multiplication, the constant must precede the variable; thus, 2*K is correct)
2+JOB	(for addition, the variable must precede the constant; thus, JOB+2 is correct)

Subscripted Variables

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

Examples:

A(I)
K(3)
ALPHA (I, J+2)
BETA (5*J-2, K-2, L+3)

Expressions

Expressions appear on the right-hand side of arithmetic statements and in certain control statements. Expressions are used to specify a computation between constants and variables.

Arithmetic Expressions

The simplest arithmetic expression consists of a single constant, variable, or subscripted variable. If the quantity is an integer quantity, the expression is said to be in the integer mode. If the quantity is a real quantity, the expression is said to be in the real mode.

Examples:

EXPRESSION	TYPE OF DATA	MODE OF EXPRESSION
3	Integer Constant	Integer
I	Integer Variable	Integer
3.0	Real Constant	Real
A	Real Variable	Real
A(I)	Real Variable	Real

In the last example, note that the subscript (which is always an integer quantity) does not affect the mode of the expression. The mode of the expression is determined solely by the mode of the quantity itself.

An arithmetic expression is usually a combination of constants, subscripted or nonsubscripted variables, function names (see Subprogram Statements), and arithmetic operation symbols.

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

Examples:

```
A+3.0
B**2
C-D
E/F
A*(X**2)+B*X-C
```

Rules for Construction of Arithmetic Expressions

Rule 1. All constants, variables, and functions that form an arithmetic expression need not be of the same mode or type. It should be noted, however, that a mixed expression is computed in the real mode. This means that some inefficiencies exist in mixed mode computations since all integer values will be converted to real values.

Examples: The following are valid expressions:

Expression	Mode
F	Real
5* JOB+ITEM/(2*ITAX)	Integer
5. *AJOB+BITEM/(2. *TAX)	Real
J+1	Integer
A**I+B(J)+C(K)	Real
A**B	Real
I**J+K(L)	Integer
A+B(I)/ITEM	Mixed
DEV+I	Mixed
ITA**2.5	Mixed

Rule 2. Any expression may be enclosed in parentheses. The use of parentheses does not affect the mode of the expression. Thus, A, (A), and ((A)) are all valid real expressions.

Parentheses may also be used in arithmetic expressions, as in algebra, to specify the order in which the various arithmetic operations are to be performed. Within parentheses, or where parentheses are omitted, the order of operations is as follows:

1. Evaluation of Functions
2. Exponentiation
3. Multiplication and Division (left to right)
4. Addition and Subtraction (left to right)

For example, the expression:

A*B/(C+D)**I+D

is effectively evaluated in the following order:

1. AxB
2. $C+D$
3. $(C+D)^I$
4. $(AxB)/(C+D)^I$
5. $((AxB)/(C+D)^I)+D$

NOTE: Parentheses may not be used to imply multiplication; the asterisk arithmetic operator must always be used for this purpose. Therefore, the algebraic expression:

$(AxB)\ (-C^D)$

must be written as:

(A*B) * (-C**D)

Rule 3. No two operators may appear in sequence (e.g., A*-B is invalid).

Rule 4. No operation symbol may be assumed (e.g., 3A will not be taken as 3.*A).

Rule 5. The expression A**B**C is permitted and evaluated as A**(B**C).

ARITHMETIC STATEMENTS

The Arithmetic statement is similar to a mathematical equation.

General Form:

A = B

where:

A is any variable (subscripted or nonsubscripted), and B is an arithmetic expression.

In an Arithmetic statement, the equal sign means: is to be replaced by, rather than, is equal to. This distinction is important; for example, suppose the integer variable I has the value 3. Then, the statement:

I = I + 1

would give I the value 4. This technique enables the programmer to keep counts and perform other required operations in the solution of a problem.

Examples:

```
K = X + 2.5
ROOT = (-B+(B**2-4.*A*C)**.5)/(2. *A)
ANS (I) = A(J) + B(K)
```

In each of the above Arithmetic statements, the arithmetic expression to the right of the equal sign is evaluated, converted to the mode of the variable to the left of the equal sign (if there is a difference), and this converted value is stored in the storage location associated with the variable name to the left of the equal sign.

In the first example, K=X+2.5, assume that the current value of X is 232.18. Upon execution of this statement, 2.5 is added to 232.18, giving 234.68. This value is then truncated (because K is an integer variable) to 234, and this value replaces the value of K. If K were defined as a real variable by a Type statement, truncation would not occur and the value of K would be 234.68.

Examples:

A = I	Convert I to real value and store it in A.
A = B	Store the value of B in A.
A = 3.*B	Multiply 3 by B and store the result in A.
I = B	Truncate B to an integer and store it in I.

CONTROL STATEMENTS

The second class of Fortran statements is composed of control statements that enable the programmer to control the course of the program. Normally, statements are executed sequentially; that is, after one statement has been executed, the statement immediately following it is executed. However, it is often undesirable to proceed in this manner. The following statements may be used to alter the sequence of a program.

Unconditional GO TO Statement

This statement interrupts the sequential execution of statements, and specifies the number of the next statement to be performed.

General Form:

GO TO n

where:

n is a statement number.

Examples:

```
GO TO 25
GO TO 63468
```

The first example causes control to be transferred to the statement numbered 25; the second example causes control to be transferred to the statement numbered 63468.

Computed GO TO

This statement also indicates the statement that is to be executed next. However, the statement number that the program is transferred to can be altered during execution of the program.

General Form:

GO TO (n_1, n_2, . . . , n_m), i

where:

$n_1, n_2, \ldots, n_m$ are statement numbers and
i is an integer variable whose value is greater than or equal to 1 and less than or equal to the number of statement numbers within the parentheses.

This statement causes control to be transferred to statement $n_1, n_2, \ldots, n_m$, depending on whether the current value of i is 1, 2, . . . , or m, respectively.

NOTE: If $i>m$ or $i<1$, the results are unpredictable.

Example:

```
GO TO (10, 20, 30, 40), ITEM
```

In this example, if the value of ITEM is 3 at the time of execution, a transfer occurs to the statement whose number is third in the series (30). If the value of ITEM is 4, a transfer occurs to the statement whose number is fourth in the series (40), etc.

IF Statement

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

General Form:

IF (a) n_1, n_2, n_3

where:

a is an expression and n_1, n_2, and n_3 are statement numbers. The expression, a, must be enclosed in parentheses; the statement numbers must be separated from one another by commas.

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

Example:

```
        IF ((B+C)/(D**E)-F) 12, 72, 10
10      .
        .
12      .
        .
72      .
```

which means: if the result of the expression is less than zero, transfer to the statement numbered 12; if the result is zero, transfer to 72; otherwise, transfer to the statement numbered 10.

DO Statement

The ability of a computer to repeat the same operations using different data is a powerful tool that greatly reduces programming effort. There are several ways to accomplish this when using the Fortran language. For example, assume that a manufacturer carries 1,000 different parts in inventory. Periodically, it is necessary to compute the stock on hand of each item (STOCK) by subtracting stock withdrawals of that item (OUT) from the previous stock on hand. These results could be achieved by the following statements:

```
        .
        .
        .
 5      I=0
10      I=I + 1
25      STOCK (I) = STOCK (I) - OUT (I)
15      IF (I-1000) 10,30,30
```

The three statements (5, 10, and 15) required to control this loop could be replaced by a single DO statement.

General Form:

DO n i = m_1, m_2
or
DO n i = m_1, m_2, m_3

where:

n is any statement number, i is a nonsubscripted integer variable, and m_1, m_2, m_3 are unsigned integer constants or nonsubscripted integer variables. If m_3 is not stated (it is optional), its value is assumed to be 1. In this case, the preceding comma must also be omitted.

Examples:

```
DO 50 I = 1, 1000
DO 10 I = J, K, L
DO 11 I = 1, K, 2
```

The DO statement is a command to repeatedly execute the statements that follow, up to and including the

statement n. The first time the statements are executed, i has the value m_1, and each succeeding time, i is increased by the value of m_3. After the statements have been executed with i equal to the highest value that does not exceed m_2, control passes to the statement following statement number n. This is called a normal exit from the DO statement.

The range (n) is the series of statements to be executed repeatedly. It consists of all statements following the DO, up to and including statement n. The range can consist of any number of statements.

The index (i) is an integer variable that is incremented for each execution of the range of statements. Throughout the range of the DO, the index is available for use either as a subscript or as an ordinary integer variable. However, the index may not be changed by a statement within the range of the DO. Upon the completion of the DO, the index must be redefined before being used again. When transferring out of the range of a DO, the index is available for use and is equal to the last value it attained.

The initial value (m_1) is the value of the index for the first execution of the range. The initial value cannot be equal to zero.

The test value (m_2) is the value that the index must not exceed. After the range has been executed with the highest value of the index that does not exceed the test value, the DO is completed and the program continues with the first statement following the range. The test value is compared with the index value at the end of the range; therefore, a DO loop will always be executed at least once.

The increment (m_3) is the amount by which the value of the index will be increased after each execution of the range. The increment may be omitted, in which case it is assumed to be 1.

Example:

```
      DO 25 I=1, 10
  5   .
 10   .
 15   .
 20   .
 25   A=B+C
 26   .
```

This example shows a DO statement that will execute statements 5, 10, 15, 20, and 25 ten times. Upon each execution, the value of I will be incremented by 1 (1 is assumed when no increment is specified). After completion of the DO, statement 26 is executed.

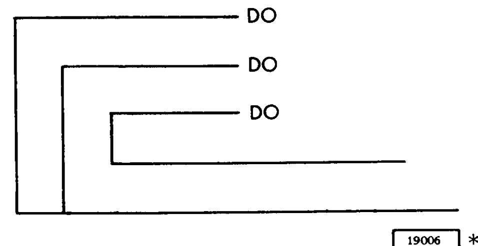
In some cases, the DO is completed before the test value is reached. Consider the following:

```
DO 5 K=1, 9, 3
```

In this example, the range is executed three times (i.e., K equal to 1, 4, and 7). The next value of K would be 10. Since this exceeds the test value, the DO is completed after three iterations.

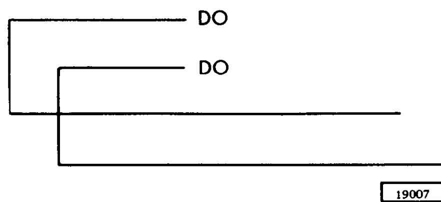
Restrictions. The restrictions on statements in the range of a DO are:

1. Within the range of a DO may be other DOs. When this is so, all statements in the range of the inner DO must be in the range of the outer DO. A set of DOs satisfying this rule is called a nest of DOs. The maximum depth of a single nest of DOs is 25. For example, the following configuration is permitted (brackets are used to indicate the range of the DOs):



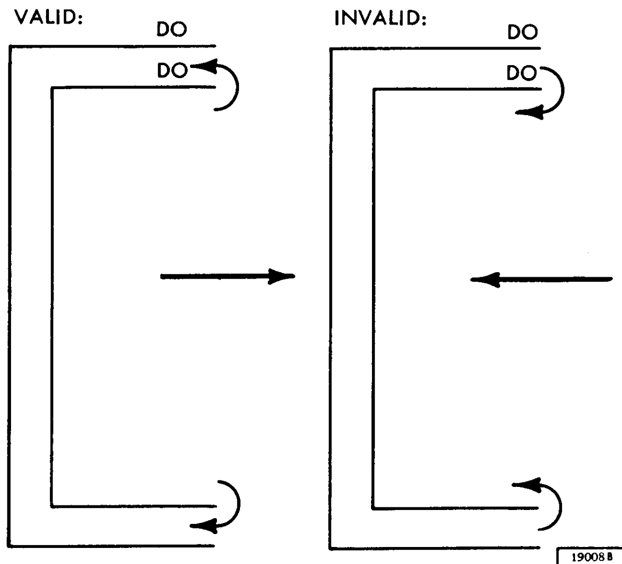
*

but, the following configuration is not permitted:



2. Transfer of control from inside the range of a DO to outside its range is permitted at any time. If, and only if, a transfer is made from the range of an innermost DO loop, transfer back into the range of that innermost DO loop is allowed provided none of the indexing parameters (i, m_1, m_2, m_3) are changed outside the range of the DO. A transfer back into the range of any other DO in the nest of DOs is not permitted. The following illustrations show those transfers that are valid and those that are invalid.

*NOTE: The illustrations in this manual have a code number in the lower corner. This is a publishing control number and is unrelated to the subject matter.



3. The last statement in the range of a DO loop must be an executable statement; however, it must not be a GO TO, IF, STOP, PAUSE, RETURN, or another DO statement.
4. Any statement that redefines the value of the index or any of the indexing parameters (i.e., m_1, m_2, m_3) is not permitted in the range of a DO.

CONTINUE Statement

CONTINUE is a dummy statement that does not produce any executable instructions. It is used as the last statement of a DO loop to provide a branch address (statement number) for GO TO statements that are intended to begin another repetition of the DO range.

General Form:

CONTINUE

In the following example, the DO loop is executed 20 times. The CONTINUE statement provides the branch address to begin the DO loop again when I<20. When I=20, the DO loop is executed once more and the CONTINUE statement then provides the branch address for the next sequential statement outside the DO loop, that is, statement 40.

```
      DO 30 I = 1, 20
      D = D + 5. 0
 7    IF (A - B) 10, 30, 30
10    A = A + 1. 0
      B = B - 2. 0
      GO TO 7
30    CONTINUE
40    C = A + B
        .
        .
```

PAUSE Statement

General Form:

PAUSE or PAUSE n

where:

n is an unsigned integer constant whose value is equal to or less than 9999.

The PAUSE statement causes the program to stop on a Wait instruction. If n is specified, its hexadecimal value will be displayed on the console by the accumulator lights. Pressing the Start key on the console causes the program to resume execution, starting with the next executable statement following the PAUSE statement.

STOP Statement

General Form:

STOP or STOP n

where:

n is an unsigned integer constant whose value is equal to or less than 9999.

The STOP statement terminates the program. If n is specified, its hexadecimal value will be displayed on the console by the accumulator lights.

END Statement

General Form:

END

The END statement defines the end of a program or subprogram for the compiler. Physically, it must be the last statement of each program or subprogram. The END statement is not executable. Any source program cards following the END card will not be compiled.

INPUT/OUTPUT STATEMENTS

The Input/Output (I/O) statements control the transmission of information between the computer and I/O units such as the card reader, card punch, or printer, etc. I/O statements are classified as follows:

1. General I/O Statements. These statements cause transmission of information between the computer and I/O units. They are READ and WRITE.
2. FORMAT Statements. These are non-executable statements that specify the arrangement of the data to be transferred, and the editing transformation required between internal and external forms of the data. The FORMAT statements are used in conjunction with the general I/O statements.

General I/O Statements

READ Statement

The READ statement is used to transfer information from any input unit to the computer. Two forms of the READ statement may be used, as follows:

READ (a, b) List

READ (a, b)

where:

a is an unsigned integer constant or integer variable that specifies the logical unit number to be used for input data. The logical unit numbers for the 1130 System are:

2	1442 Card Reader
4	1054 Paper-Tape Reader
6	Console Keyboard

b is the statement number of the FORMAT statement describing the type of data conversion.

List is a list of variable names, separated by commas, for the input data.

The READ (a, b) List form is used to read a number of items (corresponding to the variable names in the list) from the file on unit a, using FORMAT statement b to specify the internal representation of these data (see FORMAT Statement).

The List specifies the number of items to be read and the locations into which the items are to be placed. For example, assume that a card is punched as follows:

Card Columns	Contents
1-2	25
5-7	102
61-64	-101
70-71	10
80	5

If the following statement appears in the source program:

READ (2, 25) I, J, K, L, M

the card is read (assuming that 25 is the number of an appropriate FORMAT statement), and the program operates as though the following statements had been written:

I = 25
J = 102
K = -101
L = 10
M = 5

For the next execution of the READ statement, I, J, K, L, and M will have new values, depending upon what is punched in the next card to be read.

Any number of quantities may appear in a single list. Integer and real quantities may be transmitted by the same statement.

If there are more quantities to be transmitted than there are items in the list, only the number of quantities equal to the number of items in the list are transmitted; remaining quantities are ignored. Thus, if a card contains three quantities and a list contains two, the third quantity is lost. Conversely, if a list contains more quantities than the input record, succeeding cards (or other input records) are read until all the items specified in the list have been transmitted.

When an array name appears in an I/O list in nonsubscripted form, all of the quantities in the array are transmitted in the order in which they are stored (see Arrangements of Arrays in Storage). For example, assume that A is defined as an array of 25 quantities. Then, the statement:

READ (2, 15) A

causes all of the quantities A(1), . . . , A(25) to be read into storage (in that order) from the 1442 Card Reader.

Indexing I/O Lists

Variables within an I/O list may be indexed and incremented in the same manner as with a DO statement.

For example, suppose it is desired to read data into the first five positions of the array A. This may be accomplished by using an indexed list, as follows:

```
     READ (2, 15) (A(I), I=1, 5)
15   FORMAT (F10. 3)
```

This is equivalent to:

```
     DO 12 I=1, 5
12   READ (2, 15) A(I)
15   FORMAT (F10. 3)
```

As with DO statements, a third indexing parameter may be used to specify the amount by which the index is to be incremented at each iteration. Thus,

```
     READ (2, 15) (A(I), I=1, 10, 2)
```

causes transmission of values for A(1), A(3), A(5), A(7), and A(9). Furthermore, this notation may be nested. For example, the list:

```
     ((C(I, J), D(I, J), J=1, 5), I=1, 4)
```

would transmit data in the following order, reading from left to right:

```
C(1, 1), D(1, 1), C(1, 2), . . . , C(1, 5), D(1, 5)
C(2, 1), D(2, 1), C(2, 2), . . . , C(2, 5), D(2, 5)
C(3, 1), D(3, 1), C(3, 2), . . . , C(3, 5), D(3, 5)
C(4, 1), D(4, 1), C(4, 2), . . . , C(4, 5), D(4, 5)
```

The READ (a, b) form may be used in conjunction with a FORMAT statement to read H-type alphameric data into an existing H-type field in core storage (see <u>Conversion of Alphameric Data</u>). The size of the data field determines the amount of data to be read. For example, the statements:

```
10   FORMAT (23HTHIS IS ALPHAMERIC DATA)
       .
       .
       .
     READ (INPUT, 10)
```

cause the next 23 characters to be read from the file on the unit named INPUT and placed into the H-type alphameric field whose contents were:

THIS IS ALPHAMERIC DATA

WRITE Statement

The WRITE statement is used to transfer information from the computer to any of the output units (tape, printer, card punch, etc.). Two forms of the WRITE statement may be used as follows:

```
     WRITE (a, b) List
     WRITE (a, b)
```

where:

a is an unsigned integer constant or integer variable that specifies the logical unit number to be used for output data. The logical unit numbers for the 1130 System are:

1	Console Printer
2	1442 Card Punch
3	1132 Printer
4	1055 Paper-Tape Punch

b is the statement number of the FORMAT statement describing the type of data conversion.

List is a list of variable names separated by commas for the output data.

The WRITE (a, b) List form of the WRITE statement is used to write the data specified in the list on the file on unit a, using FORMAT statement b to specify the external representation of the data (see <u>FORMAT Statement</u>).

NOTE: The 1442 Card Read Punch has one input hopper. Therefore, if a READ or WRITE statement references a 1442, care should be taken to avoid punching a card that was only meant to be read or reading a card that was only meant to be punched.

The WRITE (a, b) form is used to write alphameric data (see <u>Conversion of Alphameric Data</u>). The actual data to be written is specified within the FORMAT statement; therefore, an I/O list is not required. The following statements illustrate the use of this form:

```
25   FORMAT (24HWRITE ANY DATA IN H TYPE)
       .
       .
       .
     WRITE (2, 25)
```

Specifying Format

In order for quantities to be transmitted from an external storage medium (e. g. , cards or paper tape) to the computer or from the computer to an external medium (cards, paper tape, or printed line), it is necessary that the computer know the form in which the data exists. This is accomplished by data conversion specifications within a FORMAT statement (see Conversion of Numeric Data).

FORMAT Statement

The I/O statements require, in addition to a list of quantities to be transmitted, reference to a FORMAT statement. The FORMAT statement describes the type of conversion to be performed between the internal and the external representation of each quantity in the list by the use of data conversion specifications (see Conversion of Numeric Data).

General Form:

m FORMAT $(k_1, k_2, \ldots, k_n/t_1, t_2, \ldots, t_n/\ldots)$

where:

$k_1, k_2, \ldots, k_n$ and $t_1, t_2, \ldots, t_n$ represent data conversion specifications.
/ represents the beginning of a new record, and m represents a statement number.

Examples:

```
 5  FORMAT (I5, F8.4)
18  FORMAT (I4/F6.2, F8.4)
20  FORMAT (E10.4/I8)
```

FORMAT statements are not executed but they must be given a statement number.

Slashes are used in a FORMAT statement to delimit unit records, which must be one of the following.

1. A punched card or paper tape record with a maximum of 80 characters.
2. A printed line with a maximum of 120 print characters and 1 carriage control character.
3. A typewritten line with a maximum of 120 characters.

Thus, the statement:

```
5   FORMAT (F9.2/E14.5)
```

specifies the data conversion specification F9. 2 for the first unit record, and the data conversion specification E14. 5 for the second unit record.

Successive items in the I/O list are transmitted according to successive specifications in the FORMAT statement, until all items in the list are transmitted. If there are more items in the list than there are specifications in the FORMAT statement, control transfers to the preceding left parenthesis (including any preceding repeat constant) of the FORMAT statement and the same specifications are used again with the next unit record. For example, suppose a program contains the following statements:

```
10  FORMAT (F10.3,E12.4,F12.2)
         .
         .
         .
    WRITE (3,10) A,B,C,D,E,F,G
```

The following table shows the data transmitted in the column on the left and the specification by which it is converted in the center column. The column on the right shows the number of the record which contains the data.

Data Transmitted	Specification	Record Number
A	F10.3	1
B	E12.4	1
C	F12.2	1
D	F10.3	2
E	E12.4	2
F	F12.2	2
G	F10.3	3

A specification may be repeated as many times as desired (within the limits of the output unit) by preceding the specification with an unsigned integer constant. Thus,

(2F10.4)

is equivalent to:

(F10.4, F10.4)

A limited, one-level, parenthetical expression is permitted to enable repetition of data fields according to certain format specifications within a longer FORMAT statement. For example, the statement:

```
10  FORMAT (2(F10.6, E10.2), I4)
```

is equivalent to:

```
10  FORMAT (F10.6,E10.2,F10.6,E10.2,I4)
```

If there had been 8 items in the list, the above FORMAT statement would have been equivalent to:

```
10   FORMAT (F10.6,E10.2,F10.6,E10.2,I4/
             F10.6,E10.2,F10.6)
```

The specifications in a FORMAT statement need not correspond in mode with the list items in the I/O statement; automatic input conversion will convert external values (I-type, E-type, and F-type) to the correct internal representation depending on the type of the variable in the READ statement list. The same type of conversion will be handled for variables in the WRITE statement list.

Conversion of Numeric Data

Three types of specifications (or conversion codes) are available for the conversion of numeric data. These types of conversions are specified in the following form:

Iw
Fw.d
Ew.d

where:

I, F, and E specify the type of conversion.
w is an unsigned integer constant specifying the total field length of the data. (This specification may be greater than that required for the actual digits in order to provide spacing between numbers.)
d is an unsigned integer constant specifying the number of decimal places to the right of the decimal point.

NOTE: The decimal point between the w and d portions of the specification is required.

For purposes of simplification, the following discussion of conversion codes deals with the printed line. The concepts developed apply to all permissible input/output media.

I-Conversion (Iw)

The specification I6 may be used to print a number in integer form; 6 print positions are reserved for the number. It is printed in this 6-position field right-justified (that is, the units position is at the extreme right). If the number to be converted is greater than 5 positions, an error condition will exist because one position must be reserved for the sign. If the number has less than 5 digits, the leftmost print positions are filled with blanks. If the quantity is negative, the position preceding the leftmost digit contains a minus sign.

For input it is not necessary to reserve a sign position. For example, I5 can be used to read in a 5-digit integer.

The following examples show how each of the quantities on the left is printed, according to the specification I3:

Internal Value	Printed
721	***
-721	***
-12	-12
8114	***
0	0
-5	-5
9	9
1.7	1

NOTE: All error fields are filled in with asterisks.

F-Conversion (Fw.d)

For F-type conversion, w is the total field length reserved and d is the number of places to the right of the decimal point (the fractional portion). The total field length reserved must include sufficient positions for a sign and a decimal point. The sign, if negative, is printed. An output error condition will result if w ≤ d+1.

If insufficient positions are reserved by d, the fractional portion is truncated from the right. If excessive positions are reserved by d, zeros are filled in from the right. The integer portion of the number is handled in the same fashion as numbers converted by I-type conversion on input and output.

The following examples show how each of the quantities on the left is printed according to the specification F5.2:

Internal Value	Printed
12.17	*****
-41.16	*****
-.2	-.20
7.3542	7.35†
-1.	-1.00
9.03	9.03
187.64	*****
5	5.00

†Last two digits of accuracy lost due to insufficient specification.

NOTES:

1. All error fields are filled in with asterisks.

2. Numbers for F-conversion input need not have their decimal points appearing in the input field. If no decimal point appears, space need not be allocated for it. The decimal point will be supplied when the number is converted to an internal equivalent; the position of the decimal point will be determined by the format specification. However, if the decimal point does appear within the field and it is different from the format specification, this position overrides the position indicated in the format specification.

3. Fractional numbers for which F-type output conversion is specified are normally printed without a leading zero. If F-conversion is used and zero decimal width is specified (for example, F5.0), a fractional value is printed as a sign and a decimal point; however, a zero value is printed with a zero preceding the decimal point if the field width is sufficient.

E-Conversion (Ew.d)

For E-conversion, the fractional portion is again indicated by d. The w includes field d, spaces for a sign, a decimal point, and the letter E, plus space(s) for the exponent. Space must be reserved for each of these on output. An output error condition will result if w ≤d+5. For input, it is not necessary to reserve a sign position.

The exponent is a signed or unsigned one- or two-digit integer constant not greater than 38. Ten (10) raised to the power of the exponent is multiplied by the number to obtain its true internal value.

The following examples show how each of the quantities on the left is printed, according to the specification E9.3:

Internal Value	Printed
238.	.238Eb03
-.002	-.200E-02
.00000000004	.400E-10
-21.0057	-.210Eb02†

†Last three digits of accuracy lost due to insufficient specification.
b represents a blank.

NOTES:

1. For input, the start of the exponent field must be marked by an E, or, if that is omitted, by a + or - sign (not blank). Thus, E2, E+2, +2, +02, E02, and E+02 are all permissible exponent fields for input.

2. For input, the exponent field may be omitted entirely (i.e., E-conversion will accept input data in F-type format).

3. Numbers for E-conversion input need not have their decimal points appearing in the input field. If no decimal point appears, space need not be allocated for it. The decimal point will be supplied when the number is converted to an internal equivalent; the position of the decimal point will be determined by the format specification. However, if the decimal point does appear within the field and it is different from the format specification, this position overrides the position indicated in the format specification.

Conversion of Alphameric Data

There are two specifications available for input/output of alphameric data: H-conversion or literal data enclosed in quotes, and A-conversion. H-conversion is used for alphameric data that is not going to be changed by the object program (e.g., printed headings); A-conversion is used for alphameric data in storage which is to be operated on by the program (e.g., modifying a line to be printed).

H-Conversion

The specification nH is followed in the FORMAT statement by n alphameric characters. For example:

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

Blanks are considered alphameric data and must be included as part of the count n. A comma following the last alphameric character is optional.

The effect of nH depends on whether it is used with an input or output statement.

Input: n characters are extracted from the input record and replace the n characters included in the specification. For example,

```
        READ (4, 5)
    5   FORMAT (8HHEADINGS)
```

would cause the next 8 characters to be read from the input file on the Paper-Tape Reader; these characters would replace the data HEADINGS in storage.

Output: The n characters following the specification are written as part of the output record. Thus, the statements:

```
        WRITE (1,6)
    6   FORMAT (15H CUST. NO. NAME)
```

would cause the following record to be written on the Console Printer:

```
        CUST. NO. NAME
```

Literal Data Enclosed in Quotes

Literal data can consist of a string of alphameric and special characters written within the FORMAT statement and enclosed in quotes (a comma following the last quote is optional). For example:

```
25   FORMAT (' 1966 INVENTORY REPORT')
```

A quote character within literal data is represented by two successive quote marks. For example, the characters DON'T are represented as:

```
DON''T
```

The effect of the literal format code depends on whether it is used with an input or output statement.

Input: A number of characters, equal to the number of characters between the quotes, are read from the designated I/O unit. These characters replace, in storage, the characters within the quotes. For example, the statements:

```
          .
          .
5    FORMAT ('  HEADINGS')
          .
          .
          .
     READ (4, 5)
          .
          .
```

would cause the next 9 characters to be read from the Paper-Tape Reader, these characters would replace the blank and the 8 characters H, E, A, D, I, N, G, and S in storage.

Output. All characters (including blanks) within the quotes are written as part of the output data. Thus the statements:

```
          .
          .
          .
5    FORMAT ('  THIS IS ALPHAMERIC DATA')
          .
          .
          .
     WRITE (1, 5)
          .
          .
          .
```

would cause the following record to be written on the Console Printer:

```
THIS IS ALPHAMERIC DATA
```

A-Conversion

The specification Aw is used to transmit alphameric data to/from variables in storage. It causes the first w characters to be read into, or written from, the area of storage specified in the I/O list. For example, the statements:

```
10   FORMAT (A4)
          .
          .
          .
     READ (4, 10) ERROR
```

would cause four alphameric characters to be read from the Paper-Tape Reader and placed (left-justified) into the field in storage named ERROR.

The following statements:

```
     INTEGER OUT
15   FORMAT (3HXY=, F9.3, A4)
          .
          .
          .
     WRITE (OUT, 15)A, ERROR, B, ERROR
```

may produce the following lines:

```
XY= 5976.214----
XY= 6173.928----
```

where ---- represents the contents of the field ERROR.

Thus, A-conversion provides the facility for reading alphameric data into a field in storage, manipulating the data as required, and printing it out.

If the number of alphameric characters is less than the length of the field in storage into which they are to be read, then the remaining rightmost characters in the field are loaded with blanks. However, if the number of characters is greater than the length of the field in storage, only the rightmost characters are read in and the excessive leftmost characters are lost. It is important, therefore, to allocate enough area in storage to handle the alphameric characters being read in. Each real variable has

sufficient space for 4 or 6 characters (the precision of real variables is specified at compile time); each integer variable has space for 2 characters. For example, 10 characters could be read into, or written from, the first five positions of the array I (I is an integer variable). Thus, two characters are contained in each of the five consecutive positions: I(1), I(2), I(3), I(4), I(5).

Arithmetic operations involving variables containing alphameric characters should be performed in integer mode. Alphameric characters are represented internally in eight-bit EBCDIC code (refer to the IBM 1130 Subroutine Library, Form C26-5929, for a description of the EBCDIC code used for internal representation of alphameric characters).

Blank Fields

Blank characters may be provided in an output record, or characters of an input record may be skipped, by means of the specification, nX; n is the number of blanks desired or the number of characters to be skipped.

When the nX specification is used with an input record, n characters are skipped over before the transmission of data begins.

For example, if a card has six 10-column fields of integers, the statement:

```
5    FORMAT (I10, 10X, 4I10)
```

would be used, along with the appropriate READ statement, to avoid reading the second quantity.

When this specification is used with an output record, n positions are left blank. Thus, the facility for spacing within a printed line is available. For example, the statement:

```
10   FORMAT (3(F6.2, 5X))
```

may be used with the appropriate WRITE statement to print a line as follows:

```
-23.45bbbbbb17.32bbbbbb24.67bbbbb
```

where b represents a blank.

Multiple Field Format

Blank lines may be introduced between output records, or input records may be skipped, by using consecutive slashes (/) in a FORMAT statement. The number of input records skipped, or blank lines inserted between output records, depends upon the number and placement of the slashes within the statement.

If there are n consecutive slashes at the beginning or end of a format specification, n input records are skipped or n blank lines are inserted between output records. If n consecutive slashes appear anywhere else in a format specification, the number of records skipped or blank lines inserted is n-1. For example, the statements:

```
10   FORMAT (///I6)
     READ (INPUT, 10) MULT
```

cause 3 records to be skipped on the input file before data is read into MULT.

The statements:

```
15   FORMAT (I5,////,F5.2,I2//)
     WRITE (IOUT, 15) K,A,J
```

result in the following output:

```
Integer
(blank line)
(blank line)
(blank line)
Real Number     Integer
(blank line)
(blank line)
```

NOTE: The comma before or after the / is optional.

To obtain a multiline listing in which the first two lines are to be printed according to a special format and all remaining lines according to another format, the last-line specification should be enclosed in a second pair of parentheses. For example, in the statement:

```
FORMAT (I2,3E12.4/2F10.3,3F9.4/(3F12.4))
```

when data items remain to be transmitted after the format specification has been completely used, the format repeats from the last left parenthesis. Thus, the listing would take the following form:

```
I2, E12.4, E12.4, E12.4
F10.3, F10.3, F9.4, F9.4, F9.4
F12.4, F12.4, F12.4
F12.4, F12.4, F12.4
  .      .      .
```

Carriage Control

If a printed line is being edited, the first character of the line will be used for controlling the printer carriage. Under program control, this character will control spacing of the printer and will not be printed. The control characters and their effects are:

blank	-	Single space before printing
0	-	Double space before printing
1	-	Sheet eject before printing
+	-	Suppress space before printing

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

Data Input to the Object Program

Data input to the object program is contained in unit records, as described in the section FORMAT Statement. The following information should be considered when preparing input data on punched cards:

1. The input data record must correspond to the field width specifications defined in the FORMAT statement.
2. Blanks within a number are not allowed; however, blanks may precede the number in the field. Thus, all numbers must be right-justified in a field.
3. A plus sign may be implied by no sign or indicated by a plus sign; a negative number, however, must be preceded by a minus sign.

SPECIFICATION STATEMENTS

The Specification statements are nonexecutable because they do not cause the generation of instructions in the object program. Instead, they provide the compiler with information about the nature of the constants and variables used in the program. In addition they supply the information required to allocate locations in storage for certain variables and/or arrays.

All specification statements must precede the first executable statement of the source program. The Specification statements must appear in the following order:

Type Statements (REAL, INTEGER)
EXTERNAL Statements
DIMENSION Statements
COMMON Statements
EQUIVALENCE Statements

Type Statements (REAL, INTEGER)

General Form:

```
INTEGER a, b, c,...
REAL a, b, c,...
```

where:

a, b, c, ... are variable, array, FUNCTION subprogram or arithmetic statement function names appearing in a program or subprogram. Arrays named in this statement must also be dimensioned in this statement.

Examples:

```
INTEGER   DEV, JOB, XYZ12, ARRAY(5,2,6)
REAL      ITA, SMALL, ANS, NUMB(3, 14)
```

The REAL and INTEGER statements explicitly define the type of variable, array, or function. In the first example, the variable DEV (implicitly defined as a real variable, because its initial letter is not I, J, K, L, M, or N) is explicitly defined as an integer variable and is, therefore, handled as an integer variable in the program. The appearance of a variable name in either of these statements overrides any implicit type specification determined by the initial letter of the variable.

Type statements must precede any other Specification statements and all executable statements in the source program.

EXTERNAL Statement

General Form:

```
EXTERNAL a,b,c,...
```

where:

a,b,c, ... are the names of FUNCTION subprograms, SUBROUTINE subprograms, or Fortran-supplied subprograms that appear in any argument list.

Example:

EXTERNAL SIN, MATRX, INVRT

Any subprogram named in the EXTERNAL statement may be used as an argument for other subprograms (see SUBPROGRAM STATEMENTS). Subprograms named in an EXTERNAL statement are loaded at execution time.

DIMENSION Statement

General Form:

DIMENSION $a(k_1)$, $b(k_2)$, $c(k_3)$, ... $x(k_n)$

where:

a, b, c, ... x are names of arrays.

$k_1, k_2, k_3, \ldots k_n$ are each composed of 1, 2, or 3 unsigned integer constants that specify the maximum value for 1, 2, or 3 subscripts, respectively.

Example:

DIMENSION A(10), B(5, 15), C(9, 9, 9)

The DIMENSION statement provides information to allocate storage for arrays in an object program (unless the information appears in a Type or COMMON statement). It defines the maximum size of each array listed.

Each variable that appears in subscripted form in a source program must appear in a Type, DIMENSION, or COMMON statement contained within the source program. The first of these statements that refers to the array must give dimension information. (See COMMON Statement - With Dimensions.)

COMMON Statement

General Form:

COMMON a, b, c, ... n

where:

a, b, c, ... n are variable or array names.

Variables or arrays that appear in the main program or a subprogram may be made to share the same storage locations with variables or arrays of the same type and size in other subprograms, by use of the COMMON statement. For example, if one program contains the statement:

COMMON TABLE

and a second program contains the statement:

COMMON LIST

the variable names TABLE and LIST refer to the same storage locations (assuming the data associated with the names TABLE and LIST are equal length and type).

If the main program contains the statement:

COMMON A, B, C

and a subprogram contains the statement:

COMMON X, Y, Z

and A, B, and C are equal in length to X, Y, and Z, respectively, then A and X refer to the same storage locations, as do B and Y, and C and Z.

Within a specific program or subprogram, variables and arrays are assigned storage locations in the sequence in which their names appear in a COMMON statement. Subsequent sequential storage assignments within the same program or subprogram are made with additional COMMON statements.

A dummy variable can be used in a COMMON statement to establish shared locations for variables that would otherwise occupy different locations. For example, the variable S can be assigned to the same location as the variable Z of the previous example with the following statement:

COMMON Q, R, S

where Q and R are dummy names that are not used elsewhere in the program.

Redundant COMMON entries are not allowed. For example, the following is invalid:

COMMON A, B, C, A

COMMON Statement—With Dimensions

General Form:

COMMON $a(k_1), b(k_2), c(k_3), \ldots n(k_n)$

where:

a, b, c, ... n are array names and

$k_1, k_2, k_3, \ldots k_n$ are each composed of 1, 2, or 3 unsigned integer constants that specify the dimensions of the array.

Example:

COMMON A(10), B(5,5,5), C(5,5,5)

This form of the COMMON statement, besides performing the functions discussed previously for the COMMON statement, performs the additional function of specifying the size of arrays.

NOTES:

1. Dummy arguments for SUBROUTINE or FUNCTION statements cannot appear in COMMON statements.

2. A single COMMON statement may contain variable names, array names, and dimensioned array names. For example, the following are valid:

```
DIMENSION B(5,15)
COMMON A, B, C(9,9,9)
```

3. All dimensioned arrays in a main program or subprogram and all items in COMMON are stored in descending storage locations.

EQUIVALENCE Statement

Different variables and arrays are usually assigned unique storage locations. However, it may be desirable to have two or more variables of the same type and size share the same storage locations. This facility is provided by the EQUIVALENCE statement.

General Form:

EQUIVALENCE (a, b, ...), (d, e, ...), ...

where:

a, b, d, e, ... are simple variables or subscripted variables. Subscripted variables may have either multiple subscripts (which must agree with the DIMENSION statement) or single subscripts. The subscripts must be integer constants.

Each pair of parentheses in the EQUIVALENCE statement encloses two or more variable names that refer to the same location during the execution of the object program.

Any number of variables may be listed in a single EQUIVALENCE statement.

Examples:

```
EQUIVALENCE (A, B, SAVE, AREA),
                (E(1), F(1)), (G(1), H(5))
EQUIVALENCE (A(4), C(2), D(1))
```

In the second example, making A(4), C(2), and D(1) equivalent to one another sets up an equivalence among the elements of each array as follows:

```
A(1)
A(2)
A(3)   C(1)
A(4)   C(2)   D(1)
A(5)   C(3)   D(2)
 .      .      .
 .      .      .
```

An EQUIVALENCE statement must not contradict any previously established equivalences.

Within an EQUIVALENCE List, there may be no more than one variable which previously has been:

1. EQUIVALENCED, or
2. placed in COMMON

The following sequence of statements is invalid:

```
COMMON D
EQUIVALENCE (A, B, C)
EQUIVALENCE (X, Y, Z)
EQUIVALENCE (A, Z)
EQUIVALENCE (D, X, P)
```

The following is valid:

```
COMMON D
EQUIVALENCE (D, X, P)
EQUIVALENCE (A, B, C, X)
EQUIVALENCE (X, Y, Z)
```

COMMON with EQUIVALENCE Statements

No two elements that appear in a COMMON statement may be made equivalent. Both of the following examples are invalid:

```
COMMON A, B             COMMON A, B
EQUIVALENCE (A, B)      EQUIVALENCE (A, R),
                          (R, D), (D, B)
```

However, EQUIVALENCE statements may extend the size of the COMMON area. For example, the following is valid:

```
DIMENSION C(4)
COMMON A, B
EQUIVALENCE (B, C(2))
```

for it would produce the following relationship in the COMMON area:

```
A    C(1)
B    C(2)
     C(3)
     C(4)
```

Since arrays must be stored in descending storage locations, a variable may not be made equivalent to an element of an array in such a manner as to cause the array to extend beyond the beginning of the COMMON area. For example, the following coding is invalid:

```
DIMENSION C(4)
COMMON A, B
EQUIVALENCE (A, C(2))
```

for it would force C(1) to precede A in the COMMON area, as follows:

```
     C(1)    (outside the COMMON area)
A    C(2)
B    C(3)
     C(4)
```

Conversion to Single Subscripts

Two- and three-dimensional arrays actually appear in storage in a one-dimensional sequence of core storage words.

In an EQUIVALENCE statement it is possible to refer to elements of multi-dimensioned arrays by single-subscripted variables. For example, in an array dimensioned A (3, 3, 3), the fourth element of the array can be referenced as A(1, 2, 1) or as A(4).

The rules for converting multiple subscripts to single subscripts are as follows:

1. For a two-dimensional array, dimensioned as A(I, J): the element A(i, j) can also be referenced as A(n), where $n = i + I(j-1)$.
2. For a three-dimensional array, dimensioned as A(I, J, K): the element A(i, j, k) can also be referenced as A(n), where $n = i + I(j-1) + I * J(k-1)$.

SUBPROGRAM STATEMENTS

Suppose that a program is being written which, at various points, requires the same computation to be performed with different data for each calculation. It would simplify the writing of that program if the statements required to perform the desired computation could be written only once and then could be referred to freely. Each reference to the statements would have the same effect as if the statements were written at the point in the program where the reference was made. For example, if a general program were written to take the square root of any number, it would be desirable to be able to incorporate that

program (or subprogram) into other programs where square root calculations are required.

The Fortran language provides for the preceding situation through the use of subprograms. There are three classes of subprograms: Statement functions, FUNCTION Subprograms, and SUBROUTINE Subprograms. In addition, there is a group of Fortran supplied subprograms.

The first two classes of subprograms are called functions. Functions differ from the SUBROUTINE subprograms in that functions always return a single value to the calling program, whereas, a SUBROUTINE subprogram can return any number of values to the calling program. A function is employed (or called) by writing the name of the function (see Subprogram Names) and an argument list in a standard arithmetic expression. A SUBROUTINE subprogram must be called by a special Fortran statement, namely, the CALL statement.

The Statement function is written and compiled as part of the program in which it appears. The other subprograms are written and compiled separately and linked to the main program at the time they are loaded for execution.

Subprogram Names

A subprogram name consists of 1-5 alphameric characters, the first of which must be alphabetic. The type (real or integer) of a subprogram can be indicated in the same manner as variables.

The type of a statement function may be indicated implicitly by the initial character of the name or explicitly by the REAL or INTEGER Type statement.

The type of a Fortran supplied subprogram is indicated implicitly by the initial character of its name.

The type of a FUNCTION subprogram may be indicated implicitly by the initial character of the name or explicitly by a Type specification (see Type Specification of the FUNCTION Subprogram). In the latter case, the implicit type is overridden by the explicit specification.

The type of a SUBROUTINE subprogram is not defined, because the result returned to the main program is dependent only on the type of the variable names in the argument list.

A list of the presently used names for subprograms is contained in the publication, IBM 1130 Subroutine Library (Form C26-5929).

Functions

In mathematics, a function is a statement of the relationship between a number of variables; the value of the function depends upon the values assigned to the variables (or arguments) of the function. The same definition of function is true in Fortran. To use a function in Fortran, it is necessary to:

1. Define the function, that is:
   a. Assign a unique name by which it may be called.
   b. State the arguments of the function.
   c. State the procedure for evaluating the function.
2. Call the function, where required, in the program.

When the name of a function appears in any Fortran arithmetic expression, program control is transferred to the function routine. Thus, the appearance of the function with its arguments causes the computations indicated by the function definition to be performed. The resulting quantity replaces the function reference in the expression and assumes the mode of the function. The mode of a function, as with variables, is determined either implicitly by the initial character of its name, or explicitly by a Type statement.

Statement Function

General Form:

a = b

where:

a is a function name followed by parentheses enclosing its arguments, which must be distinct, non-subscripted variables separated by commas.

b is an expression that does not involve subscripted variables.

Examples:

```
FIRST(X) = A*X+B
OTHER(D) = FIRST (E)+D
```

If the statement Y = OTHER(Z) appears in a program in which the above functions are defined, the current values of A, B, E, and Z will be used in a calculation which is equivalent to:

```
Y = A*E+B+Z
```

Since the arguments of "a" are dummy arguments, their names may be the same as names appearing elsewhere in the program. Those variables in b that are not included in the dummy argument list are the parameters of the function and are defined as the ordinary variables appearing elsewhere in the source program. The type of each dummy argument is defined implicitly. A maximum of fifteen variables appearing in the expression may be used as arguments of the function.

Any Statement function appearing in b must have been previously defined. All definitions of Statement functions must follow the Specification statements and precede the first executable statement of the source program.

Statement functions are compiled as internal subprograms; therefore, they will appear only once in the object program.

NOTE: The same dummy arguments may be used in more than one Statement function definition and may also be used as variables outside Statement function definitions.

Fortran Supplied Subprograms

Fortran supplied subprograms are predefined subprograms that are part of the system library. A list of all the Fortran supplied subprograms is given in Table 1. Note that the type (real or integer) of each subprogram and its arguments are predefined and cannot be changed by the user.

To use a Fortran supplied subprogram, simply use the function name with the appropriate arguments in an arithmetic statement. The arguments may be arithmetic expressions, subscripted or simple variables, constants, or other Fortran supplied subprograms.

Examples:

```
DISCR = SQRT(B**2-4.0*A*C)
A = ABS (COS(B))
```

Table 1. Fortran Supplied Subprograms

Name	Function Performed	No. of Arguments	Type of Argument(s)	Type of Function
SIN	Trigonometric sine	1	Real	Real
COS	Trigonometric cosine	1	Real	Real
ALOG	Natural logarithm	1	Real	Real
EXP	Argument power of e (i.e.. e^x)	1	Real	Real
SQRT	Square root	1	Real	Real
ATAN	Arctangent	1	Real	Real
ABS	Absolute value	1	Real	Real
IABS	Absolute value	1	Integer	Integer
FLOAT	Convert integer argument to real	1	Integer	Real
IFIX	Convert real argument to integer	1	Real	Integer
SIGN	Transfer of sign (Sign of Arg_2 times Arg_1)	2	Real	Real
ISIGN	Transfer of sign (Sign of Arg_2 times Arg_1)	2	Integer	Integer
TANH	Hyperbolic tangent	1	Real	Real

19010 A

The use of the SQRT function in the first example causes the calculation of the value for the square root of the expression (B**2-4.0*A*C). This value replaces the current value of DISCR.

In the second example, cosine B is evaluated and its absolute value replaces the current value of A.

The Fortran Compiler adds an E or an F in front of the names of real Fortran supplied programs to specify required precision. Refer to IBM 1130 Subroutine Library (Form C26-5929) for descriptions of Fortran supplied subprograms and subprogram error detection routines.

FUNCTION Subprogram

The FUNCTION subprogram is a Fortran subprogram consisting of any number of statements. It is like a Fortran supplied subprogram in that it is an independently written program that is executed whenever its name appears in another program. In other words, if a user needs a function that is not available in the library, he can write it with Fortran statements.

General Form:

```
FUNCTION name (a1, a2, a3, ... an)
(FORTRAN statements)
     .
     .
RETURN
END
```

where:

name is a subprogram name, and $a_1, a_2, a_3, \ldots a_n$ are nonsubscripted variable names, array names, or other subprogram names (except that they may not be Statement function names).

The FUNCTION subprogram may contain any Fortran statement except a SUBROUTINE statement or another FUNCTION statement and must return control to the calling program with a RETURN statement.

The arguments of the Fortran subprogram may be considered to be dummy variable names. These are replaced at the time of execution by the actual arguments supplied in the function reference in the main program. The actual arguments must correspond in number, order, and type to the dummy arguments. The arguments in a FUNCTION subprogram may be any of the following: any type of constant, any type of subscripted or nonsubscripted variable, an arithmetic expression, or a subprogram name (except that they may not be Statement function names).

The relationship between variable names in the calling program and the dummy names in the FUNCTION subprogram is illustrated in the following example:

```
   Calling             FUNCTION
   Program             Subprogram

      .
      .                FUNCTION SOMEF (X, Y)
      .                    .
      .                SOMEF = X/Y
A = SOMEF (B, C)       RETURN
      .                END
      .
```

In the preceding example, the value of the variable B of the calling program is used in the subprogram as the value of the dummy variable X; the value of C is used in place of the dummy variable Y. Thus, if B = 10.0 and C = 5.0, then A = 2.0, that is, B/C.

When a dummy argument is an array name, an appropriate DIMENSION statement must appear in the FUNCTION subprogram.

When an argument is a subprogram name, it must be declared in an EXTERNAL statement in the calling program. The following example illustrates the use of the EXTERNAL and DIMENSION statements with subprograms.

Calling Program:

```
EXTERNAL    ABS
DIMENSION   A(4)
    .
    .
    .
I = 3
B = COMP(A, I, ABS)
    .
    .
    .
```

Called Subprogram:

```
   FUNCTION COMP(X, J, FUNCT)
   DIMENSION X(4)
   TEMP = 0
   DO 10 K = 1, J
10 TEMP = TEMP + X(K)
   COMP = FUNCT (TEMP)
   RETURN
   END
```

In this example, the resulting value of B returned to the calling program is equivalent to:

```
B = ABS(A(1) + A(2) + A(3))
```

The value of the formal arguments of a FUNCTION subprogram must not be redefined in the subprogram. That is, they must not appear on the left side of an arithmetic statement, or in an input list, or as the index in a DO statement. Variables that appear in common storage may not be redefined either. For example, the following violates this rule:

```
FUNCTION SAM (A, B, K)
COMMON J
J = J + 1
K = J
```

The name of the function must appear at least once as the variable name on the left side of an

arithmetic statement, in a READ statement, or in the argument list of a CALL statement. For example:

Calling Program:

```
ANS = ROOT1*CALC (X, Y, I)
```

FUNCTION Subprogram:

```
FUNCTION CALC (A, B, J)
    .
    .
I = J*2
    .
    .
CALC = A**I/B
    .
    .
RETURN
END
```

In this example, the values of X, Y, and I are used in the FUNCTION subprogram as the values of A, B, and J, respectively. The value of CALC is computed and this value is returned to the calling program where the value of ANS is computed.

Type Specification of the FUNCTION Subprogram

The type of function may be explicitly stated by the inclusion of the word REAL or INTEGER before the word FUNCTION. For example:

```
REAL FUNCTION SOMEF (A, B)
    .
    .
    .
RETURN
END
INTEGER FUNCTION CALC (X, Y, Z)
    .
    .
    .
RETURN
END
```

NOTE: The function type, if explicitly stated, must be defined in the calling program by use of the INTEGER or REAL type statement.

END and RETURN Statements

Note that all of the preceding examples of FUNCTION subprograms contain both an END and at least one RETURN statement. The END statement specifies the end of the subprogram for the compiler; the RETURN statement signifies the conclusion of a computation and returns any computed value and control to the calling program. There may, in fact, be more than one RETURN statement in a FUNCTION or SUBROUTINE subprogram. For example:

```
      FUNCTION DAV (D, E, F)
      IF(D-. 1)2, 3, 2
          .
          .
          .
2     DAV = . . . .
          .
          .
          .
      RETURN
3     DAV = . . . .
          .
          .
          .
      RETURN
      END
```

SUBROUTINE Subprogram

The SUBROUTINE subprogram is similar to the FUNCTION subprogram in many respects: the naming rules are the same, they both require a RETURN statement and an END statement, and they both contain the same sort of dummy arguments. Like the FUNCTION subprogram, the SUBROUTINE subprogram is a set of commonly used operations; but the SUBROUTINE subprogram does not restrict itself to a single value for the result, as does the FUNCTION subprogram. A SUBROUTINE subprogram can be used for almost any operation with as many results as desired.

The SUBROUTINE subprogram is called by a special FORTRAN statement, the CALL statement. It consists of the word CALL followed by the name of the subprogram and its parenthesized arguments.

General Form:

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

```
    .
    .
    .
RETURN
END
```

where:

name is the subprogram name (see Subprogram Names).

$a_1, a_2, a_3, \ldots a_n$ are the arguments (arguments are not necessary). Each argument used must be a nonsubscripted variable name, array name, or other subprogram name (except that it may not be a Statement function name).

Because the SUBROUTINE is a separate subprogram, the variables and statement numbers do not relate to any other program (except the dummy argument variables). The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. Any arguments so used must appear on the left side of an arithmetic statement or in an input list within the subprogram.

The arguments may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. The actual arguments must correspond in number, order and type to the dummy arguments. None of the dummy arguments may appear in an EQUIVALENCE statement in a SUBROUTINE subprogram. When the argument is an array name, an appropriate DIMENSION statement must appear in the SUBROUTINE subprogram.

CALL Statement

The CALL statement is used only to call a SUBROUTINE subprogram.

General Form:

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

where:

name is the symbolic name of a SUBROUTINE subprogram.

$a_1, a_2, a_3, \ldots a_n$ are the actual arguments that are being supplied to the SUBROUTINE subprogram.

Examples:

```
CALL MATMP (X,5,40,Y,7,2)
CALL QDRTI  (X,Y,Z,ROOT1,ROOT2)
```

The CALL statement transfers control to the SUBROUTINE subprogram and replaces the dummy variables with the value of the actual arguments that appear in the CALL statement. The arguments in a CALL statement may be any of the following: any type of constant, any type of subscripted or nonsubscripted variable, an arithmetic expression, or a subprogram name (except that they may not be Statement function names).

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

Subprograms Written in Assembler Language

Subprograms can be written in the 1130 Assembler language to be called by a FORTRAN program. In order to write such subprograms, the user must know the linkage generated by the FORTRAN Compiler and the location of the arguments.

The linkage to all three types of routines (SUBROUTINE Subprograms, FUNCTION Subprograms, Fortran supplied subprograms) is assembled and executed in the same way as the Assembler language CALL statement (see Program Linking Statements in the publication, IBM 1130 Assembler Language, Form C26-5927).

The arguments in the linkage are located as follows: At execution time, the Branch instruction corresponding to the CALL is followed in memory by a list of the addresses of the arguments.

Examples:

SUBROUTINE Subprogram CALL:

CALL JOE (A,B,C)

Result in memory at execution:

2 word CALL BSI L (Address of Entry Point of JOE)
ADDRESS OF A
ADDRESS OF B
ADDRESS OF C
First Word of Next Instruction.

Subprogram should return here.

When a SUBROUTINE subprogram CALL is used, results of the computations within the subprogram will be returned by means of the arguments. The Assembler coded SUBROUTINE subprogram must return control to the calling program at the next location following the last argument in the list.

FUNCTION Subprogram call or Fortran supplied subprogram call:

X = Y + JOE(A, B, C)

The underlined section of the above statement produces the same result in core storage as the SUBROUTINE subprogram example. It must be noted, however, that the Assembler coded FUNCTION or Fortran supplied subprogram must return a single result to the calling program by means of the pseudo floating accumulator or the machine accumulator, depending on whether the FUNCTION type is real or integer. (The floating accumulator is the last three words of the Transfer Vector area. See Program Linking Statements in the publication, IBM 1130 Assembler Language, Form C26-5927.) The argument list must not be used to return a result of the subprogram computation.

Machine Indicator Tests

The Fortran language provides machine indicator tests even though some of the machine components referred to by the tests do not physically exist. The machine indicators that do not exist are simulated by SUBROUTINE subprograms located in the system library.

To use any of the following machine indicator tests, the user supplies the proper arguments and writes a CALL statement. In the following listing, i is an integer expression; j is an integer variable.

General Form	Function
SLITE (i)	If i=0, all sense lights are turned off. If i=1, 2, 3, or 4, the corresponding sense light is turned on.
SLITET (i, j)	Sense light i (equal to 1, 2, 3, or 4) is tested. If i is on, j is set to 1; if i is off, j is set to 2. After the test, sense light i is turned off.
OVERFL (j)	This indicator is on if an arithmetic operation with real variables and constants results in an overflow or underflow condition; that is, j is set to 1 if the result of an arithmetic operation is greater than 2^{127} (10^{38}); j is set to 2 if no overflow condition exists; j is set to 3 if the result of an arithmetic operation is not zero but less than 2^{-129} (10^{-39}). The machine is left in a no overflow condition.
DVCHK (j)	This indicator is set on if an arithmetic operation with real constants and variables results in the attempt to divide by zero. If the indicator is on, j is set to 1; if off, j is set to 2. The indicator is set off after the test is made.
DATSW (i, j)	Data entry switch i is tested. If i is on, j is set to 1; if i is off, j is set to 2.
TSTOP	The TSTOP subroutine may be used to stop the tracing mode if trace control has been specified at compile time.
TSTRT	The TSTRT subroutine may be used to re-establish the trace mode if trace control has been specified at compile time.
FCTST (i, j)	The FCTST subroutine checks an indicator word that is set on if any Fortran supplied subprogram detects an error. If the indicator is off, i is set to 2 and j is set to 0; if the indicator is on, i is set to 1 and j is set equal to the contents of the indicator. The indicator is set to 0 after the test. Refer to IBM 1130 Subroutine Library (Form C26-5929) for descriptions of errors detected by Fortran supplied subroutines and the contents of resulting indicator words.

NOTE: SLITET and OVERFL contain six characters in order to be compatible with other IBM Fortrans; SLITET and OVERFL are changed by the Fortran Compiler to SLITT and OVERF, respectively.

Examples:

```
CALL SLITE (3)
CALL SLITET (K*J, L)
CALL OVERFL (J)
CALL DVCHK (I)
```

```
CALL DATSW (15, N)
CALL TSTOP
CALL TSTRT
CALL FCTST (IM, JM)
```

As an example of how the sense lights can be used in a program, assume that it is desired to continue with the program if sense light 3 is on and to write results if sense light 3 is off. This can be accomplished by using the IF statement or a Computed GO TO statement, as follows:

```
      CALL SLITE (3)
        .
        .
      CALL SLITET (3, KEN)
 5    IF (KEN-2) 10, 9, 10
 9    WRITE (3, 36)(ANS(K), K=1, 10)
10      .
        .
        .
      CALL SLITET (3, KEN)
24    GO TO (26, 25), KEN
25    WRITE (3, 36)(ANS(K), K=1, 10)
26      .
        .
```

In statement 5, if KEN is not equal to 2, statement 9 is not executed. In statement 24, if KEN equals 2, statement 25 is executed.

The following Fortran statements and features apply only to the IBM 1130 Monitor System. These statements and features are not valid for use with the card/paper tape Fortran Compiler.

CALL EXIT Statement

This statement is used in a Fortran program when control is to be returned to the Supervisor portion of the Monitor System; that is, the CALL EXIT statement must be used as the last logical statement of a Fortran program. (The END statement must still be used as the last physical statement of each program or subprogram.)

DEFINE FILE Statement

The DEFINE FILE statement specifies to the Fortran Compiler the size and quantity of disk data records within files that will be used with a particular program and its associated subprograms. This statement must not appear in a subprogram, and it may appear only in a main program. Therefore, all subprograms used by the main program must use the defined files of the main program.

The purpose of the DEFINE FILE statement is to divide the disk unit into files to be used in the disk READ, WRITE, and FIND statements.

General Form:

DEFINE FILE a_1 (m_1, l_1, U, v_1),

a_2 (m_2, l_2, U, v_2),...

where:

a_i is an integer constant $\leq$ 32,767 that is the symbolic designation for this file.

m_i is an integer constant that defines the number of file records in this symbolic file.

l_i is an integer constant that defines the length (in words) of each file record in this symbolic file. The value of l_i must be less than or equal to 320.

U is a fixed letter used to designate that the file must be read/written with the disk READ/WRITE statements which will handle no data conversion.

v_i is a nonsubscripted integer variable name which is set at the conclusion of each disk READ/WRITE statement referencing this symbolic file. It is set to the value of the next available file record. This variable must also appear in COMMON if it is to be referenced by more than one program at execution time.

NOTE: Since records which require no data conversion are transmitted, care must be exercised to ensure that the programs using this file have the same precision. A disk READ/WRITE statement always starts transmittal at the beginning of a file record.

Disk READ, WRITE, and FIND Statements

The generalized READ and WRITE statements and the FIND statement for disk I/O appear as:

```
READ (a'b) List
WRITE (a'b) List
FIND (a'b)
```

where:

a (an unsigned integer constant or integer variable) is the symbolic file number,

b (an integer expression) is the record number where transmittal will start, and

List is a list of variable names, separated by commas, for the input or output data.

NOTE: Only information which requires no data conversion can be transmitted to and from disk storage.

The purpose of the FIND statement is to move the disk read/write mechanism to the a'b record. The use of the FIND statement is optional.

The user should be aware that the effect of the FIND statement will usually be nullified if any disk operations occur before the associated READ or WRITE statement (such disk operations occur as a result of a CALL LINK, calls to LOCAL subprograms, etc.). Therefore in certain cases there may be no advantage to a FIND statement preceding a READ or WRITE statement.

CALL LINK Statement

The CALL LINK statement calls a new main program from disk storage into core storage and transfers control to the first executable statement in that program.

General Form:

CALL LINK (NAME)

where:

NAME is the name of a Fortran main program as contained in the Location Equivalence Table (LET). The program name consists of 1-5 alphameric characters of which the first must be alphabetic.

Examples:

```
CALL LINK (JOE)
CALL LINK (PROG3)
```

The program that is called causes all subprograms and library subroutines that it references to read into core storage. Any program called by this statement must already be in disk storage. If the logic of the program allows any one of several Links to be called, it is necessary that all of the Link programs be on disk storage prior to execution or loading initialization.

The COMMON area is not destroyed during the loading of the Link programs. If the size of COMMON differs between programs, the COMMON area size that remains undestroyed is determined by the newest Link program.

APPENDIX A: TABLE OF SOURCE PROGRAM CARD CHARACTER CODES

Character	Card Punches	Character	Card Punches
0	0	Q	11-8
1	1	R	11-9
2	2	S	0-2
3	3	T	0-3
4	4	U	0-4
5	5	V	0-5
6	6	W	0-6
7	7	X	0-7
8	8	Y	0-8
9	9	Z	0-9
A	12-1	.	12-8-3
B	12-2	<	12-8-4
C	12-3	(	12-8-5
D	12-4	+	12-8-6
E	12-5	&	12
F	12-6	$	11-8-3
G	12-7	*	11-8-4
H	12-8	)	11-8-5
I	12-9	-	11
J	11-1	/	0-1
K	11-2	,	0-8-3
L	11-3	%	0-8-4
M	11-4	#	8-3
N	11-5	@	8-4
O	11-6	'	8-5
P	11-7	=	8-6

NOTES:

1. At compilation time, the following character punches are treated as being equal; however, the characters to the left of the "and" must be used if compatibility with IBM System/360 Fortran is desired.

   ' and @) and <
   + and & (and %
   = and #

2. Only the 52 characters shown above can be handled at execution time through A- or H-type formatting in the Fortran Input/Output routines. Any other character is replaced with a blank.

3. No transformations, such as & converted to +, etc., are made through A- or H-conversion; however, the & is converted to + when read with I-, E-, or F-conversion.

APPENDIX B: NONSTANDARD ITEMS

The items listed below, which are a part of the language described in this publication, are not contained in the standards defined by the ASA committee, X 3.4.3 - Fortran.

Expression of the form A**B**C
Machine Indicator Tests
Mixed mode expressions
Monitor Disk Statements
Automatic input/output conversion
(e.g. I type to a real variable)

APPENDIX C: SOURCE PROGRAM STATEMENTS AND SEQUENCING

Every executable statement in a source program (except the first) must have some programmed path of control leading to it. Control originates at the first executable statement in the program and is passed as follows:

Statement	Normal Sequence
a = b	Next executable statement
CALL	First executable statement of called subprogram
COMMON	Nonexecutable
CONTINUE	Next executable statement
DEFINE FILE	Nonexecutable
DIMENSION	Nonexecutable
DO	DO sequencing, then the next executable statement
EQUIVALENCE	Nonexecutable
EXTERNAL	Nonexecutable
FORMAT	Nonexecutable
FUNCTION	Nonexecutable
GO TO n	Statement n
GO TO (n_1, $n_2, \ldots n_m$), i	Statement n_i
IF(a)S_1, S_2, S_3	Statement S_1 if arithmetic a < 0 Statement S_2 if arithmetic a = 0 Statement S_3 if arithmetic a > 0
INTEGER	Nonexecutable
PAUSE	Next executable statement
READ	Next executable statement
REAL	Nonexecutable
RETURN	The first statement, or part of a statement, following the reference to this program.
STOP	Terminate execution
SUBROUTINE	Nonexecutable
WRITE	Next executable statement

The IBM 1130 Fortran compilers do not require reserved words; however, the words listed below are reserved by two IBM System/360 Fortran Compilers (Special Support and "E" Level). These words should be used in a Fortran source program only as specified in the publications describing those compilers if compatibility with those compilers is desired. Also, in that case, these words should not be used as names. (In addition, the same two compilers require significant blanks around certain identifiers, and no embedded blanks within the identifiers. This restriction should also be observed if compatibility is desired.) In other words, if these restrictions are observed, the source program could be compiled for either the 1130 or System/360.

ABS
AINT
ALOG
AMAX0
AMAX1
AMIN0
AMIN1
AMOD
ATAN

BACKSPACE

CALL
COMMON
CONTINUE
COS

DEFINE
DIM
DIMENSION
DFLOAT
DO
DOUBLE
DSIGN
DVCHK

END
EQ
EQUIVALENCE
EXP
EXTERNAL

FILE
FLOAT
FORMAT
FUNCTION

GE
GO
GT

IABS
IDIM
IF
IFIX
INT
INTEGER
ISIGN

LE
LT

MAX0
MAX1
MIN0
MIN1
MOD

NE

OVERFL

PAUSE

READ
REAL
RETURN
REWIND

SIGN
SIN
SLITE
SLITET
SQRT
STOP
SUBROUTINE

TANH
TO

WRITE

INDEX

A-conversion, 15
ABS subprogram (Table 1), 22
ALOG subprogram (Table 1), 22
Alphameric data conversion, 14
Arguments,
  dummy, 22, 23
  function, 23
Arithmetic,
  expressions, 5
  operation symbols, 5
  statements, 6
  Statement functions, 21
Arrays, 3, 4
  arrangement, 3
  dimensioning, 18
  element equivalence, 19
ATAN subprogram (Table 1), 22

Blanks, 1
  Blank fields, 16
  Blank lines, 16

CALL DATSW statement, 26
CALL DVCHK statement, 26
CALL EXIT statement, 28
CALL FCTST statement, 26
CALL LINK statement, 29
CALL OVERFL statement, 26
CALL SLITE statement, 26
CALL SLITET statement, 26
CALL statement, 25
CALL TSTOP statement, 26
CALL TSTRT statement, 26
Card punches for source program, 30
Carriage control, 17
Comments, 1
COMMON statement, 18, 29
Computed GO TO statement, 6
Constants, 1
  Integer, 2
  Real, 2
Continuation line, 1
CONTINUE statement, 9
Control statements, 6
  GO TO, 6, 7
  IF, 7
  DO, 7
  CONTINUE, 9
  END, 9
  PAUSE, 9
  STOP, 9
Conversion of alphameric data, 14
Conversion of numeric data, 13
Conversion to Single Subscripts, 20

Data conversion,
  alphameric, 14
  numeric, 13
Data input to object program, 17
DATSW subprogram, 26
DEFINE FILE statement, 28
DIMENSION statement, 18
Disk READ, WRITE, and FIND statements, 28
DO statement, 7
DVCHK subprogram, 26

E-conversion, 14
END statement, 9, 24
EQUIVALENCE statement, 19
EXTERNAL statement, 17
EXP subprogram (Table 1), 22
Explicit specification, 2
  (see Type Statements), 17
Expressions, 4
  rules for construction, 5, 6

F-conversion, 13
FCTST subprogram, 26
Feature requirements (of compiling machine), ii
FIND statement, 28
FLOAT subprogram (Table 1), 22
FORMAT statement, 12
Fortran Supplied Subprograms, 22
FUNCTION subprogram, 22
Functions,
  definition, 21
  Statement functions, 21

General I/O statements, 10
GO TO statement,
  computed, 6
  unconditional, 6

H-conversion, 14

I-conversion, 13
IABS subprogram (Table 1), 22
IF statement, 7
IFIX subprogram (Table 1), 22
Implicit specification, 2
Increment of a DO statement, 8
Index of a DO statement, 8
Indexing I/O lists, 10
Initial value of a DO statement, 8
Input data, 17
  conversion, 13, 14
Input/Output statements,
  disk, 28
  general, 10
Integer constants, 2
INTEGER statement (type), 17
ISIGN subprogram (Table 1), 22

List, 10
Literal data, 15
Location Equivalence Table (LET), 29

Machine configuration and feature
requirements, ii
Machine indicator tests, 26
Monitor statements, 28
Multiple field format, 16

Nesting of a DO statement, 8, 9
Nonstandard Items (Appendix B), 31
Numeric data conversion, 13

Object program input, 17
Operation symbols, 5
Order of arithmetic operations, 5
Order of specification statements, 17
OVERFL subprogram, 26

Parentheses, 5
PAUSE statement, 9
Printer Carriage Control, 17

Range of a DO statement, 8
READ statement, 10
(see also Disk Read and Write statements), 28
Reading alphameric data, 14
Real Constants, 2
REAL statement (Type), 17
Reserved words (IBM System/360), 33
Restrictions of a DO statement, 8
RETURN statement, 23

Sequence of source statements, 32
Sequence of specification statements, 17
Simulated machine indicators, 26
SIN subprogram (Table 1), 22
SIGN subprogram (Table 1), 22
SLITE subprogram, 26
SLITET subprogram, 26
Source program card code characters (Appendix A), 30
Specification statements, 17
COMMON, 18
DIMENSION, 18
EQUIVALENCE, 19
Type (REAL, INTEGER), 17
SQRT subprogram (Table 1), 22
Statement,
comments, 1
continuation, 1
format, 1
numbers, 1
Function, 21
Statements,
arithmetic, 6
control, 6
format, 12
input/output, 9
specification, 17
subprogram, 20
STOP statement, 9
Subprograms,
functions, 21
definition, 21
END statement, 9, 24
FUNCTION, 22
naming, 21
RETURN statement, 23
SUBROUTINE, 24
writing in assembler language, 25
SUBROUTINE subprogram, 24
Subscripted variables, 4
Subscripts, 3, 4
System/360 Reserved Words, 33

TANH subprogram (Table 1), 22
Test value of a DO statement, 8
TSTOP subprogram, 26
TSTRT subprogram, 26
Type statements (REAL, INTEGER), 17

Variables, 2
explicit specification, 2, 17
implicit specification, 2
names, 2
rules for naming, 3
types, 2

WRITE statement, 11
(see also Disk Read and Write statements), 28

X-conversion (blank fields), 16

C26-5933-2

IBM 1130 Printed in U.S.A. C26-5933-2

IBM®

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

READER'S COMMENT FORM

IBM 1130 FORTRAN Language — Form C26-5933-2

- Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. All comments will be handled on a non-confidential basis.

	Yes	No
• Does this publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

- What is your occupation? ______________________________
- How do you use this publication?

As an introduction to the subject?	☐	As an instructor in a class?	☐
For advanced knowledge of the subject?	☐	As a student in a class?	☐
For information about operating procedures?	☐	As a reference manual?	☐

Other ______________________________

- Please give specific page and line references with your comments when appropriate. If you wish a reply, be sure to include your name and address.

COMMENTS

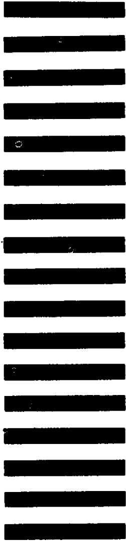
- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

C26-5933-2

FOLD

FOLD

FIRST CLASS
PERMIT NO. 2078
SAN JOSE, CALIF.



BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

POSTAGE WILL BE PAID BY

IBM CORPORATION
MONTEREY & COTTLE RDS.
SAN JOSE, CALIFORNIA
95114

Attention: Programming Publications, Dept. 234

FOLD

FOLD

IBM 1130 Printed in U.S.A. C26-5933-2

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