File No. 1130/1800-25
Form C26-3715-3

IBM Systems Reference Library

IBM 1130/1800 Basic FORTRAN IV Language

This publication presents the specifications and programming rules for the Basic FORTRAN IV Language used under the following programming systems:

IBM 1130 Card/Paper Tape Programming System

IBM 1130 Disk Monitor System

IBM 1130 Disk Monitor System, Version 2

IBM 1800 Card/Paper Tape Programming System

IBM 1800 Time-Sharing Executive System

IBM 1800 Multiprogramming Executive System

Appendix A of this publication lists the FORTRAN statements described and specifies to which of the above programming systems they apply.

This publication should not be used as a FORTRAN primer. For general information about FORTRAN, refer to IBM FORTRAN II General Information Manual (Form F28-8074).

Fourth Edition

This publication (Form C-26-3715-3) is a revision of the previous edition (Form C26-3715-2), which is now obsolete. This edition updates the publication to include the IBM 1800 Multiprogramming Executive System, and makes certain other corrections and clarifications to the manual.

Specifications contained herein are subject to change from time to time. Any such change will be reported in subsequent revisions or Technical Newsletters.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to IBM Nordic Laboratory, Technical Communications Department, Vesslevägen 3, Lidingö, Sweden.

This publication supports the following programming systems:

IBM 1130 Card/Paper Tape Programming System

IBM 1130 Disk Monitor System

IBM 1130 Disk Monitor System, Version 2

IBM 1800 Card/Paper Tape Programming System

IBM 1800 Time-Sharing Executive System (TSX)

IBM 1800 Multiprogramming Executive Operating System (MPX)

Each of these programming systems includes a FORTRAN Compiler that converts a source program consisting of statements written in the 1130/1800 Basic FORTRAN IV Language into an object program executable under that same programming system. This publication provides the specifications and programming rules for the writing of source program statements in the 1130/1800 Basic FORTRAN IV Language.

COREQUISITE PUBLICATIONS

Assembler Language publications:

IBM 1130 Assembler Language (Form C26-5927) supports the IBM 1130 Card/Paper Tape Programming System, Version 1 of the IBM 1130 Disk Monitor System, and the IBM 1130 Disk Monitor System, Version 2.

IBM 1800 Assembler Language (Form C26-5882) supports the IBM 1800 Card/Paper Tape Programming System and the IBM 1800 Time-Sharing Executive System.

Subroutine Library publications:

IBM 1130 Subroutine Library (Form C26-5929) supports the IBM 1130 Card/Paper Tape Programming System, Version 1 of the IBM 1130 Disk Monitor System, and the IBM 1130 Disk Monitor System, Version 2.

IBM 1800 Subroutine Library (Form C26-5880) supports the IBM 1800 Card/Paper Tape Programming System.

IBM 1800 Time-Sharing Executive System Subroutine Library (Form C26-3723) supports the IBM 1800 Time-Sharing Executive System.

IBM 1800 Multiprogramming Executive Operating System: Subroutine Library (Form C26-3754) supports the IBM 1800 Multiprogramming Executive System.

Operating procedures publications:

IBM 1130 Card/Paper Tape Programming System Operator's Guide (Form C26-3629)

IBM 1130 Disk Monitor System (Version 1) Reference Manual (Form C26-3750)

IBM 1130 Disk Monitor System, Version 2, Programming and Operator's Guide (Form C26-C26-3717)

IBM 1800 Card/Paper Tape Programming System Operator's Guide (Form C26-3751)

IBM 1800 Time-Sharing Executive System Operating Procedures (Form C26-3754)

IBM 1800 Multiprogramming Executive Operating System Programmer's Guide (Form C26-3720)

Additional publications:

IBM 1800 Time-Sharing Executive System Concepts and Techniques (C26-3703)

MACHINE CONFIGURATION AND FEATURE REQUIREMENTS

The minimum machine configuration and feature requirements needed to compile source programs written in the Basic FORTRAN IV Language are specified below.

Under the IBM 1130 Card/Paper Tape Programming System:

IBM 1131 Central Processor Unit, Model 1, with a minimum of 4096 words of core storage

IBM 1442 Card Read Punch, Model 6 or 7
or
IBM 1134 Paper Tape Reader in combination with IBM 1055 Paper Tape Punch

Under the IBM 1130 Disk Monitor System, Version 1:

IBM 1131 Central Processor Unit, Model 2, with a minimum of 4096 words of core storage

IBM 1442 Card Read Punch, Model 6 or 7
or
IBM 1134 Paper Tape Reader in combination with IBM 1055 Paper Tape Punch

Under the IBM 1130 Disk Monitor System, Version 2:

IBM 1131 Central Processor Unit, Model 2, with a minimum of 4096 words of core storage

IBM 1442 Card Read Punch, Model 6 or 7
or
IBM 1134 Paper Tape Reader in combination with
IBM 1055 Paper Tape Punch
or
IBM 2501 Card Reader in combination with
IBM 1442 Card Punch, Model 5

Under the IBM 1800 Card/Paper Tape Programming System:

IBM 1801 or 1802 Processor-Controller with a minimum of 4096 words of core storage

IBM 1442 Card Read Punch, Model 6 or 7
or
IBM 1054 Paper Tape Reader in combination with
IBM 1055 Paper Tape Punch
IBM 1053 Printer
or
IBM 1443 Printer
or
IBM 1816 Printer-Keyboard

Under the IBM 1800 Time-Sharing Executive (TSX) System:

IBM 1801 or 1802 Processor-Controller with a minimum of 8192 words of core storage

IBM 2310 Disk Storage

IBM 1442 Card Read Punch, Model 6 or 7

IBM 1053 Printer
or
IBM 1443 Printer
or
IBM 1816 Printer-Keyboard

Under the IBM 1800 Multiprogramming Executive (MPX) System:

IBM 1801 or 1802 Processor-Controller with a minimum of 24596 words of core storage.

IBM 2310 Disk Storage, Model A2 (standard - two disk drives) or C2 (fast access - two disk drives)

IBM 1442 Card Read Punch, Model 6 or 7

IBM 1053 Printer
or
IBM 1816/1053 Printer-Keyboard

CONTENTS

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

The FORTRAN language is a set of statements, composed of expressions and operators, which are used in writing the source program. The IBM 1130 and 1800 Programming Systems provide a FORTRAN Compiler, a program that translates the source program statements into a form suitable for execution under the respective programming 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 or printer. At the user's option, the compiler also produces a listing of the source program and storage allocations.

The basic elements of the FORTRAN language are: constants, variables, arrays and subscripts, expressions, and statements.

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. However, statement numbers may not be zero. Statement numbers may appear anywhere in the statement number field but must not contain any non-numeric 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. Statement numbers on specification statements are ignored.

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. Likewise, blank records in a source program are ignored by the FORTRAN Compiler.

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.

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 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. An integer constant may have any value in the range -32768 (-2^{15}) to 32767 ($2^{15}-1$), including zero.

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
```

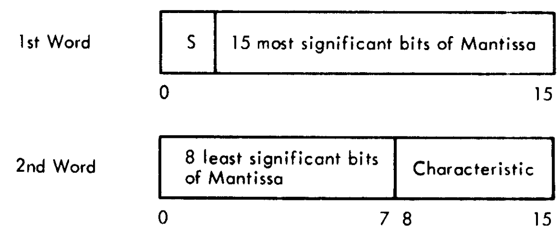
The following are not valid integer constants:

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

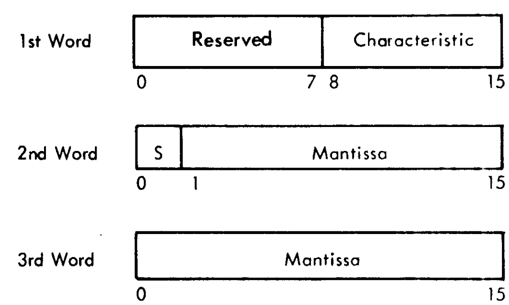
REAL CONSTANTS

A real constant is a number written with a decimal point and consisting of 1-6 or 1-9 significant decimal digits.

Standard precision provides up to 23 significant bits of precision (6 plus significant digits) stored in core storage as shown below:



Extended precision provides up to 31 significant bits of precision (9 plus significant digits) stored in core storage as shown below:



NOTE: Normalization can in some cases cause the loss of one bit of significance.

(The precision is specified to the compiler by optional use of an *EXTENDED PRECISION control record. See the section describing FORTRAN control records in the appropriate corequisite publication, as listed in the Preface.)

The magnitude of a real constant must not be greater than 2^{127} or less than 2^{-128} (approximately 10^{38} and 10^{-39}). It may be 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)
5E02	(no decimal point)

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, excluding special characters, the first of which must be alphabetic. (See Appendix C.)

Examples:

```
M
DEV86
I2
```

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
```

but it would be more meaningful to write:

```
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 represents an integer or real value and by using succeeding characters as an aid to the user's memory.

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.

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)
```

ARRAYS AND SUBSCRIPTS

An array is an ordered set of data that is referred to by a single name. Each individual element in the set is referred to in terms of its position in the set. For example, assume that the following is an array named NEXT:

15
12
18
42
19

To refer to the second element in the group in ordinary mathematical notation, the form $NEXT_2$ would be used. In FORTRAN the form 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 $NEXT_i$ 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

To refer to the number in row 2, column 3, $LIST_{2,3}$ would be used in ordinary mathematical notation. In FORTRAN, the form LIST(2,3) would be used 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 notation uses $LIST_{i,j}$ to represent any element of the two-dimensional 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 reference to the array in one of these statements must specify its size (see Specification Statements).

ARRANGEMENT OF ARRAYS IN STORAGE

Arrays are stored by column 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 0508 appears in storage as follows:

Address	Element
0500	J(5)
0502	J(4)
0504	J(3)
0506	J(2)
0508	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 0228 (assuming each element occupies two words).

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.

The value of a subscript (including the added or subtracted constant, if any) must be greater than zero and not greater than the corresponding array dimension. Each subscripted variable must have the size of its array (i.e., the maximum values that its subscripts can attain) specified in a DIMENSION, COMMON, or Type Statement.

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)

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 subscripted expression is determined solely by the mode of the variable.

An arithmetic expression is 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. The minus symbol (-) is also used to denote unary minus.

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. However, in a mixed expression, parts of the expression involving purely integer operations are computed in the integer mode. Then these integer results are converted to real values and the entire expression is computed in the real mode.

For example, in the expression:

```
A + (I * J) + (A / J) + I**2
```

I*J and I**2 are computed in the integer mode and these results are then converted to real values. However, the J in A/J will be converted to real before A/J is computed.

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. Unary minus
4. Multiplication and Division (left to right)
5. 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).

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 provide linkage to and from subprograms.

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 STATEMENT

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 a non-subscripted 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. Under the 1800 TSX and MPX Systems an execution error results and the program is aborted.

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
30
```

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

```
        .
        .
        .
        DO 25 I = 1, 1000, 1
25      STOCK(I) = STOCK(I) - OUT(I)
```

General Form:

DO n i = m_1, m_2

or

DO n i = m_1, m_2, m_3

where:

n is a statement number.
i is a nonsubscripted integer variable.
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 limit (n) defines the range of the DO. The range 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. 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 or negative.

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 limit. 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.

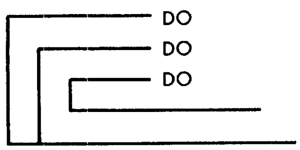
In some cases, the DO is completed before the test value is reached. Consider the following:

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

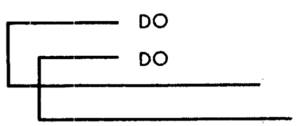
In this example, the range is executed three times (i.e., K equal to 1, 4, and 7). The next value of K would be 10. Since this exceeds the test value, the DO is completed after three iterations.

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

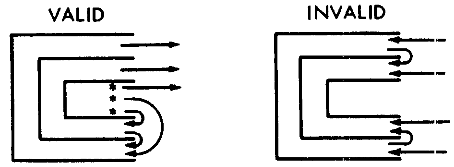
1. Within the range of a DO may be other DOs. When this is so, all statements in the range of the inner DO must be in the range of the outer DO. A set of DOs satisfying this rule is called a nest of DOs. The maximum depth of a single nest of DOs is 25. For example, the following configuration is permitted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



but, the following configuration is not permitted:



2. A transfer out of the range of any DO loop is permissible at any time. A transfer into a DO range is permissible only as described in item 3.
3. When a transfer is made out of the range of the innermost DO loop, a transfer back into the range of that loop is allowed if, and only if, neither the index nor any of the indexing parameters (i.e., m_1, m_2, m_3) are changed outside the range of the DO loop. This transfer back into a DO loop is permitted only to the innermost DO loop. 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.



*Return (opposite of arrow direction) is also permitted if no indexing parameters are changed.

4. The last statement in the range of a DO loop must not be a GO TO, IF, STOP, PAUSE, FORMAT, RETURN, or another DO statement.
5. 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

The CONTINUE statement is a dummy statement that does not produce any executable instructions. It can be inserted anywhere into a program; it simply indicates that the normal execution sequence continues with the statement following.

General Form:

```
CONTINUE
```

The CONTINUE statement is principally used as the range limit of DO loops in which the last statement would otherwise be a GO TO, IF, PAUSE, STOP, or RETURN statement. It also serves as a transfer point for IF and GO TO statements within the DO loop that are intended to begin another repetition of the loop. An example of these two functions follows:

```
        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 decimal integer constant whose value is equal to or less than 9999.

The PAUSE statement causes the program to stop on a Wait instruction. To resume execution the START key must be pressed. Execution starts with the next executable statement following the PAUSE statement. If n is specified, it is treated as a hexadecimal number and displayed on the console in the accumulator (A-register in the IBM 1800) lights.

STOP STATEMENT

General Form:

STOP
or
STOP n

where:

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

The STOP statement terminates program execution. If n is specified it is treated as a hexadecimal number and displayed on the console in the accumulator (A-register in the IBM 1800) lights.

In FORTRAN under the IBM 1130 Disk Monitor Systems, the IBM 1800 TSX and MPX Systems, the STOP statement is equivalent to a PAUSE statement followed by a CALL EXIT statement. Under the IBM 1800 TSX and MPX Systems, the STOP statement is valid only in nonprocess programs.

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 statements following the END statement will not be compiled.

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 values 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, any other kind of 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, and type with the corresponding arguments in the SUBROUTINE subprogram.

Note that the constants should not be used as parameters in a CALL statement if the subroutine is returning a value through that parameter. For example:

```
    Calling            SUBROUTINE
    Program            Subprogram

     .                 SUBROUTINE JOE (A, B)
     .                 A = B + 10
    CALL JOE (5,6)     RETURN
     .                 END
     .
100 C = 5
     .
     .
```

In this case the constant 5 in the calling program is replaced by the value of A as computed in the subroutine (A = B + 10). Subsequent execution of statement 100 in the calling program results in the variable C being assigned a value other than 5.

For descriptions of the SUBROUTINE subprograms that can be called in FORTRAN under the IBM 1130 and 1800 Programming Systems, see the appropriate Subroutine Library publication as listed in the Preface, above.

Special CALLs

CALL EXIT Statement

In FORTRAN under the IBM 1130 Disk Monitor Systems, the CALL EXIT statement is used when control is to be returned to the Supervisor portion of the system.

In FORTRAN under the IBM 1800 TSX System, the CALL EXIT statement is used when control is to be returned to the Supervisor portion of the Nonprocess Monitor. The CALL EXIT statement is therefore valid only in nonprocess programs.

For use of CALL EXIT in the MPX System see IBM 1800 Multiprogramming Executive Operating System Programmer's Guide. The CALL EXIT statement is not valid in FORTRAN under the IBM 1130 and 1800 Card/Paper Tape Programming Systems.

CALL LINK Statement

The CALL LINK statement is used when control is to be transferred from one program (link) to the next.

General Form:

CALL LINK (Name)

where:

Name is the name of the program to be loaded into core storage and given control. The program name consists of 1-5 alphameric characters (excluding special characters) the first of which must be alphabetic.

The link program that is called is loaded with all subprograms and library subroutines that it references. Any link 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 in disk storage prior to execution.

NOTE: Link programs called under the IBM 1800 TSX and MPX Systems must be in disk storage in core image format.

The COMMON area of the program relinquishing control is not destroyed during the loading of the link program. If the size of COMMON differs between programs, the COMMON area size that remains undisturbed is determined by the link program called.

In FORTRAN under the IBM 1800 Time-Sharing Executive System, the CALL LINK statement is valid only in nonprocess programs. Also, the name specified in the CALL LINK statement may be the name of a nonprocess program only.

The CALL LINK statement is not valid in the IBM 1130 and 1800 Card/Paper Tape Programming Systems. For use of CALL LINK in the 1800 MPX System see IBM 1800 Multiprogramming Executive Operating System Programmer's Guide.

CALL LOAD Statement

The CALL LOAD statement, which is valid only in FORTRAN for the card forms of the IBM 1130 and 1800 Card/Paper Tape Programming Systems, is used to link to another program without requiring the core image loader to precede the link program. CALL LOAD causes the next program in the card reader to be read in and executed.

For example:

```
.
.
.
CALL LOAD
STOP
END
```

The CALL LOAD statement may only be used in a core image program and may only call a core image program. (See the description of the *SAVE LOADER control record in the appropriate corequisite publication, as listed in the Preface, above.)

CALL PDUMP Statement

In FORTRAN for the IBM 1130 Disk Monitor System, Version 2, the dump program PDUMP can be called to print the contents of all of or one or more parts of core storage.

General Form:

CALL PDUMP (a_1, b_1, f_1,..., a_n, b_n, f_n)

where:

a_i and b_i are variable data names, subscripted or non-subscripted, indicating the inclusive limits of a block of core storage to be dumped. Either a_i or b_i can indicate the upper or lower limit of the block to be dumped.

f_i is an integer constant indicating the format in which the associated block of core storage is to be dumped. The dump formats are specified as follows:

Format	Value of f_i
Hexadecimal	0
Integer	4
Real	5

Machine and Program Indicator Tests

The FORTRAN language provides machine and program 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 subroutines provided in the system library.

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

General Form and Function:

CALL 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.

CALL 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.

CALL OVERFL (j) This indicator is on if an arithmetic operation with real variables and/or constants results in an overflow or underflow condition; that is, j is set to 1 if the absolute value of 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.

CALL SSWTCH (i, j) Sense switch i is tested. If i is on, j is set to 1; if i is off, j is set to 2.

This CALL is valid only in FORTRAN under the IBM 1800 Card/Paper Tape Programming System, and the IBM 1800 TSX and MPX Systems.

CALL DVCHK (j) This indicator is set on if an arithmetic operation with real constants and/or 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.

CALL DATSW (i, j) Data entry switch i is tested. If data entry switch i is on, j is set to 1; if data entry switch i is off, j is set to 2.

CALL TSTOP The TSTOP subroutine may be used to stop the tracing mode if trace control has been specified to the compiler.

CALL TSTRT The TSTRT subroutine may be used to re-establish the trace mode if trace control has been specified to the compiler.

CALL FCTST (j, k) The FCTST subroutine checks an indicator word that is set on if a FORTRAN-supplied FUNCTION subprogram detects an error or an end-of-file condition is detected during an unformatted I/O operation. k is set to the value of the indicator word. If the indicator word is zero, j is set to 2; otherwise, j is set to 1. The indicator word is set to 0 after the test.

NOTE: SSWTCH, SLITET and OVERFL contain six characters in order to be compatible with other IBM FORTRANs; SSWTCH, SLITET, and OVERFL are changed by the FORTRAN compiler to SSWTC, SLITT, and OVERF, respectively.

Examples:

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

INPUT/OUTPUT STATEMENTS

The input/output (I/O) statements control the transmission of information between the computer and the I/O units. On the IBM 1130 Computing System these units are: 2310 Disk Storage; 1442 Card Read Punch, Models 6 and 7; 1442 Card Punch, Model 5; 2501 Card Reader; 1132 Printer; 1403 Printer; 1134 Paper Tape Reader; 1055 Paper Tape Punch; Console Printer; Keyboard; and 1627 Plotter. On the IBM 1800 Data Acquisition and Control System these units are: 2310 Disk Storage; 2401 and 2402 Magnetic Tape Units; 1442 Card Read Punch, Models 6 and 7; 1053 Printer; 1443 Printer; 1054 Paper Tape Reader; 1055 Paper Tape Punch; 1816 Printer Keyboard; and 1627 Plotter.

I/O statements are classified as follows:

1. Non-disk I/O Statements. These statements cause transmission of formatted information between the computer and I/O units other than the disk. They are READ and WRITE.
2. Disk I/O Statements. These statements cause transmission of information between the computer and the disk. They are READ, WRITE, and FIND.
3. Unformatted I/O Statements. These statements cause transmission of unformatted information as follows:
   a) under the IBM 1800 Card/Paper Tape Programming and TSX Systems: between the computer and magnetic tape units in FORTRAN;
   b) under the IBM 1800 MPX System: between the computer and magnetic tape units or disk storage units;
   c) under the IBM 1130 Disk Monitor System, Version 2: between the computer and a special disk area for the simulation of magnetic tape I/O in FORTRAN.
   These statements are READ and WRITE.
4. Manipulative I/O Statements. These statements manipulate magnetic tape units in FORTRAN under the IBM 1800 Card/Paper Tape Programming System, the IBM 1800 TSX and MPX Systems; they manipulate the unformatted I/O area on disk in FORTRAN under the IBM 1130 Disk Monitor System, Version 2. These statements are BACKSPACE, REWIND, and END FILE.
5. FORMAT Statements. These are nonexecutable 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 non-disk I/O statements.

NON-DISK 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
      or
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 (see Logical Unit Numbers).

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 external 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 statements appear in the source program:

```
READ (2,25) I,J,K,L,M
25 FORMAT(I2,2x,I3,53x,I4,5x,I2,8x,I1)
```

the card is read (assuming that 2 is the unit number associated with the card reader), and the program operates as though the following statements had been written:

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

After 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 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 in an input record 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 number of input records, succeeding 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 non-subscripted form, all of the quantities in the array are transmitted in column order (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) with an appropriate FORMAT statement.

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 Conversion of Alphameric Data). 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. Two forms of the WRITE statement may be used, as follows:

WRITE (a,b) List
or
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 (see Logical Unit Numbers).

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 data format (see FORMAT Statement).

NOTE 1: The 1442 Card Read Punch, Model 6 or 7, has one input hopper. Therefore, if a READ or WRITE statement references a 1442-6 or -7, 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.

NOTE 2: If the first I/O operation is a WRITE to the 1442 Card Read Punch, Model 6 or 7, and the 1442 contains cards that are not to be punched, one of the following two options may be used to avoid punching the cards remaining in the 1442:

Option 1. Stack no cards behind the last card to be read. This causes an interrupt to occur when the WRITE is encountered. The cards remaining in the 1442 can then be run out (NPRO) and blank cards placed in the hopper before the WRITE is executed.

Option 2. Place blank cards behind the last card to be read; in addition, include in the program, as the first I/O operation, a dummy READ such as

READ (a,b)

where a is the logical unit number of the 1442 and b is any FORMAT statement number. The dummy READ causes the last cards read to be fed through the 1442 and the first blank card to be positioned for punching. The WRITE to the 1442 can then be executed.

Option 2 is preferable since it allows uninterrupted execution and requires no operator intervention.

The WRITE (a,b) form is used to write alphameric data (see Conversion of Alphameric Data). 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)
```

DISK I/O 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 that the symbolic file number and record number (a and b) must be separated by an apostrophe.

An example is:

```
READ (IFILE'200) OUTX, OUTY, OUTZ
```

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

The READ (a'b) List form is used to read information from the disk. The List specifies the number of items to be read and the locations into which the items are to be placed. It functions the same as the List in the non-disk I/O READ/WRITE statements. For example, assume a file defined as:

```
DEFINE FILE 3 (400, 2, U, K)
```

contains the following information:

RECORD NUMBER	CONTENTS Word 1	Word 2
253	4800	0084
254	5000	0084
255	6800	0084

Then, if X, Y, and Z are two-word standard precision real variables,

```
READ (3'253) X, Y, Z
```

would result in the following values being read into X, Y, and Z:

X = 48000084
Y = 50000084
Z = 68000084

or, converting from binary to decimal:

X = 9.0
Y = 10.0
Z = 13.0

As is the case in the non-disk I/O statements, if there are more quantities in an input file than there are items in the list, only the number of items in the list are transmitted. Thus, in the above example, only records 253, 254, and 255 were transmitted; the rest were ignored. If a list contains more quantities than the input file, an error results.

Variables within an I/O list may be indexed and incremented in the same manner as with a DO statement. For example, if we have:

```
DIMENSION X(400)
DEFINE FILE 3(400, 2, U, K)
    .
    .
    .
READ (3'1) (X(I), I = 1,5)
    .
    .
    .
```

records 1 through 5 of file 3 will be read into the first 5 elements of the array X (see <u>Indexing I/O Lists</u>).

The WRITE (a'b) List form operates in the same way as the READ (a'b) List statement and is used to transmit data to the disk.

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

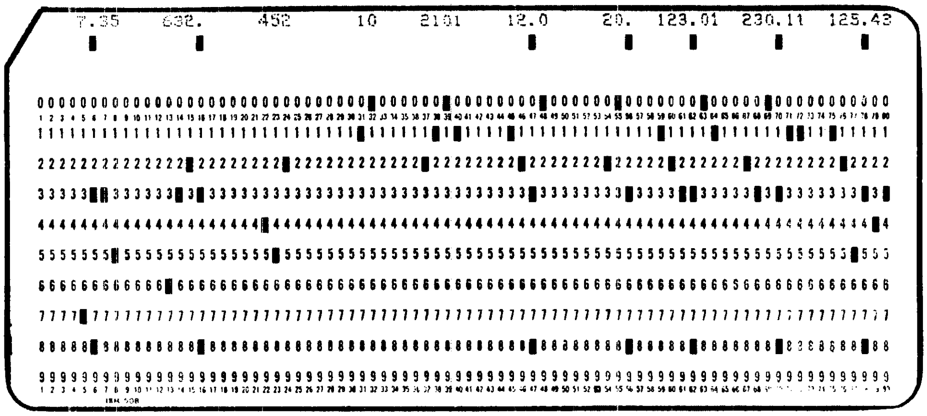
The user should be aware that disk operations such as calls to LOCAL subprograms on the same disk drive may move the access mechanism and nullify the effect of the FIND statement. Therefore in certain cases there may be no advantage to a FIND statement preceding a READ or WRITE statement.

There are several ways of using the disk facility in FORTRAN programs. There may be an already created file in the disk Fixed Area that the FORTRAN program may read data from or write data into, or the program may create a temporary file in the Working Storage area of the disk. The following example shows the use of the disk in both ways.

<u>Example.</u> Assuming (1) an already existing data file on the disk occupying one sector and having the symbolic file number 4 and (2), a card reader having the logical unit number 2, a FORTRAN program to read in, in standard precision, real values from 10 cards and write these values on the file may look as follows. Note however that the FORTRAN defined file must be associated with the existing data file on disk, in order that the first program shares a file with the second program.

```
      DIMENSION A(10)
      DEFINE FILE 4 (10,20,U,J)
      J=1
      DO 5 I=1,10
      READ (2,100)A
  100 FORMAT (10F8.0)
      WRITE (4'J)A
    5 CONTINUE
      CALL EXIT
      END
```

where a typical data card may be:



Assuming (1) the data file on disk written in the above program and (2) a printer having the logical unit number 3, a program that reads this file from disk and prints the results on the printer may be:

```
      DIMENSION A(10)
      DEFINE FILE 4(100,2,U,K)
      K=1
      DO 5 I=1,10
      READ (2'K)A
      WRITE (3,100)A
  100 FORMAT (6F20.8)
    5 CONTINUE
      CALL EXIT
      END
```

UNFORMATTED I/O STATEMENTS

The READ and WRITE statements for unformatted I/O, i.e., I/O without data conversion, appear as:

```
READ (a) List
READ (a)
WRITE (a) List
```

where:

a is an unsigned integer constant or integer variable that specifies a logical unit number to be used for I/O data (see Logical Unit Numbers). p22 .

List is a list of variable names, separated by commas, for the I/O data.

The READ (a) List form is used to read a core-image record, without data conversion, into core storage from unit a. No FORMAT statement is required; the amount of data that is read corresponds to the number of list items. The total length of the list of variable names must not be longer than the logical record length. If the length of the list is equal to the logical record length, the entire record is read. If the length of the list is shorter than the logical record length, the unread items in the record are skipped.

The READ (a) form is used to skip an unedited record on unit a.

The WRITE (a) List form is used to write a core-image record, without data conversion, on unit a.

For detailed information concerning the creation and use of the unformatted I/O area under the IBM 1130 Disk Monitor System, Version 2, see the corequisite publication for that system as listed in the Preface.

For the specific use of unformatted I/O for MPX, see IBM 1800 Multiprogramming Executive Operating System Programmer's Guide.

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)
```

MANIPULATIVE I/O STATEMENTS

The statements BACKSPACE, REWIND, and END FILE are used in FORTRAN under the IBM 1800 Card/Paper Tape Programming System and the IBM 1800 TSX System to manipulate magnetic tape units. In the IBM 1800 MPX System, manipulative I/O statements are used to manipulate both disk units and magnetic tape units in unformatted mode. In FORTRAN under the IBM 1130 Disk Monitor System, Version 2, these statements are used to manipulate the unformatted I/O area on disk.

BACKSPACE Statement

General Form:

BACKSPACE n

where:

n is an unsigned integer constant or integer variable specifying the logical unit number (see Logical Unit Numbers).

In FORTRAN under the IBM 1130 Disk Monitor System, Version 2, the BACKSPACE statement causes a pointer to the next available logical record in the unformatted I/O area to be decremented by one. The statement has no effect if this pointer indicates the first logical record in the area.

In FORTRAN under the IBM 1800 MPX System, the BACKSPACE statement causes the following actions to occur.

Unformatted Disk I/O. A backspace over one logical record is accomplished by decrementing a pointer in the device table. This pointer always points to the sector address of the next available logical record in process or batch process Working Storage. A BACKSPACE statement has no effect if the unformatted disk pointer is set at the beginning of Working Storage.

Magnetic Tape. Tape unit n is backspaced one logical record. If the tape unit is at load point, the BACKSPACE statement has no effect.

REWIND Statement

General Form:

REWIND n

where:

n is an unsigned integer constant or integer variable specifying the logical unit number (see Logical Unit Numbers).

In FORTRAN under the IBM 1130 Disk Monitor System, Version 2, the REWIND statement causes a pointer to the next available logical record in the unformatted I/O area to be reset to one. The statement has no effect if this pointer already indicates the first logical record in the area.

In FORTRAN under the IBM 1800 Card/Paper Tape Programming System and the IBM 1800 TSX System, the REWIND statement causes the tape on unit n to be rewound to its load point. The statement has no effect if the tape is positioned at its load point. The statement does not cause the tape on unit n to be unloaded.

In FORTRAN under the IBM 1800 MPX System the REWIND statement causes either a pointer to the next logical record to be set to one for unformatted disk, or the tape on unit n to be rewound to its load point for unformatted tape. If the logical record pointer in the unformatted disk area is already one or if the tape on unit n is already at its load point, the statement has no effect.

END FILE Statement

General Form:

END FILE n

where:

n is an unsigned integer constant or integer variable specifying the logical unit number (see Logical Unit Numbers).

In FORTRAN under the IBM 1130 Disk Monitor System, Version 2, the END FILE statement causes an end-of-file record to be written in the unformatted I/O area.

In FORTRAN under IBM 1800 Card/Paper Tape Programming System and the IBM 1800 TSX, the END FILE statement causes an end-of-file mark to be written on the tape on unit n.

In backspacing and in skipping forward over records, the end-of-file record or mark is equivalent to one logical record.

In FORTRAN under the IBM 1800 MPX System the END FILE statement causes either an end-of-file record to be written in the unformatted I/O area for unformatted disk operations or an end-of-file record to be written on the tape on tape unit n.

LOGICAL UNIT NUMBERS

The logical unit numbers used in FORTRAN I/O statements under the IBM 1130 Card/Paper Tape Programming System and the IBM 1130 Disk Monitor System are:

1 Console Printer
2 1442 Card Read Punch, Model 6 or 7
3 1132 Printer
4 1134 Paper Tape Reader/1055 Paper Tape Punch
6 Keyboard
7 1627 Plotter

The logical unit numbers used in FORTRAN I/O statements under the IBM 1130 Disk Monitor System, Version 2, are:

1 Console Printer
2 1442 Card Read Punch, Model 6 or 7
3 1132 Printer
4 1134 Paper Tape Reader/1055 Paper Tape Punch
5 1403 Printer
6 Keyboard
7 1627 Plotter
8 2501 Card Reader
9 1442 Card Punch, Model 5
10 Unformatted I/O area on disk

The logical unit numbers used in FORTRAN I/O statements under the IBM 1800 Card/Paper Tape Programming System are assigned by each installation during system edit.

The logical unit numbers used in FORTRAN I/O statements under the 1800 TSX and MPX Systems are assigned by each installation during system generation.

FORMAT STATEMENT

In order for data 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 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 an I/O list by the use of data conversion specifications (see Conversion of Numeric Data). FORMAT statements may appear any place within the source program after all Specification statements.

General Form:

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

where:

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

Examples:

```
 5  FORMAT (I5, F8.4)
18  FORMAT (I4/F6.2, F8.4)
20  FORMAT (E11.4/I8)
```

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

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 that 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 unit record size) 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 must correspond in mode with the list items in the I/O statement. Numeric data read into integer variables require an I-type format specification, and numeric data read into real variables require an F-type or an E-type specification. Alphameric data may be read into either integer or real variables by using the A-type format specification. This requirement holds for variables in both the READ and the WRITE statement list. For a more detailed description of I-, E-, F-, and A-type formats see Conversion of Numeric Data and Conversion of Alphameric Data.

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 Iw may be used to print a number in integer form; w print positions are reserved for the number. It is printed in this w-position field right-justified (that is, the units position is at the extreme right). If the number to be converted is greater than w-1 positions, an error condition will exist if the number is negative. A print position must be reserved for the sign if negative values are printed, but positive values do not require a position for the sign. If the number has less than w digits, the leftmost print positions are filled with blanks. If the quantity is negative, the position preceding the leftmost digit contains a minus sign.

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

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

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). For output, the total field length reserved must include sufficient positions for a sign, if any, a digit to the left of the decimal point, and a decimal point. The sign, if negative, is printed. In general w should be at least equal to d + 3 for output.

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 to the extent of the specified precision. 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	12.17
-41.16	*****
-.2	-0.20
7.3542	7.35†
-1.	-1.00
9.03	9.03
187.64	*****

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

NOTES:

1. All error fields are filled 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 with 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, a zero, and a decimal point. A zero value is printed with a zero preceding the decimal point.
4. F-conversion will accept input data in E-type format.

E-Conversion (Ew.d)

For E-conversion, the fractional portion is again indicated by d. For output, the w includes the field d, a space for a sign, space for a digit preceding the decimal point, a decimal point, and four spaces 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 all of these positions. In general, w should be at least equal to d+7.

The exponent is a signed or unsigned one- or two-digit integer constant not greater than 38 and preceded by the letter E. 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	0.238Eb03
-.002	*********
.00000000004	0.400E-10
-21.0057	*********

If the last example above had been printed with a specification of E10.3, it would appear as:

    -0.210Eb02†

NOTES:

1. All error fields are filled with asterisks.
2. 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.
3. For input, the exponent field may be omitted entirely (i.e., E-conversion will accept input data in F-type format).
4. 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.
5. A leading zero is always printed to the left of the decimal point.

Conversion of Alphameric Data

There are two specifications available for input/output of alphameric data: H-conversion (including literal data enclosed in apostrophes), 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 that is to be operated on by the program (e.g., modifying a line to be printed). The characters that can be handled are listed in Appendix C.

H-Conversion

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

    24H THIS IS ALPHAMERIC DATA

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

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 data characters to be read from the input file on the I/O unit associated with the logical unit number 4 (Paper Tape Reader on the 1130); 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 I/O unit associated with the logical unit number 1 (Console Printer on the 1130):

    CUST. NO. NAME

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 I/O unit associated with the logical unit number 4 (Paper Tape Reader on the 1130) and placed (left-justified) into the variable named ERROR.

The following statements:

```
      INTEGER OUT
15    FORMAT (4HbXY=, F9.3,A4)
            .
            .
            .
      WRITE (OUT, 15)A, ERROR, B, ERROR
```

may produce the following lines:

```
XY=b5976.214----
XY=b6173.928----
```

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 capacity of the field in storage into which they are to be read, the remaining rightmost characters in the field are loaded with blanks. However, if the number of characters is greater than the capacity 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 -- see Real Constants); each integer variable has space for 2 characters. For example, 10 characters could be read into, or written from, the first five variables of the array I if the following format is used:

```
101   FORMAT (5A2)
        .
        .
        .
      READ (IN,101) I
        .
        .
      WRITE(IOUT,101) I
```

Thus, two characters are contained in each of the five consecutive positions: I(1), I(2), I(3), I(4), I(5). On output the leftmost character is written first. Note that the format

```
101    FORMAT (A10)
```

would not work since 10 characters would be read from an array element of two words, causing the last 8 alphameric characters to be ignored.

Arithmetic operations involving variables containing alphameric characters should be performed in integer mode. Alphameric characters are represented internally in eight-bit EBCDIC (refer to the appropriate Subroutine Library publication, as listed in the Preface, above, for a description of the EBCDIC used for internal representation of alphameric characters).

Literal Data Enclosed in Apostrophes

Literal data can consist of a string of alphameric and special characters written within the FORMAT statement and enclosed in apostrophes. For example:

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

A comma following the last apostrophe is optional.

An apostrophe character within literal data is represented by two successive apostrophes. 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 specified between the apostrophes, are read from the designated I/O unit. These characters replace, in storage, the characters within the apostrophes. For example, the statements:

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

would cause the next 9 characters to be read from the I/O unit associated with the logical unit number 4 (Paper Tape Reader on the 1130). These characters would replace the blank and the 8 characters H,E,A,D,I,N,G, and S in storage.

Ouput. All characters (including blanks) within the apostrophes 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 I/O unit associated with the logical unit number 1 (Console Printer on the 1130):

```
THIS IS ALPHAMERIC DATA
```

X-Type Format

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.

T-Format Code

Input and output may begin at any position by using the format code Tw where w is an unsigned integer constant specifying the position in a FORTRAN record where the transfer of data is to begin. Only when the output is printed on an 1132, 1403, or 1443 Printer does the correspondence between w and the actual print position differ. In this case, because of the carriage control character, the print position corresponds to w-1, as may be seen in the following example:

```
5 FORMAT (T40, '1964 INVENTORY REPORT'
     T80, 'DECEMBER' T2, 'PART NO. 10095')
```

The preceding FORMAT statement would result in a printed line as follows:

Print Position 1	Print Position 39	Print Position 79
↓	↓	↓
PART NO. 10095	1964 INVENTORY REPORT	DECEMBER

The following statements:

```
5     FORMAT (T40, 'bHEADINGS')
        .
        .
        .
      READ (2,5) or READ (I,5)
```

would cause the first 39 characters of the input data to be skipped, and the next 9 characters would then replace the blank and the characters H, E, A, D, I, N, G and S in storage.

The T-format code may be used in a FORMAT statement with any type of format code. For example, the following statement is valid:

```
5     FORMAT (T100, F10.3, T50, E9.3, T1,
      'bANSWER IS')
```

where b represents a blank.

NOTE: If the T-format code is being used with a FORTRAN record that is being punched on paper tape or printed on the typewriter, the integer constant (w) must point to the last character to be punched or printed. This is necessary since blanks or additional characters are not output to save time. For example, the following statement:

```
FORMAT (4HABCD, T3)
```

would cause only AB to be punched or printed. To output ABCD, a T-format code of T5 would be required.

Multiple Field Format

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

1. A punched paper tape record with a maximum of 80 characters (1054 Paper Tape Reader, 1055 Paper Tape Punch, or 1134 Paper Tape Reader).
2. A punched card record with a maximum of 80 characters (1442 Card Read-Punch, Model 6 or 7, or 1442 Card Punch, Model 5).
3. A printed line with a maximum of 120 print characters and 1 carriage control character (1132 Printer or 1403 Printer).
4. A printed line with a maximum of 144 print characters and 1 carriage control character (1443 Printer).
5. An output typewritten line with a maximum of 120 characters (Console Printer, 1053 Printer, or 1816 Printer-Keyboard).
6. An input record from the keyboard with a maximum of 80 characters (Console Keyboard or 1816 Printer-Keyboard).
7. A plotted output record with a maximum of 120 characters (1627 Plotter).
8. A magnetic tape record with a maximum length of 145 characters (2401 and 2402 Magnetic Tape Units).

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.

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 records are skipped or n blank lines are inserted between printed 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)
```

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 unit record is to be printed on an 1132, 1403, or 1443 Printer, the first character in that unit record is used for carriage control. Normally the character is specified at the beginning of the format specification for the unit record as 1Hx, where x is a blank, 0, 1, or +. This character is not printed; it only controls carriage spacing as follows:

blank	causes a single space before the unit record is printed
0	causes a double space before the unit record is printed
1	causes a skip to channel 1 before the unit record is printed
+	causes all spacing or skipping to be suppressed before the unit record is printed

Data Input to the Object Program

Data Input to the object program is contained in unit records, as described under Multiple Field Format, above. 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. Leading blanks are ignored. All other blanks are treated as zeros.
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 provide the compiler with information about:

1. The nature of the variables used in the program.
2. The allocation in storage for certain variables and/or arrays.
3. The names of subprograms to be used at object time.

The Specification statements are non-executable because they do not cause the generation of instructions in the object program.

All Specification statements must precede any statement function definition statement and the first executable statement of the source program. They should appear in the following order:

Type Statements (REAL, INTEGER)
EXTERNAL Statements
DIMENSION Statements
COMMON Statements
EQUIVALENCE Statements
DATA Statements
DEFINE FILE Statements
Statement Function Definition Statements
First Executable Statement

TYPE STATEMENTS (REAL, INTEGER)

General Form:

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

where:

a, b, c, ... are variable, array, FUNCTION subprogram or statement function names appearing in a program or subprogram. Arrays named in this statement must also be dimensioned in this statement. Array dimensions specified in this statement should not be included in references to the array in DIMENSION or COMMON statements. Repetition or respecification of the array dimensions results in an error.

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.

EXTERNAL STATEMENT

General Form:

EXTERNAL a,b,c,....

where:

a,b,c,...are the names of subprograms that appear in any other subprogram argument list. Only the subprogram name is used with the EXTERNAL statement. Other subprogram parameters must not be included. Subprograms declared external may be FUNCTION subprograms, SUBROUTINE subprograms, FORTRAN supplied FUNCTION subprograms, or subprograms written in Assembler Language.

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 when the executable core load is built, not during compilation.

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

Blank COMMON

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 in 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

Named COMMON

Named COMMON is valid only in FORTRAN under the IBM 1800 TSX and MPX Systems, where the name INSKEL specifies Skeleton COMMON. The Skeleton COMMON is located in the low core addressed Skeleton Area. It is not altered by the IBM System and provides the capability for complete communications between process core loads, non-process core loads, INSKEL interrupt subroutines, in-core-with-mainline interrupt subroutines (TSX only), interrupt core loads, and special core loads. Process and non-process programs (either as part of mainline or interrupt core loads) can refer to the Skeleton COMMON area by the following statement:

COMMON/INSKEL/a, b, c,...n

where:

INSKEL is the name of Skeleton COMMON. INSKEL must be enclosed in slashes.

a, b, c,...n are variable or array names as described for the blank COMMON statement.

NOTE: Non-process core loads should reference INSKEL COMMON only for process-oriented functions such as updating conversion factors after time-shared instrument calibration.

The assignment of variables or constants to the COMMON areas can be mixed in the same COMMON statement by preceding the Skeleton COMMON items with /INSKEL/ and by preceding the blank COMMON items with //. For example, in the statement

COMMON/INSKEL/A, B, C//D, E, F

the variables A, B, and C will be assigned locations in the Skeleton COMMON area and D, E, and F will be assigned locations in the blank COMMON area. The same assignment could be made with the following statement.

COMMON D, E, F/INSKEL/A, B, C

In this case, the double slashes are not necessary because the blank COMMON items were not preceded by a Skeleton COMMON assignment.

NOTE: INSKEL COMMON may be used in one word integer programs only.

COMMON Statement With Dimensions

General Form:

COMMON A(k_1), B(k_2), C(k_3),...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(1), 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. Array dimensions may be specified for both blank COMMON and named COMMON variables.

NOTES:

1. Dummy arguments for SUBROUTINE or FUNCTION statements cannot appear in COMMON statements, if they appear on the SUBROUTINE or FUNCTION statement.

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 share the same storage location. 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 information) or single subscripts. The subscripts must be integer constants.

Each pair of parentheses in the EQUIVALENCE statement encloses a list of 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 (X,Y,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)
.	.	.
.	.	.

NOTE: Any EQUIVALENCE statement that lists an array must reference elements of that array. That is, if A and B are both 30 element arrays to be equated,

```
EQUIVALENCE (A,B)
```

is not allowed. The arrays may be equated by a statement of the form:

```
EQUIVALENCE (A(1), B(1))
```

The combination of all equivalence lists in a program must not:

1. Equate two variables or array elements that are already assigned to COMMON.
2. Contradict any previously established equivalences.
3. Extend an array beyond the dimensions defined in a DIMENSION, TYPE, or COMMON statement.

Example 1: Violating Rule 1

```
DIMENSION A(10), B(5)
COMMON A, B
EQUIVALENCE (A(1), B(1))
```

Example 2: Violating Rule 2

```
EQUIVALENCE (A(10), B(1))
EQUIVALENCE (B(10), C(1))
EQUIVALENCE (A(10), C(1))
```

Example 3: Violating Rule 3

```
DIMENSION A(3), B(3)
EQUIVALENCE (A(4), B(1))
```

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).

NOTE: Conversion to single subscripts is permitted only in EQUIVALENCE statements.

DATA STATEMENT

The DATA statement is used to define initial values of variables and array elements assigned to areas other than COMMON. Values assigned to variables or array elements during execution override values assigned via the DATA statement.

General Form:

DATA $V_1, V_2, \ldots, V_n/i_1*d_1, i_2*d, \ldots, i_m*d_m/,$
$V_{n+1}, \ldots, V_r/i_{m+1}*d_{m+1}, \ldots, i_s*d_s/, \ldots/\ldots/$

where:

$V_1, \ldots, V_r$ are variables or subscripted variables (subscripts must be integer constants).

$d_1, \ldots, d_s$ are data constants. They may be integer, real, hexadecimal, or literal data constants. Integer and real constants may be specified as negative. See Data-Variable Combinations for the valid name and constant combinations.

$i_1, \ldots, i_s$ are optional unsigned integer constants that indicate the number of variables and/or array elements that are to be assigned the value of the data constant. They are separated from the data constants by asterisks. Each data constant must be of the same type (integer or real) as its corresponding variable. The value assigned to i must not be >4095.

The slash is used to separate and enclose data constants.

When an unsubscripted array name is specified, constants are assigned from the first element toward the end of the array.

Example 1:

```
DATA A/5*1.0, 2.0, 3*3.0/
```

If A is a nine-element array for real variables, the first five elements are initialized to 1.0, the sixth to 2.0, and the remaining three to 3.0.

If a given constant is not exhausted by assignment to a given variable or array, the remainder will be assigned to succeeding variables or arrays.

Example 2:

```
DATA A, B/12*1.0/
```

If A is a nine-element array for real variables and B is an array containing positions for at least three real variable elements, all nine elements of A and the first three elements of B will be initialized to 1.0.

An error condition occurs if all constants are not exhausted when the last variable or array has been satisfied. Similarly, an error occurs when a variable or array is specified for which no constants are available.

When an array name is specified with a subscript, only the element specified may be assigned a value.

Example 3:

```
DATA A(3)/5.0/
```

If A is an array for real variables of at least 3 elements, the third element will be initialized to 5.0.

Hexadecimal Constants

Hexadecimal constants are written as the letter Z followed by one to four hexadecimal digits (0 through F). Each constant is assigned one word and the constant is right-justified if three or less hexadecimal digits are used. Each constant must be separated by a comma.

Any variable, array, or array element to which a hexadecimal constant value is assigned by the DATA statement must be an integer variable, integer array, or an element of an integer array.

Example 4:

```
DATA I/6* Z24, ZAB19/
```

The first 6 elements of array I will be initialized to the following configuration:

0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
0				0				2				4			

The seventh element will be initialized to:

1	0	1	0	1	0	1	1	0	0	0	1	1	0	0	1
A				B				1				9			

Literal Data

Literal data must be enclosed in single quotes. A quote mark within a literal field is represented by two consecutive quote marks. A literal constant may not exceed the element length of the variable or array to which it is assigned. Where necessary, blanks are included, with the constant left-justified. Literal data is written in 8-bit EBCDIC, packed two characters per word.

Example 5:

```
DATA A/3*'ABCD', 2*'AB', 'A''BC', 'A.BC'/
```

If the array A contains at least seven elements, and is of standard (two word) precision, the first three elements will be assigned the value ABCD, the fourth and fifth the value ABbb, where the b's are blanks, the sixth element the value A'BC, and the seventh A.BC.

Example 6:

```
DATA KEYWD/2*'AB', 'A''','B. ', 'AB', 'X'/
```

If literal data is assigned to an integer array, a maximum of two characters per element may be specified, regardless of the precision of the program. In the array KEYWD, which consists of at least 6 elements, the first two elements are assigned the value AB, the third element A', the fourth element B., the fifth element AB, and the sixth element Xb where b is a blank.

Data-Variable Combinations

variable \ data	real	integer	hexadecimal	literal
real	yes	no	no	yes
integer	no	yes	yes	yes

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; 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 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 is an integer constant ≤ 32767 that is the symbolic designation for this file.
m is an integer constant that defines the number of file records in this symbolic file.
l 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. It is used to designate that the file must be read/written with the disk READ/WRITE statements and will handle no data conversion.
v is a non-subscripted integer variable name. This variable, called the associated variable, is set at the conclusion of each disk READ, WRITE, and FIND statement referencing this symbolic file. After a READ or WRITE statement, it is set to the value of the next available file record. After a FIND statement, it is set to the value of the indicated record.

This variable must be set initially by the user if it is to be used in disk I/O statements as a symbolic record number. This variable must appear in COMMON if it is to be referenced by more than one program during execution.

NOTE: The associated variable for a defined file cannot appear in the I/O list of a disk READ or WRITE statement referencing that file.

An example of defining a data file is:

```
DEFINE FILE 3 (400, 60, U, K)
```

The DEFINE FILE statement furnishes execution time FORTRAN I/O subroutines with the necessary parameters to manipulate data files that are user-generated or system-generated. The user-generated data files are a result of Disk Utility Program functions requested by the user (refer to the sections describing the *FILES control record and the STOREDATA function of the Disk Utility Program or MPX Disk Management Program, in the appropriate operating procedures publication as listed in the Preface, above). The *FILES control records supply at the time the executable core load is built those parameters not supplied by DEFINE FILE statements. That is they provide a correlation between the file numbers found on the DEFINE FILE statements and data file names on the disk. System-generated data files are temporary disk storage areas allocated by the Core Load Builder. They are a result of DEFINE FILE statements for which no matching file numbers exist on *FILES control records.

NOTE: Since records that require no data conversion are transmitted, care must be exercised to ensure that the programs using a data file have the same precision (standard or extended).

SUBPROGRAM STATEMENTS

Suppose that a program is being written that, 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 FUNCTION 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, excluding special characters, the first of which must be alphabetic. The type (real or integer) of a subprogram (except SUBROUTINE) can be indicated in the same manner as variables.

The type of 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 FUNCTION 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 overriden 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.

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 a 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 subroutine. 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 Definition Statement

General Form:

a = b

where:

- a is a function name followed by parentheses enclosing its arguments, which must be distinct, nonsubscripted 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.

FUNCTION Subprogram

The FUNCTION subprogram is a FORTRAN subprogram consisting of any number of statements. It is like a FORTRAN-supplied FUNCTION 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.

$a_1, a_2, a_3, \ldots a_n$ are dummy arguments to be replaced at execution time by nonsubscripted variable names, array names, or other subprogram names (except that they cannot be statement function names). None of the dummy arguments may appear in an EQUIVALENCE statement in a FUNCTION subprogram.

The FUNCTION subprogram may contain any FORTRAN statement except a SUBROUTINE statement, a DEFINE FILE statement, or another FUNCTION statement and must return control to the calling program with a RETURN statement. Because the FUNCTION is a separate subprogram, the variables and statement numbers do not relate to any other program (except the dummy argument variables).

The arguments of the FUNCTION 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. They may be any of the following: any type of constant, any type of subscripted or nonsubscripted variable, any other kind of arithmetic expression, or a subprogram name (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, a DIMENSION statement must appear in the FUNCTION subprogram. The DIMENSION statement in the FUNCTION subprogram permits the dummy argument to be subscripted. Thus, if B is a 40-element array defined in a calling program, a method of passing elements of the array to a FUNCTION subprogram would be:

```
  Calling               FUNCTION
  Program               Subprogram

     .                FUNCTION SOMEF (X,ITER)
     .                DIMENSION X(40)
DIMENSION B(40)       SOMEF = 0
     .                DO 5 I = 1,ITER
     .              5 SOMEF=SOMEF+X(ITER)
D = SOMEF (B,J)       RETURN
                      END
```

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 dummy 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, in the I/O list of a READ statement, or as the index in a DO statement. Variables that appear in COMMON 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 the I/O list of 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.

FORTRAN-supplied FUNCTION Subprograms

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

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

Examples:

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

The use of the SQRT function in the first example causes the calculation of 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 FORTRAN-supplied FUNCTION subprograms to specify required precision. The prefix is added to any variable name that is the same as the FORTRAN-supplied FUNCTION subprogram names.

For detailed descriptions of the FORTRAN-supplied FUNCTION subprograms, refer to the appropriate Subroutine Library publication as listed in the Preface).

Table 1. FORTRAN-supplied FUNCTION Subprograms

Name	Function Performed	No. of Arguments	Type of Argument(s)	Type of Function
SIN	Trigonometric sine (argument in radians)	1	Real	Real
COS	Trigonometric cosine (argument in radians)	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 (Arg_1 given sign of Arg_2)	2	Real	Real
ISIGN	Transfer of sign (Arg_1 given sign of Arg_2)	2	Integer	Integer
TANH	Hyperbolic tangent	1	Real	Real

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 is not restricted to a single result, as is 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 the special FORTRAN statement, the CALL statement (see CALL Statement).

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 or may be located in COMMON). 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).

The SUBROUTINE subprogram may contain any FORTRAN statement except a FUNCTION statement, another SUBROUTINE statement, a DEFINE FILE statement, or any other statement in which the SUBROUTINE name is used as a variable in an expression or list.

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 the I/O list of a READ statement 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, a DIMENSION statement must appear in the SUBROUTINE subprogram.

END AND RETURN STATEMENTS IN SUBPROGRAMS

Note that all of the preceding examples of 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
```

SUBPROGRAMS WRITTEN IN ASSEMBLER LANGUAGE

Subprograms can be written in the 1130 or 1800 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 subprograms (SUBROUTINE subprograms, FUNCTION subprograms, FORTRAN-supplied FUNCTION subprograms) is assembled and executed in the same way as the Assembler Language CALL statement (see the appropriate Assembler Language publication as listed in the Preface).

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

Examples:

SUBROUTINE subprogram CALL:

CALL JOE (A,B,C)

Contents of core storage at execution:

BSI L (Address of Transfer Vector, which contains 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 reference or FORTRAN-supplied FUNCTION subprogram reference:

X = Y + <u>JOE(A,B,C)</u>

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 subprogram <u>must</u> return a single result to the calling program by means of the real number pseudo-accumulator, referred to as FAC, or the machine accumulator (A-register in the 1800), depending on whether the function type is real or integer. That is, assuming JOE is a real function in the above example, the computed result of JOE(A, B, C) must be placed in FAC by the Assembler coded subprogram, since the contents of FAC will be added to Y to yield X. (For a description of FAC, refer to the Real Number Pseudo-Accumulator in the applicable Subroutine Library manual.) The argument list must not be used to return a result of the subprogram computation.

In the table below, the FORTRAN statements described in this publication are listed alphabetically at the left. There is one column at the right for each of the IBM 1130 and 1800 programming systems supported. An 'x' in a column indicates that the statement on the left applies to the programming system named at the top of that column.

	1130 Card/Paper Tape Systems	1130 Disk Monitor Systems (Version 1)	1130 Disk Monitor Systems (Version 2)	1800 Card/Paper Tape Systems	1800 TSX/MPX Systems
Arithmetic Statement	X	X	X	X	X
BACKSPACE			X	X	X
CALL EXIT		X	X		X
CALL LINK		X	X		X
CALL LOAD	X[2]			X[2]	
CALL name	X	X	X	X	X
CALL PDUMP			X		
CALL SSWTCH				X	X
Comments statement	X	X	X	X	X
COMMON	X	X	X	X	X
COMMON/INSKEL/					X
CONTINUE	X	X	X	X	X
DATA			X		X
DEFINE FILE		X	X		X
DIMENSION	X	X	X	X	X
DO	X	X	X	X	X
END	X	X	X	X	X
END FILE			X[1]	X	X
EQUIVALENCE	X	X	X	X	X
EXTERNAL	X	X	X	X	X
FIND		X	X		X
FORMAT	X	X	X	X	X
FUNCTION	X	X	X	X	X
GO TO, computed	X	X	X	X	X
GO TO, unconditional	X	X	X	X	X
IF	X	X	X	X	X
INTEGER	X	X	X	X	X
INTEGER FUNCTION	X	X	X	X	X
PAUSE	X	X	X	X	X
READ, disk I/O		X	X		X
READ, non-disk I/O	X	X	X	X	X
READ, unformatted I/O			X	X	X[4]
REAL	X	X	X	X	X
REAL FUNCTION	X	X	X	X	X
RETURN	X	X	X	X	X
REWIND			X[1]	X	X[3]
Statement Function Statement	X	X	X	X	X
STOP	X	X	X	X	X
SUBROUTINE	X	X	X	X	X
WRITE, disk I/O		X	X		X
WRITE, non-disk I/O	X	X	X	X	X
WRITE, unformatted I/O			X	X	X[4]

1. Simulated in the 1130 Disk Monitor System, Version 2.
2. Card version only.
3. Simulated for unformatted disk in 1800 MPX
4. Both unformatted disk and magnetic tape operations are supported under 1800 MPX.

APPENDIX B. COMPARISON OF USA STANDARD FORTRAN AND IBM 1130/1800 FORTRAN LANGUAGES

This appendix compares the USA Standard FORTRAN, as found in the following documents:

BASIC FORTRAN	X 3.10-1966
FORTRAN	X 3.9-1966

with the FORTRAN language for the IBM 1130 Card/Paper Tape Programming System, the IBM 1130 Disk Monitor System, Version 1, the IBM Disk Monitor System, Version 2, the IBM1800 Card/Paper Tape Programming System, and the IBM 1800 TSX and MPX Systems.

	USA Standard, Full	USA Standard, Basic	1130 Card/Paper Tape	1130 Monitor, Version 1	1130 Monitor, Version 2	1800 Card/Paper Tape	1800 TSX/MPX
Character Set							
A-Z	Yes	Yes	Yes	Yes	Yes	Yes	Yes
0-9	Yes	Yes	Yes	Yes	Yes	Yes	Yes
blank = + - * / ( ) , .	Yes	Yes	Yes	Yes	Yes	Yes	Yes
$	Yes	No	Yes	Yes	Yes	Yes	Yes
' (apostrophe)	No	No	Yes	Yes	Yes	Yes	Yes
Statement Continuation Lines	19	5	5	5	5	5	5
Numeric Statement Label	1 to 5	1 to 4	1 to 5	1 to 5	1 to 5	1 to 5	1 to 5
Variable Name	1 to 6	1 to 5	1 to 5	1 to 5	1 to 5	1 to 5	1 to 5
Data Types							
Integer	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Real*	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Double Precision	Yes	No	No	No	No	No	No
Complex	Yes	No	No	No	No	No	No
Logical	Yes	No	No	No	No	No	No
Hollerith	Yes	No	No	No	No	No	No
Real Constant							
Basic Real Constant	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Integer Constant followed by a decimal exponent	Yes	No	No	No	No	No	No
Double Precision Constant							
Real Constant with 'D' in place of 'E'	Yes	No	No	No	No	No	No
Number of Array Dimensions	3	2	3	3	3	3	3
Relational Expressions	Yes	No	No	No	No	No	No
Logical Operators	Yes	No	No	No	No	No	No
Assigned GO TO	Yes	No	No	No	No	No	No
Logical IF	Yes	No	No	No	No	No	No
DO - Extended Range	Yes	No	Yes	Yes	Yes	Yes	Yes
READ and WRITE							
READ/WRITE (Formatted)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
READ/WRITE (Unformatted)	Yes	Yes	No	No	Yes	Yes	Yes
Auxiliary I/O Statements							
REWIND	Yes	Yes	N.A.	N.A.	Yes	Yes	Yes
BACKSPACE	Yes	Yes	N.A.	N.A.	Yes	Yes	Yes
ENDFILE	Yes	Yes	N.A.	N.A.	Yes	Yes	Yes
Formatted Records							
1st character not printed	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Space before printing							
blank 1 line	Yes	No	Yes	Yes	Yes	Yes	Yes
0 2 lines	Yes	No	Yes	Yes	Yes	Yes	Yes
1 1st line, new page	Yes	No	Yes	Yes	Yes	Yes	Yes
+ no advance	Yes	No	Yes	Yes	Yes	Yes	Yes
Adjustable Dimension	Yes	No	No	No	No	No	No

* Precision specified at compile time.

	USA Standard, Full	USA Standard, Basic	1130 Card/ Paper Tape	1130 Monitor, Version 1	1130 Monitor, Version 2	1800 Card/ Paper Tape	1800 TSX/MPX
Common							
Blank	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Named	Yes	No	No	No	No	No	Yes**
Array Size Declared	Yes	No	Yes	Yes	Yes	Yes	Yes
External Statement	Yes	No	Yes	Yes	Yes	Yes	Yes
Type Statement	Yes	No	Yes	Yes	Yes	Yes	Yes
Dimension Information	Yes	No	Yes	Yes	Yes	Yes	Yes
Data Statement	Yes	No	No	No	Yes	No	Yes
Format Types							
A	Yes	No	Yes	Yes	Yes	Yes	Yes
D	Yes	No	No	No	No	No	No
E	Yes	Yes	Yes	Yes	Yes	Yes	Yes
F	Yes	Yes	Yes	Yes	Yes	Yes	Yes
G	Yes	No	No	No	No	No	No
H	Yes	Yes	Yes	Yes	Yes	Yes	Yes
I	Yes	Yes	Yes	Yes	Yes	Yes	Yes
L	Yes	No	No	No	No	No	No
X	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Format (Parenthesis Levels)	2	1	1	1	1	1	1
Scale Factor	Yes	No	No,	No	No	No	No
Blanks in Numeric Conversions							
High-Order	Zero	Zero	Zero	Zero	Zero	Zero	Zero
Within the field	Zero	Error	Zero	Zero	Zero	Zero	Zero
Real Conversions							
Integer plus Exponent							
E Type Exponent	Yes	Yes	Yes	Yes	Yes	Yes	Yes
D Type Exponent	Yes	No	No	No	No	No	No
Format During Execution	Yes	No	No	No	No	No	No
Statement Function must precede the first executable statement and follow the specification statements	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Type Specification in a Function Statement	Yes	No	Yes	Yes	Yes	Yes	Yes
Function May Define or Redefine its Arguments	Yes	No	No	No	No	No	No
Transmit in a Call							
Hollerith Arguments	Yes	No	No	No	No	No	No
External Subprogram Names	Yes	No	Yes	Yes	Yes	Yes	Yes
Block Data Subprogram	Yes	No	No	No	No	No	No
Specification Statements							
Precede first executable Statement	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Must be ordered DIMENSION COMMON EQUIVALENCE	No	Yes	Yes	Yes	Yes	Yes	Yes

** Only the name INSKEL, specifying the Skeleton COMMON area, is allowed.

	USA Standard, Full	USA Standard, Basic	1130 Card, Paper Tape	1130 Monitor, Version 1	1130 Monitor, Version 2	1800 Card Paper Tape	1800 TSX/MPX
External Function may alter variables in COMMON	Yes	No	No	No	No	No	No

LANGUAGE FEATURES NOT IN USA STANDARD FORTRAN

Feature	1130 Card/ Paper Tape	1130 Monitor, Version 1	1130 Monitor, Version 2	1800 Card Paper Tape	1800 TSX/MPX
Mixed mode Arithmetic	Yes	Yes	Yes	Yes	Yes
Disk Statements	No	Yes	Yes	No	Yes
T format	No	No	Yes	No	Yes
Literal Format code	Yes	Yes	Yes	Yes	Yes
Expression of the form A**B**C	Yes	Yes	Yes	Yes	Yes
Machine indicator tests	Yes	Yes	Yes	Yes	Yes
Source characters @, &, #, <, %	Yes	Yes	Yes	Yes	Yes

APPENDIX C. 1130/1800 FORTRAN SOURCE PROGRAM CHARACTER CODES

Character	IBM Card Code	PTTC/8 Hex (U = Upper Case) (L = Lower Case)
Numeric Characters*		
0	0	1A (L)
1	1	01 (L)
2	2	02 (L)
3	3	13 (L)
4	4	04 (L)
5	5	15 (L)
6	6	16 (L)
7	7	07 (L)
8	8	08 (L)
9	9	19 (L)
Alphabetic Characters*		
A	12-1	61 (U)
B	12-2	62 (U)
C	12-3	73 (U)
D	12-4	64 (U)
E	12-5	75 (U)
F	12-6	76 (U)
G	12-7	67 (U)
H	12-8	68 (U)
I	12-9	79 (U)
J	11-1	51 (U)
K	11-2	52 (U)
L	11-3	43 (U)
M	11-4	54 (U)
N	11-5	45 (U)
O	11-6	46 (U)
P	11-7	57 (U)
Q	11-8	58 (U)
R	11-9	49 (U)

Character	IBM Card Code	PTTC/8 Hex (U = Upper Case) (L = Lower Case)
Alphabetic Characters*		
S	0-2	32 (U)
T	0-3	23 (U)
U	0-4	34 (U)
V	0-5	25 (U)
W	0-6	26 (U)
X	0-7	37 (U)
Y	0-8	38 (U)
Z	0-9	29 (U)
Special Characters*		
.	12-8-3	6B (L)
<	12-8-4	02 (U)
(	12-8-5	19 (U)
+	12-8-6	70 (U)
&	12	70 (L)
$	11-8-3	5B (L)
*	11-8-4	08 (U)
)	11-8-5	1A (U)
-	11	40 (L)
/	0-1	31 (L)
,	0-8-3	3B (L)
%	0-8-4	15 (U)
#	8-3	0B (L)
@	8-4	20 (L)
'	8-5	16 (U)
=	8-6	01 (U)
Space	Blank	10 ( )

NOTES:

1. At compilation time, the following character punches are treated as being equal, and the characters to the left of the "and" are printed. Any invalid character is printed as an ampersand on all systems except 1800 TSX and MPX. If the FORTRAN compiler in the TSX system uses the card routine in the skeleton, a blank will be printed out. If the TSX FORTRAN compiler uses its own card routine, an ampersand is printed out.

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

2. Only the 53 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 (space).
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.

---

*The term, alphameric characters, as used in this publication, includes Special Characters.

1. No FORTRAN statement can be compiled that contains more than 15 different subscript expressions.
2. Certain very long FORTRAN statements cannot be compiled since they expand to a size that is too long to be scanned. This expansion by the compiler occurs in handling subscript expressions and in generating temporary storage locations for arithmetic expressions.
3. FORTRAN supplied subprograms, FLOAT, and IFIX may not be used in EXTERNAL statements.
4. Within A, H, I, T, and X specifications in FORMAT statements, the field width "w" may not be greater than 145.
5. Within E and F specifications the field width "w" may not be greater than 127 and the number of decimal places specified for "d" may not be greater than 31.
6. The repetition specification for groups and fields and the total width specification for a record may not be greater than 145.
7. The size of COMMON specified in a mainline program must be at least as large as the largest COMMON specified in any subprogram.
8. A maximum of 75 files can be specified in DEFINE FILE statements per program.
9. When standard precision is used, it is possible for two quantities representing the same value to yield a non-zero result when subtracted from one another due to the extra eight bits of precision in FAC not used by standard precision. The non-zero result, although not reflected in the first seven significant digits, will effect an IF statement test.
10. Variables used in subscript expressions should not be equivalenced to other variables which may change their value. If they are equivalenced, the new value assumed by the equivalenced variable may be disregarded by the variable in the subscript expression.
11. In a DATA statement, the maximum value of the constant repeat index is 4095, i. e. DATA V/i*d/≤ i≤.

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 or first statement of a DO loop
DATA	Nonexecutable
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

C26-3715-3

IBM 1130/1800 Printed in U.S.A. C26-3715-3

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

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

READER'S COMMENT FORM

IBM 1130/1800 Basic FORTRAN IV Language | Form C26-3715-3

- 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. Comments and suggestions become the property of IBM.

	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.

COMMENTS

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

YOUR COMMENTS PLEASE . . .

This SRL bulletin is one of a series which serves as reference sources for systems analysts, programmers and operators of IBM systems. Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

Fold Fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N. Y.

BUSINESS REPLY MAIL
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
112 East Post Road
White Plains, N. Y. 10601

Attention: Department 813

Fold Fold

IBM®

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

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

IBM 1130/1800 Printed in U.S.A. C26-3715-3