CALL/360

SBC®

# BASIC Reference Handbook

65-2211-1

# CALL/360:

## BASIC Reference Handbook

SBC®

# CONTENTS

# PREFACE

This publication describes the SBC CALL/360: BASIC programming language. It is a reference document, primarily intended for experienced BASIC users seeking specific information.

This manual describes features inherent to the SBC CALL/360: BASIC language only, and includes topics such as program structure, program statements, program limits and error messages. It does not describe system features which are common to all SBC CALL/360 languages. The reader is referred to the *Command Language Reference Manual* (form no. 65-2403) for a description of topics such as system commands, system messages, correction procedures and so forth.

Additional information about CALL/360: BASIC may be found in the following publications:

- *CALL/360: BASIC Introduction* (form no. 65-2204-1)
- *CALL/360: BASIC Terminals Reference Manual* (form no. 65-2210)
- *CALL/360: BASIC Reference Card* (form no. 65-2394-1)

A Reader's Comment Form is included at the end of this publication, and comments regarding the manual are welcomed.

# INTRODUCTION

This manual describes SBC CALL/360: BASIC, a powerful problem-solving language based upon the original language developed at Dartmouth College, Hanover, New Hampshire.

The manual is divided into four sections:

- Section I, *CALL/360: BASIC Program Structure,* describes the BASIC character set and the composition of a BASIC program.
- Section II, *Elements of CALL/360: BASIC Statements,* describes the components of the BASIC language and the data structure involved in the process.
- Section III, *Input and Output,* deals with the methods by which the user may enter and manipulate data in a BASIC program.
- Section IV, *CALL/360: BASIC Statements,* describes each statement of the SBC CALL/360: BASIC language. The BASIC statements are organized functionally with each statement assigned to one of eight functional blocks. A general example at the beginning of each functional block illustrates a program using each of the BASIC statements described within that particular block. Each description of a program statement includes the general form of the statement, the effect of the statement and at least one example of its use.

Two appendices are also included. *Appendix A* deals with program limits. *Appendix B* lists the three types of CALL/360: BASIC diagnostic error messages: compilation error messages resulting from errors detected by the BASIC language processor during program compilation; execution error messages resulting from errors detected during program execution which are severe enough to terminate further execution; exception error messages warning the user of an error condition which is not severe enough to interfere with program execution.

System messages, resulting from a system command, may be found in the *Command Language Reference Manual.* A system message may be either an indication of an error condition or a normal response to a particular command.

# SECTION I

# CALL/360:BASIC PROGRAM STRUCTURE

The SBC CALL/360: BASIC language is composed of a set of statements and a set of commands. BASIC statements are used to create program solutions. Commands are used to direct the system in performing tasks such as providing program listings, modifying programs under construction or in library storage, and in general are used to communicate with the system.

## CALL/360:BASIC STATEMENTS

A BASIC statement defines the type of operation performed and the kinds of data manipulated by the program. Two types of statements are provided: executable and nonexecutable. An executable statement specifies a program action (e.g., LET X = 5); a nonexecutable statement provides information necessary for program execution (e.g., DATA 1, 2, 5, 6E-7).

Every BASIC statement must be prefaced by a line number. The line number associated with each statement identifies the line and also determines its placement in the user's work area. A line number may consist of one through five digits, and always begins at the leftmost carrier position. It cannot contain embedded blanks or nonnumeric characters.

A statement line is composed of a single CALL/360: BASIC statement prefaced by a line number. For example:

$$\underset{\substack{\text{Line}\\ \text{Number}}}{\underset{\uparrow}{\texttt{10}}}\ \underbrace{\texttt{LET X = 2*Y+7*Z}}_{\substack{\text{BASIC}\\ \text{Statement}}}$$

## CALL/360 SYSTEM COMMANDS

The system commands used with CALL/360: BASIC are common to all CALL/360 languages and are described in detail in the *Command Language Reference Manual.*

A system command always begins at the leftmost carrier position and a command line is composed of a single system command (in some cases followed by information supplied by the user). For example:

$$\underbrace{\texttt{NAME}}_{\substack{\text{System}\\ \text{Command}}}\ \underbrace{\texttt{INTEREST}}_{\substack{\text{Information supplied}\\ \text{by the user}}}$$

## CALL/360:BASIC PROGRAMS

A CALL/360:BASIC program is a group of statement lines arranged according to the following general rules:

1. A statement line may occupy no more than one print line.
2. A print line may contain only one statement.
3. Program statements are executed in the sequence in which they are numbered, and may be entered in any order.
4. Executable and nonexecutable statements may be intermixed. Transfer of control to a nonexecutable statement causes control to pass to the next executable statement.
5. The first reference to an array establishes a declaration for the array (e.g., LET A(I,J) = C1).
6. An END statement must have the highest line number in the source program.

## CONVENTIONS OF STATEMENT SPECIFICATIONS

The following conventions are used in this manual to describe the formats of CALL/360: BASIC statements:

1. Uppercase letters, digits and special characters must appear exactly as shown.
2. Information in lowercase letters must be supplied by the user.
3. Information contained within brackets [ ] represents an option that may be omitted by the user.
4. An ellipsis (a series of three periods) indicates that a variable number of items may be included in a list. A list whose length is variable is specified by the format $x_1, x_2, x_3, \ldots, x_n$. This format indicates that x may be repeated from 1 to n times.
5. The appearance of one or more items in sequence indicates that the items, or their replacements, should also appear in the specified order.
6. A vertical bar (|) indicates that a choice must be made between the item to the left of the bar and the item to the right of the bar.

## CALL/360:BASIC CHARACTER SET

A CALL/360:BASIC program is written using the following character set:

| | |
|---|---|
| Letters: | A B C D E F G H I J K L M N O P Q R S T U V W X Y Z @ # $ |
| Digits: | 1 2 3 4 5 6 7 8 9 0 |

Special characters:

| | | | |
|---|---|---|---|
| ' | Single quote | * | Asterisk (Multiplication) |
| " | Double quote | / | Right oblique (slash) (Division) |
| < | Less than | ↑ | Up arrow (Exponentiation) |
| ≤ | Less than or equal to | ( | Left parenthesis |
| = | Equal to | ) | Right parenthesis |
| ≥ | Greater than or equal to | ! | Exclamation mark |
| > | Greater than | , | Comma |
| ≠ | Not equal to | . | Period |
| & | Ampersand | ; | Semicolon |
| + | Plus | : | Colon |
| - | Minus | | Blank |

Any valid terminal character not listed is a non-BASIC character and may be used only where specifically noted.

# SECTION II

# ELEMENTS OF CALL/360:BASIC STATEMENTS

All arithmetic computations in CALL/360: BASIC are performed as floating-point numbers. A floating-point number is a machine approximation of the value of the real number.

A BASIC program may be compiled and executed using either short-form arithmetic or long-form arithmetic; the mode is specified via the ENTER command. The system command ENTER BASIC specifies short-form floating-point computations, while ENTER BASICL specifies that all computations be performed in long-form floating-point arithmetic. The long-form computations are more accurate, but also require more machine time to compute. If the ENTER command is not typed, BASIC short-form is automatically assigned by default.

## SHORT-FORM DATA

An integer format (I format) is used to print integer values. Up to eight decimal digits may be printed for integers whose absolute value is less than 16777216. For example:

```
17
203167
5
9993456
```

Decimal numbers are printed in either fixed-point form (F format) or exponential form (E format). The F format is used to specify decimal numbers of up to six digits. The F format specification has a sign and a decimal point. If no sign is used, the number is positive. For example:

```
1.2076
+783347.
-.003424
```

The E format is used for numbers whose magnitude is less than $10^{-1}$ or greater than $10^7$. The number is of the form:

[sign] d.dddddE±ee

The sign is optional. If omitted, the number is positive. The d signifies a digit, and the E specifies the exponent followed by an optionally signed exponent, ee.

If the exponential notation (E format) is used, the value of the constant is equal to the number on the left of the E multiplied by 10 raised to the power of the number following the E.

Examples of the data formats:

| E Format | F Format | I Format | Equivalent Number |
|---|---|---|---|
| -1.70834E+02 | -170.834 | -170 | -170.834 |
| 5.43311E-05 | +.000054 | +0 | +.000054311 |
| 2.17787E+00 | +2.17787 | +2 | +2.17787 |
| -6.72136E-02 | -.067214 | +0 | -.0672136 |
| 9.68E-07 | .000000 | +0 | +.000000968 |

## LONG-FORM DATA

An I format is used to print an integer value up to 15 digits whose absolute value is less than $10^{15}$, using either the PRINT or the PRINT USING statement.

Decimal numbers written in F format are used to print decimal values of up to 15 digits with a decimal point. The F format in long-form data (BASICL) may only be printed with the PRINT USING statement.

Exponential numbers written in E format are used to print a value with a sign, a decimal point, ten decimal digits, the letter E, and a signed characteristic (exponent).

## NUMERIC CONSTANTS

A numeric constant is a string of characters whose value is a decimal number. The defined value cannot be changed throughout program execution. The two general forms of a numeric constant are

```
[+|-] d . . . [.] [d . . . ] [E [+|-] d . . . ]
[+|-] [d . . . ] [.] d . . . [E[+|-] d . . . ]
```

where d is a digit.

Any of the above formats (e.g., E format, F format, etc.) may be used to write numeric constants in a program statement. If the exponential notation (E format) is used, the value of the constant is equal to the number to the left of the E multiplied by 10 to the power of the number to the right of the E.

The magnitude of a numeric constant must be less than 1E+75 and greater than 1E-78. If long-form arithmetic is specified, up to 15 significant digits are retained after conversion. If short-form arithmetic is specified, six significant digits are retained after conversion. If the number of digits written exceeds machine usable values, the system will discard the remaining digits.

Storage is allocated for a numerical constant every time the constant appears in a source program (see note below). For example, if the constant 1.5 appears three times in a source program, three separate storage areas are allocated.

NOTE: Separate storage allocation does not pertain to integers which have ten addresses reserved for storage (0 - 9). If the same integer appears more than once in a source program, that particular address is merely referenced again.

## INTERNAL CONSTANTS

Three internal constants are provided in CALL/360: BASIC. They represent pi, e, and the positive square root of 2. The names of the internal constants may be used in calculations where the values of the constants are needed. They are called &PI, &E and &SQR2. The values inserted by the system are:

| Name | Short-form value | Long-form value |
|---|---|---|
| &PI | 3.141593 | 3.141592653589793 |
| &E | 2.718282 | 2.718281828459045 |
| &SQR2 | 1.414214 | 1.414213562373095 |

For example:

```
10 LET X = &PI*Y/2
20 LET R = &E+4*Z↑3
30 LET Y = &SQR2*C↑4
```

## LITERAL CONSTANTS

A literal constant is a character string enclosed by a pair of single or double quotation marks. The two general forms of a literal constant are:

"[c . . . ]"
'[c . . . ]'

where c is any character.

A single quote may appear in a character string bounded by double quotes, and a double quote may appear in a character string bounded by single quotes. However, when a boundary character appears in the character string, it must be represented as two consecutive boundary characters.

The following examples illustrate how character strings may be represented as literal strings:

| Character String | Literal String |
|---|---|
| ABCD | "ABCD" or 'ABCD' |
| ABC'D | "ABC'D" or 'ABC' 'D' |
| ABC"D | "ABC" "D" or 'ABC"D' |

A literal constant containing less than 18 characters is padded with blanks on the right. A literal constant containing more than 18 characters is truncated on the right. A literal constant containing no characters is interpreted as 18 blank characters.

## VARIABLE NAMES

A variable name is represented by a letter (a character from the extended alphabet), a letter followed by a digit, or a letter followed by the character $. A variable name represents a data item. The data value of any variable may be set or modified by the CALL/360: BASIC statements. There are two types of variables in CALL/360: BASIC: simple and array.

### Simple Variables

#### Simple Numeric Variable

A simple numeric variable is named by a letter (a character from the extended alphabet) or a letter followed by a digit. Examples are:

```
A, B1, @, #4, $9
```

A simple numeric variable can be assigned only a numeric value. The initial value of all simple numeric variables is zero.

#### Simple Alphameric Variable

A simple alphameric variable is named by a letter (a character from the extended alphabet) followed by the character $. Examples are:

```
A$, B$, X$
```

A simple alphameric variable can only be assigned a literal containing a maximum of 18 characters. The initial value of all simple alphameric variables is 18 blank characters.

### Array Variables

An array is an ordered set of data members. Arrays in BASIC may be either one-dimensional or two-dimensional. An array member is referenced by the subscripted array name. A subscript is an expression evaluated in floating-point arithmetic and then truncated to an integer. (For instance, 3.61727E+00 would be truncated to the integer value of 3.) Subscripts of an array variable must be enclosed in parentheses. If two subscripts are used, they must be separated by a comma. The number of subscripts used to reference an array member must equal the number of dimensions specified for the array. The maximum value of the subscript must be within the bounds defined for the array.

The general form of an array variable is:

$a(x_1 [, x_2])$

where a is an array name, and x is an expression.

## Numeric Array Names

A numeric array variable is named by a letter (a character from the extended alphabet). As many as 29 numeric arrays may be specified in a CALL/360: BASIC program.

A numeric array may have one or two dimensions. Numeric arrays may contain only numeric values. The initial value of each numeric array member is zero. Examples are:

```
I(10,12),J(I),A(Z),L(8,14)
```

## Alphameric Array Names

An alphameric array is named by a letter (a character from the extended alphabet), followed by the character $. Alphameric arrays have one dimension. They contain only members whose value is a character string containing 18 characters. The initial value of each alphameric array member is 18 blank characters. Examples are:

```
I$(10),J$(I),L$(8)
```

## Array Declarations

An array declaration states that an array with a specified name and dimensions should be allocated to a user program. Arrays may be defined explicitly in a DIM (dimension) statement or implicitly through usage.

An array is implicitly declared by the first reference to one of its members if the specified array has not been previously defined by a DIM statement. The array is declared to have one dimension (10) when a member is referenced by an array variable with one subscript. The array is declared to have two dimensions (10, 10) when a member is referenced by an array variable with two subscripts.

Array dimensioning and referencing always start at one. A one-dimensional array is a one-column list containing the number of rows given by the subscript. For example, A(10) defines a one-dimensional array having one column and ten rows.

For a two-dimensional array, the first subscript defines the number of rows and the second subscript defines the number of columns. For example, A(4, 6) defines a two-dimensional array having four rows and six columns, or 24 members. The members of such an array are shown below:

(4, 6)

| | | | | | | |
|---|---|---|---|---|---|---|
| row 1 | (1, 1) | (1, 2) | (1, 3) | (1, 4) | (1, 5) | (1, 6) |
| row 2 | (2, 1) | (2, 2) | (2, 3) | (2, 4) | (2, 5) | (2, 6) |
| row 3 | (3, 1) | (3, 2) | (3, 3) | (3, 4) | (3, 5) | (3, 6) |
| row 4 | (4, 1) | (4, 2) | (4, 3) | (4, 4) | (4, 5) | (4, 6) |
| | col. 1 | col. 2 | col. 3 | col. 4 | col. 5 | col. 6 |

Array values are stored in the system by column order. That is, the value for the first row, first column is stored first, the value for the second row, first column is stored next, and so forth. For example, elements of a 2 by 2 array would be stored in the following order:

1, 1
2, 1
1, 2
2, 2

## MATRIX OPERATIONS

A BASIC matrix is a two-dimensional numeric array. The limits of a matrix must be defined by a DIM statement before the matrix is used in any MAT operations. A matrix may then be redimensioned by appending two subscripts (enclosed in parentheses and separated by a comma) to the following matrix statements:

Matrix CON function
Matrix IDN function
Matrix ZER function
MAT GET
MAT READ

Redimensioning, however, must not increase the limits of the array originally declared in the DIM statement. For example:

```
120 DIM A(20,40)
130 DIM B(15,100)
.
.
.
250 MAT READ A(10,40)
260 MAT READ B(1,15)
.
.
.
```

Matrix A was originally a 20 by 40 matrix. Line 250 redefines the limits to 10 by 40. Similarly, matrix B is redefined from a 15 by 100 matrix to a 1 by 15 matrix. Refer to *Section IV: Array Declarations and Matrix Operations* for a further description and examples of matrix operations.

## INTRINSIC FUNCTIONS

An intrinsic function is one whose meaning is predefined by the BASIC language processor. These functions are provided to facilitate the writing of CALL/360:BASIC. The available functions may be used very much as a variable would be used. For example:

```
10 LET A = SIN(23)
20 LET Z = LOG(X) + LOG(Y)
```

The intrinsic functions provided are:

| | |
|---|---|
| SIN(x) | Sine of x radians |
| COS(x) | Cosine of x radians |
| TAN(x) | Tangent of x radians |
| COT(x) | Cotangent of x radians |
| SEC(x) | Secant of x radians |
| CSC(x) | Cosecant of x radians |
| ASN(x) | Angle (in radians) whose sine is x |
| ACS(x) | Angle (in radians) whose cosine is x |
| ATN(x) | Angle (in radians) whose tangent is x |
| HSN(x) | Hyperbolic sine of x radians |
| HCS(x) | Hyperbolic cosine of x radians |
| HTN(x) | Hyperbolic tangent of x radians |
| DEG(x) | Convert x from radians to degrees |
| RAD(x) | Convert x from degrees to radians |
| EXP(x) | Natural exponent of x (e to the power x) |
| ABS(x) | Absolute value of x ($\|x\|$) |
| LOG(x) | Logarithm of x to the base e (ln x) |
| LTW(x) | Logarithm of x to the base 2 |
| LGT(x) | Logarithm of x to the base 10 |
| SQR(x) | Positive square root of x |
| RND(x) | A random number between 0 and 1 (x is a meaningless but necessary entry) |
| INT(x) | Integral part of x |
| SGN(x) | Sign of x, defined as: if $x<0$, SGN(x) = -1; if $x=0$, SGN(x) = 0; if $x>0$, SGN(x) = +1 |

## USER FUNCTIONS

A user function is one whose meaning is defined by the user via the DEF statement. The user function is named by the characters FN followed by a letter. For example, FNA(x) could be defined as:

```
10 DEF FNA(X) = 2+3*X-5*X↑2
```

FNA(x) can then be used in the same manner as an intrinsic function.

## CALL/360:BASIC OPERATORS

### Unary Operators

The unary operators are:

| | |
|---|---|
| + | The value of |
| - | The negative value of |

### Arithmetic Operators

The arithmetic operators are:

| | |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| ↑ or ** | Exponentiation |

### Relational Operators

The relational operators are:

| | |
|---|---|
| < | Less than |
| <= or ≤ | Less than or equal |
| > | Greater than |
| >= or ≥ | Greater than or equal |
| = | Equal |
| <> or ≠ | Not equal |

## EXPRESSIONS

An expression is a combination of identifiers (e.g., A, B, C) and arithmetic operators (e.g., +, -, *) which represents a decimal number. An expression is evaluated by performing the indicated operations as shown below. When not defined, operations are performed from left to right in the expression. The rules are:

1. Operations within parentheses are performed before operations not within parentheses.
2. Operations on the same level are performed in the order in which they appear from left to right in the expression.

3. Operations are performed in sequence from highest level to lowest level. The levels of operation are:
   a. Operations within parentheses
   b. ↑ or ** (exponentiation)
   c. * or /
   d. + or -
4. Alphameric variables or literals are not allowed.
5. Numeric constants may be used.

For example, in the expression X = A+B*C/D+E–F↑2, the order of operation is:

1. F↑2
2. B*C
3. Result of No. 2 divided by D
4. A plus the result of No. 3
5. E plus the result of No. 4
6. Result of No. 5 minus result of No. 1

If the expression were changed to X=(A+B)*C/D+E–F↑2, the order of operation would be:

1. A+B
2. F↑2
3. Result of No. 1 times C
4. Result of No. 3 divided by D
5. E plus the result of No. 4
6. Result of No. 5 minus result of No. 2

Examples of expressions are:

```
A1
-6.4
SIN(R)
X+Y–Z
X3/(–6)
-(X–X↑2/2+X↑(Y*Z))
```

This last expression corresponds to the algebraic expression:

$$-(x-\frac{x^2}{2}+x^{yz})$$

Expressions resulting in an imaginary or mathematically undefined value are not evaluated. The system generates an appropriate error message and terminates execution. If the value of an expression falls outside the limits of machine representable values, the system continues program execution after taking the specified action. Refer to *Appendix B* for a list of diagnostic error messages.

# SECTION III

# INPUT AND OUTPUT

To solve problems with the BASIC language, it is often necessary to enter and manipulate large groups of data. There are three methods of entering data into a BASIC program:

1. Program input
2. Terminal input/output
3. Data file input/output

The statements associated with each of the three methods are described in detail in *Section IV;* a summary of each method is presented here.

## PROGRAM INPUT

The program input statements are READ, DATA and RESTORE. The use of these statements causes data to be compiled into the program. The data may be stored with the program if the program is saved. A brief description of the program input statements follows.

The DATA statement is used to create tables of data values in the program. For example:

```
10 DATA 100.7,-23.2,438.8,201.3,816.9,537.8
```

The values listed after the DATA statement can then be accessed by use of a READ statement in the same program. For example:

```
10 DATA 100.7,-23.2,438.8,201.3,816.9,537.8
20 READ X
30 READ Y,A
40 RESTORE
50 READ Z
```

Line 20 causes the first value in the DATA list (100.7) to be stored in the variable X. Line 30 assigns the second value in the list (-23.2) to the variable Y, and the third value (438.8) to A. The RESTORE statement in line 40 sets the list pointer back to the beginning of the data list. Line 50, therefore, assigns the first variable in the list (100.7) to the variable Z.

DATA statements may appear anywhere in the program. Each time a READ statement is executed, the next sequentially available data value will be assigned to the variable(s) following the READ statement. This continues until all of the data values are exhausted or until a RESTORE statement is executed.

## TERMINAL INPUT/OUTPUT

### Terminal Input

The INPUT statement is provided for terminal input. The use of this statement permits data to be entered from the terminal during program execution. For example:

```
        10 INPUT H,W,L
```

When the line shown above is executed, a question mark (?) is printed at the terminal and the system pauses to allow the values for H, W and L to be entered from the terminal. After the user strikes the carrier return, program execution resumes, using the data values entered for the variables H, W and L.

### Terminal Output

Terminal output consists of various forms of the PRINT statement. The designation of which form to use depends upon the type of printing format desired. The terminal output statements are PRINT, PRINT USING and MAT PRINT.

The PRINT statement may be used to form blank, partial or complete print lines at the terminal. The example shown below illustrates one use of the PRINT statement in a program listing.

```
10 INPUT H,W,L
20 PRINT 'THE VALUES OF H, W AND L ARE'
30 PRINT
40 PRINT H,W,L
50 END
RUN

              09:03    05/04/70   MONDAY      SJ2

? 10,20,30
THE VALUES OF H, W AND L ARE

 10                 20                 30

TIME      0 SECS.
```

Line 20 causes the information enclosed within the quotes to be printed at the terminal. Line 30 specifies a blank line to be inserted between the information print line and the printed values. Line 40 causes the input values for H, W and L to be printed.

The PRINT USING statement specifies data output using an Image statement to establish the printing format. The following illustrates the same example with the PRINT USING statement replacing the PRINT statement.

```
10 INPUT H,W,L
20 PRINT USING 30, H,W,L
30 :THE VALUE FOR H IS ##,THE VALUE FOR W IS ##, THE VALUE FOR L IS ##
40 END
RUN

            09:07    05/04/70  MONDAY     SJ2

? 10,20,30
THE VALUE FOR H IS 10,THE VALUE FOR W IS 20, THE VALUE FOR L IS 30

TIME    0 SECS.
```

Line 20, the PRINT USING statement, specifies the values to be printed in line 30, the Image statement. The colon (:) in line 30 defines it as an Image statement line.

The third type of terminal output, the MAT PRINT statement, is used to print the values of a matrix without the need to specify each element of the matrix. Consider a three-by-three matrix A with the following values:

| | | |
|---|---|---|
| A(1,1) = 11 | A(1,2) = 12 | A(1,3) = 13 |
| A(2,1) = 21 | A(2,2) = 22 | A(2,3) = 23 |
| A(3,1) = 31 | A(3,2) = 32 | A(3,3) = 33 |

The MAT PRINT statement could be used to print the matrix with the following statement:

```
50 MAT PRINT A
```

This would cause the values stored in A to be printed in row and column order:

```
11          12          13
21          22          23
31          32          33
```

## DATA FILE INPUT/OUTPUT

A collection of data items treated as a unit is called a data file. Data files may be created and accessed with the following BASIC statements and system commands:

The GET statement
The PUT statement
The OPEN statement
The CLOSE statement
The RESET statement
The FILE command

The PUT statement is used to write data from a program into a data file, and the GET statement is used to transmit data from a file and read it back into the program. The OPEN statement is used to activate a data file preparatory to data transmission. The OPEN statement associates a data file reference number with a named data file; the named data file is referenced by this number in the subsequent GET, PUT, RESET and CLOSE statements.

For example:

```
FILE BFILE
READY

10 OPEN 21, 'AFILE', INPUT
12 OPEN 22, 'BFILE', OUTPUT
.
.
30 GET 21: V,D,T,X,S,F
.
.
40 PUT 22: D,T,F
.
.
50 CLOSE 21,22
```

The FILE command creates a catalog entry named BFILE. The system places the file in the user's library and allocates storage for the data file on disk. AFILE is already in the user's library and contains data entered from some previous program; therefore, it is not necessary to initiate AFILE with the FILE command. If attempted, the message AFILE ALREADY EXISTS is printed at the terminal.

Line 10 opens AFILE as input and assigns it to 21. Line 12 opens BFILE as output and assigns it to 22. When a data file is OPENed, it is referred to as an active file.

Line 30 accesses AFILE and reads the data values assigned to the variables V, D, T, X, S and F, respectively. Line 40 accesses BFILE and writes an output record consisting of the three values for D, T and F. The CLOSE statement shown in line 50 causes data files 21 and 22 to be deactivated. The CLOSE is normally used only after all desired input and output have been performed on a data file. After a file is CLOSEd, it cannot be referenced again until it is reopened.

If a file is CLOSEd, then reopened, the file pointer is reset to the first record in the file. A file should be left open until all necessary transmission is completed between the program and the file. The RESET statement is provided to reset the file pointer to the first record in the file. A maximum of four (4) files may be active at any one time (i.e., OPENed but not yet CLOSEd).

## DATA FILE STORAGE

A data file is a disk file composed of storage units. Each storage unit consists of 3440 bytes (of which 3333 bytes are available to the user). A maximum of 250 storage units may be allocated for a single data file.

The FILE command is the method by which the user names the data file and allocates storage for that particular file. For example:

```
FILE AFILE,20
```

The system responds by creating a data file named AFILE and reserving 20 units of storage for subsequent data entries. The user has the option of omitting the storage units specification in the FILE command. If storage units are not specified, the system then assumes 50 storage units by default.

If data file storage is exhausted during program construction, the system will prompt the user to increase his file size with the following message (provided that less than 250 storage units were specified originally):

filename EXCEEDED ORIGINAL ALLOCATION OF specified storage units ENTER NEW MAXIMUM - -

The user would then enter a maximum three-digit number (≤250) to increase his file size accordingly.

The user may compute the number of storage units required for a particular BASIC program and thereby determine his estimated file size. Data items are stored sequentially in each data file. The storage requirements for data items are:

1. 18 bytes for alphameric items
2. 4 bytes for short-form data items
3. 8 bytes for long-form data items

Each file can contain a combination of alphameric data items, short-form data items and long-form data items. Simple numeric data is written into a file as floating-point values with each value allocated four bytes of file storage for BASIC short-form, and eight bytes of storage for BASIC long-form. Alphameric variables are written into a file as character-string values with each value allocated 18 bytes of storage. Floating-point values and character-string values are retained in the file as separate data groupings. Consider the following example written in BASIC short-form:

```
30 PUT 10: A,B
.
50 PUT 10: C
.
.
80 PUT 10: A$
90 PUT 10: D
```

The values for the simple numeric variables A, B, C and D are converted to floating-point values, each requiring four bytes of storage. The alphameric variable A$ is converted to a character-string value requiring 18 bytes of storage. Since the variables are written into the file sequentially, three data groupings would be allocated. The formats are:

| | 1 | 1 | 4 | 4 | 4 |
|---|---|---|---|---|---|
| 14 bytes | Floating-point | 3 | A | B | C |

| | 1 | 1 | 18 |
|---|---|---|---|
| 20 bytes | Character-string | 18 | A$ |

| | 1 | 1 | 4 |
|---|---|---|---|
| 6 bytes | Floating-point | 1 | D |

The first byte in each data group is reserved for the data type; the second byte indicates the number of data elements comprised in the data group, and the remaining bytes are reserved for the actual data.

The first data group contains the numeric variables A, B and C converted to floating-point values, and is 14 bytes long. The second data group contains the alphameric variable A$ converted to a character-string value, and is 20 bytes long. The third data group contains the floating-point value for D and is 6 bytes long. Thus, the total file for this program takes up 40 bytes of storage and consumes much less than one storage unit (one storage unit = 3440 bytes).

# SECTION IV

# CALL/360:BASIC STATEMENTS

## INTRODUCTION

This section contains a description of each statement of the SBC CALL/360:BASIC language. The statements are presented in a functional order with each BASIC statement assigned to one of eight major functional blocks. The reader is introduced to each functional block with a summary and general example showing the use of the statements described within that particular block. The general form, general effect and at least one example of usage are provided for each statement.

An alphabetical listing of the BASIC statements contained in this section, together with their functional assignment and page number reference, can be found in Table 1 of this section.

## GENERAL CONSIDERATIONS

Each BASIC statement is preceded by a line number which specifies the order in which the program statements are to be executed. Statements may be entered in any order. The system sorts the program statements into ascending line number sequence before commencing program execution. For example:

Program listing before compilation

```
90 END
10 INPUT A,B,C
30 PRINT D
20 LET D = A+B+C
RUN
```

Program listing after compilation

```
10 INPUT A,B,C
20 LET D = A+B+C
30 PRINT D
90 END
```

Where applicable, various system commands are illustrated in program examples in this section. Refer to the *Command Language Reference Manual* for a complete listing and description of the system commands.

Table 1. Alphabetical Listing of BASIC Statements

| BASIC Statement | Functional Assignment | Page No. |
|---|---|---|
| CLOSE | Data file I/O | |
| DATA | Program and terminal input | |
| DEF | Subroutines and user functions | |
| DIM | Array declarations and matrix operations | |
| END | Program remarks and miscellaneous control statements | |
| FOR | Control statements | |
| GET | Data file I/O | |
| GOSUB | Subroutines and user functions | |
| GOTO (computed) | Control statements | |
| GOTO (simple) | Control statements | |
| IF | Control statements | |
| Image | Terminal output | |
| INPUT | Program and terminal input | |
| LET | Program assignment | 25 |
| Matrix Addition | Array declarations and matrix operations | |
| Matrix CON Function | Array declarations and matrix operations | |
| MAT GET | Array declarations and matrix operations | |
| Matrix IDN Function | Array declarations and matrix operations | |
| Matrix INV Function | Array declarations and matrix operations | |
| Matrix Multiplication | Array declarations and matrix operations | |
| Matrix Multiplication (scalar) | Array declarations and matrix operations | |
| MAT PRINT | Array declarations and matrix operations | |
| MAT PUT | Array declarations and matrix operations | |
| MAT READ | Array declarations and matrix operations | |
| Matrix Subtraction | Array declarations and matrix operations | |
| Matrix TRN Function | Array declarations and matrix operations | |
| Matrix ZER Function | Array declarations and matrix operations | |
| NEXT | Control statements | |
| OPEN | Data file I/O | |
| PAUSE | Terminal output | |
| PRINT | Terminal output | |
| PRINT USING | Terminal output | |
| PUT | Data file I/O | |
| READ | Program and terminal input | |
| REM | Program remarks and miscellaneous control statements | |
| RESET | Data file I/O | |
| RESTORE | Program and terminal input | |
| RETURN | Subroutines and user functions | |
| STOP | Program remarks and miscellaneous control statements | |

## PROGRAM ASSIGNMENT

### • LET

| Statement | Effect | Usage |
|---|---|---|
| LET | Causes a value to be assigned to one or more variables specified in the LET statement line. | `LET A,X(Y+4)=345` |

**General Example**

```
10 LET A1 = 100
20 LET A2 = 90
30 LET A3 = 80
40 LET A4 = 70
50 A5 = 90
60 LET A = (A1+A2+A3+A4+A5)/5
70 PRINT A
80 END
```

This example computes the average of five data values. Lines 10 through 50 assign data values for variables A1 through A5. Line 60 computes the average of the five data values and assigns the average to the variable A. Line 70 requests printing of the value of A.

## LET

**General Form**

[LET] $v_1, v_2, v_3, \ldots, v_n = x$

where v is a variable and x is an expression, an alphameric variable or a literal constant. LET is an optional entry and if omitted will be assigned by default.

**Effect**

The LET statement assigns the value of x to each of the variables $v_1, v_2, v_3, \ldots, v_n$. That is, the variables to the left of the equal sign assume the value of x to the right.

The following syntax rules apply:

1. If x is an expression, all variables to the left of the equal sign must be numeric.
2. If x is an alphameric variable (A$) or a literal constant ('ONE'), all variables to the left of the equal sign must be alphameric.

The subscripts of the replaced variables are computed before the evaluation of x. Therefore, the value of x that replaces the value of the variables does not become effective for computation until execution is begun for the next sequential statement.

The LET statement may be used to implicitly declare a one- or two-dimensional numeric array provided that the referenced array subscript does not exceed ten members. If, for example, a LET statement referenced an array having a dimension containing more than ten members, such as:

```
10 FOR I = 1 TO 15
20 FOR J = 1 TO 10
30 C1 = C1 + 1
40 A(I,J) = C1
50 NEXT J
60 NEXT I
.
.
90 END
```

the following error message would be printed out at the terminal at execution time:

```
        LINE 40:  SUBSCRIPT OUT OF BOUNDS
```

The DIM statement (see page 56) may be used to declare an array having more than ten members in any one dimension.

Subscript checking for a two-dimensional array is not performed on individual subscripts, but rather the two subscripts are reduced to one subscript which is then checked to assure that it is within the bounds of the array. Thus, any one of the subscripts could be zero, negative or greater than the stated dimensions as long as the end result lies within the bounds of the array.

**Examples**

1.
```
10 LET X(Y+3), Z, Y, X(42) = 100.0967
20 LET A, B, C, D, E, F = 0.0
30 LET D$, T$, P$ = B$
```

2.
```
20 LET A1 = Z(3)/Y(A+4)
30 X1 = 49+Z(4)
40 LET A = 5
50 LET G$ = N$
```

## PROGRAM REMARKS AND MISCELLANEOUS CONTROL STATEMENTS

- REMARK
- STOP
- END

| Statement | Effect | Usage |
|---|---|---|
| REMARK | Enables user to add comments to a program without affecting execution. | `REM: SOLVE FOR X` |
| STOP | Terminates program execution. | `STOP` |
| END | Specifies completion of a source program. END must be assigned the highest line number in the program. | `END` |

**General Example**

```
10 REMARK: M EQUALS MASS IN GRAMS
20 REMARK: V EQUALS VELOCITY IN CM/SEC.
30 REM:    T EQUALS KINETIC ENERGY
40 LET M = 20.2
50 LET V = 10
60 LET T = .5*M*V↑2
70 PRINT T
80 END
```

The REMARK statement in this kinetic energy example establishes definitions for the variables M, V and T. It is permissible to truncate the REMARK statement as shown in line 30.

The END statement must be the last physical statement in any program listing. A STOP statement introduced anywhere in the program would have terminated program execution at that point.

The answer to T is 1010.

## REMARK

**General Form**

REMARK: [c]

where c is a character string.

**Effect**

The REMARK statement is used to add comments to a program listing. It does not affect program execution.

If a GOTO, GOSUB or THEN statement references the line number of a REMARK statement, the next executable statement following the REMARK statement will be executed.

**Example**

```
100 REMARK: M EQUALS MASS IN GRAMS
110 REMARK: V EQUALS VELOCITY IN CM/SEC.
120 REM:    T EQUALS KINETIC ENERGY
```

## STOP

**General Form**

STOP [c]

where c is a character string.

**Effect**

This statement causes program execution to terminate.

The character string c may be entered as a comment; it is ignored during compilation and execution.

**Example**

```
30 STOP
```

## END

**General Form**

END [c . . . ]

where c is any character.

**Effect**

This statement causes program compilation and execution to terminate. It must be the last physical statement in the program.

The character string c may be entered as a comment; it is ignored during compilation and execution.

**Example**

```
99 END
```

## CONTROL STATEMENTS

- GOTO (Simple)
- GOTO (Computed)
- FOR
- NEXT
- IF

| Statement | Effect | Usage |
|---|---|---|
| Simple GOTO | Unconditionally transfers control to an executable program statement. | `GOTO 50` |
| Computed GOTO | Transfers control to a program statement(s) depending upon the integer value specified in the GOTO statement line. | `GOTO 10,20 ON A/3` |
| FOR | Initiates a loop. | `FOR I = 1 TO 10` |
| NEXT | Terminates a loop. | `NEXT I` |
| IF | Tests for two conditions of a given value. If the test is satisfied, control branches to a program statement; otherwise it falls through to the next executable statement. | `IF I = 10 THEN 80` |

## General Example

```
010 FOR J = 5 TO 25 STEP 5
020 GOTO 30,50,80,80,70 ON J/5
030 PRINT 'YOU NOW HAVE A NICKLE'
040 GOTO 80
050 PRINT 'YOU NOW HAVE A DIME'
060 GOTO 80
070 PRINT 'YOU NOW HAVE A QUARTER'
080 NEXT J
090 IF J = 25 THEN 110
100 PRINT 'SOMETHING WENT WRONG'
110 END
RUN

            11:51   05/04/70  MONDAY     SJ2

YOU NOW HAVE A NICKLE
YOU NOW HAVE A DIME
YOU NOW HAVE A QUARTER

TIME     0 SECS.
```

Line 10 illustrates a counting statement in which the variable J is incremented five positions from 5 to 25.

Line 20 is a switching statement which transfers control to line 30 when the value for J is 5 (first step), to line 50 when J is 10 (second step), to line 80 when J is 15 and 20 (third and fourth steps) and finally to line 70 when J is 25 (fifth step).

Lines 40 and 60 transfer control past the PRINT statements to line 80, where control is returned to the FOR loop.

Line 90 terminates program execution if the value for J is 25. Line 100 is included for error detection only, and will print if the value for J was other than 25 when the loop was terminated.

## GOTO (Simple)

**General Form**

GOTO n

where n is a line number.

**Effect**

The simple or unconditional GOTO statement causes control to be transferred to the line numbered n.

**Examples**

1.

```
80 GOTO 230
```

2.

```
10 LET X = 5
20 PRINT X
30 GOTO 10
40 END
RUN

               11:52    05/04/70   MONDAY     SJ2

 5
 5

BREAK    TIME      1 SECS.
```

Example 2 illustrates the improper use of a simple GOTO statement in a program. During program execution, line 30 repeatedly returns control to line 10, resulting in an infinite loop. The program will remain in the loop until the user strikes the ATTN (BREAK) key. Inserting an IF statement in the loop is a means of branching out of the loop and executing the program successfully.

## GOTO (Computed)

**General Form**

GOTO $n_1, n_2, n_3, \ldots, n_n$ ON x

where n is a line number and x is an expression.

**Effect**

The computed or conditional GOTO statement causes control to be transferred to the statement numbered $n_1, n_2, n_3, \ldots, n_n$ (line number) depending on whether the truncated integer value of x is 1, 2, 3, . . . , n respectively. If the truncated integer value of x is less than 1 or greater than n, control passes to the next sequential statement.

**Example**

```
40 GOTO 34,60,1,34,10,45 ON 3-4/X-Z
```

In this example, when 3-4/X-Z equals 1 or 4, control is transferred to line 34. If 3-4/X-Z equals 2, control is transferred to line 60.

## FOR

**General Form**

FOR v = $x_1$ TO $x_2$ [STEP $x_3$]

where v is a simple numeric variable and x is an expression.

**Effect**

The FOR v statement initiates repeated looping through the statements that physically follow, up to and including a corresponding NEXT v statement. The FOR statement is always used in conjunction with the NEXT statement, and the range of a FOR is that set of statements, up to and including the corresponding NEXT, that will be executed repeatedly.

The statements within the range of the FOR and NEXT loop are executed repeatedly with v equal to $x_1$, then with v equal to $x_1+x_3$, then with v equal to $x_1+2x_3$, and so forth, until the value of v reaches the limit specified by $x_2$. When the STEP option is omitted the value of $x_3$ is assumed to be +1.

Before execution of the loop begins, the simple numeric variable v is set to $x_1$, and the values of $x_2$ and $x_3$ are computed and stored throughout the life of the loop. The v is tested against $x_2$ before each execution of the loop. The nature of the test depends upon the value of the STEP function $x_3$. For example:

- If $x_3$ is positive, the loop is terminated if $v > x_2$; if not, control passes to the first statement within the loop.
- If $x_3$ is negative, the loop is terminated if $v < x_2$; if not, control passes to the first statement within the loop.

The index of a FOR statement is the simple numeric variable v. The index is available for computation throughout the range of the FOR, either as an ordinary variable or as the variable in a subscript. Upon exiting from the FOR the index v is available for computation and is equal to the last value it attained.

FOR loops may contain up to 14 other FOR loops nested within the outer loop for a total of 15 allowable loops in a program. In all cases the inner loop is executed before the outer loop. The numeric variable v must be a different name for each of the nested FOR loops. Unnested or multiple loops contained in a program may use the same variable name.

**Example**

```
10 A = 0
20 FOR I = 1 TO 10
30 A(I) = I
40 NEXT I
50 PRINT A(I)
60 END
RUN

              15:37    05/11/70   MONDAY      SJ2

 10

TIME      0 SECS.
```

In this example the series of statements computes and prints the final value for the array A(I).

Line 10 sets A to zero. Line 20 initiates a FOR/NEXT loop. The loop variable is I with an initial value of 1. I is incremented by 1 and has a terminal value of 10.

Line 30 sets the array A(I) equal to I. The first time through the loop I is 1 and A(1) is calculated; the next time, I is 2 and A(2) is calculated, and so forth.

Line 40 transfers control back to the FOR statement for the incrementing and testing of I. I is increased by 1 and tested against 10; if I is greater than 10, control passes to line 50; otherwise, control passes to line 30. Line 50 requests printing of the output value for A(I).

## NEXT

### General Form

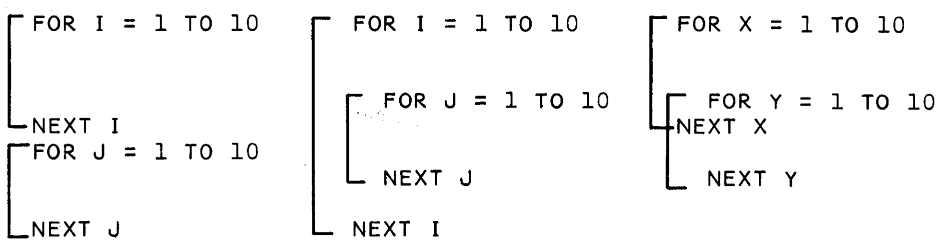
NEXT v

where v is a simple numeric variable.

### Effect

This statement terminates the range of a FOR v loop. The variable name appearing in the NEXT statement must be the same as that appearing in the corresponding FOR v statement. During loop execution the NEXT statement returns control to the beginning of the loop (the corresponding FOR statement). Control transfers to the statement following the NEXT at the completion of loop execution.

### Examples



```
FOR I = 1 TO 10

NEXT I
FOR J = 1 TO 10

NEXT J
```

Multiple Loops

```
FOR I = 1 TO 10

   FOR J = 1 TO 10

   NEXT J

NEXT I
```

Nested Loops

```
FOR X = 1 TO 10

   FOR Y = 1 TO 10
NEXT X
   NEXT Y
```

Incorrect Nesting

In the multiple loop example, the loop on I is repeated ten times, followed by ten repetitions of the loop on J.

In the nested loop example, the J loop is repeated ten times for each of the ten different values of I for a total of 100 J-loop executions.

In the incorrect nesting example, a diagnostic error message is printed when the system attempts to compile the program.

## IF

**General Forms**

IF $x_1$ $\Delta$ $x_2$ THEN n
IF $x_1$ $\Delta$ $x_2$ GOTO n

where x is an expression, an alphameric variable or a literal constant, $\Delta$ is a relational operator and n is a line number.

**Effect**

If $x_1$ is related to $x_2$ as specified by the relational operator, the IF statement is true and control passes to the statement numbered n. Otherwise, the IF condition is false and control passes to the next sequential statement.

If $x_1$ is an expression, then $x_2$ must be an expression. If $x_1$ is an alphameric variable or a literal constant, then $x_2$ must be an alphameric variable or a literal constant.

**Example**

```
10 A = 20
20 B = 60
.
.
50 IF A*3 = B THEN 100
60 Y = 4
.
.
100 Y = 8
```

In this example lines 10 and 20 provide initial values for the variables A and B. Line 50 evaluates the expression A*3 = B. Assuming that the initial values of A and B are unaltered, the expression is true (A*3 = 60 = B) and control passes to line 100. If the evaluation of A*3 = B was false, control would have passed to line 60, where Y would be assigned the value of 4.

## PROGRAM AND TERMINAL INPUT STATEMENTS

- DATA
- READ
- RESTORE
- INPUT

| Statement | Effect | Usage |
|---|---|---|
| DATA | Creates a table of data values in a program. | `DATA 13, 4.6, .066` |
| READ | Fetches values from a data table and assigns them to variables. | `READ A, B, C` |
| RESTORE | Resets the data table pointer to the first data value. | `RESTORE` |
| INPUT | Permits keyboard entry of data during program execution. | `INPUT X, Y, Z` |

## General Example

```
010 DATA 1000,2000,3000
020 RESTORE
030 INPUT R
040 IF R = 10 THEN 130
050 READ P
060 FOR T = 30 TO 180 STEP 30
070 LET I = P*R/100*T/360
080 LET A = P+I
090 PRINT "TIME=";T, "AMOUNT =";A
100 NEXT T
110 IF P ≥ 3000 GOTO 20
120 IF T = 180 GOTO 50
130 END
RUN
```

This example illustrates the use of program and terminal input statements in solving a simple interest problem.

Line 10 constructs a data table consisting of three principal amounts.

Line 20 resets the data pointer to the beginning of the data table after three consecutive READ cycles.

Line 30 requests a keyboard entry for the variable R.

Line 40 permits the operator to terminate program execution. If the INPUT value for R is ten percent, control passes to the END statement.

Line 50 instructs the system to read the data table.

Line 60 establishes a FOR/NEXT loop by incrementing the time period from 30 to 180 days in 30-day STEPs.

Line 70 computes the simple interest I for each value of P at each time-period step (I = PRT).

Line 80 computes the amount for each time period (amount equals principal plus interest).

Line 110 transfers control back to the RESTORE statement after all data values have been read for the input variable R.

At the completion of the FOR/NEXT loop, line 120 returns control to the READ statement where the second and third data values are read.

The following results would be printed if a five percent rate was entered in response to the ?:

```
? 5
TIME= 30                 AMOUNT = 1004.17
TIME= 60                 AMOUNT = 1008.33
TIME= 90                 AMOUNT = 1012.5        First data value
TIME= 120                AMOUNT = 1016.67
TIME= 150                AMOUNT = 1020.83
TIME= 180                AMOUNT = 1025
TIME= 30                 AMOUNT = 2008.33
TIME= 60                 AMOUNT = 2016.67
TIME= 90                 AMOUNT = 2025          Second data value
TIME= 120                AMOUNT = 2033.33
TIME= 150                AMOUNT = 2041.67
TIME= 180                AMOUNT = 2050
TIME= 30                 AMOUNT = 3012.5
TIME= 60                 AMOUNT = 3025
TIME= 90                 AMOUNT = 3037.5        Third data value
TIME= 120                AMOUNT = 3050
TIME= 150                AMOUNT = 3062.5
TIME= 180                AMOUNT = 3075
```

When output printing is completed the system prints another ? and a new rate may be entered.

## DATA

**General Form**

DATA $c_1, c_2, c_3, \ldots, c_n$

where c is a numeric or literal constant.

**Effect**

The DATA statement instructs the system to construct a data table containing the values (i.e., $c_1, c_2, \ldots, c_n$) appearing in the DATA statement line. The data values are entered in the data table in the same order in which they originally appeared in the DATA statement(s). The number of values per DATA statement line is restricted only by the space of the line.

The data table is read when the system encounters a READ statement.

**Example**

```
10 DATA 10,15,17
20 DATA 34E-51,532,3.021
30 DATA 'JOHNSON','SMITH','BROWN'
```

Lines 10 through 30 establish a data table containing the values specified in the three DATA statement lines. The data values are entered in the data table in the same order as they appear in the three DATA statements (refer to the READ statement for an illustration of the data table).

## READ

**General Form**

READ $v_1, v_2, v_3, \ldots, v_n$

where v is a variable.

**Effect**

The READ statement instructs the system to read the data table beginning with variable $v_1$ through $v_n$.

The data table is a one-column table containing data values entered by way of a DATA statement. When a READ statement is encountered in the program it instructs the system to read the data values sequentially. The read position is advanced one data item for each value read. Numeric variables must correspond to numeric data and alphameric variables must correspond to literal data.

Program execution is terminated if a READ statement is executed when insufficient data remains in the data table.

**Example**

```
10 DATA 10,15,17
20 DATA 'ONE','TWO','THREE','FOUR'
•
•
60 READ A,B
70 READ C,D$
80 READ E$,F$,G$
```

|  | Data Table |
|---|---|
| First READ Statement | 10 |
|  | 15 |
| Second READ Statement | 17 |
|  | ONE |
| Third READ Statement | TWO |
|  | THREE |
|  | FOUR |

Lines 10 and 20 establish values for the data table. Line 60 instructs the system to read the first two data values (10 and 15). Lines 70 and 80 read the remaining data values. Note that the alphameric variables (e.g., D$) correspond with the literals (e.g., ONE) contained in the data table.

## RESTORE

**General Form**

RESTORE [c]

where c is a character string.

**Effect**

This statement causes the next READ statement to begin reading at the first DATA element in the program. The character string c may be entered as a comment; it is ignored during compilation and execution.

**Example**

```
100 DATA 20,40,60,80
110 DATA 70,90
.
.
140 READ A,B,C
150 RESTORE
160 READ D,E
```

The DATA statements referenced in lines 100 and 110 establish six data values in the data table. When line 140 is executed, the values 20, 40 and 60 are read into variables A, B and C, respectively, and the data pointer is positioned to read the value 80 at its next request for data.

When the RESTORE statement at line 150 is executed, the pointer is repositioned to the beginning of the data table at value 20. Thus, at line 160 the values 20 and 40 are read into variables D and E respectively.

## INPUT

**General Form**

INPUT $v_1, v_2, v_3, \ldots, v_n$

where v is a variable.

**Effect**

When the INPUT statement is encountered by an executing program, a question mark is printed at the terminal. Data in the form of numeric and/or literal constants may then be entered from the terminal.

The variables specified assume the values of the data in order of entry; the number of items entered must equal the number of variables in the INPUT statement list. Numeric constants must be entered for numeric variables; literal constants must be entered for alphameric variables.

If a literal data entry is not empty or does not contain a comma, the entry need not be bounded by quotation marks; leading blanks are ignored, but embedded blanks are significant.

**Examples**

1.
```
10 INPUT X,Y(X),Z(R+3),C1
.
.
.
90 END
RUN

               15:35 05/04/70   MONDAY   SJ2

?20,15,4,.35


10 INPUT A$,R
.
.
.
90 END
RUN

               15:40 05/04/70   MONDAY   SJ2

? YES,20
```

2.
```
10 REM: INTEREST = PRINCIPAL * RATE * TIME
20 INPUT P,R,T
30 I = P*R/100*T/360
40 A = P + I
50 PRINT 'INTEREST IS $';I, 'AMOUNT IS $';A
60 END
RUN

               15:48    05/11/70  MONDAY      SJ2

? 1000,7.5,90
INTEREST IS $ 18.75               AMOUNT IS $ 1018.75

TIME      0 SECS.
```

Line 10 of Example 2 defines the simple interest equation used in this series of program statements. Line 20, the INPUT statement, requests values for the three variables P, R and T. Line 30 establishes the simple interest equation and line 40 computes the amount A. Line 50 requests printing of the output values for I and A.

In response to the ?, the values for the principal, rate and time are entered from the terminal.

## TERMINAL OUTPUT

- PRINT
- PRINT USING
- IMAGE
- PAUSE

| Statement | Effect | Usage |
|---|---|---|
| PRINT | Causes output printing according to the format specified in the PRINT line. | `PRINT 'VALUE =', V` |
| PRINT USING | Specifies output printing using an Image statement to establish the printing format. | `PRINT USING 80, A` |
| Image | Establishes printing format for the PRINT USING statement. | `:THIS YR TOTAL IS ##` |
| PAUSE | Causes program execution to halt, allowing one line of comments to be entered from the keyboard. | `PAUSE` |

### General Example

```
10 DATA 1000,2000,3000
20 RESTORE
30 INPUT R
35 PAUSE
40 IF R = 10 THEN 140
50 READ P
60 FOR T = 30 TO 180 STEP 30
70 LET I = P*R/100*T/360
80 LET A = P + I
90 PRINT USING 100,T,A
100 :NUMBER OF DAYS IS ###, TOTAL AMOUNT IS $####.##
110 NEXT T
120 IF P ≥ 3000 GOTO 20
130 IF T = 180 GOTO 50
140 END
RUN
```

This is the same example shown on page 38 except that a PRINT USING statement replaces the PRINT statement and a PAUSE statement has been introduced at line 35.

Line 35 causes program execution to pause, permitting a message to be entered from the keyboard. Since the PAUSE statement appears after the INPUT statement, the system will first ask for input and then print the following message:

```
? 5
  PAUSE AT LINE 35
YOU MAY NOW ENTER ONE LINE OF DATA. THIS DATA WILL NOT BE EXECUTED.
```

Line 90 specifies the values to be printed in line 100, the Image statement. The Image statement establishes the printing format as follows:

```
      NUMBER OF DAYS IS 30, TOTAL AMOUNT IS $1004.17
```

Compare this format with the PRINT statement in the original example on page 38:

```
      TIME = 30             AMOUNT = 1004.17
```

## PRINT

### General Form

PRINT $f_1 t_1 f_2 t_2 f_3 t_3 \ldots f_n [t_n]$

where f is an expression, an alphameric variable, a literal constant or null, and t is a comma or a semicolon.

### Effect

This statement causes the program to convert each print field f to a specified output format, print the converted field, and position the carrier according to the terminator character.

Each print line is divided into zones. Two types of print zones are provided: full and packed. A full print zone consists of 18 characters. The type of print zone assigned is specified by the terminator character t. A full print zone is controlled by commas, and a packed print zone is controlled by semicolons. A full and a packed print zone may be combined in one PRINT statement.

If the print field f is an expression, the size of a packed zone is shown below:

| Print Field Length | Packed Zone Length |
|---|---|
| 1-4 characters | 6 characters |
| 5-7 characters | 9 characters |
| 8-10 characters | 12 characters |
| 11-13 characters | 15 characters |
| 14-16 characters | 18 characters |

If the print field is an alphameric variable, the size of a packed print zone is 18 characters minus the number of trailing blanks.

If the print field is a literal constant, the size of a packed print zone equals the size of the converted field.

The f field is printed at the terminal as follows:

1. If the print field is an alphameric variable or a literal constant
   a. and t is a comma with at least 18 spaces remaining on the print line, printing starts at the current carrier position. If the end of the print line is encountered before the print field is exhausted, printing of remaining characters starts on the next print line.
   b. and t is a comma with less than 18 spaces remaining on the print line, printing starts at the beginning of the next print line. If the end of the print line is encountered before the field is exhausted, printing of remaining characters starts on the next line.
   c. and t is a semicolon, printing starts at the current carrier position. If the end of the print line is encountered before the field is exhausted, printing of remaining characters starts on the next line.
2. If the print field is an expression, printing starts at the current carrier position unless the print line does not contain sufficient space to accommodate the value, in which case printing starts at the beginning of the next line.

After the converted print field has been printed, the carrier is positioned as specified by the terminator character.

1. If the print field is an expression or an alphameric variable
   a. and t is a comma, the carrier is moved past any remaining spaces in the full print zone; if the end of the print line is encountered, the carrier is moved to the beginning of the next print line.
   b. and t is a semicolon, the carrier is moved past any remaining spaces in the packed print zone; if the end of the print line is encountered, the carrier is moved to the beginning of the next print line.
   c. and t is omitted following the last print field in the statement, the carrier is moved to the beginning of the next print line.

2. If the print field is a literal constant
   a. and t is a comma, the carrier is moved past any remaining spaces in the full print zone; if the end of the print line is encountered,the carrier is moved to the beginning of the next print line.
   b. and t is a semicolon, the carrier is not moved unless the end of the print line is encountered, in which case the carrier is moved to the beginning of the next print line.
   c. and t is omitted following the last print field in the statement, the carrier is moved to the beginning of the next print line.
3. If the print field is null
   a. and t is a comma, the carrier is moved 18 spaces; if the end of the print line is encountered, the carrier is moved to the beginning of the next print line.
   b. and t is a semicolon, the carrier is moved three spaces; if the end of the print line is encountered,the carrier is moved to the beginning of the next print line.

**Examples**

1.
```
50 PRINT "X= "; 5, -6.78; (X/2+4*Z)
60 PRINT Y$
```

2.
```
100 LET A1 = 100
110 LET A2 = 90
120 LET A3 = 80
130 LET A4 = 70
140 LET A = (A1+A2+A3+A4)/4
150 PRINT "AVERAGE =",A
160 END
RUN
```

In Example 2, the comma in the PRINT line specifies two full print zones, each 18 characters long. The printed result appears as follows:

```
AVERAGE =          85
|__________________||__________________|

  First Print zone   Second Print Zone
  (18 Spaces)         (18 Spaces)
```

If the PRINT line specified a packed print zone (by use of the semicolon)

```
150 PRINT "AVERAGE =;A
```

two packed print zones would then be allocated. The size of the first print zone would be identical to the literal defined in the first print field ("AVERAGE ="). Six spaces would be allocated for the second print zone because the character range of the second print field (A) is one to three digits:

```
|AVERAGE =||85    |

First Print  Second
   Zone      Print
              Zone
```

## PRINT USING

**General Form**

PRINT USING n [, $f_1$, $f_2$, $f_3$, . . . , $f_n$]

where n is the line number of an Image statement and f is an expression, an alphameric variable or a literal constant.

**Effect**

The PRINT USING statement forms one or more print lines according to the format specifications of the Image statement and prints the lines at the terminal. The actual line printing format for each print field f is established by the Image statement referenced in line number n.

If the number of print fields ($f_1$, $f_2$, . . . , $f_n$) contained in the PRINT USING statement exceeds the number of format specifications contained in the Image statement, the carrier returns at the end of the Image statement and the same format specifications are reused for the remaining print fields. This procedure is repeated until the values for all of the print fields are printed.

If the number of print fields in the PRINT USING statement is less than the number of format specifications in the Image statement, the print line is terminated at the first unused format specification.

Each print field is converted to output format as follows:

1. The meaning of an alphameric variable or a literal constant is extracted from the specified string and edited into the print line, replacing all of the elements in the conversion specification (including sign, #, decimal point and !). If the edited string is shorter than the conversion specification, blank padding occurs to the right. If the edited string is longer than the conversion specification, truncation occurs to the right. A null string results in blank padding of the entire conversion specification.
2. An expression is converted in accordance with its conversion specification:
    a. If the conversion specification contains a plus sign and the expression value is positive, a plus sign is edited into the print line.
    b. If the conversion specification contains a plus sign and the expression value is negative, a minus sign is edited into the print line.
    c. If the conversion specification contains a minus sign and the expression value is positive, a blank is edited into the print line.
    d. If the conversion specification contains a minus sign and the expression value is negative, a minus sign is edited into the print line.
    e. If the conversion specification does not contain a sign and the expression value is negative, a minus sign is edited into the print line in front of the first printed digit, and the length of the conversion specification is reduced by one.
    f. The expression value is converted according to the type of its conversion specification:
        (1) I format: the value of the expression is converted to an integer, truncating any fraction.

(2) F format: the value of the expression is converted to a fixed-point number, rounding the fraction or extending it with zeros in accordance with the conversion specification.

(3) E format: the value of the expression is converted to a floating-point number with one decimal digit to the left of the decimal point, rounding the fraction or extending it with zeros in accordance with the conversion specification.

3. If the length of the resultant field is less than or equal to the length of the conversion specification, the resultant field is edited, right-justified, into the print line. If the length of the resultant field is greater than the length of the conversion specification, asterisks are edited into the print line instead of the resultant field.

**Example**

```
10 A = 342.7
20 B = 42.0399
30 PRINT USING 40,A,B
40 : RATE OF LOSS ##### EQUALS ####.## POUNDS
```

```
Results ──► RATE OF LOSS    342 EQUALS     42.04 POUNDS
                             ↑              ↑
                          Value for A   Value for B
```

Line 30 specifies the values to be printed in line 40, the Image statement. The Image statement establishes the format of the print line. Lines 10 and 20 establish the values 342.7 and 42.0399 for the variables A and B respectively.

## Image

**General Form**

$: [c_1 s_1 c_2 s_2 c_3 s_3 \ldots c_n s_n]$

where c is a character string and s is a conversion specification.

**Effect**

This statement specifies a format picture for a single PRINT USING line. The character string c can contain any character other than #, or it can be null. The conversion specification s can specify I, F or E format, or it can be null. A colon immediately following the line number identifies the Image statement.

The conversion specifications for the various formats are as follows:

1. I format consists of an optional sign followed by one or more # characters. For example, ## is an I-format specification.
2. F format consists of an optional sign followed by the optional occurrence of one or more # characters, a decimal point, and the optional occurrence of one or more # characters. There must be at least one # character in the specification. For example, +#.##### is an F-format specification.

3. E format consists of an optional sign followed by one or more # characters, a decimal point, the optional occurrence of one or more # characters, and four ! characters. For example, #.##!!!! is an E-format specification.

**Example**

```
10 A = 10.25
20 B = 20.25 - A
30 C = A + B↑2
40 PRINT USING 50,C
50 :THE VALUE FOR C IS ######
60 PRINT USING 70,C
70 :THE VALUE FOR C IS ####.####
80 PRINT USING 90,C
90 :THE VALUE FOR C IS ####.####!!!!
100 END
RUN

           16:10   05/11/70  MONDAY     SJ2

THE VALUE FOR C IS   110
THE VALUE FOR C IS  110.2500
THE VALUE FOR C IS    1.1025E+02

TIME     0 SECS.
```

This example illustrates the use of the three format specifications for the Image statement. The PRINT USING statements in lines 40, 60 and 80 request printing of the value for C.

The Image statement in line 50 specifies that the value of C be shown as an integer (I format). In line 70, the value for C is converted to a fixed-point number (F format), and in line 90 the value for C is shown in exponential form (E format).

## PAUSE

**General Form**

PAUSE [c]

where c is a character string.

**Effect**

The PAUSE statement causes program execution to halt and the following message to be printed at the terminal:

PAUSE AT LINE n

where n is the line number of the PAUSE statement.

Program execution may be resumed by striking the carrier return or by entering a character string followed by a carrier return. The character string c is entered as a comment; it is ignored during compilation and execution.

**Example**

```
100 P = 1000.00
110 R = 5/100
120 N = 4
130 L = 2
140 I = P*((1+R/N)↑L-1)
150 PAUSE
160 PRINT USING 170,I
170 :COMPOUND INTEREST = $######.##
180 END
RUN

             16:17    05/11/70   MONDAY      SJ2

  PAUSE AT LINE  150
COMP RATE = PRINC*((1+YRLY INT RATE/INT PERIODS PER YR)↑# OF PERIODS $ HELD-1
COMPOUND INTEREST = $     25.16

TIME     0 SECS.
```

The PAUSE statement in the compound interest example is used to define the compound interest equation. If the PAUSE statement was introduced after the PRINT USING statement, the system would have printed the result before halting program execution.

# ARRAY DECLARATIONS AND MATRIX OPERATIONS

- DIM
- MAT GET
- MAT PRINT
- MAT PUT
- MAT READ
- Arithmetic Operations
- Functional Operations

| Statement | Effect | Usage |
|---|---|---|
| DIM | Defines and allocates storage for a one- or two-dimensional array. A matrix must be declared by a DIM statement prior to usage. | `DIM A(4,4), B(15)` |
| Matrix Addition | Replaces the elements of $m_1$ with the sum of the elements of $m_2$ and $m_3$. | `MAT A = B + C` |
| Matrix CON Function | Causes elements of the specified matrix to assume values of all ones. | `MAT A = CON` |
| MAT GET | Reads matrix values from a file. | `MAT GET 10: A` |
| Matrix IDN Function | Causes the specified matrix to assume the form of an identity matrix. | `MAT A = IDN` |
| Matrix INV Function | Replaces each element of $m_1$ with the inverse of $m_2$. | `MAT A = INV B` |

| | | |
|---|---|---|
| Matrix Multiplication | Replaces each element of $m_1$ with the sum of the products of $m_2$ and $m_3$. | `MAT D = E*F` |
| Matrix Multiplication (Scalar) | Multiplies each element of $m_2$ by a given value and places the product in $m_1$. | `MAT A = (5)*C` |
| MAT PRINT | Causes output printing of the matrix elements according to the format specified in the MAT PRINT line. | `MAT PRINT A, B, C` |
| MAT PUT | Writes matrix values into a file. | `MAT PUT 20: C` |
| MAT READ | Places numeric data originally created by a DATA statement into the specified matrices. | `MAT READ A, B, C` |
| Matrix Subtraction | Replaces each element of $m_1$ with the difference of the corresponding elements of $m_2$ and $m_3$. | `MAT E = F - G` |
| Matrix TRN Function | Replaces each element of $m_1$ with the transpose values of $m_2$. | `MAT X = TRN(Y)` |
| Matrix ZER Function | Replaces all elements of the specified matrix with zeros. | `MAT C = ZER` |

**General Examples**

1.
```
10 DIM S(5,1),T(7,1),R(5,1)
20 DATA 5,15,35,65,95
30 DATA 1,3,5,7,11
40 MAT READ S, T(5,1)
50 MAT R = ZER
60 MAT R = S+T
70 MAT PRINT R
80 MAT R = S-T
90 MAT PRINT R
100 MAT R = (5)*R
110 MAT PRINT R
120 END
RUN
```

This example illustrates the use of the DIM statement and six matrix operations in one program listing. The matrix operations are MAT READ, Matrix ZER Function, Matrix Addition, Matrix Subtraction, Matrix Multiplication (scalar) and MAT PRINT.

Line 10, the DIM statement, allocates storage for three separate arrays S, T and R. Although the referenced arrays are declared to have two dimensions, each is essentially a one-dimensional numeric array because the column subscript is 1 (e.g., S(5,1)).

Lines 20 and 30 construct a data table consisting of ten data values.

Line 40 specifies that the data values defined in lines 20 and 30 be read into matrices S and T. Note that matrix T is redimensioned to a 5 by 1 matrix.

|  | S(5,1) |  | T(5,1) |
|---|---|---|---|
| S(1,1) | 5 | T(1,1) | 1 |
| S(2,1) | 15 | T(2,1) | 3 |
| S(3,1) | 35 | T(3,1) | 5 |
| S(4,1) | 65 | T(4,1) | 7 |
| S(5,1) | 95 | T(5,1) | 11 |

Line 50 causes all elements of matrix R (the result table) to assume the value of zero (0).

|  | R(5,1) |
|---|---|
| R(1,1) | 0 |
| R(2,1) | 0 |
| R(3,1) | 0 |
| R(4,1) | 0 |
|  | 0 |

Line 60 adds the corresponding elements of matrices S and T and places the result in matrix R.

|  | R(5,1) |
|---|---|
| R(1,1) | 6 |
| R(2,1) | 18 |
| R(3,1) | 40 |
| R(4,1) | 72 |
| R(5,1) | 106 |

R = S + T

Line 80 subtracts the corresponding elements of matrices S and T and places the result in matrix R.

|  | R(5,1) |
|---|---|
| R(1,1) | 4 |
| R(2,1) | 12 |
| R(3,1) | 30 |
| R(4,1) | 58 |
| R(5,1) | 84 |

R = S - T

Line 100 multiplies each element of matrix R by 5 and places the result in matrix R.

|  | R(5,1) |
|---|---|
| R(1,1) | 20 |
| R(2,1) | 60 |
| R(3,1) | 150 |
| R(4,1) | 290 |
| R(5,1) | 420 |

R = R * 5

Thus, the printed results would be as follows:

$$\left.\begin{array}{r} 6 \\ 18 \\ 40 \\ 72 \\ 106 \end{array}\right\} \text{Result of line 70: MAT PRINT R}$$

```
  4  )
 12  (
 30  >    Result of line 90:  MAT PRINT R
 58  (
 84  )

 20  )
 60  (
150  >    Result of line 110:  MAT PRINT R
290  (
420  )
```

2.
```
10 DIM A(4,4),S(4,1),C(4,1),B(4.4),D(4,4)
20 DATA 1,-2,3,4
30 DATA 3,-1,2,5
40 DATA 2,4,-5,1
50 DATA 4,2,-1,3
60 DATA 4.5,9.5,15,12
70 MAT READ A,C
80 MAT B = INV(A)
90 MAT S = B*C
100 MAT PRINT S
110 MAT D = B*A
120 MAT PRINT D
130 MAT B = IDN
140 MAT PRINT B
150 END
RUN
```

This second example of matrix operations illustrates the use of matrix statements in solving problems in matrix analysis. Four equations are given containing the four unknowns x, y, z and t for which solution values must be determined. The equations are

$$\begin{aligned} x - 2y + 3z + 4t &= 4.5 \\ 3x - y + 2z + 5t &= 9.5 \\ 2x + 4y - 5z + t &= 15 \\ 4x + 2y - z + 3t &= 12 \end{aligned}$$

Line 10, the DIM statement, declares the five arrays A, S, C, B and D. C and S are vectors.

Lines 20 through 50 construct a data table containing the data values assigned to the left-hand side of the equations, and line 60 constructs a data table containing the result values assigned to the right-hand side of the equations.

Line 70 reads the data values assigned to the left-hand side of the equations into matrix A and the values assigned to the right-hand side into vector C.

Lines 80 and 90 compute the values for the unknowns x, y, z and t. Line 80 causes matrix B to be replaced by the inverse of matrix A. Line 90 computes the products of matrix B and the right-hand-side vector C, and places the sum of the resultant products into the solution vector S. Vector S now contains the values for the unknowns x, y, t and z. The computations performed by these two matrix operations may be represented by the equation:

$$\text{Solution} = \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} = A^{-1} \begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \end{pmatrix}$$

Line 110 checks the validity of the solution by multiplying the original left-hand-side matrix A by its calculated inverse matrix B and placing the sum of the products into matrix D. If matrix D assumes the values of an identity matrix (consisting of values close to one in the diagonal and zeros or small numbers in the remaining elements), the solution is valid. Line 120 requests printing of the elements of matrix D.

Line 130 causes matrix B to assume the form of an identity matrix and line 140 requests printing of the identity matrix. These two matrix operations (lines 130 and 140) are incorporated in the program only to illustrate the use of the Matrix IDN Function.

Thus, the printed output would be as follows:

Result of line 100: MAT PRINT S

```
-.500018   ←——— Value for x
 2.00002   ←——— Value for y
-.999968   ←——— Value for z
 3.        ←——— Value for t
```

Result of line 120: MAT PRINT D

```
.999995         2.86102E-06      -4.76837E-06      -9.89437E-06
2.86102E-06      .999999          1.90735E-06       9.53674E-06
3.81470E-06     -1.90735E-06      1.                1.04904E-05
1.78814E-07     -2.38419E-07     -2.98023E-07       1.
```

Result of line 140: MAT PRINT B

```
1               0                0                 0
0               1                0                 0
0               0                1                 0
0               0                0                 1
```

The solution may also be checked by inserting the values found for x, y, z and t into all of the equations. The following program computes the value of the expression $4x + 2y - z + 3t$. The value computed should be 12, which corresponds to the fourth equation.

```
20 X = -.500018
30 Y = 2.00002
40 Z = -.999968
50 T = 3.
60 A = 4*X+2*Y-Z+3*T
70 PRINT 'RESULT VALUE IS';A
80 END
RUN

              16:49    05/07/70   THURSDAY    SJ2

RESULT VALUE IS 11.9999

TIME     0 SECS.
```

## DIM

**General Form**

DIM $a_1$ ($d_{11}$ [,$d_{12}$]), $a_2$ ($d_{21}$ [,$d_{22}$]), . . . , $a_n$($d_{n1}$ [,$d_{n2}$])

where a is an array name and d is an unsigned integer.

**Effect**

The DIM statement explicitly defines one or more arrays and causes allocation of storage space for the named arrays with their specified dimensions. An array is declared to have one dimension (e.g., (10)) when the member is referenced by an array variable with one subscript, and two dimensions (e.g., (10,10)) when the member is referenced by an array variable with two subscripts.

Any number of DIM statements may appear in a program; however, a specific array name can only appear in a single DIM statement.

In matrix operations, a DIM statement must have first defined the matrix prior to its usage. Once a matrix has been defined by the DIM statement, the matrix may be redimensioned by appending two subscripts (enclosed in parentheses and separated by a comma) within the matrix statement. If redimensioning exceeds the number of matrix elements originally declared in the DIM statement, program execution is terminated and an error message printed.

A BASIC program written in short-form arithmetic may have up to 29 numeric arrays (A through Z plus @, # and $) defined in a source program, provided that the sum of the array elements does not exceed 7167 numeric values. For example, DIM A(5,1433) defines a matrix composed of 7165 elements and is perfectly legal provided that matrix A is the only array defined in the program. If, however, matrix A was defined as a 5 by 1792 table resulting in a total of 7168 elements, execution would be terminated and the following error message would be printed at the terminal:

line number: TOO MANY ARRAY ELEMENTS
COMPILATION TERMINATED

Alphameric variables are converted to character string values and take up four and one-half times more array storage than simple numeric values. Thus, the limits for a BASIC program defining alphameric arrays exclusively (e.g., A$(10)) is 1592 elements. For example, DIM A$(1592) defines a one-dimensional array containing 1592 character string values and is legal provided that no other arrays are defined in the program. If alphameric arrays and simple numeric arrays are combined in one BASIC program, the user may compute the array limits by using the ratio of 4½:1. Additional information regarding array limits may be found in Appendix A.

**Example**

```
10 DIM A(10),B(2,3),C(10,50)
•
•
40 DIM D(9)
```

In line 10,

List A is allocated space for 10 elements, A(1) . . . A(10).
Table B is allocated space for 6 elements, B(1,1) . . . B(2,3).
Table C is allocated space for 500 elements, C(1,1) . . . C(10,50).

In line 40, list D is allocated space for 9 elements, D(1) . . . D(9).

## Matrix Addition

**General Form**

MAT $m_1 = m_2 + m_3$

where m is a matrix.

**Effect**

This statement causes each element of the matrix $m_1$ to be replaced by the sum of the corresponding elements of $m_2$ and $m_3$. If the matrices are not conformable, program execution is terminated and an error message is printed at the terminal.

**Example**

```
10 DIM A(10,10),B(10,10),C(10,10)
20 MAT A = B + C
30 FOR I = 1 TO 10
40 FOR J = 1 TO 10
50 LET A(I,J) = B(I,J) + C(I,J)
60 NEXT J
70 NEXT I
•
•
90 END
```

Lines 30 through 60 produce results identical to those of line 20.

## Matrix CON Function

**General Form**

MAT m = CON [$(d_1, d_2)$]

where m is a matrix and d is an expression which is used for redimensioning the matrix.

**Effect**

This statement causes all elements of the specified matrix to assume the value of one (1).

**Examples**

1.
```
20 MAT A = CON
30 MAT B = CON(J,K)
```

2.
```
10 DIM A(3,3)
20 DATA 20,3.4,69,7,10,678,3,.89,389
30 MAT READ A
40 MAT PRINT A
50 MAT A = CON
60 MAT PRINT A
70 END
RUN
                  16:58   05/07/70  THURSDAY    SJ2

 20                 3.4               69

 7                  10                678

 3                  .89               389


 1                  1                 1

 1                  1                 1

 1                  1                 1


TIME     0 SECS.
```

Line 10 allocates storage for a 3 by 3 matrix A. Line 20 constructs a data table containing nine data values. These data values are read into matrix A by line 30, the MAT READ statement.

Line 50, the Matrix CON Function, replaces the current matrix values with all ones (1). Both matrix values are shown in the printed output.

## MAT GET

**General Form**

MAT GET [u:] $m_1$ [($d_{11}$ , $d_{12}$)] , $m_2$ [($d_{21}$ , $d_{22}$)] , . . . , $m_n$ [($d_{n1}$ , $d_{n2}$)]

where u is an expression, m is a matrix, and d is an expression which is used for redimensioning the matrix.

**Effect**

This statement is similar to the GET statement. It allows numeric data to be read into the specified matrices without referencing each member individually. Elements are read by rows from the data file specified by the value of u. The entry of u: is optional; if u: is omitted it is assumed to be 1.

If a MAT GET statement is executed when the specified data file is not active or is assigned as an output file, program execution is terminated.

If a MAT GET statement is executed which causes the data file to be exhausted before a specified matrix is filled, program execution is terminated.

**Examples**

1.
```
20 MAT GET F1: X, Y, Z
```

2. This example is an extension of Example 2 of the MAT PUT statement (page 66).

```
10 OPEN 10, 'ITEMFILE', INPUT
20 DIM A(2,4)
30 MAT GET 10: A(2,3)
40 MAT PRINT A
50 CLOSE 10
60 END
RUN

            10:04   05/08/70   FRIDAY     SJ2

 1                 1.1                 1.2

 1.3               1.4                 1.5


TIME     0 SECS.
```

Line 10 activates the file ITEMFILE as input and assigns it to file number 10. Line 20 declares a 2 by 4 matrix A, and line 30, the MAT GET statement, redimensions the matrix and reads the matrix values originally PUT into the data file. Line 40 requests a printed output for matrix A and line 50 deactivates the data file.

## Matrix IDN Function

**General Form**

MAT m = IDN [($d_1$, $d_2$)]

where m is a matrix and d is an expression which is used for redimensioning the matrix.

**Effect**

The IDN Function causes the specified matrix m to assume the form of an identity matrix. The IDN Function cannot be used to initialize an array which is not symmetrical. Any attempt to initialize a nonsymmetrical array (i.e., one with a different number of rows and columns) with the IDN Function terminates program execution and causes the following error message to be printed at the terminal:

line number: MATRICES NOT CONFORMABLE

Redimensioning a matrix with the IDN Function is permitted. If redimensioning exceeds the number of matrix elements declared in the DIM statement, program execution is terminated and the following error message is printed at the terminal:

line number: INVALID MATRIX DECLARATION

**Examples**

1.
```
20 MAT A = IDN
30 MAT B = IDN (4,4)
```

2.
```
10 DIM A(4,3)
20 DATA 2,4,6,8,10,12,14,16,18,20,22,24
30 MAT READ A
40 MAT PRINT A
50 MAT A = IDN (3,3)
60 MAT PRINT A
70 END
RUN

            10:17   05/08/70  FRIDAY      SJ2

 2                  4                  6

 8                  10                 12

 14                 16                 18

 20                 22                 24


 1                  0                  0

 0                  1                  0

 0                  0                  1


TIME      0 SECS.
```

Line 10, the DIM statement, allocates storage space for a 4 by 3 matrix A. Line 20 constructs a data table containing 12 values. These values are read into matrix A by line 30, the MAT READ statement. Line 50 redimensions matrix A to a 3 by 3 matrix and causes it to assume the form of an identity matrix.

Both the original and identity matrix values are shown in the printed output.

## Matrix INV Function

### General Form

MAT $m_1$ = INV($m_2$)

where m is a matrix.

### Effect

This statement causes the matrix $m_1$ to be replaced by the inverse of matrix $m_2$. The subscripts assigned to matrix $m_1$ and matrix $m_2$ must be identical (e.g., A(3,3), B(3,3)) for execution to occur. If the subscripts are not identical (e.g., A(4,6), B(4,6)), execution is terminated and the following error message is printed at the terminal:

line number: MATRICES NOT CONFORMABLE

Matrix $m_1$ cannot be assigned the same name as matrix $m_2$.

### Examples

1.
```
20 MAT A = INV(B)
```

2.
```
10 DIM A(2,2),B(2,2)
20 DATA 1,2,3,4,5,6,7,8
30 MAT READ A,B
40 MAT PRINT A
50 MAT A =INV(B)
60 MAT PRINT A
70 END
RUN

                 10:20    05/08/70   FRIDAY      SJ2


 1                  2

 3                  4


-4                  3

 3.5               -2.5


TIME     0 SECS.
```

Line 10, the DIM statement, allocates storage for matrix A and matrix B. Line 20 constructs a data table containing eight (8) values. Line 30 reads the data table values into matrices A and B. Line 50 causes matrix A to be replaced by the inverse of matrix B.

The printed output displays both conditions of matrix A.

## Matrix Multiplication

### General Form

MAT $m_1 = m_2 * m_3$

where m is a matrix.

### Effect

This statement causes matrix $m_1$ to be replaced by the sum of the products of matrices $m_2$ and $m_3$. If the matrices are not conformable, program execution is terminated and an error message is printed at the terminal.

The rules of matrix multiplication are

$$m_{1ij} = m_{2j1} * m_{3i1} + m_{2j2} * m_{3i2} \dots + m_{2jn} * m_{3jn}$$

where

i = row
j = column
n = number of columns in $m_2$ and number of rows in $m_3$.

Matrix $m_1$ cannot be assigned the same name as matrices $m_2$ and $m_3$.

### Examples

1.
```
20 MAT Q = P*R
```

2.
```
10 DIM A(4,1),B(4,4),C(4,1)
20 DATA 2,4,6,8
30 DATA 1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
35 MAT READ A,B
40 MAT PRINT A
50 MAT PRINT B
60 MAT C = B*A
70 MAT PRINT C
80 END
RUN
```

```
           10:26    05/08/70  FRIDAY      SJ2

2  A(1,1)
4  A(2,1)
6  A(3,1)
8  A(4,1)

1  B(1,1)          3  B(1,2)          5  B(1,3)          7  B(1,4)
9  B(2,1)          11 B(2,2)          13 B(2,3)          15 B(2,4)
17 B(3,1)          19 B(3,2)          21 B(3,3)          23 B(3,4)
25 B(4,1)          27 B(4,2)          29 B(4,3)          31 B(4,4)

100   C(1,1)
260   C(2,1)
420   C(3,1)
580   C(4,1)

TIME      0 SECS.
```

Line 10 allocates storage space for matrices A, B and C. Lines 20 and 30 construct a data table containing 20 values. Line 35 reads the data values into matrices A and B, and lines 40 and 50 request printing of the two matrices.

Line 60 computes the products of matrices A and B and places the sum of the products into matrix C. Line 70 requests printing of matrix C. The solution used to compute the first value of 100 in matrix C is

$$B(1,1) * A(1,1) + B(1,2) * A(2,1) + B(1,3) * A(3,1) + B(1,4) * A(4,1) = 100$$

or

$$(1 * 2) + (3 * 4) + (5 * 6) + (7 * 8) = 100$$

## Matrix Multiplication (Scalar)

**General Form**

MAT $m_1$ = (x) * $m_2$

where m is a matrix and x is an expression.

**Effect**

This statement causes each element of matrix $m_1$ to be replaced by each corresponding element of matrix $m_2$ multiplied by the expression x. The expression x is evaluated before any scalar multiplication. If the matrices are not conformable, program execution is terminated.

**Examples**

1.
```
20  MAT A =(A(3,2))*D
```

2.
```
10 DIM A(2,2)
20 DATA 4,4,4,4
30 MAT READ A
40 MAT PRINT A
50 MAT A = (5)*A
60 MAT PRINT A
70 END
RUN
                10:32    05/08/70   FRIDAY      SJ2

 4                 4

 4                 4



 20                20

 20                20



TIME      0 SECS.
```

Line 10 allocates storage for a 2 by 2 matrix. Line 20 constructs a data table containing four values and line 30 reads the data values into matrix A. Line 50 multiplies each element of matrix A by 5 and places the product back into matrix A.

Both conditions of matrix A are shown in the printed output.

## MAT PRINT

**General Form**

MAT PRINT $m_1 t_1 m_2 t_2 m_3 t_3 \ldots m_n [t_n]$

where m is a matrix and t is a comma or semicolon.

**Effect**

This statement causes each element of each specified matrix to be converted to a specified output format and then printed. After the element has been printed, the carrier is positioned as specified by the terminator character t.

The matrix is printed in order by rows. All of the elements of a row are printed with single line spacing on as many print lines as are required. A blank print line is used to separate rows. Printing of the first element of a row always starts at the beginning of a new print line.

The rules for printing as described under the PRINT statement apply to the MAT PRINT statement; however, literal strings are not allowed, and an omitted final terminator character is treated as a comma.

**Example**

```
20 MAT PRINT A,B,C
```

Refer to other matrix operations in this section for additional examples of the MAT PRINT statement.

## MAT PUT

**General Form**

MAT PUT [u:] $m_1, m_2, m_3, \ldots m_n$

where u is an expression and m is a matrix.

**Effect**

This statement causes the specified matrices to be written on an output data file without referencing each member individually. Elements are written by row on the data file specified by the truncated integer value of u. If u: is omitted, the value 2 is assumed.

If a MAT PUT statement is executed when the specified data file is not active or is assigned as an input file, program execution is terminated. If a MAT PUT statement is executed which causes the size of the data file to be exceeded, program execution is terminated.

**Examples**

1. 
```
20 MAT PUT F1: X, Y, Z
```

2.

```
10 OPEN 10, 'ITEMFILE', OUTPUT
20 DIM A(2,4)
30 DATA 1.00,1.10,1.20,1.30,1.40,1.50,1.60,1.70
40 MAT READ A
50 MAT PUT 10:A
60 CLOSE 10
70 END
RUN

                09:56   05/08/70  FRIDAY     SJ2


TIME      0 SECS.
```

Line 10 activates ITEMFILE as output and assigns it to file number 10. Line 20 allocates storage for a 2 by 4 matrix A. Line 30 constructs a data table consisting of eight data values; these values are read into matrix A by line 40.

Line 50, the MAT PUT statement, causes matrix A to be written onto the output file ITEMFILE. Line 60 deactivates the data file. Refer to the MAT GET statement (page 59) for a further illustration of this example.

## MAT READ

**General Form**

MAT READ $m_1$ $[(d_{11}, d_{12})]$, $m_2$ $[(d_{21}, d_{22})]$, . . . , $m_n$ $[(d_{n1}, d_{n2})]$

where m is a matrix and d is an expression which is used for redimensioning the matrix.

**Effect**

This statement allows numeric data to be read into the specified matrices without individually referencing each member. Elements are read by row from a table created by the DATA statements. If the data table is exhausted before a specified matrix is filled, program execution is terminated.

**Example**

```
20 MAT READ A(4,3),B,C(4,5)
```

Refer to other matrix operations in this section for additional examples of the MAT READ statement.

## Matrix Subtraction

**General Form**

MAT $m_1 = m_2 - m_3$

where m is a matrix.

**Effect**

This statement causes each element of matrix $m_1$ to be replaced by the difference of the corresponding elements of $m_2$ and $m_3$. If the matrices are not conformable, program execution is terminated.

**Examples**

1.

```
20 MAT D = A - B
```

2.

```
10 DIM A(1,3),B(1,3),C(1,3)
20 DATA 2.4,3.8,33,14,43,1.9
30 MAT READ A,B
40 MAT C = B - A
50 MAT PRINT B,A,C
60 END
RUN
                10:42    05/08/70   FRIDAY      SJ2

 14                 43                 1.9 ←—Matrix B


 2.4                3.8                33  ←— Matrix A


 11.6               39.2              -31.1 ←—Matrix C


TIME     0 SECS.
```

Line 10 allocates storage for the three matrices A, B and C. Line 20 constructs a data table containing six data values, and line 30 reads the data values into matrices A and B.

Line 40 causes each element of matrix C (in this case, zeros) to be replaced by the difference of the corresponding elements of matrices B and A. Line 50 requests printing of the output values for all three matrices.

It would also have been permissible to subtract matrix A from matrix B and place the output values into matrix A.

## Matrix TRN Function

**General Form**

MAT $m_1$ = TRN($m_2$)

where m is a matrix.

**Effect**

This statement causes matrix $m_1$ to be replaced by the transpose of matrix $m_2$. If the matrices are not conformable, program execution is terminated.

Matrix $m_1$ cannot be the same as $m_2$.

**Examples**

1.
```
20 MAT D = TRN(X)
```

2.
```
10 DIM A(3,3),B(3,3)
20 DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18
30 MAT READ A,B
40 MAT PRINT A,B
50 MAT A = TRN(B)
60 MAT PRINT A
70 END
RUN

                10:48    05/08/70   FRIDAY      SJ2

1                2                3
4                5                6     Matrix A
7                8                9

10               11               12
13               14               15    Matrix B
16               17               18

10               13               16
11               14               17    Matrix A
12               15               18

TIME      0 SECS.
```

Line 10 allocates storage for matrices A and B. Line 20 constructs a data table containing 18 values and line 30 reads the values into matrices A and B. Line 50 causes each element of matrix A to be replaced by the transpose of matrix B.

For example:

A(1,1) = B(1,1), A(1,2) = B(2,1), A(1,3) = B(3,1), A(2,1) = B(1,2)

and so forth.

The values assigned to matrix B and both conditions of matrix A are shown in the printed output.

## Matrix ZER Function

**General Form**

MAT m = ZER [($d_1$, $d_2$)]

where m is a matrix and d is an expression which is used for redimensioning the matrix.

**Effect**

This statement causes all elements of the specified matrix to assume the value zero (0).

**Examples**

1.
```
20 MAT A = ZER
30 MAT B = ZER(4,8)
```

2.
```
10 DIM A(2,3)
20 DATA 13,3,1.8,481,38,.038
30 MAT READ A
40 MAT PRINT A
50 MAT A = ZER(2,2)
60 MAT PRINT A
70 END
RUN
                    10:52    05/08/70   FRIDAY      SJ2

 13                  3                  1.8

 481                 38                 3.80000E-02


 0                   0

 0                   0



TIME      0 SECS.
```

Line 10 allocates storage for a 2 by 3 matrix A. Line 20 constructs a data table containing six data values, and line 30 reads the values into matrix A. Line 50 redimensions the matrix and causes all elements to assume the value of zero. Both conditions of matrix A are displayed in the printed output.

## SUBROUTINES AND USER FUNCTIONS

- DEF
- GOSUB
- RETURN

| Statement | Effect | Usage |
|---|---|---|
| DEF | Defines an expression in a shorter form. | `DEF FNA(X) = 3*Y↑2` |
| GOSUB | Transfers control to a subroutine. | `GOSUB 180` |
| RETURN | Transfers control to the statement immediately following GOSUB. | `RETURN` |

## General Example

```
50 GOSUB 100
60 DEF FNA(X)=2*X↑2+14
70 IF FNA(X)=85 THEN 200
80 GOTO 380
.
.
100 INPUT A$,A,B
.
.
190 RETURN
200 GOSUB 240
.
.
380 END
```

Line 50 transfers control to a subroutine beginning at line 100. Lines 100 through 190 comprise the subroutine referred to by the first GOSUB statement.

Line 190 returns control to line 60, the statement following GOSUB. Line 60 is a DEF statement defining the user function FNA(X). If the function value is 85, control is transferred to a second subroutine beginning at line 200; otherwise, control is passed to the END statement.

## DEF

### General Form

DEF FNz(v) = x

where z is a letter, v is a simple numeric variable, and x is an expression.

### Effect

The DEF statement may be used to define a mathematical equation (expression) in a shorter form, insert new values into an existing expression, and/or alter an existing expression.

This statement defines the evaluation of the user function FNz at the time of execution. The function is evaluated by substituting a specified user expression for each occurrence of the dummy variable v into the expression x, and then evaluating x.

The expression x specifies the computation to be performed in order to arrive at the function value. A function may reference another function if it does not directly or indirectly reference itself.

A DEF statement may appear anywhere in the program since it is executed only as a result of a reference to the function which it defines.

**Example**

```
70 DEF FNB(X)=5*X↑2+27
80 DEF FNA(X)=FNB(X)+X↑3

140 LET R=FNA(Z)+23
```

Line 140 is equivalent to R = ( (5*Z↑2+27) + Z↑3) + 23.

## GOSUB

**General Form**

GOSUB n

where n is a line number.

**Effect**

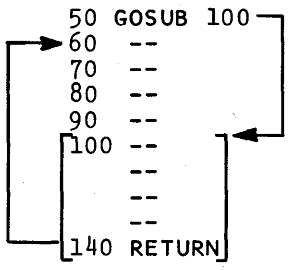
The GOSUB statement is used to enter a subroutine in a program. This subroutine capability is a convenient tool when a particular operation within a program is very repetitive.

When the GOSUB statement is executed, control transfers to the line number indicated by n.

When a RETURN statement is executed, control transfers to the statement immediately following the last GOSUB executed. Exit from a subroutine must be made with the RETURN statement.

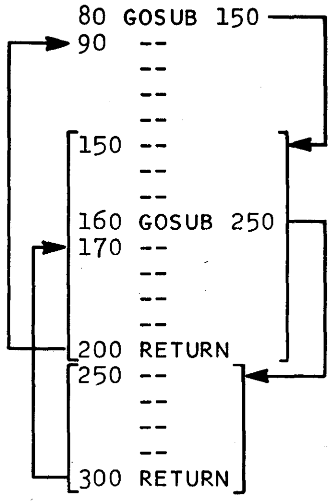
**Examples**

1.



```
50 GOSUB 100
60 --
70 --
80 --
90 --
100 --
    --
    --
    --
140 RETURN
```

2.



```
80 GOSUB 150
90 --
   --
   --
   --
150 --
    --
    --
160 GOSUB 250
170 --
    --
    --
    --
200 RETURN
250 --
    --
    --
    --
300 RETURN
```

## RETURN

**General Form**

RETURN [c]

where c is a character string.

**Effect**

RETURN provides the means of exiting from a subroutine. RETURN transfers control to the statement following the last GOSUB executed.

More than one GOSUB statement may be executed before a RETURN statement is executed, but when a RETURN statement is executed there must be at least one active GOSUB. The character string c may be entered as a comment; it is ignored during compilation and execution.

**Example**

(See GOSUB examples on facing page.)

## DATA FILE INPUT/OUTPUT

- OPEN
- PUT
- GET
- RESET
- CLOSE

| Statement | Effect | Usage |
|---|---|---|
| OPEN | Activates a data file previously named by the FILE command. | `OPEN 10, 'AFILE'` |
| PUT | Writes data into a file. | `PUT 10: A, B, C` |
| GET | Reads data from a file. | `GET 10: A, B, C` |
| RESET | Restores file pointer to the first data item. | `RESET 10` |
| CLOSE | Deactivates a data file. | `CLOSE 10` |

## General Example

Two program listings are illustrated in this example; the first program creates a data file containing prices for various items and the second program uses this data file to perform further computations to arrive at an average price per item.

### Writing Data into a File

```
FILE ITEMFILE
READY

10 OPEN 10, 'ITEMFILE', OUTPUT
20 INPUT A$, A,B,C,D
30 PUT 10: A$,A,B,C,D
40 IF A$ ≠ 'BEARING' THEN 20
50 CLOSE 10
60 END
RUN

             14:30    05/11/70   MONDAY      SJ2

? GASKET,1.10,1.15,1.20,1.40
? BUSHING,1.65,1.70,1.85,2.00
? BEARING,2.00,2.20,2.30,2.40

TIME     0 SECS.
```

The FILE command allocates storage space for a file named ITEMFILE. Line 10 assigns ITEMFILE to file number 10 and defines it as an output file. Line 20 requests input values for the variables A$, A, B, C and D, and line 30 writes the data values into the file.

Line 40 tests the A$ value against BEARING, the final value. If the keyed value was not BEARING, control returns to line 20 and a new set of values is requested. When BEARING and its associated data values are keyed, the file is closed and the program ends.

### Reading Data from a File

```
10 OPEN 10, 'ITEMFILE', INPUT
20 GET 10: A$,A,B,C,D
30 LET A1 = (A+B+C+D)/4
40 PRINT USING 50, A$,A1
50 :AVERAGE PRICE FOR A ######## IS $#.##
60 GOTO 20
70 CLOSE 10
80 END
RUN

             14:34    05/11/70   MONDAY      SJ2

AVERAGE PRICE FOR A GASKET   IS $1.21
AVERAGE PRICE FOR A BUSHING  IS $1.80
AVERAGE PRICE FOR A BEARING  IS $2.22

LINE    20:    END OF FILE

TIME     0 SECS.
```

Line 10 activates the file ITEMFILE, assigns it to file number 10 and defines it as an input file. Line 20 reads the data values, and line 30 computes the average price for each item read. Line 40 specifies the values to be printed in line 50, the Image statement. Line 60 returns control to line 20 where a new item is read and the average price is computed. The loop is maintained until the file is exhausted and the END OF FILE message is printed.

## OPEN

### General Forms

OPEN u, f, INPUT
OPEN u, f, OUTPUT

where u is an expression and f is an alphameric variable or a literal constant.

### Effect

This statement causes the data file named by f to be assigned to the file number specified by u. It sets the status of the file to active, resets the file pointer to the beginning of the data file, and specifies the mode of data transfer (input or output).

If the truncated integer value of u is less than 1 or greater than 255, program execution is terminated.

If the file named by u or f in the OPEN statement is currently active, that file is closed prior to opening the specified file. If an attempt is made to open more than four concurrent data files, program execution is terminated. (See *CLOSE.*)

The name specified by f must correspond to a valid file name.

### Examples

1.
```
100 OPEN F1,'SYSIN', INPUT
100 OPEN 1, A$, OUTPUT
```

2.
```
10 OPEN F1, A$, INPUT
.
.
.
90 END
```

Line 10, the OPEN statement, activates the data file A$, defines it as an input file and assigns it to file number F1.

## PUT

### General Form

PUT [u:] $x_1, x_2, x_3, \ldots, x_n$

where u is an expression and x is an expression, alphameric variable or literal constant.

### Effect

The PUT statement is used to write data onto a file previously called out by an OPEN statement, and it causes the next n values of x to be written out on the data file specified by u.

The file specified by u in the PUT statement must be identical to the file number specified in the OPEN statement; otherwise, program execution is terminated when the system attempts to execute the PUT statement. The value assigned to u is truncated to an integer and may be a numeric constant, variable or an arithmetic expression.

The specification of u: may be omitted in the PUT statement *only* if the file specified in the OPEN statement is assigned the integer of 2. A u value must be assigned in all other cases.

The PUT statement may be used to write data onto an output file only. Any attempt to PUT data onto an input file terminates program execution and causes an error message to be printed at the terminal. Program execution is also terminated if a PUT statement attempts to exceed the size of the data file.

**Examples**

1.
```
100 PUT F1: Z3,F*A-7,A,C,W$
110 PUT 2: 'DATA','STAT','LOG1'
```

2.
```
100 OPEN 15, 'DATA1',OUTPUT
110 A = 3
120 B = 4
130 C = 5
140 PUT 15: A,B,C
150 CLOSE 15
160 END
```

Line 100 activates a file called DATA1, defines it as an output file and assigns it to file number 15.

Lines 110 through 130 assign data values to variables A, B and C.

Line 140 writes the values for A, B and C into file 15.

Line 150 deactivates file 15.

## GET

**General Form**

GET [u:] $v_1, v_2, v_3, \ldots, v_n$

where u is an expression and v is a variable.

**Effect**

The GET statement is used to read data from an input file previously called out by an OPEN statement. The variables v are set to the next n data values read from the file specified by u.

The input file specified by u in the GET statement must be identical to the file number assigned in the OPEN statement; otherwise, program execution is terminated and an error message is printed at the terminal. The value assigned to u is truncated to an integer and may be a numeric constant, variable or arithmetic expression.

The specification of u: may be omitted in the GET statement only if the file specified in the OPEN statement is assigned the integer of 1. A u value must be assigned in all other cases.

The variables v will be replaced by the input data in the order of specification. That is, the first v will be set equal to the first input from the data file; the second v will be set equal to the second data input; and so forth. This continues until all of the variables specified in the data list have been filled.

If any variable v is a numeric variable, the corresponding data input item must also be numeric; alphameric variables must correspond with literal data input.

The GET statement may be used to read data from an input file only. Any attempt to read data from an output file terminates program execution and causes an error message to be printed at the terminal. Program execution is also terminated if insufficient data remains in the input file.

**Examples**

1.
```
100 GET A,B,C
110 GET F1: D(I,J),A$
```

2.
```
100 OPEN 15, 'DATA1', INPUT
110 GET 15: A,B,C
.
.
.
140 PRINT A,B,C
150 CLOSE 15
160 END
```

This example accesses the same file illustrated in the PUT statement (see page 77).

Line 100 activates file DATA1, assigns it to file number 15 and defines it as an input file.

Line 110 reads the data values originally PUT into the file in the program listing shown in the PUT statement example. If desired, these data values may be used for further computations in the current program.

Line 140 prints the data values. Line 150 deactivates file number 15.

## RESET

### General Form

RESET [$u_1, u_2, u_3, \ldots, u_n$]

where u is an expression.

### Effect

This statement causes the data file specified by the value of u to be reset to the beginning of the file. Any subsequent GET or PUT statement references the first item in the file.

If u is omitted, the value 1 is assumed.

If the specified data file is not active, the RESET command is ignored.

### Examples

1.
```
100 RESET F1, F2, F3
200 RESET 200
```

2.
```
100 OPEN 15, 'DATA1', INPUT
110 GET 15: A,B,C
.
.
150 RESET 15
```

Line 110 reads the first three data items and assigns them to the variables A, B and C. Line 150 restores the file pointer to the first data item A.

## CLOSE

### General Form

CLOSE [$u_1, u_2, u_3, \ldots, u_n$]

where u is a constant, variable or expression.

### Effect

The CLOSE statement causes the data file specified by the file number u to be removed from the list of active (i.e., OPENed) data files and placed back into the user's library. If the specified data file is not active, the system will ignore the CLOSE statement.

When the system encounters an END statement (signifying program termination), all currently active data files are closed automatically regardless of whether a CLOSE statement is included in the program.

The entry of u may be a numeric constant, variable or arithmetic expression. The value of u is truncated to an integer.

**Examples**

1.
```
100 CLOSE 10
200 CLOSE 12, 14, A1
```

2.
```
10 OPEN F1, A$, INPUT
•
•
90 CLOSE F1
```

Line 90 removes file number F1 from working storage and places it back into the user's library.

# APPENDIX A: PROGRAM LIMITS

## USER PROGRAM LIMITS

The limits of a user program are as follows:

| Program Element | Limit |
|---|---|
| Statement lines | 800 [1] |
| Characters (program size) | 29,176 [1] |
| Number of Image statements | 25 |
| Number of FOR loops: | |
| Program limit | 80 |
| Nest limit | 15 |
| Number of function references and GOSUBs per nest | 47 [2] |
| Number of files open at once | 4 |
| Number of storage units per file | 250 |
| Number of bytes per storage unit | 3,440 |
| Array limits: | |
| BASIC short-form: | |
| Maximum numeric elements | 7,167 |
| Maximum alphameric elements | 1,592 |
| BASIC long-form: | |
| Maximum numeric elements | 3,583 |
| Maximum alphameric elements | 1,592 |

[1] **Limit is determined by whichever limit is reached first.**

[2] **Limit = x + 2y = 47 where x is the number of nested GOSUBs and y is the number of nested functions.**

## INTRINSIC FUNCTION LIMITS

The following table gives the allowable limits for the arguments to the intrinsic functions.

| Function | Valid Arguments (Minimum < x < Maximum) | | | |
|---|---|---|---|---|
| | Short-Form Arithmetic | | Long-Form Arithmetic | |
| | Min. Value | Max. Value | Min. Value | Max. Value |
| SIN(x) | $-PI * 2^{18}$ | $PI * 2^{18}$ | $-PI * 2^{50}$ | $PI * 2^{50}$ |
| COS(x) | $-PI * 2^{18}$ | $PI * 2^{18}$ | $-PI * 2^{50}$ | $PI * 2^{50}$ |
| TAN(x) | $-PI * 2^{18}$ | $PI * 2^{18}$ | $-PI * 2^{50}$ | $PI * 2^{50}$ |
| COT(x) | $-PI * 2^{18}$ | $PI * 2^{18}$ | $-PI * 2^{50}$ | $PI * 2^{50}$ |
| SEC(x) | $-PI * 2^{18}$ | $PI * 2^{18}$ | $-PI * 2^{50}$ | $PI * 2^{50}$ |
| CSC(x) | $-PI * 2^{18}$ | $PI * 2^{18}$ | $-PI * 2^{50}$ | $PI * 2^{50}$ |
| ASN(x) | -1 | 1 | -1 | 1 |
| ACS(x) | -1 | 1 | -1 | 1 |
| ATN(x) | -1E75 | 1E75 | -1E75 | 1E75 |
| HSN(x) | -174.673 | 174.673 | -174.673 | 174.673 |
| HCS(x) | -174.673 | 174.673 | -174.673 | 174.673 |
| HTN(x) | -1E75 | 1E75 | -1E75 | 1E75 |
| DEG(x) | -1E75 | 1E75 | -1E75 | 1E75 |
| RAD(x) | -1E75 | 1E75 | -1E75 | 1E75 |
| EXP(x) | -180.218 | 174.673 | -180.218 | 174.673 |
| ABS(x) | -1E75 | 1E75 | -1E75 | 1E75 |
| LOG(x) | 0 | 1E75 | 0 | 1E75 |
| LTW(x) | 0 | 1E75 | 0 | 1E75 |
| LGT(x) | 0 | 1E75 | 0 | 1E75 |
| SQR(x) | 0 | 1E75 | 0 | 1E75 |
| RND(x) | -1E75 | 1E75 | -1E75 | 1E75 |
| INT(x) | -1E75 | 1E75 | -1E75 | 1E75 |
| SGN(x) | -1E75 | 1E75 | -1E75 | 1E75 |

# APPENDIX B: DIAGNOSTIC ERROR MESSAGES

## COMPILATION ERROR MESSAGES

Compilation error messages are issued by the CALL/360: BASIC language processor while the program is being translated or prepared for execution. A line number is printed before any message pertaining to a particular line.

Compilation errors can be classified as syntax errors (errors in the construction of a statement), program structure errors (errors in the ordering and relationship of statement lines), or program limit errors. If any compilation error occurs the program is not executed. The language processor generally continues to scan the rest of the program for additional errors. If the error involves a program limit, compilation is usually terminated. Only one error per statement is detected for a particular compilation.

The compilation error messages are listed alphabetically.

| | |
|---|---|
| Message: | END STATEMENT MISSING |
| Cause: | The program does not contain an END statement. |
| Action: | Compilation is terminated; execution is inhibited. |

| | |
|---|---|
| Message: | EXPRESSION TOO COMPLEX |
| Cause: | The line contains an expression requiring too much work space to compile or too many temporary storage locations to compute. |
| Action: | Compilation is continued; execution is inhibited. |

| | |
|---|---|
| Message: | FOR/NEXT LOOP INCOMPLETE |
| Cause: | The program contains at least one incomplete FOR loop. |
| Action: | Compilation is terminated; execution is inhibited. |

| | |
|---|---|
| Message: | FOR/NEXT NESTED INCORRECTLY |
| Cause: | A NEXT statement does not match the preceding FOR statement. |
| Action: | Compilation is continued; execution is inhibited. |

| | |
|---|---|
| Message: | FOR/NEXT NESTED TOO DEEPLY |
| Cause: | The program contains more than 15 nested FOR loops. |
| Action: | Compilation is terminated; execution is inhibited. |

Message: FOR/NEXT OUT OF SEQUENCE

Cause: A NEXT statement appears at a point where no incomplete FOR loop exists.

Action: Compilation is continued; execution is inhibited.

Message: INVALID ARRAY DECLARATION

Cause: An array name appears in a DIM statement after the name has been implicitly or explicitly declared.

Action: Compilation is continued; execution is inhibited.

Message: INVALID ARRAY REFERENCE

Cause: The line contains an array variable with a different number of subscripts than the first reference to the array.

Action: Compilation is continued; execution is inhibited.

Message: INVALID LITERAL CONSTANT

Cause: The line contains a literal constant without a final boundary character.

Action: Compilation is continued; execution is inhibited.

Message: INVALID MATRIX OPERATION

Cause: Matrix inversion or transposition in place has been attempted, or matrix multiplication has been specified where the product matrix is the same as a multiplier or multiplicand matrix.

Action: Compilation is continued; execution is inhibited.

Message: INVALID MATRIX REFERENCE

Cause: The line contains a matrix reference to an undefined or a one-dimensional array.

Action: Compilation is continued; execution is inhibited.

Message: INVALID NUMERIC CONSTANT

Cause: The line contains a numeric constant whose absolute value is greater than 1E+75 or less than 1E–78, and/or the constant has an incorrect syntax.

Action: Compilation is continued; execution is inhibited.

Message: INVALID USER FUNCTION

Cause: A user function has been defined more than once.

Action: Compilation is continued; execution is inhibited.

| | |
|---|---|
| Message: | OBJECT PROGRAM TOO LARGE |
| Cause: | The object program exceeds the maximum storage space allowed. |
| Action: | Compilation is terminated; execution is inhibited. |
| Message: | STATEMENTS FOLLOWING END |
| Cause: | The END statement appears before the final statement in the program after the statements are sorted. |
| Action: | Compilation is terminated; execution is inhibited. |
| Message: | SYNTAX ERROR IN EXPRESSION |
| Cause: | The line does not contain a valid expression where one is expected. |
| Action: | Compilation is continued; execution is inhibited. |
| Message: | SYNTAX ERROR IN STATEMENT |
| Cause: | The line contains an error in the construction of the statement. |
| Action: | Compilation is continued; execution is inhibited. |
| Message: | SYSTEM ERROR HAS OCCURRED |
| Cause: | A language processor error has occurred during the compilation process. |
| Action: | Compilation is terminated; execution is inhibited. |
| Message: | TOO MANY ARRAY ELEMENTS |
| Cause: | The space required for array storage exceeds the maximum allocation. |
| Action: | Compilation is terminated; execution is inhibited. |
| Message: | TOO MANY FOR/NEXT LOOPS |
| Cause: | The program contains more than 80 FOR loops. |
| Action: | Compilation is terminated; execution is inhibited. |
| Message: | TOO MANY IMAGE STATEMENTS |
| Cause: | PRINT USING statements reference more than 25 Image statements. |
| Action: | Compilation is terminated; execution is inhibited. |
| Message: | TOO MANY STATEMENT LINES |
| Cause: | The program contains more than 800 statement lines. |
| Action: | Compilation is terminated; execution is inhibited. |

Message: TOO MANY UNDEFINED LINE NUMBERS

Cause: More than ten undefined line numbers have been referenced.

Action: Compilation is terminated; execution is inhibited.

Message: TOO MANY VARIABLES OR CONSTANTS

Cause: The space required to store the variables and constants exceeds the maximum allocation.

Action: Compilation is terminated; execution is inhibited.

Message: VARIABLE LIST TOO LARGE

Cause: The line contains a multiple LET statement that has too many variables to the left of the equal sign.

Action: Compilation is continued; execution is inhibited.

## EXECUTION ERROR MESSAGES

When a program error is detected during program execution, a message is printed and execution is terminated. All messages at run time are preceded by the line number of the statement being executed at the time of error, with the exception of those errors where a line number would be irrelevant.

Message: ATTEMPT TO WRITE TO INPUT ON LAST WRITE

Cause: An attempt was made to write to an input data file.

Message: *DIRECTORY IS NOT PRESENT

Cause: An attempt was made to open a data file from the *Directory when there is no directory present.

Message: END OF DATA

Cause: A READ statement has been executed with insufficient data in the data table.

Message: END OF FILE

Cause: A GET or MAT GET statement has been executed with insufficient data in the input file.

Message: ERROR IN ACS FUNCTION . . . ARGUMENT TOO LARGE

Cause: The ACS function has been called using an argument whose magnitude is greater than one (1).

| | |
|---|---|
| Message: | ERROR IN ASN FUNCTION . . . ARGUMENT TOO LARGE |
| Cause: | The ASN function has been called using an argument whose magnitude is greater than one (1). |
| Message: | ERROR IN COS FUNCTION . . . ARGUMENT TOO LARGE |
| Cause: | The COS function has been called using an argument whose short-form magnitude is equal to or greater than PI*$2^{18}$ or whose long-form magnitude is equal to or greater than PI*$2^{50}$. |
| Message: | ERROR IN COT FUNCTION . . . ARGUMENT TOO LARGE |
| Cause: | The COT function has been called using an argument whose short-form magnitude is equal to or greater than PI*$2^{18}$ or whose long-form magnitude is equal to or greater than PI*$2^{50}$. |
| Message: | ERROR IN COT FUNCTION . . . INFINITE VALUE |
| Cause: | The COT function has been called using an argument that causes the cotangent to approach infinity. |
| Message: | ERROR IN CSC FUNCTION . . . ARGUMENT TOO LARGE |
| Cause: | The CSC function has been called using an argument whose short-form magnitude is equal to or greater than PI*$2^{18}$ or whose long-form magnitude is equal to or greater than PI*$2^{50}$. |
| Message: | ERROR IN CSC FUNCTION . . . INFINITE VALUE |
| Cause: | The CSC function has been called using an argument that causes the cosecant to approach infinity. |
| Message: | ERROR IN EXP FUNCTION . . . ARGUMENT TOO LARGE |
| Cause: | The EXP function has been called using an argument whose magnitude is greater than 174.673. |
| Message: | ERROR IN HCS FUNCTION . . . ARGUMENT TOO LARGE |
| Cause: | The HCS function has been called using an argument whose magnitude is greater than 174.673. |
| Message: | ERROR IN HSN FUNCTION . . . ARGUMENT TOO LARGE |
| Cause: | The HSN function has been called using an argument whose magnitude is greater than 174.673. |

Message: ERROR IN LGT FUNCTION . . . ARGUMENT ZERO OR NEGATIVE

Cause: The LGT function has been called using an argument whose value is equal to or less than zero.

Message: ERROR IN LOG FUNCTION . . . ARGUMENT ZERO OR NEGATIVE

Cause: The LOG function has been called using an argument whose value is equal to or less than zero.

Message: ERROR IN LTW FUNCTION . . . ARGUMENT ZERO OR NEGATIVE

Cause: The LTW function has been called using an argument whose value is equal to or less than zero.

Message: ERROR IN SEC FUNCTION . . . ARGUMENT TOO LARGE

Cause: The SEC function has been called using an argument whose short-form magnitude is equal to or greater than PI*$2^{18}$ or whose long-form magnitude is equal to or greater than PI*$2^{50}$.

Message: ERROR IN SEC FUNCTION . . . INFINITE VALUE

Cause: The SEC function has been called using an argument that causes the secant to approach infinity.

Message: ERROR IN SIN FUNCTION . . . ARGUMENT TOO LARGE

Cause: The SIN function has been called using an argument whose short-form magnitude is equal to or greater than PI*$2^{18}$ or whose long-form magnitude is equal to or greater than PI*$2^{50}$.

Message: ERROR IN SQR FUNCTION . . . NEGATIVE ARGUMENT

Cause: The SQR function has been called using an argument whose value is negative.

Message: ERROR IN TAN FUNCTION . . . ARGUMENT TOO LARGE

Cause: The TAN function has been called using an argument whose short-form magnitude is equal to or greater than PI*$2^{18}$ or whose long-form magnitude is equal to or greater than PI*$2^{50}$.

Message: ERROR IN TAN FUNCTION . .... INFINITE VALUE

Cause: The TAN function has been called using an argument that causes the tangent to approach infinity.

Message: EXPONENTIATION ERROR

Cause: X↑Y has been attempted with $X = 0$ and $Y = 0$.

Message: FILE DOES NOT EXIST

Cause: An attempt has been made to open a nonexistent data file.

Message: FILE IS CLOSED OR UNASSIGNED

Cause: A GET, MAT GET, PUT or MAT PUT operation has been attempted on an inactive data file.

Message: FILE IS FOR INPUT

Cause: A PUT or MAT PUT operation has been attempted on a data file opened as an input file.

Message: FILE IS FOR INPUT ONLY

Cause: An attempt has been made to open a shared data file as an output file.

Message: FILE IS FOR OUTPUT

Cause: A GET or MAT GET operation has been attempted on a data file opened as an output file.

Message: FILE IS LOCKED

Cause: An attempt has been made to open a locked data file as an output file.

Message: FILE IS NOT A DATA FILE

Cause: An attempt was made to open a file that is a program file rather than a data file.

Message: FILE IS PROTECTED

Cause: An attempt has been made to open a shared data file that is protected.

Message: INVALID LOGICAL FILE NUMBER

Cause: An attempt has been made to access a data file whose reference number is less than 1 or greater than 255.

Message: INVALID MATRIX DECLARATION

Cause: A matrix has been redimensioned outside the original bounds of the array.

Message: INVALID RECORD FORMAT

Cause: A record in the input data file which is not a valid BASIC format has been detected.

| | |
|---|---|
| Message: | INVALID VARIABLE ASSIGNMENT |
| Cause: | A variable has been assigned a value whose data type is not valid for the specified assignment. |
| Message: | MATRICES NOT CONFORMABLE |
| Cause: | A matrix operation has been attempted and the matrices are not conformable with the specified operation. |
| Message: | NEARLY SINGULAR MATRIX |
| Cause: | Matrix inversion has been attempted and the matrix being inverted is singular or nearly singular. |
| Message: | OUTPUT FILE EXCEEDED |
| Cause: | An attempt has been made to write to a data file when there is no space left in the file. |
| Message: | RETURN WITHOUT ACTIVE GOSUB |
| Cause: | A RETURN statement has been executed with no active GOSUB. |
| Message: | SUBSCRIPT OUT OF BOUNDS |
| Cause: | The computed address of an array variable is not within the bounds of the array. |
| Message: | TOO MANY DATA FILES |
| Cause: | An attempt has been made to open more than four data files at one time. |
| Message: | TOO MANY NESTED FUNCTIONS OR SUBROUTINES |
| Cause: | GOSUB statements or function references have been nested in excess of 47. (Refer to *Appendix A* for GOSUB and function limits.) |
| Message: | UNDEFINED IMAGE STATEMENT REFERENCED |
| Cause: | A PRINT USING statement has referenced a nonexistent Image statement. |
| Message: | UNDEFINED LINE NUMBER REFERENCED |
| Cause: | A nonexistent line number has been referenced. |
| Message: | UNDEFINED USER FUNCTION REFERENCED |
| Cause: | A nonexistent user function has been referenced. |

Message: UNRECOVERABLE I/O ERROR

Cause: An unrecoverable error has occurred during the reading or writing of a data file.

Message: UNRECOVERABLE SYSTEM PROBLEM

Cause: An unrecoverable system problem occurred during the opening of a data file.

## EXCEPTION ERROR MESSAGES

When an input or arithmetic exception is detected during program execution, a message is printed, a specified action is invoked, and execution is continued. Where relevant, the message is prefaced by the line number of the statement being executed at the time the exception occurs.

Message: DIVISION BY ZERO

Cause: Division by zero has been attempted.

Action: The result of the operation causing the exception is set to the maximum machine magnitude.

Message: INVALID INPUT DATA . . . RETYPE IT

Cause: The user has provided input data which violates the format specified in the INPUT statement.

Action: The data for all items in the INPUT statement list must be reentered.

Message: OVERFLOW

Cause: An arithmetic operation has exceeded the maximum machine value.

Action: The result of the operation causing the exception is set to the maximum machine magnitude.

Message: UNDERFLOW

Cause: An arithmetic operation has exceeded the minimum machine value.

Action: The result of the operation causing the exception is set to zero.

# INDEX

## READER'S COMMENT FORM

**CALL/360: BASIC Reference Handbook** **65-2211-1**

Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. All comments and suggestions become the property of SBC.

| | Yes | No | No Opinion |
|---|---|---|---|
| Does this publication meet your needs? | ☐ | ☐ | ☐ |
| Did you find the material: | | | |
| Easy to read and understand? | ☐ | ☐ | ☐ |
| Organized for convenient use? | ☐ | ☐ | ☐ |
| Complete? | ☐ | ☐ | ☐ |
| Arranged for convenient reference? | ☐ | ☐ | ☐ |
| Written for your technical level? | ☐ | ☐ | ☐ |

What is your occupation? ______________________________

| | | | |
|---|---|---|---|
| How do you use this publication? | | | |
| As an introduction to the subject? | ☐ | ☐ | ☐ |
| For advanced knowledge of the subject? | ☐ | ☐ | ☐ |
| For information about operating procedures? | ☐ | ☐ | ☐ |
| As an instructor in a class? | ☐ | ☐ | ☐ |
| As a reference manual? | ☐ | ☐ | ☐ |

Other ______________________________

You may 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.

fold fold

First Class
Permit No. 3373
San Jose, Calif.

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

POSTAGE WILL BE PAID BY . . .

THE SERVICE BUREAU CORPORATION
Publications Department — 884
P. O. Box 5974
San Jose, California 95150

fold fold