Chapter 8

CBM BASIC

This chapter describes the syntax for all CBM BASIC statements and functions. Statements are described first, listed in alphabetic order; then functions are described, also in alphabetic order.

This chapter serves as a reference for all statements and functions. Chapters 4, 5 and 6 describe programming concepts; these three chapters also give examples of statements and functions used in programs.

Immediate and Program Modes

Most statements can be executed in immediate or program mode. Unless otherwise stated, you can assume that a statement can be used in both modes. Exceptions are identified. Some statements can be used in one mode, but not the other; other statements can be used in both modes, but only one mode is practical.

BASIC Revisions

All statements and functions are identified as available with BASIC 4.0 only, or with all versions of BASIC. Statements and functions are cross referenced where an "all versions" statement or function has a BASIC 4.0 equivalent. All BASIC 4.0 statements need DOS 2.0, or higher releases of DOS.

Format Conventions

Consistent syntax is used when defining the format for all statements and functions. The following conventions have been adopted:

UPPER CASE	Upper case words and letters must appear exactly as shown.
lower case	Lower case words and letters are variable; the exact wording or value is supplied by the programmer.
{ }	Braces indicate a choice of items; braces do not appear in an actual statement.
[ ]	Brackets indicate that the parameter is optional; brackets do not appear in an actual statement.
...	Ellipses indicate that the preceding item can be repeated; ellipses do not appear in actual statements.
line number	A beginning line number is implied for all stored statements.

Terms are used as follows in statement and function format definitions:

access	the way in which a data file is to be accessed. Use WRITE for a write access and READ for a read access.
bno	the character number within a record of a relative data file.
byte	a numeric constant variable or expression which evaluates to a number in the range 0 through 255.
condition	a relational term or expression of the type: var { < | > | = | < > | < = | > = } [expression] If the expression to the right of the relational operator is absent then = 0 is implied.
constant	any numeric or string constant.
c$	a character string or CHR$ function representing a comma, carriage return, or other legal separator in a PRINT# statement parameter list.
<CR>	a carriage return character.
d	a destination diskette drive number (0 or 1).
data	any constant, variable or expression.
datan	any numeric constant, variable or expression.
data$	any string constant, variable or expression.
Dd	a destination diskette drive number which must be specified as D0 or D1.
destfile	the name of a destination file.
dev	a physical unit device number (see Table 8-1).

Table 8-1. Physical Device Numbers

Device Number	Device
0	Keyboard
1 (default)	Cassette tape unit #1
2	Cassette tape #2
3	Video display screen
4	Printer
5-7	IEEE port devices
8	Diskette unit
9-30	IEEE port devices
31-255	Currently unassignable

Table 8-2. Secondary Address Codes

Device	Secondary Address Code	Operation
CBM Cassette Tape Units	0 (default) 1 2	Open for read Open for write Open for write and end-of-file (EOF) tape mark Write end-of tape (EOT) mark when file is closed
CBM Line Printer	0 (default) 1 2 3 4 5 6 7* 8* 9* 10*	Normal Print Print under format statement control Store the formatting data Set number of lines per page Enable printer format diagnostic messages Define a programmable character Set spacing between lines (Model 2022 only) Select lower-case Select upper-case Turn off Unit 4 Reset
CBM Diskette Unit	0 1 2-14 15	Not defined Not defined Open for Read Write as specified Access parameter

* New printer ROMs only

diskname the name assigned to a disk.

dr a diskette drive number (0 or 1).

Ds a source diskette drive number which must be specified as D0 or D1.

<ESC> the escape key or character.

expression an arithmetic expression containing any combination of operators, numeric constants and variables.

filename any file name.

lvv a diskette number which may range between 00 and 99, and must be written as 100 through 199.

lf a logical file number (an integer between 0 and 255).

line any basic program line number.

$line_i$ one of many basic program line numbers.

Ly relative file record length. y is the number of characters per record; it may range between 1 and 254. The record length must be specified using the format L1 through L254.

memadr any memory address. Memory addresses may range from 0 to 65536.

message any text string enclosed in quotes.

newname a new data file name.

nvar any numeric variable name.

oldname any old data file name.

ON Uz the standard BASIC 4.0 means of specifying a physical unit number. ON U must be present; z is the physical number. If this parameter is absent physical unit number 8 (the standard disk drive physical number) is assumed.

rno the record number within a relative data file.

<RVS> the unshifted REVERSE key.

s a source diskette drive number (0 or 1).

sa a secondary address (see Table 8-2).

sourcefile the name of a source data file.

statement any BASIC statement.

type data file type specification. SEQ represents a sequential file, PRG represents the program file, and USR represents a random access file.

var any numeric integer or string variable.

var(sub) any subscripted integer, numeric, or string variable.

vv a diskette number (between 00 and 99).

W a parameter specifying the sequential file being opened for a write access.

BASIC STATEMENTS

APPEND# (BASIC 4.0)

The APPEND# statement opens an existing sequential diskette file and allows new data to be added at the end of the file. (See also PRINT# COPY.)

Format:

```
APPEND#lf,"filename"[,Dd][ON Uz]
```

The APPEND# statement opens sequential data file "filename" on the diskette on drive d and positions file pointers beyond the current end of file. Subsequent PRINT# statements referencing logical file lf can then write additional data, which gets appended to the end of the file. If no disk drive is specified (d is absent) drive 0 is assumed.

Example:

```
APPEND#1,"CALC"
PRINT#1,A
```
Open sequential file "CALC" as logical file #1 on drive 0. Write variable A contents to the end of the file

```
APPEND#3,"TALK",D1
PRINT#3,"123"
```
Open sequential file "TALK" as logical file #3. The string "123" is added to the end of the file

BACKUP (BASIC 4.0)

The BACKUP statement duplicates an entire diskette. The duplicate and original have the same header, disk name, identification number, directory, and files. (See also PRINT# DUPLICATE.)

Format:

```
BACKUP Ds TO Dd [ON Uz]
```

The diskette in drive s is duplicated. The duplicate diskette is generated in drive d. Duplicating the entire diskette takes a couple of minutes.

Example:

```
BACKUP D0 TO D1
```
Duplicate contents of diskette in drive 0 to diskette in drive 1

```
BACKUP D1 TO D0
```
Duplicate contents of diskette in drive 1 to diskette in drive 0

Caution: All files on the diskette must be properly closed before the diskette is backed up.

CLOSE

The CLOSE statement closes a logical file. (See also DCLOSE.)

Format:

```
CLOSE lf
```

The CLOSE statement closes logical file lf. If lf is not present, all open logical files are closed by BASIC < 3.0, but BASIC 4.0 gives a syntax error.

Every file should be closed after all file accesses have been completed. An open logical file may be closed only once. The particular operations performed in response to a CLOSE statement depend on the open file's physical device and the type of access that occurred. For details see Chapter 6.

Example:

```
CLOSE 1     Close logical file 1
CLOSE 14    Close logical file 14
```

CLR

The CLR statement sets all numeric variables to zero and assigns null values to all string variables. All array space in memory is released. This is equivalent to turning the CBM computer off, then turning it back on and reloading the program into memory. CLR closes all logical files that are currently open within the executing program.

Format:

```
CLR
```

A program will continue to run following execution of a CLR statement providing the effects of the CLR statement's execution do not adversely effect program logic.

Example:

```
100 CLR
```

CMD

The CMD statement sends to physical unit 4 (the printer) all output that would have gone to the display. Output goes to the printer, instead of the display, until a PRINT# statement specifying the same logical file number is executed. At least one PRINT# statement must follow a CMD statement.

Format:

```
CMD lf
```

The CMD statement assigns a line printer output channel to logical file lf. After execution of a CMD statement, PRINT and LIST both print data instead of displaying it. See Chapter 6 for a discussion of line printer programming.

Example:

The following sequence uses CMD to print program listings.

```
OPEN 5,4    Open logical file 5 selecting the printer
CMD 5       Direct subsequent output to the printer
LIST        Print the program listing
PRINT#5     Print a carriage return and deselect the printer
CLOSE 5     Close logical file 5
```

COLLECT (BASIC 4.0)

The COLLECT statement recreates a Block Availability Map (BAM) for all files on the diskette. Improperly closed files are closed or deleted.

Format:

```
COLLECT  [Dd][ON Uy]
```

The diskette on drive d is collected. If the Dd parameter is absent, drive 0 is assumed.

Example:

```
COLLECT       Collects space on diskette in last drive accessed
COLLECT D0    Collects space on diskette in drive 0
COLLECT D1    Collects space on diskette in drive 1
```

CONCAT (BASIC 4.0)

The CONCAT statement concatenates two data files. (See also PRINT# COPY.)

Format:

```
CONCAT[Ds,]"sourcefile" TO [Dd,]"destfile"[ON Uz]
```

The contents of sourcefile on the diskette in drive s is concatenated onto the end of destfile on the diskette in drive d. The file named sourcefile does not change. The file named destfile keeps its original contents, with the contents of sourcefile tacked on at the end. If drive numbers s and/or d are not specified, then drive 0 is assumed.

Caution: Files must be closed before they are concatenated.

Example:

```
CONCAT "FIRST" TO "SECOND"      The contents of file FIRST is concatenated on the end of
                                file SECOND. Both files are on the diskette in drive 0

CONCAT D1,"ABC" TO D0,"XYZ"     The contents of file ABC on the diskette in drive 1 is
                                concatenated on the end of file XYZ on the diskette
                                in drive 0
```

CONT

The CONT statement, typed at the keyboard in immediate mode, resumes program execution after a BREAK.

Format:

```
CONT
```

A break is caused by execution of a STOP statement or an END statement that has additional statements following it. Depressing the STOP key while a program is running also causes a break. Program execution continues at the exact point where the break occurred.

Pressing the RETURN key in response to an INPUT statement will also cause a break. Typing CONT after this break re-executes the INPUT statement.

Example:

```
CONT
```

COPY (BASIC 4.0)

The COPY statement copies a single diskette file, or all the files on a diskette. (See also PRINT# COPY.)

Format:

COPY [Ds,]["sourcefile"] TO [Dd,]["destfile"][ON Uz]

If the COPY statement is used to copy a single file, then the file named sourcefile on the diskette on drive s is copied to a new file named destfile on the diskette in drive d; the file names sourcefile and destfile must be present, but if Ds and/or Dd are absent, drive 0 is assumed.

The COPY statement can also be used to copy all files from the diskette in one drive to the diskette in the other drive. To use the COPY statement in this fashion, file names sourcefile and destfile must be absent, but drive numbers Ds and Dd must be present and different.

If the name of a source file that is being copied exists on the destination diskette, then the copy will be aborted at that file, and a FILE ALREADY EXISTS error will be reported.

COPY does not modify any files previously on the destination diskette.

Caution: A file must be closed before it is copied.

Example:

```
COPY D1 TO D0                    Copy all files on the diskette in drive D1 to the
                                   diskette in drive D0. (DOS 2.0 and higher
                                   releases only)

COPY D1, "MAJOR" TO D1, "MINOR"  Create MINOR file on the diskette in drive D1
```

DATA

The DATA statement declares constants that are assigned to variables by READ statements.

Format:

DATA constant[,constant,constant,...,constant]

DATA statements may be placed anywhere in a program.

The DATA statement specifies either numeric or string contents. String constants are usually enclosed in double quotation marks; the quotes are not necessary unless the string contains graphic characters, blanks (spaces), commas, or colons. Blanks, commas, colons and graphic characters are ignored unless the string is enclosed in quotes. A double quotation mark cannot be represented in a DATA string; it must be specified using a CHR$(34) function.

The DATA statement is valid in program mode only.

Example:

```
10 DATA NAME, "C.D."     Defines two string variables

50 DATA 1E6,-10,XYZ      Defines two numeric variables and one string variable
```

See the READ statement for a description of how DATA statement constants are used within a program.

DCLOSE (BASIC 4.0)

DCLOSE closes a single file or all the files currently open on a disk unit. (Also see CLOSE.)

Format:

```
DCLOSE#lf  [ON Uz]
```

The DCLOSE statement closes logical file lf. If the logical file number is not specified, all currently open diskette files are closed.

Example:

```
DCLOSE          Closes all open diskette files
DCLOSE#1        Closes the diskette file identified by logical file 1
DCLOSE ON U8    Closes all open diskette files on physical unit #8
```

DEF FN

The DEF function (DEF FN) allows special purpose functions to be defined and used within BASIC programs.

Format:

```
DEF FNnvar(arg)=expression
```

Floating point variable nvar identifies the function, which is subsequently referenced using the name FNnvar(data). (If nvar has more than five letters a syntax error is reported. A syntax error is also reported if nvar is a string or integer variable.)

The function is specified by expression, which can be any arithmetic expression, containing any combination of numeric constants, variables, and/or operators. arg is a dummy variable name which can (and usually does) appear in expression.

arg is the only variable in expression which can be specified when FNnvar(data) is referenced. Any other variables in expression must be defined before FNnvar(data) is referenced for the first time. FNnvar(data) evaluates expression using data as the value for arg.

The entire DEF FN statement must appear on a single 80 character line; however a previously defined function can be included in expression, so user-defined functions of any desired complexity can be developed.

The function name var can be re-used, and therefore redefined by another DEF FN statement appearing later in the same program.

The DEF FN definition statement is illegal in immediate mode. However, a user-defined function that has been defined by a DEF FN statement in the current stored program can be referenced in an immediate mode statement.

Example:

```
10 DEF FNC(R)=π*R↑2
```
Defines a function that calculates the circumference of a circle. It takes a single argument R, the radius of the circle, and returns a single numeric value, the circumference of the circle

```
?FNC(1)
```
Prints 3.141159265 (the value of π)

```
A=FNC(14)
```
Assigns to A the value calculated by the user-defined function FNC, using an argument of 14

```
55 IF FNC(X)>60 GOTO 150
```
Uses the value calculated by the user-defined function FNC as a branch condition. The current contents of variable X is used when calculating the user-defined function

DIM

The Dimension statement DIM allocates space in memory for array variables.

Format:

DIM var(sub)[,var(sub),. . .,var(sub)]

The DIM statement identifies arrays with one or more dimensions as follows:

var(sub_i)	Single-dimension array
var(sub_i,sub_j)	Two-dimension array
var(sub_i,sub_j,sub_k)	Multiple-dimension array

See Chapter 4 for a complete description of arrays.

Arrays with more than eleven elements must be dimensioned in a DIM statement. Arrays with eleven elements or less (subscripts 0 through 10 for a one-dimensional array) may be used without being dimensioned by a DIM statement; for such arrays, eleven array spaces are automatically allocated in memory when the first array element is encountered in the program. An array with more than eleven elements must occur in a DIM statement before any other statement references an element of the array.

If an array is dimensioned more than once, or if an array having more than eleven elements is not dimensioned, a ?REDIM'ED ARRAY error occurs and the program is aborted.

A CLR statement allows a DIM statement to be reexecuted.

Example:

```
10 DIM A(3)
```
Dimension a single-dimensional array of 3 elements.

```
45 DIM X$(44,2)
```
Dimension a two-dimensional array of 88 elements.

```
1000 DIM MU(X,3*B),N(12)
```
*Dimension a two-dimensional array of X times 3*B elements and a single dimensional array of 12 elements. X and B must have been assigned values before the DIM statement is executed.*

DIRECTORY (BASIC 4.0)

The DIRECTORY statement displays directories for diskettes in one or both drives. The word CATALOG may be used instead of DIRECTORY (also see LOAD"$dr").

Format:

```
DIRECTORY[Dd][ON Uz]
```

The directory for the diskette in drive d is displayed. If the Dd parameter is absent, directories for the diskettes in both drives are displayed.

If a selected drive contains no diskette an error status is reported.

The DIRECTORY statement is usually executed in immediate mode.

Example:

```
DIRECTORY        Displays the directory of drive 0 and drive 1
DIRECTORY D1     Displays the directory of drive 1
```

Printing a Directory

A directory can be printed instead of being displayed by opening a printer channel before executing the DIRECTORY statement. Here is the required immediate mode statement sequence:

```
OPEN 4,4         Open the printer specifying logical file 4
CMD 4            Deflect display output to the printer
DIRECTORY        Print directories for diskettes in both drives
PRINT#4          Deflect output back to the display
CLOSE 4
```

DLOAD (BASIC 4.0)

The DLOAD statement loads a BASIC program from a diskette into memory (also see LOAD).

Format:

```
DLOAD "filename"[Dd][ON Uz]
```

The DLOAD statement loads program file "filename" from the diskette in drive Dd into computer memory. If Dd is not present, drive 0 is assumed.

Example:

```
DLOAD "CALC"                Load CALC file from drive 0
DLOAD "TIME",D1             Load TIME file from drive 1
A$="PROG"                   Load PROG file from drive 0
DLOAD A$
DLOAD"PROG",D0 ON U8        Load PROG file from drive 0 on the disk unit
```

Using BASIC 4.0, if you press the shifted RUN/STOP key, the next program encountered on the diskette is loaded and run.

DOPEN (BASIC 4.0)

DOPEN opens a data file for a read and/or write access.

Format:

DOPEN#lf,"filename"[,Ly][,Dd][ON Uz][,W]

The DOPEN statement opens data file filename on the diskette in drive d, assigning to it logical file number lf. If d is not specified then drive 0 is assumed. If Ly is not present then a sequential file is assumed. The sequential file is opened for a write access if W is not present; it is opened for a read access if W is present.

If Ly is present then a relative file is assumed with a record length of y bytes. Relative files are opened for read or write accesses, therefore the W parameter cannot be present.

Example:

```
DOPEN#1,"PRIZES"
```
Opens the sequential file named PRIZES on drive 0 for a read access

```
DOPEN#6,"SNAKE"L30,D1
```
Opens the relative file named SNAKE, with a record length of 30, for read and write accesses. The file is on drive D1

DSAVE (BASIC 4.0)

The DSAVE statement writes a BASIC program file from memory onto a diskette (also see SAVE).

Format:

DSAVE"filename"[,Dd][ON Uz]

The DSAVE statement saves the BASIC program currently in memory, writing it to a new file named filename, on the diskette in drive d. If Dd is not present, drive 0 is assumed.

Example:

```
DSAVE"TRUE"
```
Write program file TRUE to diskette in drive 0

```
DSAVE"FALSE",D1
```
Write program file FALSE to diskette in drive 1

END

The END statement terminates program execution and returns the computer to immediate mode.

Format:

END

The END statement can provide a program with one or more termination points, at locations other than the physical end of the program. END statements can be used to terminate individual programs when more than one program is in memory at the same time. An END statement at the physical end of the program is optional.

The END statement is used in program mode only.

Example:

```
20001 END
```

FOR-NEXT STEP

All statements between the FOR statement and the NEXT statement are re-executed the same number of times.

Format:

```
FOR nvar = start TO end STEP increment
[statements in loop]
NEXT[nvar]
```

where:

nvar is the index of the loop. It holds the current loop count. nvar is often used by the statements within the loop.

start is a numeric constant, variable or expression that specifies the beginning value of the index.

end is a numeric constant, variable, or expression that specifies the ending value of the index. The loop is completed when the index value is equal to the end value, or when the index value is incremented or decremented past the end value.

increment if present, is a numeric constant, variable, or expression that specifies the amount by which the index variable is to be incremented with each pass. The step may be incremental (positive) or decremental (negative). If STEP is omitted the increment defaults to 1.

The nvar may optionally be included in the NEXT statement. A single NEXT statement is permissible for nested loops that end at the same point. The NEXT statement then takes the form:

NEXT $nvar_1$,$nvar_2$...

The FOR-NEXT loop will always be executed at least once, even if the beginning nvar value is beyond the end nvar value. If the NEXT statement is omitted and no subsequent NEXT statements are found, the loop is executed once.

The start, end, and increment values are read only once, on the first execution of the FOR statement. You cannot change these values inside the loop. You can change the value of nvar within the loop. This may be used to terminate a FOR-NEXT loop before the end value is reached: set nvar to the end value, and on the next pass the loop will terminate itself. Do not jump out of the FOR-NEXT loop with a GOTO. Do not start the loop outside a subroutine and terminate it inside the subroutine.

FOR-NEXT loops may be nested. Each nested loop must have a different nvar variable name. Each nested loop must be wholly contained within the next outer loop; at most, the loops can end at the same point.

Example:

```
10 FOR IN = 0 TO 100
.
40 NEXT IN
100 FOR X = A + 14 TO C-64+D/2 STEP 4
.
150 NEXT X
60 FOR A1 = 50 TO 0 STEP -1
.
90 NEXT
100 FOR I = 0 TO 10 STEP 0.5
.
155 NEXT
250 FOR I = 1 TO 5
260 FOR J = A TO B
.
300 NEXT I,J   same as   300 NEXT I
                         310 NEXT J   same as   300 NEXT
                                                310 NEXT
```

GET

The GET statement receives single characters as input from the keyboard.

Format:

```
GET var
```

The GET statement can be executed in program mode only.

When a GET statement is executed, var is assigned a 0 value if numeric, or a null value if a string. Any previous value of the variable is lost. Then GET fetches the next character from the keyboard buffer and assigns it to var. If the keyboard buffer is empty, var retains its 0 or null value.

GET is used to handle one-character responses from the keyboard. GET accepts the RETURN key as input and passes the value (CHR$(13)) to var.

If var is a numeric variable and no key has been pressed, 0 is returned. However, a 0 is also returned when 0 is entered at the keyboard.

If var is a numeric variable and the character returned is not a digit (0-9), a ?SYNTAX ERROR message is generated and the program aborts.

The GET statement may have more than one variable in its parameter list, but it is hard to use if it has multiple parameters:

```
GET var,var,. . .,var
```

Example:

```
10 GET C$

10 GET D

10 GET A,B,C
```

GET#

The GET External statement (GET#) receives single characters as input from an external storage device identified via a logical file number.

Format:

```
GET#lf,var
```

The GET# statement can only be used in program mode. GET# fetches a single character from an external device and assigns this character to variable var. The external device is identified by logical file number lf. This logical file must previously have been opened by an OPEN or DOPEN statement.

GET# and GET statements handle variables and data input identically. For details see the GET statement description.

Example:

```
10 GET#4,C$:IF C$="" GOTO 10
```

Get a keyboard character. Re-execute if no character is present

GOSUB

The GOSUB statement branches program execution to a specified line and allows a return to the statement following GOSUB. The specified line is a subroutine entry point.

Format:

GOSUB ln

The GOSUB statement calls a subroutine. The subroutine's entry point must occur on line ln. A subroutine's entry point is the beginning of the subroutine in a programming sense; that is to say it is the line containing the statement (or statements) which are executed first. The entry point need not necessarily be the subroutine line with the smallest line number.

Upon completing execution the subroutine branches back to the line following the GOSUB statement. The subroutine uses a RETURN statement in order to branch back in this fashion.

A GOSUB statement may occur anywhere in a program; in consequence a subroutine may be called from anywhere in the program.

Subroutines may be nested; that is to say subroutines may be called from within subroutines. Twenty-six levels of nesting are allowed; that means 25 GOSUB statements may be executed before the first RETURN statement.

Example:

```
100 GOSUB 2000      Branch to subroutine at line 2000
110 A = B*C
.
.                   Subroutine branches back here

2000                Subroutine entry point
.
.

2090 RETURN         Branch back to line 110
```

GOTO

The GOTO statement branches unconditionally to a specified line.

Format:

GOTO ln

The GOTO statement causes program execution to branch to line ln.

Example:

```
10 GOTO 100
```

Executed in immediate mode, GOTO branches to the specified line in the stored program without clearing the current variable values. GOTO cannot reference immediate mode statements, since they do not have line numbers.

HEADER (BASIC 4.0)

The HEADER statement formats a diskette, assigning it a disk name and identification number. (See also PRINT# PREPARE.)

Format:

```
HEADER "diskname",Dd[,Ivv][ON Uz]
```

When formatting a diskette the HEADER statement marks off sectors on each track, then initializes the directory and Block Availability Map. The formatted diskette must be in drive d. The diskette is given the name diskname and the number vv. This name and number appears in the reverse field at the top of a diskette directory display.

The HEADER statement is usually executed in immediate mode.

The HEADER statement can be used to format a blank diskette or to reformat and clear a used diskette. Because the changes are permanent, this command requires caution in its use. If executed in immediate mode, the question ARE YOU SURE? is displayed. You must respond by typing YES (CR) to continue.

If a media error occurs when the HEADER statement is executed, a ?BAD DISK message is displayed on the screen. Media errors occur when a diskette is missing from the drive, the write protect tab is in place, or the diskette magnetic surface is defective.

Example:

```
HEADER "MASTER",D0,I02
```
Prepare and format a diskette, giving it the name "MASTER" and the number 02. The diskette is in drive 0

IF-THEN

The IF-THEN statement provides conditional execution of statements based on a relational expression.

Format:

IF condition THEN statement[:statement. . .] — *Conditionally execute statement(s)*

IF condition { THEN / GOTO } line — *Conditionally branch*

If the specified condition is true, then the statement or statements following the THEN are executed. If the specified condition is false, control passes to the statement(s) on the next line and the statement or statements following the THEN are not executed. For a conditional branch, the branch line number is placed after the word THEN, or after the word GOTO. The compound form THEN GOTO is also acceptable.

```
IF A = 1 THEN 50
IF A = 1 GOTO 50
IF A = 1 THEN GOTO 50
```
Equivalent

If an unconditional branch is one of many statements following THEN, then the branch must be the last statement on the line, and it must have "GOTO line" format. If the unconditional branch is not the last statement on the line, then statements following the unconditional branch can never be executed.

The following statements cannot appear in an immediate mode IF-THEN statement: DATA, GET, GET#, INPUT, INPUT#, REM, RETURN, END, STOP, WAIT.

If a line number is specified, or any statement containing a line number, there must be a corresponding statement with that line number in the current stored program.

The CONT and DATA statements cannot appear in a program mode IF-THEN statement.

If a FOR-NEXT loop follows the THEN, then the loop must be completely contained on the IF-THEN line. Additional IF-THEN statements may appear following the THEN as long as they are completely contained on the original IF-THEN line. However, Boolean connectors are preferred to nested IF-THEN statements. For example, the two statements below are equivalent, but the second is preferred.

```
10 IF A$ = "X" THEN IF B = 2 THEN IF C > D THEN 50
10 IF A$ = "X" AND B = 2 AND C > D THEN 50
```

Example:

```
400 IF X > Y THEN A = 1
500 IF M+1 THEN AG = 4.5:GOSUB 1000
```

INPUT

The INPUT statement receives data input from the keyboard.

Format:

INPUT { (blank) / "message"; } var [,var,. . .,var]

INPUT can be used in program mode only.

When the INPUT statement is executed, CBM BASIC displays a question mark on the screen requesting data input. The user must enter data items that agree exactly, in number and type, with the variables in the INPUT statement parameter list. If the INPUT statement has more than one variable in its parameter list, then keyboard entries must be separated by commas. The last entry must be terminated with a carriage return:

```
?1234<CR>                    Single data item response
?1234,567.89,NOW<CR>         Multiple data item response
```

If "message" is present, it is displayed before the question mark. "message" can have up to 80 characters.

If more than one but less than the required number of data items are input, CBM BASIC requests additional input with double question marks (??) until the required number of data items have been input. If too many data items are input, the message $EXTRA IGNORED is displayed. The extra input is ignored, but the program continues execution.

Example:

Statement	Operator Response	Result
`10 INPUT A,B,C$`	`?123,456,NOW`	A=123, B=456, C$="NOW"

`10 INPUT A,B,C$`	`?123`
	`??456`
	`??NOW`
		A=123
		B=456
		C$="NOW"

`10 INPUT A,B,C$`	`?NOW`
	`?REDO FROM START`
	`?123`
	`?456`
	`?789`
		
		A=123
		B=456
		C="789"

`10 INPUT "A= ";A`	`A= ?123`	A=123

Note that you must input numeric data for a numeric variable, but you can input numeric or string data for a string variable.

Caution: If the RETURN key is pressed in response to an INPUT statement with no preceding data entry, then program execution ceases and the computer enters immediate mode. To restart execution type CONT in response to the READY message.

INPUT#

The Input External statement (INPUT#) inputs one or more data items from an external device identified via a logical file number.

Format:

```
INPUT#If var[,var,. . .,var]
```

The INPUT# statement inputs data from the selected external device and assigns data items to variable(s) var. Data items must agree in number and kind with the INPUT# statement parameter list.

If an end of record is detected before all variables in the INPUT# statement parameter list have received data, then an OUT OF DATA error status is generated, but the program continues to execute.

INPUT# and INPUT statements execute identically, except that INPUT# receives its input from a logical file. Also, INPUT# does not display error messages; instead it reports error statuses which the program must interrogate and respond to.

Input data strings may not be longer than 80 characters (79 characters plus a carriage return) because the input buffer has a maximum capacity of 80 characters. Commas and carriage returns are treated as item separators by the computer when processing the INPUT# statement; they are recognized, but are not passed on to the program as data.

INPUT# is valid in program mode only.

Example:

`1000 INPUT#10,A`	*Input the next data item from logical file 10. A numeric data item is expected; it is assigned to variable A*
`946 INPUT#12,A$`	*Input the next data item from logical file 12. A string data item is expected; it is assigned to variable A$*
`900 INPUT#5,B,C$`	*Input the next two data items from logical file 5. The first data item is numeric; it is assigned to numeric variable B. The second data item is a string; it is assigned to string variable C$*

LET=

The Assignment statement, LET=, or simply =, assigns a value to a specified variable.

Format:

{(blank) / LET} var=data

Variable var is assigned the value computed by resolving data.
The word LET is optional; it is usually omitted.

Example:

```
10 A=2
450 C$="▼"

300 M(1,3)=SGN(X)
310 XX$(I,J,K,L)="STRINGALONG"
```

LIST

LIST displays one or more lines of a program. Program lines displayed by the LIST statement may be edited.

Format:

LIST { (blank) / line / line$_1$-line$_2$ / -line / line- }

The entire program is displayed in response to LIST. Use line limiting parameters for long programs to display a section of the program that is short enough to fit on the screen.

Example:

`LIST`	*List entire program*
`LIST 50`	*List line 50*
`LIST 60-100`	*List all lines in the program from lines 60 to 100, inclusive*
`LIST -140`	*List all lines in the program from the beginning of the program through line 140*
`LIST 20000-`	*List all lines in the program from line 20000 to the end of the program*

Listed lines are reformatted as follows:

1. ?'s entered as a shorthand for PRINT are expanded to the word PRINT. Example:

   ?A becomes PRINT A

2. Blanks preceding the line number are eliminated. Example:

   50 A=1 becomes 50 A=1
   100 A=A+1 becomes 100 A=A+1

3. A space is inserted between the line number and the rest of the statement if none was entered. Example:

   55A=B-2 becomes 55 A=B-2

4. The line is displayed beginning at column 2 instead of column 1.

LIST is always used in immediate mode. A LIST statement in a program will list the program, but then exit to immediate mode. Attempting to continue program execution via CONT simply repeats the LIST indefinitely.

Printing a Program Listing

To print a program listing instead of displaying it, OPEN a printer logical file and execute a CMD statement before executing the LIST statement. Here is the necessary immediate mode sequence:

```
OPEN 4,4        Open the printer specifying logical file 4
CMD 4           Deflect display output to the printer
LIST            Print the program listing
PRINT#4         Deflect output back to the display
CLOSE 4
```

LOAD

The LOAD statement loads a program from an external device into memory. (Also see DLOAD.)

Cassette Unit Format:

LOAD ["filename"][,dev]

The LOAD statement loads into memory the program file specified by filename from the cassette unit selected by device number dev. If no device is specified then device 1 is assumed by default; cassette unit 1 is then selected. If no filename is given then the next file detected on the selected cassette unit is loaded into memory.

For cassette unit operating instructions see Chapter 2.

Example:

```
LOAD            Load into memory the next program found on cassette unit #1. If
                    you start a LOAD when the cassette is in the middle of a program,
                    the cassette will read past the remainder of the current program,
                    then load the next program
LOAD "",2       Load into memory the next program found on cassette unit #2
LOAD "EGOR"     Search for the program named EGOR on tape cassette #1 and load
                    it into memory.
N$="WHEE!LS"    Search for the program named WHEE!LS on cassette unit #1 and
LOAD N$             load it into memory.
LOAD "X"        Search for a program named X on cassette unit #1 and load it
                    into memory
```

Diskette Drive Format:

```
LOAD "dr:filename",dev
```

The LOAD statement loads into computer memory the program file with the name filename on the diskette in drive dr. dev. The device number for the diskette drive unit is the value 8 in all standard CBM computer systems. If dev is not present, then the default value is 1 which selects the primary tape cassette unit.

A single asterisk can be included instead of the filename, in which case the first program found on the selected diskette drive is loaded into memory.

For diskette operating instructions see Chapter 2.

Example:

```
LOAD"0:*",8          Load the first program found on disk drive 0
LOAD"0:FIREBALL",8   Search for the program named FIREBALL on disk drive 0, and load it
                       into memory
T$="0:METEOR"        Search for the program named METEOR on disk drive 0 and load it
LOAD T$,8              into memory
```

When a LOAD is executed in immediate mode, CBM BASIC automatically executes a CLR before the program is loaded. Once a program has been loaded into memory, it can be listed, updated, and/or executed.

The LOAD statement can also be used in program mode to build program overlays. A LOAD statement executed from within a program causes that program's execution to stop and another program to be loaded. In this case the CBM computer does not perform a CLR; therefore the old program can pass on all of its variable values to the new program.

When a LOAD statement accessing a cassette unit is executed in program mode, LOAD message displays are suppressed unless the tape PLAY key is up (off). If the PLAY key is off, the PRESS PLAY ON TAPE #1 message is displayed so that the load can proceed. All LOAD messages are suppressed when loading programs from a diskette in program mode.

Using LOAD to Display the Diskette Directory

The BASIC 4.0 DIRECTORY statement displays diskette directories. To display the diskette directory using earlier releases of BASIC, you must load and list a program file name $0 (for the diskette in drive 0) or $1 (for the diskette in drive 1).

Example:

```
LOAD "$0",8
SEARCHING FOR $0
LOADING
READY
LIST
```

NEW

The NEW statement clears the current program from memory.

Format:

```
NEW
```

When a NEW statement is executed, all variables are initialized to zero or null values and array variable space in memory is released. The pointers that keep track of program statements are reinitialized, which has the effect of deleting any program in memory; in fact the program is not physically deleted. NEW operations are automatically performed when a LOAD statement is executed.

If there is a program in memory, then you should execute a NEW statement in immediate mode before entering a new program at the keyboard. Otherwise the new program will overlay the old one, replacing lines if their numbers are duplicated, but leaving other lines. The result is a scrambled mixture of two unrelated programs.

Example:

```
NEW
```

NEW is always executed in immediate mode. If a NEW statement is executed from within a program, the program will "self destruct;" it will clear itself out.

ON...GOSUB

The ON...GOSUB statement provides conditional subroutine calls to one of several subroutines in a program, depending on the current value of a variable.

Format:

```
ON byte GOSUB line1[,line2,...,linen]
```

ON...GOSUB has the same format as ON...GOTO. See the ON...GOTO statement description for branching rules. byte is evaluated and truncated to an integer number, if necessary.

For byte=1, the subroutine beginning at $line_1$ is called. That subroutine completes execution with a RETURN statement which causes program execution to continue at the statement immediately following ON...GOSUB. If byte=2, the subroutine beginning with $line_2$ is called, etc.

ON...GOSUB is normally executed in program mode. It may be executed in immediate mode as long as there are corresponding line numbers to branch to in the current stored program.

Example:

```
10 ON A GOSUB 100,200,300
```

ON...GOTO

The ON...GOTO statement causes a conditional branch to one of several points in a program, depending on the current value of a variable.

Format:

ON byte GOTO $line_1$[,$line_2$,...,$line_n$]

byte is evaluated and truncated to an integer number, if necessary.

If byte=1, a branch to line number $line_1$ occurs. If byte=2, a branch to line number $line_2$ occurs, etc.

If byte=0, no branch is taken. If byte is in the allowed range but there is no corresponding line number in the program, then no branch is taken. If a branch is not taken, program control proceeds to the statement following the ON...GOTO; this statement may be on the same line as the ON...GOTO (separated by a colon), or on the next line.

If index has a non-zero value outside of the allowed range, the program aborts with an error message. As many line numbers may be specified as will fit on the 80-character line.

ON...GOTO is normally executed in program mode. It may be executed in immediate mode as long as there are corresponding line numbers in the current stored program that may be branched to.

Example:

```
40 A=B<10
50 ON A+2 GOTO 100,200
```

Branch to statement 100 if A is true (-1) or branch to statement 200 if A is false (0)

```
50 X=X+1
60 ON X GOTO 500,600,700
```

Branch to statement 500 if X=1, to statement 600 if X=2, or to statement 700 if X=3. No branch is taken if X>3.

OPEN

The OPEN statement opens a logical file and readies the assigned physical device. (Also see DOPEN.)

Cassette Tape Format:

OPEN lf[,dev][,sa][,"filename"]

The file named filename on the tape cassette unit identified by dev is opened for the type of access specified by the secondary address sa; the access is assigned the logical file number lf.

If no filename is specified then the next file encountered on the selected tape cassette is opened. If no device is specified then device number 1 is selected by default; this device number selects cassette unit 1. If no secondary address is specified then a default value of 0 is assumed and the file is opened for a read access only. A secondary address of 1 opens the file for a write access while a secondary address of 2 opens the file for a write access with an end-of-tape mark written when the file is subsequently closed.

Example:

```
OPEN 1                    Open logical file 1 at cassette drive # 1 (default) for a read access (default)
                            from the first file encountered on the tape (no filename specified)
OPEN 1,1                  Same as above
OPEN 1,1,0                Same as above
OPEN 1,1,0,"DAT"          Same as above but access the file named DAT
OPEN 3,1,2                Open logical file 3 for cassette # 1, for a write with EOT (End Of
                            Tape) access. The new file is unnamed and will be written at the
                            current physical tape location
OPEN 3,1,2,"PENTAGRAM"    Same as above but access the file named PENTAGRAM
```

Disk Unit Format:

OPEN lf,dev,sa, "dr:filename,type[,access]"

The file named filename on the diskette in drive dr is opened and assigned logical file number lf. type identifies the file as sequential (SEQ), program (PRG), or random (USR). If the file is sequential then access must be WRITE to specify a write access or READ to specify a read access. access is not present for a program or random access file.

An existing sequential file can be opened for a write access if dr is preceded by an @ sign. The existing sequential file contents are replaced entirely by new written data.

dev, the device number, must be present; it is 8 for all standard disk units. If dev is absent then a default value of 1 is assumed and the primary tape cassette unit is selected.

For a data file the secondary address sa can have any value between 2 and 14, however every open data file should have its own unique secondary address. A secondary address of 15 selects the disk unit command channel. Secondary addresses of 0 and 1 are used to access program files. Secondary address 0 is used to load a program file; secondary address 1 is used to save a program file.

Example:

```
OPEN 1,8,2,"0:DAT,SEQ,READ"       Open logical file 1 on a diskette in drive 0. Read from
                                    sequential file DAT
OPEN 5,8,3,"1:NEWFILE,SEQ,WRITE"  Open logical file 5 on a diskette in drive 1. Write to
                                    sequential file NEWFILE
OPEN 4,8,4,"@1:NEWFILE,SEQ,WRITE" Open logical file 4 on diskette drive 1. Write to
                                    sequential file NEWFILE replacing prior contents
```

See Chapter 6 for a discussion of files and file handling.

POKE

The POKE statement stores a byte of data in a specified memory location.

Format:

POKE memadr,byte

A value between 0 and 255, provided by byte, is loaded into the memory location with the address memadr.

Example:

```
10 POKE 1,A                  POKE value of variable A into memory at address 1
POKE 32768,ASC("A")-64       POKE 1 (the value of ASC ("A")-64) into memory at address
                               32768
```

PRINT

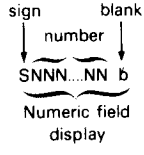
The PRINT statement displays data; it is also used to print to the line printer.

Format:

{PRINT / ?} data[{; / ,} data... {; / ,} data]

Print Field Formats:

Numeric fields are displayed using standard numeric representation for numbers greater than 0.01 and less than or equal to 999999999. Scientific notation is used for numbers outside of this range. Numbers are preceded by a sign character and are followed by a blank character:



The sign is blank for a positive number and minus sign (−) for a negative number.

Strings are displayed without additions or modifications.

PRINT Formats:

First data item. The first data item is displayed at the current cursor position. The PRINT format character (comma or semicolon) following the first data item specifies the location of the second data item's display. The location of each subsequent data item's display is determined by the punctuation following the preceding data item. Data items may be in the same PRINT statement, or in a separate PRINT statement.

New line. When no comma or semicolon follows the last data item in a PRINT statement, a carriage return occurs after the last data item is displayed.

Tabbing. A comma following a data item causes the next data item to be displayed at the next default tab column. Default tabs are at columns 1, 11, 21 and 31 for a 40 column display, continuing at 41, 51, 61 and 71 for an 80 column display. If a comma precedes the first data item, then a tab will precede the first item display.

Continuous. A semicolon following a data item causes the next display to begin immediately, in the next available column position. Numeric data always has one trailing blank character. For string data, items are displayed continuously with no forced intervening spaces.

Example:

```
40 PRINT A
40 PRINT A,B,C
40 PRINT A;B;C
40 PRINT, A;B;C
40 PRINT "NUMBERS",A;B;C
40 PRINT "NUM";"BER";
41 PRINT "S",A;B;C
```

PRINT#

The Print External statement (PRINT#) outputs one or more data items from the CBM computer to an external device (cassette tape unit, disk unit, or printer) identified by a logical file number.

Format:

```
PRINT #lf,data;c$;data;c$;...data
```

Data items listed in the PRINT# statement parameter list are written to the external device identified by logical unit number lf.

Very specific punctuation rules must be observed when writing data to external devices. A brief summary of punctuation rules is given below but for complete details see Chapter 6.

PRINT# Output to Cassette Files

Every numeric or string variable written to a cassette file must be followed by a carriage return character. This carriage return character is automatically output by a PRINT# statement that has a single data item in its parameter list. But a PRINT# statement with more than one data item in its parameter list must include c$ characters that force carriage returns. For example, use CHR$(13) to force a carriage return, or a string variable which has been equated to CHR$(13) wherever c$ appears.

PRINT# Output to Diskette Files

The cassette output rules described above apply also to diskette files with one exception: groups of string variables can be separated by comma characters (CHR$(44)). The comma character separators, like the carriage return separators, must be inserted using c$. String variables written to diskette files with comma character separators must subsequently be read back by a single INPUT# statement. The INPUT# statement reads all text from one carriage return character to the next.

PRINT# Output to the Line Printer

When the PRINT# statement outputs data to a line printer c$ must equal CHR$(29). No punctuation characters should separate c$ from data items as illustrated in the PRINT# format definition.

Caution: The form ? #cannot be used as an abbreviation for PRINT#.

Using BASIC <3.0, the PRINT# statement terminates every line output with a carriage return character. Using BASIC 4.0, this occurs only for file numbers of 127 or less, no automatic carriage return is output. Some non-Commodore printers require a carriage return character to be output at the end of a line. If you have such a printer, then using BASIC 4.0, choose a file number greater than 127, or output the carriage return as a separate terminating character.

Example:

```
100 PRINT#1,A              Output numeric variable A and a RETURN code to logical file 1
200 PRINT#4,A$             Output string variable A$ and a RETURN code to logical file 4
300 PRINT#10,B%;",";C$     Output numeric variable B%, a comma, string variable C$, and a
                             RETURN code to logical file 10

10 OPEN 1,1,2              Open logical file # 2 on cassette # 1 for write
20 PRINT#1,"HI"            Output HI to logical file # 1 on cassette # 2
```

The PRINT# statement also performs a variety of disk-handling operations. These uses of PRINT# are summarized below. BASIC 4.0 has individual statements that perform the same operations.

Disk files must be closed before being subject to any disk-handling operation.

COPY

Use PRINT# to copy and/or merge files. (Also see BASIC 4.0 COPY and CONCAT statements.

Format:

PRINT #lf,"C[OPY]d:destfile=s:sourcefile[,s:sourcefile....]"

Up to four source files can be concatenated and copied to a destination file. The source files are not changed. The source files are identified by their file name sourcefile and drive s. If more than one source file is specified then files are concatenated in the order in which they appear in the PRINT# statement. The newly created destination file is identified by its file name destfile and drive d.

Example:

```
OPEN 1,8,15                          Open the diskette command channel
PRINT#1,"C1:FILE1=C0:FILE0"          Copy FILE0 on drive 0 to a new file named
                                       FILE1 on drive 1
PRINT#1,"C0:NEWFIL=C1:FILEA,C0:FILEB"  A new file named NEWFIL is created on
                                       drive 0 by concatenating file FILEB on
                                       drive 0 at the end of file FILEA on drive 1
```

DUPLICATE

Use PRINT# to duplicate a diskette and thus generate a backup copy of it. (See also the BASIC 4.0 BACKUP statement.)

Format:

PRINT #lf,"D[UPLICATE]d=s"

The diskette in drive d becomes a duplicate of the diskette in drive s. Diskette name and number are copied, along with all data files.

Before duplicating a diskette it is wise to put write protect tabs on the diskette which is to be duplicated. Then if you put diskettes in the wrong drives, or if you mix the source and destination drive numbers in the PRINT# statement, you will simply get a diskette write error; you will not wipe out the diskette that you were trying to duplicate.

Example:

```
OPEN 1,8,15                 Open the diskette command channel
.
PRINT#1,"D0=1"              The diskette in drive 1 is duplicated; the duplicate is
.                             generated in drive 0
PRINT#1,"DUPLICATE0=1"      Same as above
```

INITIALIZE

Use PRINT# to initialize a diskette before performing any operation on it. You do not need to initialize diskettes if you are using a DOS release 2.0 or higher, and BASIC 4.0.

Format:

PRINT#file,"I[NITIALIZE][dr]

The diskette in drive dr is initialized. If the dr parameter is not present, diskettes in both drives are initialized.

Versions of DOS preceding release 2.0 require diskettes to be initialized before any file on the diskette is opened. BASIC 3.0 and earlier versions were used with these revisions of DOS. DOS 2.0 and subsequent releases automatically initialize diskettes when they are loaded into drive. BASIC 4.0 should be used with DOS 2.0 and subsequent releases.

You do not need to initialize a diskette after preparing it; the preparation process also initializes the diskette.

Example:

```
OPEN 1,8,15                 Open the diskette command channel
.
PRINT#1,"I"                 Initialize diskettes in drives 0 and 1
.
PRINT#1,"INITIALIZE1"       Initialize the diskette in drive 1
```

NEW

Use PRINT# to prepare and format a new diskette, or to erase and reformat an old diskette. (See also the BASIC 4.0 HEADER statement.)

Format:

PRINT#lf,"N[EW]dr:diskname,vv"

The diskette in drive dr is prepared. When a diskette is prepared, sectors are laid out on the diskette surface. The diskette directory and Block Availability Map (BAM) are initialized. The diskette is assigned the name diskname and the number vv.

The diskette name and number is displayed in the reverse field at the top of a directory display.

Example:

```
OPEN 1,8,15                 Open the diskette command channel
.
PRINT#1,"NO:NEWDATA,02"     A diskette has been prepared for use in drive 0. The diskette is given
                              the name NEWDATA and the number 02
```

When preparing an old diskette, you can specify a new diskette name, while keeping the old diskette number; or you can keep the old diskette name and number. For example, suppose a diskette has the name NEWDATA and the number 02. The following preparation statements are legal:

```
OPEN 1,8,15                    Open the diskette command channel
.
PRINT#1,"NEW0"                 Prepare an old diskette in drive 0. Keep its old name and number.
PRINT#1,"N1:NEWDISK"           Prepare an old diskette in drive 1. Rename the diskette
                                 NEWDATA but keep the old diskette number
PRINT#1,"N1:NEWDATA,01"        As above but give the diskette the number 01
```

The following statement is illegal:

```
PRINT#1,"N0:02"
```

This statement is attempting to give the old diskette a new number while keeping the old name.

RENAME

Use PRINT# to rename a diskette file. (See also the BASIC 4.0 RENAME statement.)

Format:

PRINT#lf,"R[ENAME]dr:newname=oldname"

A file on the diskette in drive dr has its name changed from oldname to newname.

Example:

```
OPEN 1,8,15                    Open the diskette command channel
.
PRINT#1,"R1:BACKUP=CURRENT"    The file on the diskette in drive 1 which was named
                                 CURRENT is renamed BACKUP
```

SCRATCH

Use PRINT# to scratch one or more files on a diskette. (See also the BASIC 4.0 SCRATCH statement.)

Format:

PRINT#lf,"Sdr:filename[,dr:filename]"

A single PRINT# statement can delete one file, many files or all files, on a single diskette, or on both diskettes.

To scratch one or more files, specify the drive number and file name for each file that is to be scratched.

When a number of similarly named files are to be scratched, use the asterisk (*) and question mark (?) characters to name the files.

The asterisk (*) is used to scratch a number of files whose names have the same beginning characters. Enter the common beginning file name characters, followed by an asterisk. For example the name "FILE*" will scratch all files whose names begin with the four letters FILE. The name "F*" will scratch all files whose names begin with the letter F. The name "*" will scratch all files on the diskette. The asterisk (*) may be used in the same way to specify names for OPEN, DOPEN and DLOAD statements.

Use the question mark (?) in file name character positions that are allowed to differ. For example the name "FILE?.SRC" will scratch all files named "FILEX.SCR" where X can be any character. The name "F???NO" will scratch any file whose name begins with an F, ends with NO and has any three characters in between. The name F???N* will scratch any file whose name has an F in the first character position and an N in the fifth character position.

Example:

```
OPEN 1,8,15                          Open the diskette command channel
.
PRINT#1,"S0:FILENAME"                Scratch the file on drive 0 named FILENAME
PRINT#1,"S0:FILENAME,1:NEWFILE"      As above but also scratch the file on drive 1 named NEWFILE
PRINT#1,"S0:FILENAME,0:NEW*"         As above but also scratch all files on drive 0 whose
                                       names begin with the letters NEW
PRINT#1,"S1:A???"                    Scratch all files on drive 1 whose names begin with A and
                                       have a total of 4 characters in the name
PRINT#1,"S0:*"                       Scratch all files on the diskette in drive 0
```

VALIDATE

Use PRINT# to validate a diskette. (See also the BASIC 4.0 COLLECT statement.)

Format:

PRINT#lf,"V[ALIDATE][dr]"

The diskette in drive dr is validated. If the dr parameter is absent, then the diskette in the most recently selected drive is validated.

When a diskette is validated, a new Block Availability Map is created for all valid data files on the diskette. Any files that were improperly closed, or were not closed become invalid files; they are deleted from the diskette and their diskette space is released.

Do not validate a diskette that contains random access files; validation will erase the random access file.

If a read error occurs during validation, the validation operation is aborted and the diskette is left in its initial state.

A diskette must be initialized after it is validated.

Example:

```
OPEN 1,8,15                          Open the diskette command channel
PRINT#1,"I0"                         Initialize the diskette in drive 0
PRINT#1,"V0"                         Validate the diskette in drive 0
```

READ

The READ statement assigns values from a DATA statement to variables named in the READ parameter list.

Format:

READ var[,var,. . .,var]

READ is used to assign values to variables. READ can take the place of multiple assignment statements (see LET=).

READ statements with variable lists require corresponding DATA statements with lists of constant values. The data constants and corresponding variables have to agree in type. A string variable can accept any type of constant; a numeric variable can accept only numeric constants.

The number of READ and DATA statements can differ, but there has to be an available DATA constant for every READ statement variable.

There can be more data items than READ statement variables, but if there are too few data items the program aborts with an ?OUT OF DATA error message.

READ is generally executed in program mode. It can be executed in immediate mode as long as there are corresponding DATA constants in the current stored program to read from.

Example:

```
10 DATA 1,2,3              On completion, A=1, B=2, C=3
20 READ A,B,C
150 READ C$,D,F$           On completion, C$="STR", D=14.5, F$="TM"
160 DATA STR
170 DATA 14.5,"TM"
```

RECORD (BASIC 4.0)

The RECORD statement adjusts a relative file pointer to select any byte (character) of any record in the relative file. The RECORD statement is used before GET#, INPUT# or PRINT# statements.

Format:

RECORD#lf,rno[,bno]

The RECORD statement selects byte number bno in record rno of the file identified by logical file lf.

If the RECORD statement sets the file pointer beyond the end of the file, and a PRINT# statement attempts to write another record, the file is extended to include these additional records. If an INPUT# statement is executed after the RECORD statement has set the record pointer beyond the last record, INPUT# will return null data and an end of file status is generated in ST, the status word variable.

Example:

```
10 DOPEN#1,"DATAFILE",L20,6: REM RELATIVE FILE DATAFILE HAS 20 BYTES PER RECORD
20 RECORD#1,20,6: REM SELECT THE 6TH BYTE RECORD NO. 20
30 GET#1,A$:IF A$="" THEN 30: REM LOAD THIS BYTE INTO A$
40 STOP
```

REM

The Remark statement (REM) allows comments to be placed in the program for program documentation purposes.

Format:

REM comment

where:

comment is any sequence of characters that will fit on the current 80 column line.

REM statements are reproduced in program listings, but they are otherwise ignored. A REM statement may be placed on a line of its own or it may be placed as the last statement on a multiple statement line.

A REM statement cannot be placed ahead of any other statements on a multiple-statement line, since all text following the REM is treated as a comment.

REM statements may be placed in the path of program execution, and they may be branched to.

Example:

```
10 REM *** * * * * * * * * * * * * * * * * ****
20 REM ***PROGRAM EXCALIBUR***
30 GOTO 55:REM BRANCH IF OUT OF DATA
```

RENAME (BASIC 4.0)

The RENAME statement changes the name of a file on a diskette without altering the file. (See also PRINT# RENAME.)

Format:

RENAME[dr]"oldname" TO "newname"[ON Uz]

The file named oldname on the diskette in drive dr has its name changed to newname. If dr is absent, drive 0 is assumed.

If you have any trouble renaming a file, try to validate the file, then rename it.

Caution: A file must be closed before it is renamed.

Example:

```
RENAME "PET" TO "CBM"          Rename PET file on drive 0. The new file name is CBM
RENAME D1, "ONE" TO "TWO"      Rename ONE file on drive 1. The new file name is TWO
```

RESTORE

The RESTORE statement resets the DATA statement pointer to the beginning of data.

Format:

RESTORE

RESTORE may be given in immediate or program mode.

Example:

```
10 DATA 1,2,N44
20 READ A,B,B$      A=1, B=2, B$="N44"
30 RESTORE
40 READ X,Y,Z$      X=1, Y=2, Z$="N44"
```

RETURN

The RETURN statement branches program control to the statement in the program following the most recent GOSUB call. Each subroutine must terminate with a RETURN statement.

Format:

RETURN

Example:

```
100 RETURN
```

Note that the RETURN statement returns program control from a subroutine, whereas the RETURN key moves the cursor to the beginning of the next display line. The two are not related in any way.

RUN

RUN begins execution of the program currently stored in memory. RUN closes any open files, and initializes all variables to 0 or null values.

Format:

RUN[line]

When RUN is executed in immediate mode, the CBM computer performs a CLR of all program variables and resets the data pointer in memory to the beginning of data (see RESTORE) before executing the program.

If RUN specifies a line number, the CBM computer still performs the CLR and RESTOREs the data, but execution begins at the specified line number.

RUN specifying a line number should not be used following a program break — use CONT or GOTO for that purpose.

The RUN may also be used in program mode. It restarts program execution from the beginning of the program with all variables cleared and data pointers re-initialized.

Example:

```
RUN              Initialize and begin execution of the current program

RUN 1000         Initialize and begin execution of the program starting
                   at line 1000
```

SAVE

The SAVE statement writes a copy of the current program from memory to an external device. (Also see DSAVE.)

Cassette Unit Format:

SAVE ["filename"][,dev][,sa]

The SAVE statement writes the program which is currently in memory to the tape cassette drive specified by dev. If the dev parameter is not present then the assumed value is 1 and the primary cassette drive is selected. The filename, if specified, is written at the beginning of the program. If a non-zero secondary address (sa) is specified, then an end of file mark is written on the cassette after the saved program.

Although none of the SAVE statement parameters are required when writing to a cassette drive, it is a good idea to name all programs. A named program can be read off cassette tape either by its name, or by its location on the cassette tape. A program with no name can be read off cassette tape by its location only.

The SAVE statement is most frequently used in immediate mode, although it can be executed from within a program.

For cassette operating instructions when using the SAVE statement see Chapter 2.

Example:

```
SAVE                          Write the current program onto the cassette in drive 1,
                                 leaving it unnamed
SAVE "RED"                    Write the current program onto the cassette in drive 1,
                                 assigning the file name of RED
A$="RED"                      Same as above
SAVE A$

SAVE "BLACKJACK",2,1          Write the current program onto the cassette in drive 2
                                 naming the program BLACKJACK. Write and end of
                                 file mark after the program
```

Diskette Drive Format:

SAVE "dr:filename",dev

The SAVE statement writes a copy of the current program from memory to the diskette in the drive specified by dr. The program is given the name filename. dev must be present; in all standard CBM computer systems it has the value 8. If dev is absent, a default value of 1 is assumed and the primary cassette is selected.

The file name assigned to the program must be new. If a file with the same name already exists on the diskette, a syntax error is reported. However a program file can be replaced; if an @ sign precedes dr in the SAVE statement text string, then using DOS 2.0 or higher, the program replaces the contents of a current file named filename.

The diskette SAVE statement is also used primarily in immediate mode although it can be executed out of a program.

For diskette operating instructions see Chapter 2.

Example:

```
SAVE "0:BLACKJACK",8      Write the current program to the diskette on drive 0 and name the
                            program file BLACKJACK
SAVE "@0:BLACKJACK",8     Write the current program to the diskette on drive 0, replacing prior
                            contents of program file BLACKJACK
```

SCRATCH (BASIC 4.0)

The SCRATCH statement erases a single file from a diskette. (Also see PRINT# SCRATCH.)

Format:

SCRATCH [Dd],"filename"[ON Uz]

The file named filename on the diskette in drive d is deleted. If the Dd parameter is not present, drive 0 is assumed.

The SCRATCH statement is used in immediate mode and in program mode. In immediate mode the statement is used to perform general diskette housekeeping operations. When executed the message ARE YOU SURE? is displayed. You must key the response YES <CR> or Y <CR>, or the file will not be scratched.

When the SCRATCH statement is executed out of a program, no prompt messages are displayed. Temporary data files are frequently created by a program to hold transient data that will not fit in available memory. Temporary data files should be scratched before the program completes execution; otherwise a FILE EXISTS syntax error will be generated when the program is run next.

Files must be closed before they are scratched. If you attempt to scratch an open file the CBM computer may perform complex, erroneous diskette operations.

If using DOS 2.0 it is a good idea to COLLECT the diskette in immediate mode before scratching any files (see COLLECT).

Example:

```
SCRATCH D0,"DUMMY1"       Scratch file DUMMY1 on diskette drive 0
SCRATCH "DUMMY1"          Same as above
SCRATCH D1,"FILE1"        Scratch FILE1 on diskette drive 1
```

STOP

The STOP statement causes the program to stop execution and return control to CBM BASIC. A break message is displayed on the screen.

Format:

STOP

Example:

```
655 STOP                  Will cause the message BREAK IN 655 to be displayed
```

VERIFY

The VERIFY statement compares the current program in memory with the contents of a program file.

Cassette Unit Format:

VERIFY ["filename"][,dev]

The program currently in memory is compared with the program named filename on the cassette in the unit specified by dev. If dev is not present, a default of 1 is assumed and cassette unit 1 is selected. If filename is not present, the next file on the cassette in the selected unit is verified.

You should always verify a program immediately after saving it.

The VERIFY statement is almost always executed in immediate mode. For cassette operating instructions see Chapter 2.

Example:

```
VERIFY                  Verify the next program found on the tape

VERIFY "CLIP"           Search for the program named CLIP on cassette unit # 1, and verify it

A$="CLIP"               Same as above
VERIFY A$
```

Diskette Drive Format:

VERIFY "dr:filename",dev

The program currently stored in memory is compared with the program file named filename on the diskette in drive dr. The dev parameter must be present and in all standard CBM computer systems it must have the value 8. If the dev parameter is absent a default value of 1 is assumed and the primary cassette drive is selected.

In order to verify the program most recently saved, use the following version of the VERIFY statement:

```
VERIFY "*",8
```

You should always verify programs as soon as you have saved them.

The VERIFY statement is nearly always executed in immediate mode. For diskette operating instructions see Chapter 2.

Example:

```
VERIFY "*",8            Verify the program just saved

VERIFY"0:SHELL",8       Search for the program named SHELL on disk drive 0, and verify it

C$="0:SHELL"            Same as above
VERIFY C$
```

WAIT

The WAIT statement halts program execution until a specified memory location acquires a specified value.

Format:

```
WAIT memadr, mask[,xor]
```

where:

mask	is a one-byte mask value
xor	is a one-byte mask value

The WAIT statement executes as follows:

1. The contents of the addressed memory location are fetched.
2. The value obtained in step 1 is Exclusive-ORed with xor, if present. If xor is not specified, it defaults to 0. When xor is 0, this step has no effect.
3. The value obtained in step 2 is ANDed with the specified mask value.
4. If the result is 0, WAIT returns to step 1, remaining in a loop that halts program execution at the WAIT.
5. If the result is not 0, program execution continues with the statement following the WAIT statement.

The STOP key will not interrupt WAIT statement execution.

FUNCTIONS

CBM BASIC functions are described below in alphabetic order. Names and abbreviations used are described at the beginning of this chapter.

A few functions are available only on CBM 8000 series computers; these functions are described in the next section.

ABS

ABS returns the absolute value of a number. This is the value of the number with-tions are described in the next section.

Format:

```
ABS(datan)
```

Example:

```
A=ABS(10)                          Results in A=10
A=ABS(-10)                         Results in A=10
PRINT ABS(X),ABS(Y),ABS(Z)
```

ASC

ASC returns the ASCII code number for a specified character.

Format:

ASC(data$)

If the string is longer than one character, ASC returns the ASCII code for the first character in the string. The returned argument is a number and may be used in arithmetic operations. ASCII codes are listed in Appendix A.

Example:

```
?ASC("A")        Prints 65
N=ASC(B$)
X=ASC("S")
#X               Prints the ASCII value of "S", which is 83
```

ATN

ATN returns the arctangent of the argument.

Format:

ATN(datan)

ATN returns the value in radians in the range ±17.

Example:

```
A=ATN(AG)
?180/π*ATN(A)
```

CHR$

CHR$ returns the string representation of the specified ASCII code.

Format:

CHR$(byte)

CHR$ can be used to specify characters that cannot be represented in strings. These include a carriage return and the double quotation mark.

Example:

```
IF C$=CHR$(13) GOTO 10          Branch if C$ is a carriage return (CHR$(13))

?CHR$(34);"HOHOHO";CHR$(34)     Print the eight characters "HOHOHO" (where CHR$(34)
                                  represents a double quotation mark)
```

COS

COS returns the cosine of the argument.

Format:

COS(datan)

Example:

```
AG=0.25
A=COS(AG)      A is assigned the value 0.968912422
```

DS (BASIC 4.0)

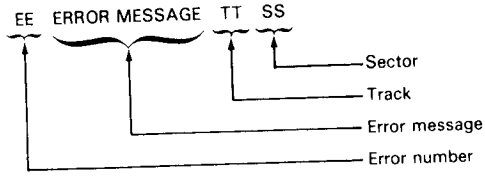
Whenever the variable DS is referenced by any BASIC statement, an integer number is returned specifying the status of the most recent disk access operation. See Table 8-1 for DS interpretations.

Example:

```
20 IF DS<>0 THEN PRINT "ERROR" :STOP
```

DS$ (BASIC 4.0)

When the string variable named DS$ is referenced by any BASIC statement the status of the most recent disk access is returned with the following format:



See Appendix B for a summary of diskette error messages.

Example:

```
20 IF DS>20 THEN PRINT DS$ :STOP
```

If DS has a value of 1, a file has been scratched; any other value less than 20 is no error.

EXP

EXP returns the value e^{arg}. The value of e used is 2.71828183.

Format:

EXP(argn)

argn must have a value in the range ±88.029691. A number larger than +88.029691 will result in an overflow error message. A number smaller than −88.029691 will yield a zero result.

Example:

```
?EXP(0)                 Prints 1
?EXP(1)                 Prints 2.71828183
EV=EXP(2)               Results in EV=7.3890561
EB=EXP(50.24)           Results in EV=6.59105247E+21
?EXP(88.0296919)        Largest allowable number, yields 1.70141183E+38
?EXP(-88.0296919)       Smallest allowable number, yields 5.87747176E-39
?EXP(88.029692)         Out of range, overflow error message
?EXP(-88.029692)        Out of range, returns 0
```

FRE

FRE is a system function that collects all unused bytes of memory into one block (called "garbage collection") and returns the number of free bytes.

Format:

FRE(arg)

arg is a dummy argument. It may be string or numeric.

FRE can be used anywhere a function may appear, but it is normally used in an immediate mode PRINT statement.

Example:

```
?FRE(1)                 Institute garbage collection and print the number
                          of free bytes
```

INT

INT returns the integer portion of a number, rounding to the next lower signed number.

Format:

INT(argn)

For positive numbers, INT is equivalent to dropping the fractional portion of the number without rounding. For negative numbers, INT is equivalent to dropping the fractional portion of the number and adding 1. Note that INT does *not* convert a floating point number (5 bytes) to integer type (2 bytes).

Example:

```
A=INT(1.5)              Results in A=1
A=INT(-1.5)             Results in A=-2
X=INT(-0.1)             Results in X=-1
```

A caution here: Since floating point numbers are only close approximations of real numbers, an argument may not yield the exact INT function value you might expect. For instance, consider the number 3.89999999. The function *INT(3.89999999) would yield a 3 answer, not 4 as would be expected:

```
?INT(3.89999999)
3
```

LEFT$

LEFT$ returns the leftmost characters of a string.

Format:

```
LEFT$(arg$,byte)
```

byte specifies the number of leftmost characters to be extracted from the arg$ character string.

Example:

```
?LEFT$("ARG",2)        Prints AR
A$=LEFT$(B$,10)        Prints leftmost ten characters of B$ string
```

LEN

LEN returns the length of the string argument.

Format:

```
LEN(arg$)
```

LEN returns a number that is the count of characters in the specified string.

Example:

```
?LEN("ABCDEF")         Displays 6
N=LEN(C$+D$)           Displays the sum of characters in strings C$ and D$
```

LOG

LOG returns the natural logarithm, or log to the base e. The value of e used is 2.71828183.

Format:

```
LOG(argn)
```

An ILLEGAL QUANTITY ERROR message is returned if the argument is zero or negative.

Example:

```
?LOG(1)                Prints 0
A=LOG(10)              Results in A=2.30258509
A=LOG(1E6)             Results in A=13.8155106
A=LOG(N)/LOG(10)       Calculates log to the base 10
```

MID$

MID$ returns any specified portion of a string.

Format:

MID$(data$,byte$_1$[,byte$_2$])

Some number of characters from the middle of the string identified by data$ are returned. The two numeric parameters byte$_1$ and byte$_2$ determine the portion of the string which is returned. String characters are numbered from the left, with the leftmost character having position 1. The value of byte$_1$ determines the first character to be extracted from the string. Beginning with this character, byte$_2$ determines the number of characters to be extracted. If byte$_2$ is absent then all characters up to the end of the string are extracted.

An ILLEGAL QUANTITY ERROR message is printed if a parameter is out of range.

Example:

```
?MID$("ABCDE",2,1)      Prints B
?MID$("ABCDE",3,2)      Prints CD
?MID$("ABCDE",3)        Prints CDE
```

PEEK

PEEK returns the contents of the specified memory location. PEEK is the function counterpart of the POKE statement.

Format:

PEEK(memadr)

Any memory location can be PEEKed except for system locations that contain the BASIC interpreter. These locations have been PEEK-protected to discourage examination of proprietary software. The protected area returns a PEEK value of 0. Locations of interest that you might want to PEEK at are discussed in Chapter 7.

Example:

```
?PEEK(1)                Prints contents of memory location 1
A=PEEK(20000)
```

POS

POS returns the column position of the cursor.

Format:

POS(data)

data is a dummy function; it is not used and therefore can have any value.

POS returns the current cursor position. If no cursor is displayed, the current character position within a program line or string variable is returned. Character positions begin at 0 for the leftmost character.

For a 40 column display POS will return a value between 0 and 39. For an 80 column display POS will return a value between 0 and 79.

Recall that program logic processes 80 character lines even if a CBM computer has a 40 character display. If program logic in such a computer is processing a character in the second half of the line, the POS function will return a value between 40 and 79, even though the computer only has a 40 character display.

By concatenation, string variables with up to 255 characters may be generated. If program logic is processing a long string, then the POS function will return the character position currently being processed. Under these circumstances the POS function will return a value ranging between 0 and 255.

Example:

```
?POS(1)                    At the beginning of a line, returns 0
?"ABCABC";POS(1)           With a previous POS value of 0, displays a
                             POS value of 6
```

RIGHT$

RIGHT$ returns the rightmost characters in a string.

Format:

```
RIGHT$(arg$,byte)
```

byte identifies the number of rightmost characters that are extracted from the string specified by arg$.

Example:

```
RIGHT$(ARG,2)              Displays RG
MM$=RIGHT$(X$+"#",5)       MM$ is assigned the last four characters of X$, plus the
                             character #
```

RND

RND generates random number sequences ranging between 0 and 1.

Format:

```
RND(argn)          Return random number
RND(-argn)         Store new seed number
```

Example:

```
A=RND(-1)          Store a new seed based on the value -1
A=RND(1)           Fetch the next random number in sequence
```

An argument of zero is treated as a special case; it does not store a new seed, nor does it return a random number. RND(0) uses the current system time value TI to introduce an additional random element into play.

A pseudo-random seed is stored by the function:

```
RND(-TI)           Store pseudo-random seed
```

RND(0) can be used to store a new seed that is more truly random, by using the following function:

```
RND(-RND(0))          Store random seed
```

For a complete discussion of the RND function see Chapter 5.

SGN

SGN determines whether a number is positive, negative, or zero.

Format:

SGN(argn)

The SGN function returns +1 if the number is positive, non-zero; 0 if the number is zero; −1 if the number is negative.

Example:

```
?SGN(-6)              Displays -1
?SGN(0)               Displays 0
?SGN(44)              Displays 1
IF A>C THEN SA=SGN(X)
IF SGN(M)>= 0 THEN PRINT "POSITIVE NUMBER"
```

SIN

SIN returns the sine of the argument.

Format:

SIN(argn)

Example:

```
A=SIN(AG)
?SIN(45*π/180)        Displays the sine of 45 degrees
```

SPC

SPC moves the cursor right a specified number of positions.

Format:

SPC(byte)

The SPC function is used in PRINT statements to move the cursor some number of character positions to the right. Text which the cursor passes over is not modified.

The SPC function moves the cursor rightward from whatever column position the cursor happens to be at when the SPC function is encountered. This is in contrast to a TAB function which moves the cursor to some fixed column measured from the leftmost column of the display. (See TAB for examples.)

SQR

SQR returns the square root of a positive number. A negative number returns an error message.

Format:

SQR(argn)

Example:

```
A=SQR(4)          Results in A=2
A=SQR(4.84)       Results in A=2.2
?SQR(144E30)      Displays 1.2E+16
```

ST

ST returns the current value of the I/O status. This status is set to certain values depending on the results of the last input/output operation.

Format:

ST

ST values are shown in Table 8-3.

Status should be checked after execution of any statement that accesses an external device. See Chapter 6 for a complete discussion of I/O status.

Example:

```
10 IF ST <>0 GOTO 500                    Branch on any error
50 IF ST=4 THEN ?"SHORT BLOCK"
```

STR$

STR$ returns the string equivalent of a numeric argument.

Format:

STR$(argn)

STR$ returns the character string equivalent of the number generated by resolving argn.

Example:

```
A$=STR$(14.6)     Displays 14.6
?A$
?STR$(1E2)        Displays 100
?STR$(1E10)       Displays 1E+10
```

Table 8-3. ST Values for I/O Devices

ST Bit Position	ST Numeric Value	Cassette Tape Read	Cassette Tape Verify and Load	IEEE Devices Read/Write
0	1			Time out write
1	2			Time out read
2	4	Short block	Short block	
3	8	Long block	Long block	
4	16	Unrecoverable read error	Any mismatch	
5	32	Checksum error	Checksum error	
6	64	End of file		EOI
7	-128	End of tape	End of tape	Device not present

SYS

SYS is a system function that transfers program control to an independent subsystem.

Format:

SYS(memadr)

memadr is the starting address at which execution of the subsystem is to begin. The value must be in the range 0<address<65535. SYS is described in Chapter 7.

TAB

TAB moves the cursor right to the specified column position.

Format:

TAB(argn)

TAB moves the cursor to the n+1 position, where n is the number obtained by resolving argn.

Example:

```
?"QUARK";SPC(10);"W"
QUARK          W

?"QUARK";TAB(10);"W"
QUARK      W
```

These two examples show the difference between SPC and TAB. SPC skips ten positions from the last cursor location, whereas TAB skips to the 10+1th position on the row

Using the TAB Key

Recent CBM computers have a TAB key. This key can be used within a PRINT statement's text string to set tabs, clear tabs, or move the cursor right to the next tab stop.

Tabs are set and cleared using the shifted TAB key, or the CHR$(9) function. A tab is cleared if the cursor is in a column where a tab was previously set; a tab is set otherwise.

Tabs may be set and cleared in immediate mode or in program mode. To set or clear tabs in immediate mode simply move the cursor to the desired screen column then

press the shifted TAB key. In program mode execute a PRINT statement that moves the cursor to the required column position, then execute a shifted tab character.

Up to 80 tabs may be set. Execution of a carriage return makes tab settings permanent until cleared.

The unshifted TAB key or the CHR$(137) function moves the cursor right to the next tab column.

Example:

The following example sets tabs at columns 15, 25, and 50, then displays the words one, two, and three at these three column positions:

```
10 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■"
20 PRINT"■ONE■TWO■T⌐REE"
```

TAN

TAN returns the tangent of the argument.

Format:

TAN(argn)

Example:

```
?TAN(3.2)                Displays 0.0584738547
XY(1)=TAN(180*π/180)
```

TI, TI$

TI and TI$ represent two system time variables.

Format:

TI	Number of jiffies since current startup
TI$	Time of day string

Example:

```
?TI
TI$="081000"
```

Usages of TI and TI$ are described in Chapter 5, under "Setting Time of Day."

USR

USR is a system function that passes a parameter to a user-written assembly language subroutine whose address is contained in memory locations 1 and 2. USR also fetches a return parameter from the subroutine.

Format:

USR(arg)

The USR function is described in more detail in Chapter 7.

VAL

VAL returns the numeric equivalent of the string argument.

Format:

VAL(data$)

The number returned by VAL may be used in arithmetic computations.

VAL converts the string argument by first discarding any leading blanks. If the first non-blank character is not a numeric digit (0-9), the argument is returned as a value of 0. If the first non-blank is a digit, VAL begins converting the string into real number format. If it subsequently encounters a non-digit character, it stops processing so that the argument returned is the numerical equivalent of the string up to the first non-digit character.

Example:

```
A=VAL("123")
NN=VAL(B$)
```

CBM 8000 EDITING FUNCTIONS

The CBM 8000 Computer also supports the following unique functions.

BELL

BELL rings the console bell of appropriately equipped CBM 8000 computers.

Format:

CHR$(7) or <ESC><RVS>g

The bell rings whenever BELL format characters appear in a PRINT statement parameter list. The bell rings automatically on power-up, or when the cursor moves through column 75 of the display. If the screen window has been narrowed using window scrolling functions, then the bell sounds when the cursor passes through the fifth column from the right edge of the window.

Example:

```
100 PRINT CHR$(7)
```

DELETE LINE (BASIC 4.0)

Delete a line on the display. Scroll up all text below the deleted line.

Format:

CHR$(21) or <ESC><RVS>u

To delete a line include one of the formats illustrated above in a PRINT statement parameter list. The line on which the cursor is currently located gets deleted. The line is deleted on the display only; memory is not modified. This function should be used in programs that create displays; it should not be used to erase data from memory.

Example:

```
PRINT"<HOME><CRSR↓><CRSR↓><CRSR↓><ESC><RVS>U"      Delete the fourth display line
```

ERASE BEGIN

ERASE BEGIN erases all text on the current cursor line from the beginning of the line up to the cursor position.

Format:

CHR$(150) or <ESC><RVS>V

To access the ERASE BEGIN function, one of the formats illustrated above must appear in a PRINT statement parameter list. The display line on which the cursor is located is erased from the beginning of the line up to the cursor position but memory is not modified. This function should only be used in programs that are controlling screen displays.

Example:

```
100 PRINT TAB(20);CHR$(150)      Erase first 20 characters of line
```

ERASE END

ERASE END erases all text on the current cursor line from the cursor position up to the end of the line.

Format:

CHR$(22) or <ESC><RVS>v

To access the ERASE END function, one of the formats illustrated above must appear in a PRINT statement parameter list. The display line on which the cursor is located is erased from the cursor position up to the end of the line, but memory is not modified. This function should only be used in programs that are controlling screen displays.

Example:

```
100 PRINT TAB(20);CHR$(22)      Erase line starting at character 20
```

GRAPHIC

The GRAPHIC function changes the screen display from text to graphic characters.

Format:

CHR$(142) or <ESC><RVS>N

The GRAPHIC function is enabled when one of the formats illustrated above is encountered in a PRINT statement parameter list. The standard character set is selected for those characters which have a graphic symbol. Also, spacing between lines is eliminated to improve the quality of graphics.

The effect of the GRAPHIC function is cancelled by the TEXT function.

Example:

```
PRINT CHR$(142)        Select graphics display
```

INSERT LINE

The INSERT LINE function inserts one blank line at the cursor position on the screen display.

Format:

CHR$(149) OR <ESC><RVS>m

A line is inserted in the screen display at the current cursor position when one of the character formats illustrated above is encountered in a PRINT statement parameter list. The display below the inserted line is scrolled down one line; the bottom display line is scrolled off the screen.

The insert line function modifies the screen display but does not alter memory. This function should be used only in programs that are creating and modifying displays.

Example:

```
PRINT "<HOME><CRSR↓><CRSR↓><CRSR↓><ESC><RVS>M"   Insert a line at display line 4
```

SCROLL DOWN AND SCROLL UP

These two functions scroll text down one line, or up one line within a display window.

Format:

Scroll Down: CHR$(153) or <ESC><RVS>Q
Scroll Up: CHR$(25) or <ESC><RVS>q

The SET BOTTOM and SET TOP functions can be used to define a window on the CBM computer display. Within this window the SCROLL DOWN function will scroll text down one line; a blank line appears at the top of the window, while the bottom line of the window is scrolled off the screen. The SCROLL UP function scrolls text up one line within the window, scrolling the top line off the screen, while a blank line is inserted at the bottom of the window. These two functions are enabled when they appear in a PRINT statement parameter list.

The SCROLL UP and SCROLL DOWN functions modify the display, but do not change memory. These two functions should only be used in programs that create displays.

Example:

```
10 PRINT CHR$(25)      Scroll up one line within window
```

SET BOTTOM AND SET TOP

These two functions define a window on the CBM computer display.

Format:

Set Bottom: CHR$(143)
Set Top: CHR$(15)

The SET BOTTOM function defines the bottom righthand corner of the screen. The SET TOP function defines the top lefthand corner of the screen. In order to define the window a PRINT statement parameter list must move the cursor to the required bottom right and top left corners of the window and then execute the SET BOTTOM and SET TOP functions respectively.

To cancel a window, execute a PRINT statement with two consecutive HOME characters in its parameter list.

Example:

Suppose a display window is to be bounded by rows 5 and 15, and columns 10 and 60. The following PRINT statement would establish the required window:

```
10 PRINT"■■■■■■";TAB(10);CHR$(15);"■■■■■■■■■■";TAB(60);CHR$(143)
```

Subsequently the following PRINT statement would cancel the window:

```
100 PRINT "<HOME><HOME>"
```

TEXT

The TEXT function cancels the effect of the GRAPHIC function. Characters that have a graphic symbol in the standard character set are switched to the alternate character set representation.

Format:

CHR$(14) or <ESC><RVS>n

The TEXT function is enabled by executing a PRINT statement with one of the formats illustrated above in its parameter list.

Example:

```
100 PRINT CHR$(14)      End graphics
```